ZBIRKA ZADATAKA IZ PREVOĐENJA PROGRAMSKIH JEZIKA

KUZMANOVIĆ VLADIMIR
VULOVIĆ ANA

Izdato od strane „Vulović Ana, Kuzmanović Vladimir”

https://theikeofficial.gitlab.io/PPJ/

Prvo izdanje, Maj 2022.
Poslednja izmena: 2022-05-15 12:15

Sadržaj

Predgovor

U okviru kursa „Prevođenje programskih jezika”, na 3. godini smera Informatika na Matematičkom fakultetu Univerziteta u Beogradu izučavaju se regularni jezici i kontekstno slobodni jezici. U cilju upoznavanja studenata sa osnovnim konceptima koji su neophodni za uspešnu leksičku i sintaksnu analizu programskog koda.

Ova zbirka zadataka sa rešenjima predstavlja kolekciju zadataka sa časova vežbi i zadataka sa nekih praktičnih provera. Prvo poglavlje je posvećena regularnim izrazima koji predstavljaju osnovu za teme iz preostalih poglavlja.

Autori

1. Regularni jezici

Regularni izraz (*eng. regular expression*), predstavlja niz karaktera kojim se definiše šablon koji treba pronaći u tekstu. Najčešće, razni programi za pretragu teksta koriste regularne izraze prilikom implementacije Find i Find & Replace operacija ili za validaciju ulaznih podataka. Regularni izrazi su tehnika razvijena u teorijskom računarstvu i formalnoj teoriji jezika. Veliki broj aplikacija i programskih jezika implementira regularne izraze na različite načine, sa manjim ili većim razlikama. U nastavku poglavlja biće pokrivena sintaksa koju podržava veliki broj popularnih aplikacija i programskih jezika. U okviru ovog poglavlja biće navođeni regularni izrazi i tekst u kome se traga za podniskom koja bi se poklopila sa regularnim izrazom. Pronađeno poklapanje biće istaknuto zelenom bojom u nisci koja će biti istaknuta bež bojom.

1.1 Literali

Najjednostavniji regularni izraz sastoji se samo od jednog karaktera, npr. karaktera `a`. U slučaju da imamo dat tekst `Danas je divan dan.`, pomoću ovog regularnog izraza prepoznaćemo prvo slovo *a* posle slova *D*.

Regularni izraz	*Primeri teksta*
`a`	Danas je divan dan.

Tabela 1.1: Literal kao regularni izraz

Isti regeks može da prepozna i naredna pojavljivanja karaktera *a*, ali samo kada se modulu za uparivanje regularnih izraza saopšti da nastavi sa pretragom nakon prvog poklapanja. U editorima teksta ovo se najčešće postiže opcijom *Find Next*. U programskim jezicima, najčešće postoji posebna funkcija koja se poziva da bi se nastavila pretraga teksta nakon prethodnog poklapanja.

1.2 Specijalni karakteri

Dvanaest karaktera ima specijalno značenje u zapisu regularnih izraza: `\` , `^`, `$`, `.`, `|`, `?`, `*`, `+`, `(`, `)`, `[` i `{`. Specijalni karakteri se često nazivaju i metakarakterima i njihova samostalna upotreba najčešće dovodi do greške. Da bi se bilo koji od specijalnih karaktera koristio kao literal u okviru regularnog izraza, potrebno je navesti obrnuto kosu crtu ispred svakog od njih. Na primer, da bismo u tekstu pronašli nisku *1+1=2*, potrebno je da zapišemo regularni izraz kao `1\+1=2` . Ako zaboravimo `\`, tada karakter `+` ima specijalno značanje.

Regularni izraz	*Primeri teksta*	
`1\+1=2`	1+1=2	11+11=22
`1+1=2`	1+1=2	11+11=22

Tabela 1.2: Specijalni karakter + : prepoznavanje karaktera + ili ponavljanje karaktera 1

1.3 Karakterske klase

Karakterska klasa se koristi kada je potrebno prepoznati tačno jedan karakter iz većeg skupa karaktera. Da bi se prepoznao karakter *a* ili karakter *e*, potrebno je zapisati regularni izraz kao `[ae]`. Upotrebimo tu karaktersku klasu da u tekstu prepoznamo, na primer,jedninu ili množinu reči *banana*, tj. *banana* ili *banane*.

Regularni izraz	*Primeri teksta*			
`banan[ae]`	banana	banane	bananaa	bananea

Tabela 1.3: Elementarna karakterska klasa

Karakterska klasa predstavlja skup dozvoljenih karaktera i samim tim redosled navođenja karaktera u karakterskoj klasi nije važan. Unutar karakterske klase moguće je koristiti karakter `-`, da bi se naveo raspon uzastopnih karaktera iz ASCII tabele koje je moguće prepoznati. Na primer, karakterskom klasom `[0-9]` moguće je prepoznati bilo koju cifru između 0 i 9. Pored toga, moguće je kombinovati veći broj raspona za karaktere između sebe kao i sa pojedinačnim karakterima. Na primer, karakterska klasa `[0-9a-fA-F]` prepoznaće heksadekadnu cifru nezavisno od velikih ili malih slova.

Regularni izraz	*Primeri teksta*	
`[0-9]`	2020	-9
`[0-9A-Fa-f]`	xf8ac	-9

Tabela 1.4: Intervali karaktera unutar karakterske klase

Navođenjem karaktera `^` nakon otvorene srednje zagrade, negira se karakterska klasa. Rezultat ove operacije jeste da karakterska klasa koja prepoznaje bilo koji karakter koji se ne nalazi u karakterskoj klasi. Na primer, regularnim izrazom `b[^e]` prepoznaje se *ba* u reči *banana*, tj. prepoznaje se reč dužine dva karaktera koja čije je prvo slovo slovo *b* i drugo slovo je različito od slova *e*. Pod karakterima se podrazumeva čitava ASCII tabela.

U reči tetreb, neće se pronaći poklapanje, jer iza jedinog slova *b* i ujedno poslednjeg slova u reči ne postoji karakter koji bi mogao da bude prepoznat negiranom karakterskom klasom.

Regularni izraz	*Primeri teksta*		
`b[^e]`	banana	beba	tetreb

Tabela 1.5: Negirane karakterske klase

1.3.1 Skraćeni zapis karakterskih klasa

S obzirom da se neke karakterske klase često koriste, uveden je određeni broj skraćenih zapisa za karakterske klase. Dostupni su u većini modernih programskih jezika, naredni zapisi:

\d - predstavlja skraćeni zapis za karaktersku klasu za dekadne cifre, `[0-9]` ;
\w - predstavlja skraćeni zapis za karaktersku klasu za karaktere reči, `[a-zA-Z0-9_]`.
\s - predstavlja skraćeni zapis za karaktersku klasu za beline, `[ \t\r\n\f\v]`.

Skraćeni zapisi se mogu koristiti izvan i u srednjim zagradama, tj. u okviru karakterske klase. Na primer regularan izraz `\s\d` prepoznaje belinu koja je praćena cifrom. Dok, regularni izraz `[\s\d]` prepoznaje tačno jedan karakter koji je ili belina ili cifra. Kada bi se ova dva regularna izraza primenila na tekst *1 + 2 = 3*, prvi regularni izraz bi prepoznao 2 (razmak 2), a drugi regeks bi prepoznao 1.

Regularni izraz	*Primeri teksta*
`\s\d`	1 + 2 = 3
`[\s\d]`	1 + 2 = 3

Tabela 1.6: Predefinisane karakterske klase

Navedene skraćene verzije karakterskih klasa imaju i odgovarajuće negirane verzije:

\D - predstavlja skraćeni zapis za karaktersku klasu za dekadne cifre, `[^\d]` ;
\W - predstavlja skraćeni zapis za karaktersku klasu za karaktere reči, `[^\w]`.
\S - predstavlja skraćeni zapis za karaktersku klasu za beline, `[^\s]`.

Potrebno je obratiti posebnu pažnju kada se koriste negirane karakterske klase unutar srednjih zagrada, tj. u drugoj karakterskoj klasi. Naime, `[\D\S]` nije isto što i `[^\d\s]`. Drugi regularni izraz prepoznaje bilo koji karakter koji nije cifra i nije belina. Dakle, prepoznaće karakter *x*, ali neće *8*. Prvi regularni izraz, prepoznaće bilo koji karakter koji nije cifra ili koji nije belina. S obzirom da nijedna cifra nije belina i nijedna belina nije cifra, regularni izraz `[\D\S]` prepoznaće bilo koji karakter, tj. cifre, beline i sve ostale karaktere.

Regularni izraz	*Primeri teksta*		
`[^\s\d]`	8	x	
`[\S\D]`	8	x	

Tabela 1.7: Negirane predefinisane karakterske klase

1.3.2 Specijalni karakteri

U okviru karakterskih klasa jedini karakteri sa specijalnim značenjem su: `]`, `^`, `\` i `-`. Zatvorena srednja zagrada se koriste za zatvaranje karakterske klase i ukoliko baš taj karakter želimo da nevedemo upotrebe unutar klase neophodno je zaobići njihovo specijalno značenje navođenjem `\` ispred specijalnog karaktera ili navesti kao prvi karakter unutar karakterske klase. Crtica, tj. `-`, ima specijalno značenje samo ako se navede između dva karaktera i na taj način se definise raspon karaktera koji se mogu prepoznati karakterskom klasom. U slučaju da se crtica nalazi na početku ili na kraju karakterske klase, tada nema specijalno značenje. Slično važi i za karakter `^`, tj. kapicu. Kapica ima specijalno značenje samo kada se nalazi na početku karakterske klase i tada definiše negaciju iste. Na bilo kom drugom mestu, kapica nema nikakvo specijalno značenje. Obrnuta kosa crta, tj. `\` je specijalni karakter kojim uklanjamo specijalno značenje ostalim specijalnim karakterima. Ukoliko je neophodno da se baš `\` karakter prepoznam neohodno je ukloniti sepcijalno značenje. Dakle, navođenjem, `\\`. Svi ostali karakteri nemaju nikakve specijalne osobine kada se koriste u okviru karakterske klase.

Regularni izraz	*Primeri teksta*
`[]]`	[0,10]
`[\s\d]`	1 + 2 = 3

Tabela 1.8: Specijalni karakteri unutar karakterske klase

1.4 Specijalni karakter .

Karakter `.`, tj. tačka, prepoznaje tačno jedan bilo koji karakter, osim novog reda `\n`. Većina uparivača regularnih izraza poseduje režim u kome tačka prepoznaje sve karaktere uključujući i novi red, *eng. single line*. Na primer, pomoću regularnog izraza `a.c` mogu se prepoznati sledeće reči *aac*, *abc*, *a1c*, *a%c* itd.

Regularni izraz	*Primeri teksta*				
`a.c`	aac	abc	a1c	a%c	a c

Tabela 1.9: Karakter . u regularnim izrazima

Očigledno, potrebno je biti posebno obazriv prilikom upotrebe karaktera tačka u definiciji regularnog izraza. Karakterska klasa, obična ili negirana, često je brži i precizniji način definisanja regularnih izraza. Pored toga, karakter tačka nema specijalno značenje u okviru karakterske klase, tj. predstavlja sam sebe. Ukoliko je potrebno prepoznati karakter tačka u tekstu, a ne želimo da koristimo karakterske klase, neophodno je zaobići specijalno svojstvo karaktera tačka u regularnom izrazu tako što ćemo navesti `\` ispred same tačke.

1.5 Sidra

Sidra ne prepoznaju nijedan karakter, već se koriste prilikom definisanja pozicije na kojoj treba prepoznati tekst. Sidro `^` govori da je potrebno izvršiti poklapanje na početku teksta, a sidro `$` govori da je potrebno izvršiti poklapanje na kraju teksta. Većina implementacija

regularnih izraza podržava višelinijski režim, koji je često i podrazumevani, u kome sidro `^` vezuje poklapanje za početak reda, a sidro `$` za kraj reda.

Pored vezivanja poklapanja za početak i kraj reda, moguće je definisati i granice reči uz pomoć sidra `\b`. Granica reči se definiše kao pozicija između karaktera reči, tj. onog karaktera koji se može prepoznati sa `\w`, i karaktera koji se ne može prepoznati sa `\w`. `\b` prepoznaje početak ili kraj reči samo kada se prvi ili poslednji karakter te reči mogu prepozanati sa `\w`. `\B` prepoznaje sve one karaktere koji se ne mogu prepoznati sa `\b`.

Regularni izraz	*Primeri teksta*			
`^b`	bob	brod	parobrod	beli brod
`\bbrod`	brod	(brod)	parobrod	beli brod
`\Bbrod`	brod	(brod)	parobrod	beli brod
`\bzdrav\b`	zdrav	Pozdrav!	(zdrav)	zdravlje

Tabela 1.10: Sidra u regularnim izrazima

1.6 Alternacija

Alternacija u slučaju regularnih izraza ekvivalentna je logičkoj operaciji ili. Karakter `|` predstavlja alternaciju. Ako je dat tekst *danas je divan dan* i regularni izraz `dan|danas`, prepoznaće se reč *danas*.

U okviru regularnog izraza moguće je definisati proizvoljno mnogo alternativa, npr. `dan|danas|divan`. Alternacija ima najniži prioritet u poređenju sa svim ostalim operatorima regularnih izraza. Regularni izraz `suncan|divan dan` može da prepozna *suncan* ili *divan dan*. Da bismo prepoznali *divan dan* ili *ruzan dan*, potrebno je da grupišemo alternative i napravimo sledeći regularan izraz `(divan|ružan) dan`.

Regularni izraz	*Primeri teksta*	
`dan|danas`	danas je divan dan	suncani dani
`suncan|divan dan`	danas je divan dan	suncani dani
`(divan|ruzan) dan`	danas je divan dan	ruzan dan

Tabela 1.11: Znak alternacije u regularnim izrazima

1.7 Ponavljanje

Najjednostavniji oblik ponavljanja je zapravo omogućavanje da se neki tekst javi opciono, tj. jednom ili nijednom u tekstu. Operator `?` se koristi kada treba naznačiti da se regularni podizraz koji mu prethodi u regularnom izrazu može poklopiti jednom ili nijednom u prepoznatom tekstu. Na primer, regularni izraz `colou?r` može da se iskoristi za prepoznavanje reči *color* ili *colour*. Regularni podizraz koji prethodi operatoru `?` je baš literal `u` i on može i ne mora biti prepoznat u tekstu nakon drugog slova `o`.

Karakter `*` u regularnom izrazu označava da se regularni podizraz koji mu prethodi mora javiti nula ili više puta u prepoznatom tekstu. Karakter `+` označava da se regularni podizraz

koji mu prethodi mora javiti jednom ili više puta u prepoznatom tekstu. Na primer, regularni izraz `<[A-Za-z][A-Za-z0-9]*>` prepoznaće bilo koji HTML ili XML tag bez atributa.

Pored ovoga, moguće je navesti i tačan broj ponavljanja određenog regularnog podizraza. karaktera. Da bismo to postigli potrebno je da koristimo velike zagrade (`{, }`) i u njima da navedemo tačan broj ili raspon ponavljivanja datog regularnog podizraza. Na primer, regularni izraz `\b[1-9][0-9]{3}\b` prepoznaće prirodan broj između 1000 i 9999. Regularni izraz `\b[1-9][0-9]{2,4}\b` prepoznaće prirodan broj između 100 i 99999.

Regularni izraz	*Primeri teksta*		
`colou?r`	color	colour	
`<[A-Za-z][A-Za-z0-9]*>`	<h1>	<a href="http://math.rs">	
`\b[1-9][0-9]{3}\b`	2019	0198	001123678
`\b[1-9][0-9]{2,4}\b`	2019	0198	001123678

Tabela 1.12: Ponavljanja u regularnom izrazu

1.7.1 Pohlepno i lenjo ponavljanje

Operatori ponavljanja su pohlepni. Uvek će pokušati da ponavljaju poklapanje podizraza koliko god je moguće i zaustaviće se u poslednjem trenutku u kojem će i ostatak regularnog izraza biti zadovoljen. Da bi ponavljanje bilo lenjo, potrebno je navesti karakter `?` iza operatora `+`.

Regularni izraz	*Primeri teksta*	
`<.+>`	Prvi link..	prva i druga rec
`<.+?>`	Prvi link..	prva i druga rec
`colou??r`	color	colour

Tabela 1.13: Pohlepna i lenja ponavljanja u regularnom izrazu

1.8 Grupisanje i izdvajanje teksta

Moguće je regularni podizraz uokviriti malim zagradama. Na taj način na taj podizraz možemo primeniti prethodno navedene operatore, ili ograditi podizraz u kom se koristi znak alternacije, zbog niskog prioriteta tog operatora, kao što je već rađeno u 1.6 Pored toga, ograđivanjem podizraza malim zagradama dobijamo mogućnost da izdvojimo podnisku prepoznate niske, koju je prepoznao regularni podizraz iz zagrade. Svakom grupisanom podizrazu dodeljuje se broj onim redom kojim su zagrade otvarane. Za svaku grupu može se čuvati i kasnije pristupiti podnisci iz prepoznate niske koju je taj podizraz prepoznao. Način na koji se može pristupiti takvim podniskama definisan je implementacijom regularnih izraza u izabranoj aplikaciji ili programskom jeziku. Grupi sa rednim brojem nula odgovara kompletan regularan izraz, grupi sa rednim brojem 1, odgovara podizraz u prvoj otvorenoj maloj zagradi, itd.

Na primer, regularni izraz `dobro( jutro)?` ima samo jednu podgrupu. U slučaju da prepoznamo nisku *dobro*, tada će grupa sa rednim brojem 1, sadržati praznu nisku, a u slučaju da je prepoznata niska *dobro jutro*, tada će grupa sadržati podnisku *jutro*.

Regularni izraz	*Tekst*	*Tekst grupe 0*	*Tekst grupe 1*
`dobro( jutro)?`	dobro jutro.	dobro jutro	jutro
	dobro vece	dobro	

Tabela 1.14: Grupisanje podizraza

1.8.1 Refereisanje unazad

Prilikom definisanja regularnog izraza, moguće je koristiti referisanje u nazad (*eng. backreference*). Pod tim se podrazumeva korišćenje niske koji je neki regularni podizraz ranije u tekstu prepoznao da bi se na nekom drugom mestu kasnije u tekstu prepoznala ista ta niska. Bekreferenca se referiše brojem grupe čiji sadržaj je neophodno ponoviti. Na primer, `\1` je bekreferenca kojom bi se prepoznao isti tekst koji je prepoznat grupom sa rednim brojem jedan.

Na primer, uz pomoć regularnog izraza `([abc])=\1` mogu se prepoznati samo tri sledeće niske: *a*=*a*, *b*=*b* i *c*=*c*.

Regularni izraz	*Tekst*	*Tekst grupe 1*
`([abc])=\1`	a=a.	a
	b=b	b
	abc=c	c

Tabela 1.15: Grupisanje podizraza i referisanje na prepoznatu podnisku

1.8.2 Imenovane grupe i referisanje u nazad

U slučaju da regularni izraz sadrži veći broj grupa, praćenje njihovih rednih brojeva može da postane prilično teško. U tom slučaju, mnogo je lakše imenovati grupe.

Na primer, regularni izraz `(?P<grupa1>[abc])=(?P=grupa1)` ekvivalentan je regularnom izrazu `([abc])=\1`, samo što se grupi obraća uz pomoć dodeljenog imena, a ne broja.

Napomena: Podrška i sintaksa za imenovane grupe zavise od izabranog alata i programskog jezika. U ovom slučaju prikazana je sintaksa programskog jezika *pajton* i njegovog `re` modula.

1.9 Preduvid i postuvid

Preduvid i postuvid predstavljaju specijalne grupe. Karakteri definisani u ovim grupama se prepoznaju najnormalnije, ali nakon što se izvrši prepoznavanje te grupe, tekst prepoznat grupom biće odbačen i biće sačuvan samo rezultat prepoznavanja ostatka regularnog izraza. Drugim rečima, preduvid i postuvid se koriste kao mehanizam provere da li tekst

ispunjava neki uslov ili ne. Preduvid i postuvid neće iskoristiti nijedan karakter iz teksta, već će samo utvrditi da li je poklapanje moguće ili ne.

1.9.1 Pozitivni i negativni preduvid

Negativni preduvid je neophodan kada je potrebno prepoznati tekst koji nije praćeno nečim drugim. U slučaju da je potrebno da prepoznamo slovo *b* posle koga ne ide slovo *e*, dovoljno je da iskoristimo negativni preduvid i zapišemo regularni izraz kao `b(?!e)`. Negativni preduvid se definiše kao par zagrada u kojima se nalazi znak pitanja, praćen znakom uzvika i regularnim izrazom preduvida. Pozitivni preduvid radi na isti način. Regularni izraz `b(?=e)` prepoznaje samo ono slovo *b* koje je prećeno slovom *e*, ali slovo *e* nije deo poklapanja. Pozitivni preduvid se definiše kao par zagrada u kojima se nalazi znak pitanja, praćen znakom jednako i regularnim izrazom.

Regularni izraz	*Primeri teksta*	
`b(?=e)`	babe	baba
`b(?!e)`	babe	bebe
`b(?=o)b`	bob	bb
`b(?=o)o`	bob	bb
`b(?!o)o`	bob	bb
`grupa_(?=(a|b))\1\1`	grupa_ab	grupa_aa

Tabela 1.16: Pozitivan i negativan preduvid

U okviru preduvida može se navesti bilo koji regularni izraz, sem postuvida. U slučaju da navedeni regularni izraz sadrži grupe za izdvajanje teksta, one će izdvojiti tekst i mogu se koristiti čak i izvan preduvida. Preduvid sam po sebi ne definiše grupu za izdvajanje teksta i ne računa se prilikom numerisanja zagrada i definisanja bekreferenci. U slučaju da je potrebno prepoznati neki regularni izraz u okviru preduvida, neophodno ga je grupisati u okviru samog preduvida, na primer na sledeći način `(?=(regeks))`. Bilo koji drugi način nije ispravan, jer će preduvid odbaciti poklapanje po tom regularnom izrazu do momenta kada bi grupa za izdvajanje teksta trebalo da upamti poklapanje.

1.9.2 Pozitivni i negativni postuvid

Postuvid ima isti efekat kao i preduvid, samo se primenjuje unazad. Postuvidom se implementaciji regularnih izraza saopštava da privremeno krene unazad u prepoznatom stringu da bi proverila da li se tekst u postuvidu može poklopiti ili ne. Regularni izraz `(?<!a)b` uz pomoć negativnog postuvida prepoznaje ono *b* ispred koga se ne nalazi slovo *a*. Ako je dat tekst *cab*, slovo *b* neće biti prepoznato, jer se ispred njega nalazi slovo *a*. U slučaju da je dat tekst *bed* ili *debt*, biće prepoznato samo slovo *b* u obe reči. Negativni postuvid se definiše kao par zagrada u kojima se nalazi znak pitanja, praćen znakom manje, zatim znakom uzvika i regularnim izrazom. Regularni izraz `(?<=a)b` uz pomoć pozitivnog postuvida prepoznaje samo ono *b* ispred koga se nalazi slovo *a*. Ako je dat tekst *cab*, uz pomoć ovog regularnog izraza biće prepoznatno samo slovo *b*, ali ako je dat tekst *bed* ili *debt* slovo *b* neće biti prepoznato. Pozitivni postuvid se definiše kao par zagrada u kojima se nalazi znak pitanja, praćen znakom manje, zatim znakom jednako i regularnim izrazom.

Regularni izraz	*Primeri teksta*		
`(?<!a)b`	cab	bed	debt
`(?<=a)b`	cab	bed	debt

Tabela 1.17: Pozitivan i negativan preduvid

Prilikom upotrebe postuvida, neophodno je voditi računa da regularni izraz definisan u samom postuvidu bude konačne dužine. Ovo je neophodno, jer regularni izrazi ne mogu da se poklapaju unazad. Da bi se ispitao postuvid, prvo je potrebno vratiti ofset unazad i onda ga poklopiti unapred kao bilo koji drugi regularni izraz. Upravo zbog vraćanja ofseta unazad, neophodno je da regeks u okviru postuvida ima unapred određenu dužinu.

1.10 Modifikatori uparivanja regularnih izraza

Uparivači regularni izraza su osetljivi na razliku između velikih i malih slova. Nekada postoji potreba da se ne pravi razlika između slova po veličini. Na primer kada je potrebno pronaći html etikete. One su ispravne čak i ako nisu napisane istom veličinom slova, jer je HTML kao jezik, neosetljiv na tu razliku. U takvim slučajevima je moguće uključiti mod uparivanja `(?i)`, (eng. *Ignore case mode*), čime će nastavak regularnog izraza bio neosetljiv na razliku slova po veličini. Ako je u nekom delu regularnog izraza potrebno isključiti, tj vratiti na podrazumevano koristi se isti modifikator ali sa -, tj. `(?-i)`.

Pored ovog, posto je još dva modifikatora `s` i `m`. Modifikator `s`, (eng. *Single line mode*), čini da se prilikom uparivanja celokupan ulazni tekst posmatra kao jedna linija. Tada se sidro za početak linije `^` može upariti samo na početku ulaza, a sidro za kraj linije `$` na samom kraju celog ulaza. Posledica ovog načina uparivanja je u tome što specijalni karakter `.` može da upari karakter za novi red, tj. `\n`. Ovaj modifikator se uključuje u regularni izraz, navođenjem `(?s)`, a kada ga želimo isključiti `(?-s)`.

Modifikator `m`, (eng. *Multi line mode*) predstavlja podrazumevano ponašanje uparivača regularnih izraza po kom ulaz posmatra kao potencijalno višelinijski tekst. Time se sidra za početak linije uparuju na početku ulaza, ali i na početku svake linije, tj. iza karaktera za nov red, sidra za kraj na samom kraju ulaza i na kraju svake linije, tj ispred karaktera za nov red. Specijalni karakter `.` ne prihvata karakter za nov red.

Modifikatori se mogu kombinovati pa se mogu u istoj zagradi uključivati i isključivati. Na primer, `(?-is)`. Zagrade koje se koriste za uključivanje modifikatora se ne broje u bekreference.

Regularni izraz	*Primeri teksta*		
`(?i)te(?-i)st`	Test TEst	test TEST	tEst tesT
`(?s).*`	Prva recenica.	Prva linija i Druga linija.	

Tabela 1.18: Modifikatori uparivanja

1.11 Razni primeri

U nastavku teksta biće prikazani razni primeri regularnih izraza:

Zadatak 1.1 Napisati regularni izraz koji će prepoznavati samo reči sledeće fraze i skraćenice engleskog jezika: *regular expression*, *regular expressions*, *regex*, *regexp* i *regexes*. ■

Rešenje. `reg(ular expressions?|ex(p|es)?)`

Zadatak 1.2 Napisati regularni izraz koji će prepoznavati samo deklinaciju reči *jezik* u srpskom jeziku. ■

Rešenje. `jezi(k[au]|om)?|ce`

Zadatak 1.3 Napisati regularni izraz koji će prepoznavati samo deklinaciju reči *jezik* u srpskom jeziku. ■

Rešenje. `jezi(k[au]|om)?|ce`

Zadatak 1.4 Napisati regularni izraz koji će prepoznavati cele brojeve bez vodećih nula. ■

Rešenje. `-?(0|[1-9][0-9]*)`

Zadatak 1.5 Napisati regularni izraz koji će prepoznavati datume iz 20. veka do danas u formatu *dd/mm/gggg*. Pretpostaviti da svaki mesec ima 31 dana. Dozvoliti da se kao separator između dana, meseca i godine koriste karakteri ., - ili /. U celom datumu se isti karakter mora koristiti kao separator. Mesec i dan ukoliko su jednocifreni imaju vodeću nulu. Na primer: *01/01/1900*, *31.12.2019* *31-04-2020* ■

Rešenje. `(0[1-9]|[12]\d|3[01])([./-])(0[1-9]|1[0-2])\2(19\d\d|20([01]\d|20))`

Zadatak 1.6 Napisati regularni izraz koji će prepoznavati cele brojeve iz intervala `[0,255]`. ■

Rešenje. `(0|[1-9]\d{1,2}|1\d{2}|2([0-4]\d|5[0-5]))`

Zadatak 1.7 Napisati regularni izraz koji će prepoznavati samoglasnike. ■

Rešenje. `[aeiou]`

Zadatak 1.8 Napisati regularni izraz koji će prepoznavati reči sastavljene samo od karaktera reči, tj slova, cifara i _, koje sadrže samo jedan samoglasnik. ■

Rešenje. `\b([^aeiou]\B)+[aeiou](\B[^aeiou])+\b`

Zadatak 1.9 Napisati regularni izraz koji će prepoznavati algebarske izraze nad pozitivnim celim brojevima, uključujući i nulu. ■

Rešenje. `(0|[1-9]\d*) *[-+/*] *(0|[1-9]\d*)`

Zadatak 1.10 Napisati regularni izraz koji će prepoznavati specijalne karaktere karakterskih klasa, `]`, `^`, `-` i `\`. ■

Rešenje. `[]^\\-]`

Zadatak 1.11 Napisati regularni izraz koji će prepoznavati četvorocifren heksadekadni broj. ■

Rešenje. `(?i)0x[\da-f]{4}`

Zadatak 1.12 Napisati regularni izraz koji će prepoznavati indentifikatore u programskom jeziku C. ■

Rešenje. `\b[a-zA-Z_][_a-zA-Z0-9]*\b`

Zadatak 1.13 Napisati regularni izraz koji će prepoznavati jednolinijske komentare u programskom jeziku C. ■

Rešenje. `(?m)//.*` Modifikator za višelinijski mod je stavljen, samo da naglasi da se čita do kraja linije. U principu, to je podrazumevano ponašanje i nije neophodan.

Zadatak 1.14 Napisati regularni izraz koji će prepoznavati višelinijske komentare u programskom jeziku C. Na primer, sledeći fragment koda sadrži komentare.

```
/* C-ovski komentar*/
int x; /* promenljiva*/
/* Dokumentacija
* u vise / linija u raznih /karaktera
redova :)
*/ /***** */
/**/ /***/
```

■

Rešenje. `(?s)/\*.*?\*/`
ili `/\*([^*]|\*[^/])*\*+/`

Zadatak 1.15 Napisati regularni izraz koji će prepoznavati brojeve u pokretnom zarezu, bilo da imaju ili nemaju razlomljen deo, bilo da imaju znak, bilo da imaju deo sa eksponentom, pa čak i kada ispred decimalne tačke nema cifara u celom delu. Na primer: -2.3, +5, +0.5, 70, 7.0, .89, -.45, 10.34e-20, -1.67E+7, 2.4e14 ■

Rešenje. `[-+]?\d*\.?\d+([eE][+-]?\d+)?`

Zadatak 1.16 Napisati regularni izraz koji će prepoznavati mejl adrese, poput: *ana.anic@goef.bg.ac.rs* ili *name-unkown@123_abc.info* ■

Rešenje. `\b[\w.-]+@[\w.-]+(\.[\w.-]+)*\.[A-Za-z]{2,4}\b`

Zadatak 1.17 Napisati regularni izraz koji će prepoznavati linkove u HTML-u, na primer *Google*. ■

Rešenje. `<a[^>]*>(.*?)</a>`

Zadatak 1.18 Napisati regularni izraz koji će prepoznavati XML etikete, na primer *<tag attribute="value»content</tag>* ■

Rešenje. `<([A-Z][A-Z0-9]*)[^>]*>(.*?)</\1>`

Zadatak 1.19 Napisati regularni izraz koji će prepoznavati predprocesorske direktive u programskom jeziku C, kako jednolinijske tako i višelinijske. ■

Rešenje. `^\s*(#.*)$` Kako se predprocesorske direktive mogu prostirati i kroz više redova, npr.

```
#include <stdio.h>
#define max(x,y) \
            ((x) > (y) ? (x): (y))
```

modifikujemo regularni izraz
`^\s*(#.*)(\\\n.*)*$`

Zadatak 1.20 Napisati regularni izraz koji će prepoznavati niske u programskom jeziku C. Niske mogu sadržati i specijalne karaktere.
Na primer `"abc"`, `"a\\nc"`, `"sa  \\"citatom\\" unutra"`. ■

Rešenje. `"[^"]*"`
ili ukoliko imamo lenje uparivače
`".*?"`

Ovo nam neće biti dobro za poslednju nisku.
`"(\\.|[^"\\])*?"`
Sledeće rešenje je efikasnije jer ima manje vraćanja u nazad i odbacivanja ulaza:
`"[^"\\]*(\\.[^"\\]*)*"`

Zadatak 1.21 Napisati regularni izraz koji će prepoznavati uzastopna ponavljanja vrednosti u listi elemenata razdvojenih zarezima. ■

Rešenje. `(?<=,|^)([^,]*)(,\1)+(?=,|$)`

Zadatak 1.22 Napisati regularni izraz koji će prepoznavati linije teksta koje u sebi sadrže neku od reči: *crvena*, *plava* i *bela*. ■

Rešenje. `^.*?\b(crvena|plava|bela)\b.*$`

Zadatak 1.23 Napisati regularni izraz koji će prepoznavati linije teksta koje u sebi sadrže reči *jedan*, *dva* i *tri* u proizvoljnom redosledu. Ne praviti sve moguće permutacije datih reči. ■

Rešenje. `^(?=.*jedan)(?=.*dva)(?=.*tri).*$`

1.12 Tipovima uparivača regularnih izraza

Postoje dva tipa uparivača regularnih izraza :

text directed koji su orijentisani prema tekstu i u čijoj pozadini je deterministički konačni automat, tj. DKA.

regex directed koji su usmereni ka regularnom izrazu i u čijoj pozadini je nedeterministički konačni automat, tj NKA.

Više o konačnim automatima će biti reči u poglavlju 4.

Prvi pomenuti uparivači, DKA uparivači ne podržavaju upotrebu referisanja u nazad (eng. *bekreference*), lenjih operatora ponavljanja, preduvida i postuvida. Sve pomenuto je naravno podržano u slučaju NKA uparivača jer imaju mogućnost bektrekinga. Bitna razlika će biti da iz istog razloga DKA brže nalaziti poklapanje.

DKA uparivači su orijentisani prema tekstu i kako god regularan izraz napisali vraća će nam najduži mogući tekst koji se poklapa sa regularnim izrazom. NKA uparivaci su nestrpljivi su da što pre vrate poklapanje kada je u pitanju operator alternacije `|`. Na primer, na tekstu *vezbanje*, različiti tipovi uparivača će se različito ponašati:

Regularni izraz	*DKA uparivač*	*NKA uparivač*
`vez|vezba`	vezbanje	vezbanje
`vezba|vez`	vezbanje	vezbanje
`one(self)?|(selfsufficient)?`	oneselfsufficient	oneselfsufficient

Tabela 1.19: Razlika u dužini prepoznatog teksta u slucaju DKA ili NKA uparivača

DKA uparivači	Flex/Lex, grep neke verzije, awk, MySQL
NKA uparivači	Emacs, Java, grep (većina verzija), less, more, .NET jezici, , Perl, PHP, Python, Ruby, sed (većina verzija), vi

Tabela 1.20: Alati i jezici podeljeni po tipovima uparivača

Međutim nije baš sve crno ili belo, postoje alati koji kombinuju oba principa. Na primer, GNU verzija *egrep* alata podržava bekreference. To postiže tako što ima dva potpuno različita uparivača ispod haube. Prvo koristi brži DKA uparivač da utvrdi ima li izgleda da se nađe poklapanje, a potom koristi NKA uparivač da potvrdi uparivanje.

2. Linux alati `grep` i `sed`

U ovom poglavlju ćemo za izdvajanje linija teksta koje odgovaraju zadatom regularnom izrazu koristiti alat *grep* koji najčešće dolazi uz `Linux` operativni sistem. Ukoliko želimo da deo linije teksta koji je prepoznat regularnim izrazom izmenimo prema zadatom šablonu, koristićemo alat *sed* koji, takođe, dolazi uz *Linux* sistem.

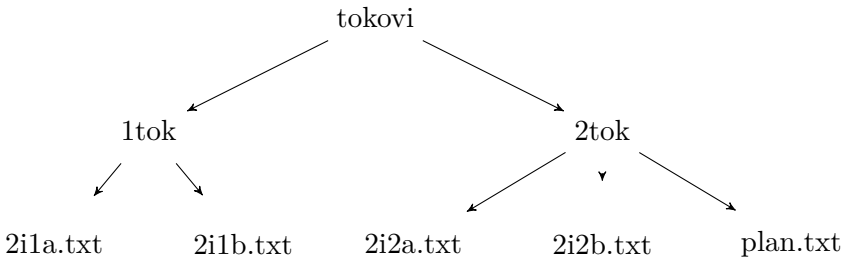
U slučaju da korisite Ubuntu operativni sistem, možete ih instalirati komandama iz konzole

```
sudo apt-get update
sudo apt-get install grep
sudo apt-get install sed
```

2.1 Alat *grep*

Grep je jedan od najstarijih uparivača regularnih izraza sa tekstom koji je još uvek u upotrebi. Pretražuje liniju po liniju ulaznih datoteka u potrazi za onima koje se delom ili u celosti mogu upariti sa regularnih izrazom. Izdvaja linije u kojima je pronađeno poklapanje. Grep je nastao iz stare komande Linuxa `g/ Regular Expression /p`, koja na engleskom predstavlja ***G**lobal **R**egular **E**xpression **P**rint*.

Zapravo, `grep` alat u svom osnovnom obliku većinu specijalnih karaktera o kojima je bilo reči u poglavlju 1, ne smatra specijalnim dok ih ne stavimo iza karaktera `\`. Jedini specijalni karakteri su `*, ., ^, $`. Primetimo da ni zagrade nisu specijalni karakteri, pa ako bismo ograđivali regularni izraz, bilo bi neophodno da sve zagrade budu u sekvenici sa `\`, npr. `\(`. Svaki malo složeniji regularan izraz biće gotovo nečitljiv zbog velikog broja karaktera `\`. Iz tog razloga koristićemo *egrep*, (skraćenica od *Extended grep*(eng.), koji ima proširen skup specijalni karaktera. *Egrep* nije ništa drugo do *grep* pozvan sa dodatnim opcijama. *Grep*-ova opcija `-E` proširuje skup specijalnih karaktera za regularne izraze. Pored, svega u grepu nep postoje imenovane karakterske klase poput `\d` ili `\w`, `\s` .



Slika 2.1: Struktura direktorijuma `tokovi`

Poziv bi generalno bio:

```
egrep [opcije] regex ulazni_sadržaj
```

Ulazni sadržaj se može izostaviti i tada će se učitavati tekst sa standardnog ulaza. Kao ulaz se može proslediti datoteka, spisak datoteka, direktorijum, itd. Neke od opcija su:

-i da ignoriše razlike između velikih i malih slova
–color=auto da boji prepoznat tekst (već je uključeno u *egrep*)
-n da izdvaja i redni broj linije u kojoj je pronađeno poklapanje
-r da rekurzivno obrađuje i datoteke u poddirektorijumima.
-h da ne ispisuje naziv datoteke iz koje je izdvojena linija.
-c da ispise samo broj izdvojenih linija.

Primer 2.1 Komandom *egrep* izdvojiti linije datoteka koja se nalaze u direktorijumu `tokovi` i njegovim poddirektorijumima koje u sebi sadrže alas naloge studenata Matematičkog fakultet upisanih od 2000. do 2019. godine i čiji je indeks iz intervala $[1-499]$, zapisan sa 3 cifre i sa vodećim nulama ako je potrebno.

Rešenje. Pišemo regex kojim ćemo izdvojiti samo linije koje ispunjavaju kompletan uslov zadatka. U pozivu *egrep*-a koristimo opciju -r da bi se pretraživale linije svih datoteka i u poddirektorijumima.

```
egrep -r '(m[irmnvl]|a[afi])[01][0-9](00[1-9]|0[1-9][0-9]|[1-4][0-9][0-9])'
      tokovi
```

Komanda je prelomljena zbog slaganja teksta. Treba je navesti u jednoj liniji ili neposredno pre prelaska u novi red otkucati karakter \.

Na primer neka je direktorijum `tokovi` i neka su poddirektorijumi struktuirani kao na slici 2.1.

I neka je sadržaj datoteka sledeći:

```
Spisak studenata sa 1. toka i grupe A
-------------------------------------
Marinko Marjanovic, mi10571
Mihailo Avramovic, mn17499
Milana Petrov, aa09090
Nikolina Tadic, mi13000
------------KRAJ SPISKA--------------
```

Rešenje 2.1: 2i1a.txt

```
Spisak studenata sa 1. toka i grupe B
-------------------------------------
Sloba Milisavljevic, ai14300
Sofija Pavlovic, af13116
Stefan Todorovic, mm15067
------------KRAJ SPISKA--------------
```

Rešenje 2.2: 2i1b.txt

```
Spisak studenata sa 2. toka i grupe A
-------------------------------------
Dejan Markovic, mi19003
Dragan Savic, mm21006
Isidora Dimitrijevic, mr99154
------------KRAJ SPISKA--------------
```

Rešenje 2.3: 2i2a.txt

```
Spisak studenata sa 2. toka i grupe B
-------------------------------------
Ugljesa Tosic, ml07001
Verica Milanov, mv01163
Vukasin Filipovic, ir10187
------------KRAJ SPISKA--------------
```

Rešenje 2.4: 2i2b.txt

Rezultat komande bi bio prikazan na ekranu:

```
tokovi/1tok/2i1b.txt:Sloba Milisavljevic, ai14300
tokovi/1tok/2i1b.txt:Sofija Pavlovic, af13116
tokovi/1tok/2i1b.txt:Stefan Todorovic, mm15067
tokovi/1tok/2i1a.txt:Mihailo Avramovic, mn17028
tokovi/1tok/2i1a.txt:Milana Petrov, aa09499
tokovi/2tok/2i2a.txt:Dejan Markovic, mi19003
tokovi/2tok/2i2b.txt:Ugljesa Tosic, ml07001
tokovi/2tok/2i2b.txt:Verica Milanov, mv01163
```

Ukoliko nas ne interesuje iz koje datoteke je izdvojena linija, tj. ne želimo u rezultatu tekst pre karaktera :, potrebno je da iskoristimo *egrep*-ovu opciju `-h`.

Ukoliko želimo, možemo preusmeriti izlaz iz konzole i izlaz komande proslediti u datoteku `studenti.txt`

```
egrep -rh '(m[irmnvl]|a[afi])[01][0-9](00[1-9]|0[1-9][0-9]|[1-4][0-9][0-9])'
      tokovi > studenti.txt
```

Možemo dodatno sačuvati komandu u *shell* skripti, sa ekstenzijom `.sh`. Na primer, `primer1.sh` i zatim je iz konzole pozvati sa:

```
sh primer1.sh
```

Grep je uparivač regularnih izraza vođen tekstom (eng. text-directed), pa nema podrške za ulenjivanje gramzivih operatora ponavljanja i uvide. Međutim backreference postoje i zadaju se sa `\1` za 1. i do najviše `\9` za 9.

Primer 2.2 Komandom *egrep* izdvojiti nazive datoteka koje se nalaze u direktorijumu `tokovi`, imaju ekstenziju `.txt` i u svom nazivu sadrže istu cifru toka kao i godinu studija. Nazivi datoteka bi trebalo da su struktuirani tako da sadrže godinu studija koja je od 1 do 5, bar 1, a najviše 2 mala slova za smer, cifru za tok, i potom slovo a ili b.

Rešenje. Razmotrimo najpre regularan izraz koji bismo poslali *egrep*-u. Najbitnije je da nam godina i tok u nazivu moraju biti iste cifre. To ćemo postići bekreferencama `([0-9])[a-z]{1,2}\1[ab]\.txt`

Ulaz za ovaj primer neće moći da bude datoteka jer nam treba spisak svih datoteka u direktorijumu `tokovi` i njegovim poddirektorijumima, tako da se svaka datoteka nalazi u svojoj liniji. Takva informacija se dobija *Linux*-ovom komandom `ls`, uz opcije `-R` zbog rekurzivnog obilaska poddirektorijuma, i `-1` za ispis svake datoteka ili direktorijuma u zasebnom redu.

```
01_grep_sed$ ls -1R tokovi/
tokovi/:
1tok
2tok

tokovi/1tok:
2i1a.txt
2i1b.txt

tokovi/2tok:
2i2a.txt
2i2b.txt
plan.txt
```

Ovaj izlaz nam nije bitan za dalji rad i ne želimo da ga sačuvamo u datoteku koju bismo mogli da prosledimo *grep*-u. Da bismo ovaj izlaz direktno prosledili komandi *egrep*-a, koristimo u konzoli karakter | (*eng. pipe*) koji će nam preusmeriti izlaz komande `ls -1R` kao ulaz u komandu *egrep*-a.

```
ls -1R tokovi | egrep  '([0-9])[a-z]{1,2}\1[ab]\.txt'
```

i očekivan izlaz na ekranu je:

```
2i2a.txt
2i2b.txt
```

Ukoliko nam je potrebno da izdvojimo linije teksta koje zadovoljavaju ili uslov1 ili uslov2, potrebno je da u jedan regularan izraz znakom alternacije objedinom regularne izraze za oba uslova.

Primer 2.3 Izdvojiti alatom *egrep*-a linije koje imaju ili slovo `a` ili slovo `b`, bez osetljivosti na razliku između velikih i malih slova.

Rešenje. `egrep -i 'a|b' tokovi/1tok/2i1b.txt`

Izlaz će biti:

```
Spisak studenata sa 1. toka i grupe B
Sloba Milisavljevic, ai14300
Sofija Pavlovic, af13116
Stefan Todorovic, mm15067
------------KRAJ SPISKA--------------
```

Ukoliko su nam potrebne linije teksta koje zadovoljavaju konjukciju više uslova, ali u bilo kom poretku, kako nemamo mogućnost korišćenja preduvida, možemo nadovezivati pozive *egrep*-a za svaki uslov. Naime, prvim pozivom izdvojimo sve linije koje zadovoljavaju uslov1, potom preusmerimo preko | izlaz u naredni poziv *egrep* za uslov2 i tako nastavimo dok imamo uslova koji su u konjukciji.

Primer 2.4 Izdvojiti alatom *egrep*-a linije iz datoteka u direkorijumu `tokovi` i njegovim poddirektorijumima koje u sebi sadrže bar 2 uzastopna samoglasnika i bar dve uzastone dekadne cifre veće od 0.

Rešenje. Shvatamo da svaka izdvojena linija treba da zadovolji oba od uslova. Prvo izdvojimo linije koje ispunjavaju prvi uslov, za bar 2 uzastopna samoglasnika. Ne želimo da se ispisuje putanja do datoteke u kojoj je linija, pa koristimo opciju `-h`, a zbog oblaska poddirektorijuma koristimo opciju `-r`.

```
egrep -rh '[aeiou]{2,}' tokovi
```

Dobijamo sledeći rezultat.

```
Sloba Milisavljevic, ai14300
Mihailo Avramovic, mn17499
Milana Petrov, aa09090
```

Taj izlaz treba da bude ulaz u naredni poziv *grep*-a, u kom ćemo zadati drugi uslov, po ciframa.

```
egrep -rh '[aeiou]{2,}' tokovi  | egrep '[1-9]{2,}'
```

Dobijamo sledeći rezultat.

```
Sloba Milisavljevic, ai14300
Mihailo Avramovic, mn17499
```

2.2 Alat *sed*

Naziv alata *sed* je skraćeno od *stream editor*(eng.) Jednostavan je za korišćenje, a opet ima mnogo funkcionalnosti i komandi. Mi ćemo se zadržati samo na komandi zamene teksta. Tada poziv *sed*-u izgleda ovako :

```
sed [opcije] 's/regex/zamena/modifikatori' ulaz
```

Od **opcija** koristićemo `-r` da bismo koristili proširen skup specijalnih operatora za regularne izraze, kao što smo radili kod *grep*-a.

Razmotrimo sada deo `'s/regex/zamena/modifikatori'`. `s` je poziv *sed*-ovoj komandi zamene, (eng. *substitute*). Karakter `/` je **delimiter**, razdvaja komandu, regularni izraz i tekst zamene. Obavezno je pojavljivanje sva tri karaktera `/` kao u primeru. Ukoliko se u regularnom izrazu ili tekstu zamene koristi baš karakter `/` onda se on mora stavljati u sekvence tipa ili jednostavno uzabrati neki drugi karakter za delimiter, na primer, : ili _ ili ∼ ili |. Ali na sva tri mesta mora biti isti karakter kao delimiter.

Tekst zamene može biti, jednostavno, tekst, ali se može i kombinovati sa bekreferencama (od `\1` do `\9`) ili sa karakterom `&` koji ima specijalno značenje i predstavlja sav tekst uparen sa regexom.

Na primer, ukoliko želimo da dupliramo svaku rec dan u rečenici, komanda bi izgledala ovako:

```
echo "Danas je topao dan! Dan danas je isto." | sed -r 's/dan/&&/'
```

Dodali smo *echo* komandu pre *sed*-a, da bismo mogli da testiramo bez pravljenja datoteke sa tekstom. Rezultat je `Danas je topao dandan! Dan danas je isto.`

Uklonimo duple reči iz linija.

```
echo "Dan danas je je jedan dan dan." | sed -r 's/([a-z]+) \1/\1/ig'
```

Rezultat je `Danas je jedan dan.`.

Od mogućih modifikatora mi ćemo koristiti `i` (od eng. termina *ignorecase*) ukoliko želimo da se ignoriše razlika između velikih i malih slova i `g` (od eng. termina *global*) da bi se zamena izvršila na svim poklapanjima u liniji, a ne samo na prvom.

Na primer, ako želimo da se svako `dan` menja sa `noc`, komanda bi bila sledeća

```
echo "Danas je topao dan! Dan danas je isto." | sed -r 's/dan/noc/gi'
```

Rezultat je `nocas je topao noc! noc nocas je ovako.`.

Ako želimo da globalna zamena krene od 2. poklapanja, umesto do 1. kombinujemo modifikator `g` sa cifrom 2.

```
echo "Danas je topao dan! Dan danas je isto." | sed -r 's/dan/noc/2gi'
```

i rezultat je `Danas je topao noc! noc nocas je isto.`

Primer 2.5 Iz HTML datoteka *nastava.html* i *nastava1.html* sadrži podatke o studentima, poput, imena prezimena i broja indeksa. Podaci su strukturirani u tabelu, i u ćelijama jednog reda nalaze se podaci za jedno studenta. Garantovano su svi podaci za jednog studenta u jednoj liniji teksta.

Koristeći alate grep, sed i sort izdvojiti podatke o studentima koji su upisani od 2000. do 2019. godine i broj indeksa im je u intervalu [1,350] bez vodecih nula. Za svakog studenta na osnovu indeksa kreirati alas nalog. Pretpostaviti da su svi sa I smera.

Izdvojiti ime, prezime i alas nalog. Nalog odvojiti karakterom ',' od ostalih podataka. Spisak urediti rastuće po imenu, numerisati i upisati u datoteku *spisak.txt*

Rešenje. Neka je sadržaj datoteke nastava.html sledeći:

```
<!-- raspored po grupama -->
<div class="right_item"><b>Podgrupa B</b></div>
<div class="left_item1">
  <table width="220">
  <tr>
  <td>Todorovic Stefan</td><td>67/2018</td>
  </tr>
  <tr>
  <td>Dimitrijevic Isidora</td><td>154/2014</td> </tr>
  <tr>  <td>Milisavljevic Sloba</td><td>332/2007</td>
  </tr>
  <tr>
  <td>Popovic Kristina</td><td>211/2012</td>
  </tr>
  <tr>
  <td>Spasovski Igor</td><td>204/2003</td>
  </tr>
  <tr>
  <td>Mitrovic Dejana</td><td>366/2015</td> </tr>
  </table>

</div>
<!-- end of entry-->
```

Rešenje 2.5: 'Sadržaj datoteke nastava.html'

i sadržaj datoteke nastava1.html sledeći:

```
<!-- raspored po grupama -->
<h3>Podela za praktikum</h3>

<div class="right_item"><b>Podgrupa A</b></div>
<div class="left_item1">   <table width="520">
    <tr>  <td>Savic Marko</td><td>357/2001</td> </tr>
    <tr>
    <td>Pavlovic Sofija</td><td>116/2017</td>
    </tr>
    <tr>
    <td>Vulovic Dejan</td><td>33/2010</td> </tr>
    <tr>
    <td>Zivkovic Teodora</td><td>228/2005</td> </tr>
    <tr>
    <td>Savic Dragan</td><td>6/2016</td></tr> <tr>

    <td>Nenadovic Dusan</td><td>26/2019</td>
    </tr>
  </table>
</div>
<!-- end of entry-->
```

Rešenje 2.6: 'Sadržaj datoteke nastava1.html'

Najpre treba da izdvojimo linije koje sadrže nama vredne podatke. Taj deo ćemo uraditi sa *egrep*.

```
egrep '<td> *[A-Z][a-z]+ +[A-Z][a-z]+ *</td> *<td> *([1-9][0-9]?|
[12][0-9][0-9]|3[0-4][0-9]|350)/20(0[0-9]|1[0-7]) *</td>' nastava*.html -h
```

Iskoristili smo opciju `-h` da nam se ne bi ispisivale informacije o nazivu dodatke iz koje je pronađena linija. Komanda je morala biti prelomljena po redovima da bi stala u okvir papira. Ako je budete testirali mora te je napisati u jednom redu.

Rezultat bi bio sledeći:

```
    <td>Pavlovic Sofija</td><td>116/2017</td>
    <td>Vulovic Dejan</td><td>33/2010</td> </tr>
    <td>Zivkovic Teodora</td><td>228/2005</td> </tr>
    <td>Savic Dragan</td><td>6/2016</td></tr> <tr>
  <td>Dimitrijevic Isidora</td><td>154/2014</td> </tr>
  <tr>  <td>Milisavljevic Sloba</td><td>332/2007</td>
  <td>Popovic Kristina</td><td>211/2012</td>
  <td>Spasovski Igor</td><td>204/2003</td>
```

Sada je potrebno da očistimo ove linije tako da ostane samo ime, prezime i indeks. Uradićemo to korišćenjem *sed*-a. Potrebno je da sve što želimo da promenimo pokrijemo regularnim izrazom, dakle celu liniju. Ipak, potrebno je da ostanu ime, prezime i indeks, tako da ćemo njih posebno ograditi zagradama i u tekstu zamene iskoristiti njihove bekreference.

Izlaz iz prethodne komande, biće ulaz u narednu. Dakle, kako smo već proverili indekse i imena i prezimena, te složene regularne izraze ne moramo pisati. Sve sto se sa njima nije poklapalo već je odbačeno.

Kako treba da napisemo regularne izraze za HTML tagove koji sadže karakter `/`, iskoristićemo $\sim$ kao delimiter.

```
egrep '<td> *[A-Z][a-z]+ +[A-Z][a-z]+ *</td> *<td> *([1-9][0-9]?|
[12][0-9][0-9]|3[0-4][0-9]|350)/20(0[0-9]|1[0-7]) *</td>' nastava*.html -h \
| sed -r 's~.*<td>([ A-Za-z]+)</td> *<td> *([0-9]{1,3})/([0-9]{4}) *
</td>.*~\1 \2 \3~'
```

Ako komandu želimo možemo je pisati u skripti ekstenzije `.sh` i kasnije izvršiti sa `sh primer3.sh`. Da bi komanda bila preglednija poželećemo da je podelimo u više linija. Pre prelaska u novi red treba otkucati karakter `\`.

Rezultat koji vidimo je:

```
Pavlovic Sofija 116 2017
Vulovic Dejan 33 2010
Zivkovic Teodora 228 2005
Savic Dragan 6 2016
Dimitrijevic Isidora 154 2014
Milisavljevic Sloba 332 2007
```

```
Popovic Kristina 211 2012
Spasovski Igor 204 2003
```

Naredni korak je da napravimo svima alas naloge i razdvojimo sa , od ostalih podataka. Znamo da su svi sa I smera, pa će prefiks biti mi, za njim treba da idu najniže dve cifre godine i broj indeksa proširen na trocifren broj sa vodećim nulama, ako već nije trocifren. Kako od broja i cifara u indeksu nam zavisi način pravljenja alas naloga, uradićemo ovo kombinovanjem tri zamene, tj. tri uzastopna poziva sed-a za svaki od slučajeva.

```
egrep '<td> *[A-Z][a-z]+ +[A-Z][a-z]+ *</td> *<td> *([1-9][0-9]?|
[12][0-9][0-9]|3[0-4][0-9]|350)/20(0[0-9]|1[0-7]) *</td>' nastava*.html -h \
|  sed -r 's~.*<td>([ A-Za-z]+)</td> *<td> *([0-9]{1,3})/([0-9]{4}) *
</td>.*~\1 \2 \3~' \
|  sed -r 's/([0-9]{3}) [0-9]{2}([0-9]{2})/, mi\2\1/'   \
|  sed -r 's/([0-9]{2}) [0-9]{2}([0-9]{2})/, mi\20\1/'  \
|  sed -r 's/([0-9]) [0-9]{2}([0-9]{2})/, mi\200\1/'
```

Rezultat komande je:

```
Pavlovic Sofija , mi17116
Vulovic Dejan , mi10033
Zivkovic Teodora , mi05228
Savic Dragan , mi16006
Dimitrijevic Isidora , mi14154
Milisavljevic Sloba , mi07332
Popovic Kristina , mi12211
Spasovski Igor , mi03204
```

Sada treba da zamenimo mesta imenu i prezimenu, jer se tako traži, a i po imenu ćemo sortirati.

```
egrep '<td> *[A-Z][a-z]+ +[A-Z][a-z]+ *</td> *<td> *([1-9][0-9]?|
[12][0-9][0-9]|3[0-4][0-9]|350)/20(0[0-9]|1[0-7]) *</td>' nastava*.html -h \
|  sed -r 's~.*<td>([ A-Za-z]+)</td> *<td> *([0-9]{1,3})/([0-9]{4}) *
</td>.*~\1 \2 \3~' \
|  sed -r 's/([0-9]{3}) [0-9]{2}([0-9]{2})/, mi\2\1/'   \
|  sed -r 's/([0-9]{2}) [0-9]{2}([0-9]{2})/, mi\20\1/'  \
|  sed -r 's/([0-9]) [0-9]{2}([0-9]{2})/, mi\200\1/' \
|  sed -r 's/([A-Za-z]+) +([A-Za-z]+) +/\2 \1/'
```

Rezultat komande je:

```
Sofija Pavlovic, mi17116
Dejan Vulovic, mi10033
Teodora Zivkovic, mi05228
Dragan Savic, mi16006
Isidora Dimitrijevic, mi14154
Sloba Milisavljevic, mi07332
Kristina Popovic, mi12211
Igor Spasovski, mi03204
```

Na redu je da ih sortiramo,tako što ćemo samo izlaz preusmeriti kao ulaz u alat *sort*. On će linije podrazumevano leksikografski sortirati u rastućemo poretku. Ukoliko želimo da

bude u opadajućem poretku, treba iskoristiti opciju `-r`. Postoji mnogo opcija za ovaj alat kojim se može kontrolisati njegov rad. Mi se nećemo dalje upuštati u opcije.

```
egrep '<td> *[A-Z][a-z]+ +[A-Z][a-z]+ *</td> *<td> *([1-9][0-9]?|
[12][0-9][0-9]|3[0-4][0-9]|350)/20(0[0-9]|1[0-7]) *</td>' nastava*.html -h \
|  sed -r 's~.*<td>([ A-Za-z]+)</td> *<td> *([0-9]{1,3})/([0-9]{4}) *
</td>.*~\1 \2 \3~' \
|  sed -r 's/([0-9]{3}) [0-9]{2}([0-9]{2})/, mi\2\1/'   \
|  sed -r 's/([0-9]{2}) [0-9]{2}([0-9]{2})/, mi\20\1/'  \
|  sed -r 's/([0-9]) [0-9]{2}([0-9]{2})/, mi\200\1/' \
|  sed -r 's/([A-Za-z]+) +([A-Za-z]+) +/\2 \1/'   \
| sort
```

Rezultat komande je:

```
Dejan Vulovic, mi10033
Dragan Savic, mi16006
Igor Spasovski, mi03204
Isidora Dimitrijevic, mi14154
Kristina Popovic, mi12211
Sloba Milisavljevic, mi07332
Sofija Pavlovic, mi17116
Teodora Zivkovic, mi05228
```

Preostalo nam je još da numerišemo. Iskoristićemo opciju `-n` *egrep*-a da nam izdvoji broj linije u kom je pronašao poklapanje, a poslaćemo mu regularan izraz koji odgovara nekom delu svake linije, jer ne želimo da neku izostavimo. U našem slučaju to može biti regularan izraz `,`, ili još generalnije `.*`.

```
egrep '<td> *[A-Z][a-z]+ +[A-Z][a-z]+ *</td> *<td> *([1-9][0-9]?|
[12][0-9][0-9]|3[0-4][0-9]|350)/20(0[0-9]|1[0-7]) *</td>' nastava*.html -h \
|  sed -r 's~.*<td>([ A-Za-z]+)</td> *<td> *([0-9]{1,3})/([0-9]{4}) *
</td>.*~\1 \2 \3~' \
|  sed -r 's/([0-9]{3}) [0-9]{2}([0-9]{2})/, mi\2\1/'   \
|  sed -r 's/([0-9]{2}) [0-9]{2}([0-9]{2})/, mi\20\1/'  \
|  sed -r 's/([0-9]) [0-9]{2}([0-9]{2})/, mi\200\1/' \
|  sed -r 's/([A-Za-z]+) +([A-Za-z]+) +/\2 \1/'   \
| sort \
| egrep -n ','
```

Rezultat komande je:

```
1:Dejan Vulovic, mi10033
2:Dragan Savic, mi16006
3:Igor Spasovski, mi03204
4:Isidora Dimitrijevic, mi14154
5:Kristina Popovic, mi12211
6:Sloba Milisavljevic, mi07332
7:Sofija Pavlovic, mi17116
8:Teodora Zivkovic, mi05228
```

Numeracija sa `:` nije baš prirodna, tako da ćemo je zameniti sa tačkom i razmakom.

```
egrep '<td> *[A-Z][a-z]+ +[A-Z][a-z]+ *</td> *<td> *([1-9][0-9]?|
```

```
[12][0-9][0-9]|3[0-4][0-9]|350)/20(0[0-9]|1[0-7]) *</td>' nastava*.html -h \
|  sed -r 's~.*<td>([ A-Za-z]+)</td> *<td> *([0-9]{1,3})/([0-9]{4}) *
</td>.*~\1 \2 \3~' \
|  sed -r 's/([0-9]{3}) [0-9]{2}([0-9]{2})/, mi\2\1/'   \
|  sed -r 's/([0-9]{2}) [0-9]{2}([0-9]{2})/, mi\20\1/'  \
|  sed -r 's/([0-9]) [0-9]{2}([0-9]{2})/, mi\200\1/' \
|  sed -r 's/([A-Za-z]+) +([A-Za-z]+) +/\2 \1/'   \
| sort \
| egrep -n ',' \
| sed -r 's/:/. /'   > spisak.txt
```

I na kraju da preusmerimo izlaz u datoteku *spisak.txt*, koristimo $>$. Rezultat komande je upisan u *spisak.txt*:

```
1. Dejan Vulovic, mi10033
2. Dragan Savic, mi16006
3. Igor Spasovski, mi03204
4. Isidora Dimitrijevic, mi14154
5. Kristina Popovic, mi12211
6. Sloba Milisavljevic, mi07332
7. Sofija Pavlovic, mi17116
8. Teodora Zivkovic, mi05228
```

3. Programski jezik Pajton

Programski jezik Pajton (eng. Python) je interpreterski programski jezik visokog nivoa i opšte namene. Kreiran je od strane Gvida van Rosuma i prvi put objavljen 1991. godine. Osnovna filozofija programskog jezika pajton naglašava izuzetno visoku čitljivost izvornog koda, pre svega ekstenzivnom i sistematičnom upotrebom belina.

Prednosti pajtona ogledaju se u dinamičkoj tipiziranosti jezika i automatskom upravljanju memorijom, kao i postojanju interpretera i podrške u okviru svih dostupnih operativnih sistema. Pored ovoga, pajton programski jezik podržava veći broj popularnih programskih paradigmi, uključujući proceduralnu, imperativnu, funkcionalnu i objektno-orijentisanu paradigmu. Velika prednost programskog jezika pajton ogleda se i u vrlo obimnoj standardnoj biblioteci.

Programski jezik Pajton i njegova referentna implementacija su softver otvorenog koda čiji razvoj donekle zavisi od zajednice korisnika, kao i razvoj svih dodatnih modula koji nisu deo standardne biblioteke.

Osnovna filozofija programskog jezika Pajton može da se sumira u nekoliko rečenica:

- Lepo je bolje od ružnog.
- Eksplicitno je bolje od implicitnog.
- Jednostavno je bolje od složenog.
- Složeno je bolje od komplikovanog.
- Čitljivost je važna.

U okviru programskog jezika Pajton, ugrađeni modul omogućava upotrebu regularnih izraza. Modul podržava klasični ASCII režim, kao i Unicode režim. Jedino ograničenje jeste to da ASCII i Unicode ne smeju da se mešaju. S obzirom da regularni izrazi koriste veliki broj specijalnih karaktera, neophodno je koristi sirove stringove prilikom definisanja regularnih izraza o čemu će biti reči u nastavku teksta.

3.1 Uvod

Kroz naredne primere biće ilustrovana upotreba elementarnih tipova podataka, kolekcije podataka i naredbe kontrola toka.

Primer 3.1 Primer ilustruje rad sa brojevima i operacije nad njima.

Rešenje.

```
# Operatori i operacije nad brojevima se ne razlikuju previse
# od C-ovskih.
# Definisemo 2 promenljive.
# x je celobrojna, a y realna promenljiva.
x = 3
y = 4.2

# Stampamo tipove promenljivih
print( type(x) )
print( type(y) )

# Stampa na ekran zbir dve promenljive
print(x + y)

# Stampa na ekran vrednost y na x-ti stepen
print(y ** x)

# Ceo deo kolicnika
print(y // x)

# Realan kolicnik  realne i celobrojne promenljive
print(y / x)

# Ostatak pri deljenju
print(y % x)

# Eksplicitna promena tipa (konverzija)
# u int, odnosno, u float
print( int(y) )
print( float(x) )

# Podrska za kompleksne brojeve
z = complex(x, y)
# ili z = (3+4.2j)

print(z)
print( type(z) )

# Sabiranje sa celobrojnom promenljivom
print(z + x)
# Kompleksnim brojem
```

```
print(z + complex(1,x))
print (z + 20j)

# Realan i imaginaran deo kompleksne promenljive z
print(z.real, z.imag)

# Prikazujemo normu komplesnog broja
print(abs(z))  # sqrt(z.real**2 + z.imag**2)
```

Rešenje 3.1: Operatori i operacije nad numeričkim podacima

Primer 3.2 Primer ilustruje štampanje niski karaktera.

Rešenje.

```
print( 'Hello world!'  )

print( "Zdravo svima!\
Nekada je lakse zapisati\
 u vise redova tekst, \
a da on bude ipak \
tretiran kao \
jedna linija \
teksta!\n")

print( "Mozemo \t koristiti \n specijalne karaktere\v!")

print( "Jel' moguce imati i ' u sredini" +" ogradjene sa \" ?")

# Ako je niska ogradjena sa ', onda se " mogu koristiti unutar
#  te niske, bez koriscenja \
print( 'levo,  a \t "ovo" je  desno' + " od reci 'levo'\n")

print( """Ukoliko je niska ogradjena sa
trostrukim navodnicima(ili apostrofima)
onda ce svaki
prelazak u novi red biti prikazan.\n""")

# raw string
print( r"ovo\tse ispisuje\"doslovno.\n")
```

Rešenje 3.2: Štampanje niski

Primer 3.3 Primer ilustruje osnovna svojstva niski i operacije nad njima.

Rešenje.

```
# Definisemo promenljive tipa niska i stampamo ih na ekran
a = "Zdravo svima!\nDobro jutro!\n"
```

```
3
# Stampamo tip promenljive a i vrednost promenljive a.
5 print( type(a) )
print( a )
7
b = "Pozdrav!"
9 print( b )

11 # Niske se mogu nadovezivati
print( a + b + "\n\n" )
13
# i same sa sobom zadat broj puta
15 print( b * 3 )

17 # pri cemu se originalna niska nece promeniti.
print( b )
19
# Karakteri stringa se ne mogu menjati.
21 # a[0]='z'

23 # Mogu se izdvojiti podniske koristeci indeksnu sintaksu
# indeksi karaktera pocinju od 0 za  prvi karakter i duzina-1
25 # za poslednji karakter niske.
# Python dozvoljava upotrebu pozitivnih i negativnih celih
27 # brojeva za indeksiranje.
# Na primeru niske "Pozdrav!" :
29 #  P   o   z   d   r   a   v  !
#  0   1   2   3   4   5   6  7
31 # -8  -7  -6  -5  -4  -3  -2 -1
print( b[2:6] )
33
# Stampa podnisku od pocetka niske do karaktera na indeksu 2,
35 # ali bez njega.
print( b[:2] )
37
# Stampa nisku od karaktera na indeksu 4 do kraja niske b
39 print( b[4:] )

41 # Stampamo duzinu niske b
print( len(b) )
43
# Stampamo sve karaktere niske b
45 print( b[:] )

47 # Stampamo nisku od 3 karaktera do pretposlednjeg,
# neukljucujuci pretposlednji karakter.
49 print( b[2:-2] )

51 # Stampamo nisku od 2 karaktera do poslednjeg, neukljucujuci ga
```

```
print( b[1:len(b)-1] )

# Stampamo prvi karakter niske b
print( b[-0] )

# Stampamo nisku koristeci samo negativne indekse
print( b[-5:-1] )

# Moramo biti oprezni da se prvi indeks se odnosi na karakter
# koji je ranije u nisci od karaktera na koji se odnosi drugi
# indeks u intervalu. Inace, imacemo praznu nisku.
print( b[-3: 1] )
print( b[4:2] )
```

Rešenje 3.3: Svojstva i operacije nad niskama

Primer 3.4 Primer ilustruje kreiranje i upotrebu listi i ntorki kao kolekcija podataka.

Rešenje.

```
# a je tipa ntorka navedenih elemenata  (eng. tuple)
# Moze se definisati i bez zagrada ()
a = (1,2,3,4,5)
print( a )
print(type(a))

# Sa ntorkama mozemo sve sto smo mogli sa niskama. I za njih
# vazi da ne mozemo menjati elemente, kao niskama karaktere.

# Kreiramo listu od ntorke u kojoj su nabrojani elementi.
l = list((1,2,3,4,5))
# Isto se postize i sledecom naredbom jer je u zagradi ista
# sekvenca kao i a.
l1 = list(a)

# Prikazujemo listu l i za njom l1.
print( l, l1 )

# Kreiramo novu listu u kojoj nisu svi elementi celi brojevi
l2 = [7.2, 4, 'seminar', 'Python', 'programiranje']
print( l2 )

# Menjamo element na indeksu 2
l2[2] = 'uvod'
print( l2 )

# Stampamo rezultat nadovezivanja dve liste
print( l1 + l2 )
```

```
30 # Na liste mozemo primeniti indeksnu notaciju da izdvojimo
   # pojedinacni element ili vise uzastopnih.
32
   # Pravimo kopiju liste l1 i cuvamo je u promenljivu b
34 b = l1[:]
   print( b )
36
   # Zamenjujemo elemente liste b pocevsi od elementa sa indeksom
38 # 3, pa do kraja liste, praznom listom. Drugim recima iz liste
   # b uklanjamo sve pomenute elemente.
40 b[3:] = []
   print( b )
42
   # l1 je nepromenjena jer smo radili sa kopijom.
44 # Ukoliko u 34. liniji sklonite [:] iz naredbe b = l1[:]
   # primeticete razliku.
46 print( l1 )

48 # Nadovezujemo b listu sa samom sobom i prikazujemo rezultat
   # nadovezivanja. Lista b ostaje nepromenjena.
50 print( b*2 )

52 # Dodajemo na pocetak liste b listu od 2 elementa
   b[:0]=['na','pocetak']
54 print( b )

56 # Dodajemo u listu b izmedju 2. i 3. elementa  2 elementa
   b[2:2] = ["nesto","izmedju"]
58
   # Nadovezujemo listu b sa listom l2
60 b.extend(l2)
   # Isto bi se moglo postici sledecom naredbom:
62 # b = b + l2
   print( b )
64
   # Dodajemo na kraj liste nov element koji je lista l
66 b.append(l)
   print( b )
68
   # Stampamo 1. element liste koja dodata na kraj liste b
70 print( b[-1][0])

72 # Uklanjanje konkretnog elementa liste je moguce samo ako je
   # element vec u listi. Inace, imacemo ValueError.
74 if 'pocetak' in b:
     b.remove('pocetak')
76 else:
     print("'pocetak' se ne nalazi u listi " + b)
78
```

```
# Uklanjamo poslednji element liste i stampamo ga.
print( b.pop() )
print( b )

# Uklanjamo element na indeksu 0.
print( b.pop(0) )
print( b )

# Obrcemo listu b i stampamo je pre i posle te modifikacije.
print( b )
b.reverse()
print( b )

# Sledece naredbe ponovo imaju smisla samo ako element vec
# postoji u listi.
# Ukoliko element nije u listi, imacemo ValueError.
#Prikazujemo indeks elementa sa vrednoscu 'uvod'.
print( b.index('uvod') )

# Broj pojavljivanja reci 'uvod' u listi b
print( b.count('uvod') )

# Uklanjamo 4 elementa liste pocevsi od elementa na indeksu 3.
b[3:7] = []
print( b )

# Sortiramo listu b u opadajucem poretku. Ona trenutno sadrzi
# samo niske i poredjenje je leksikografsko.
b.sort(reverse=True)
print( b )

# Sortiramo listu niski po duzini u opadajucem poretku.
# Kada koristimo imena argumenata prilikom poziva funkcije nije
# bitan redosled navodjenja argumenata.
# key parametar je funkcija poredjenja koja ce se primeniti
# samo na jedan element liste b
b.sort(key=len, reverse = True)
print( b )

# Za slozenija poredjenja moze se koristiti lambda izraz.
b.sort(key=lambda x: len(x), reverse = True)
print( b )
```

Rešenje 3.4: Kolekcije u Python-u. Liste i ntorke.

Primer 3.5 Primer ilustruje iteriranje kroz niske i liste.

Rešenje.

```
# Iteriranje kroz liste i niske
# Naredbe grananja i iteriranja
a = "Moze i Ovako."
b = ""

# Na osnovu niske a formiramo nisku b tako sto male samoglasnike
# menjamo odgovarajucim velikim samoglasnikom, karakter
# koji nije slovo zamenjujemo sa -, a sve ostale prepisujemo.
for x in a:
  if x.islower() and x in ('a','e', 'i', 'o', 'u'):
    b = b + x.upper()
  elif not x.isalpha() :
    b = b +  '-'
  else :
    b += x

print( a )
print( b )

# Zelimo da napisemo sve stepene broja 2 od 0. to 10.
# range(start, stop, step)
# Funkcija kreira objekat koji generise sekvencu celih brojeva
# pocevsi od celog broja start do broja stop sa celobrojnim
# korakom step.
# range(0, 11, 1) je potrebno za ovaj zadatak
# Ako je start 0, moze se izostaviti jer mu je to podrazumevana
# vrednost. Takodje, ukoliko je step 1, moze se izostaviti.
# range(0, 11,1)  <=> range(11)
for i in range(0, 11, 1):
  print( 2, "^", i, "=", 2**i, end = "\n" )
print()

# Stampanje svakog elementa liste u novom redu.
l = ['ovde', 'tamo', 'negde', 'svuda']
for x in l:
  print( x  )
print()

# Ukoliko u iteracijama planiramo da menjamo listu, posebno ako
# uticemo na njenu duzinu, bolje je ne iterirati kroz originalnu
# listu vec kroz kopiju.
for x in l[:] :
  l.insert( 0, x )
# Ukloniti [:] iza l u uslovu ostanka u petlji i naci cemo se u
# beskonacnoj petlji jer svakom iteracijom produzavamo listu za
# jedan element.
print()

# Stampamo elemente liste razdvojene zarezima
```

```
# koristeci indeksnu notaciju.
for i in range(len(l)):
  if i != len(l) - 1:
    print( l[i], end = ", " )
  else:
    print( l[i], end = "\n\n" )

# II nacin
i = 0
while i < len(l):
  if i != len(l) - 1:
    print( l[i], end = ", " )
  else:
    print( l[i], end = "\n\n" )
  i += 1

# Nacin za kreiranje liste
l = [ a for a in range(10) ]
print( l )
```

Rešenje 3.5: Kolekcije u Pythonu. Iteriranje.

Primer 3.6 Primer ilustruje skup kao kolekciju podataka i neke od metoda koje se mogu primeniti na skupove.

Rešenje.

```
# Skup je struktura podataka  bez ponavljanja elemenata

# s ce biti skup razlicitih karaktera reci abrakadabra
s = set("magija")
print( s )

b= ['magija', "pokus", 'abrakadabra', 'magija', "hokus" ]
# Stampamo skup ciji elementi su elementi liste b
print( set(b) )

# Ukoliko skup s sadrzi 'f' uklanjamo ga, ukoliko ne sadrzi
# dodajemo.
# Za uklanjanje je neophodno da se proveri da li je element
# u skupu. Ako uklanjamo 'f' pre provere da li je element
# skupa imacemo KeyError, u slucaju da nije element skupa.
if 'f' not in s:
  s.add('f')
else:
  s.remove('f')

# Definisemo skup od slova iz reci "mudro"
c = set("mudro")
```

```
24 # Stampamo skupove pre primene operacija
   print( "\n",s,"\n",c )
26
   # Presek skupova
28 print( s&c )
   print( s.intersection(c) )
30
   # Unija skupova
32 print( s|c )
   print( s.union(c) )
34
   # Razlika skupova
36 print( s-c )
   print( s.difference(c) )
38
   # Simetricna razlika
40 print( s^c )
   print( s.symmetric_difference(c))
```

Rešenje 3.6: Kolekcije u Pythonu. Skupovi.

Primer 3.7 Primer ilustruje upotrebu rečnika, tj. mapa.

Rešenje.

```
1  # Dictionaries (dict)
   dnevnik = {'Pera': 3,'Mira': 4, 'Dejan': 2}
3
   print( dnevnik )
5  print (type(dnevnik))

7  # key() i values () su metode koje vracaju listu kljuceva,
   # odnosno vrednosti sacuvanih u mapi.
9  print( dnevnik.keys() )
   print(dnevnik.values() )
11 print("\n")

13 # Stampamo sortirane kljuceve
   print( sorted(dnevnik.keys()) )
15
   # Peri menjamo ocenu na 5
17 # Ukoliko imamo Peru u dnevniku promenicemo mu ocenu, a da
   # ne postoji unos za Peru ovom naredbom bismo ga dodali
19 dnevnik['Pera'] = 5
   # Stampamo mapu
21 print( dnevnik, end="\n" )

23 # Metod get vraca vrednost u mapi za navedeni kljuc ukoliko
```

```
# postoji vratice None, ako nema unosa sa navedenim kljucem.
print( dnevnik.get('Sonja') )
print ("\n\n")

# Proverimo da li je Sonja u dnevniku. Stampamo vrednost
# ako smo sigurni da postoji, inace imamo KeyError.
if 'Sonja'  not in dnevnik.keys():
  dnevnik['Sonja']= 3
else:
  print( dnevnik['Sonja'] , "\n")

# items() je metod koji nam vraca objekat koji ce nam
# sukcesivno generisati parove kljuc vrednost iz mape.
print( dnevnik.items())

# Proveravamo da li Pera ima 1
print( ("Pera",1) in dnevnik.items() )

print( "\n\n" )

# Ispis sadrzaja mape koriscenjem kljuca kao indeksa
for k in dnevnik.keys():
  print( k, dnevnik[k] )

print ("\n\n")

# Ispis parova kljuc i odgovarajuca vrednost iz mape
# koriscenjem objekta dobijenog sa metodom items()
# k ce uvek uzimati vrednosti prvog elementa para, tj, kljuca,
# v ce uzimati uvek vrednost drugog u paru, tj, vrednosti.
for k, v in dnevnik.items():
  print( k, '\t-> ', v )
```

Rešenje 3.7: Kolekcije u Pythonu. Mape.

Primer 3.8 Naredni program ispituje da li je uneti broj prost. Na ovom primeru se može videti način definisanja korisničkih funkcija, učitavanje podatka sa standardnog ulaza kao i obrada izuzetka.

Rešenje.

```
# Ukljucujemo math biblioteku zbog funkcije za kvadratni koren
import math

# Funkcija prost(n)  treba da proveri da li je broj n prost i
# da vrati:     1, ako je prost ;
#               x, takvo da je x < n  i  n = x * (n/x), inace.
# Funkcija prost ima jedan argument sa podrazumevanom vrednosti
# 2. To omogucava da se funkcija pozove i bez zadavanja
```

```
# argumenata. U tom slucaju n ce imati vrednost 2.
def prost(n = 2):
  if n == 2:
    return 1

  if n % 2 == 0:
    return 2

  # Gornja granica nam je donji ceo deo kvadratnog korena iz n,
  # npr za n=49 je 7.
  # Neophodno je da konvertujemo u int zbog funkcije range.
  for x in range(3, int(math.sqrt(n))+1, 2):
    if n % x == 0:
      return x
      break
    # Naredna else grana se odnosi na for petlju, i izvrsava se
    # samo ako se iz petlje izaslo usled iscrpljivanja liste u
    # uslovu ostanka u petlji.
  else :
    return 1

try:
  # input() ispisuje poruku na ekran i preuzima liniju teksta
  # sa standardnog ulaza.
  # int() taj tekst konvertuje u tip int, ukoliko imamo ceo broj
  # u zapisu. Inace baci ce izuzetak.
  n = int(input('Unesite jedan ceo broj veci od 1: '))
  # Ukoliko nije unet trazen broj bacamo izuzetak.
  if n <= -1 :
    raise ValueError
except ValueError:
  exit('Potrebno je uneti prirodan broj veci od 1')

x = prost(n)
if( x == 1) :
  print( str(n) + " je prost" )
else:
  print( n, " = ", x, " * ", n//x )
```

Rešenje 3.8: Funkcija prost().

Primer 3.9 Naredni program generiše listu prvih n fibonačijevih brojeva i ispisuje ih na standardni izlaz. Na ovom primeru se može videti istovremena dodela, para vrednosti dvema promenljivana, vraćanje liste elemenata preko povratne vrednosti funkcije i ispis podataka različitih tipova.

Rešenje.

```
# Funkcija fibonacci(n) treba da nam generise i vrati
```

```
# listu prvih elemenata Fibonacijevog niza koji su manji od n.
# Na primer, za n = 10, vraca [1, 1, 2, 3, 5, 8]
def fibonacci(n = 6):
  niz = []
  a, b = 0, 1
  while a < n:
    niz.append(a)
    a, b = b, a + b

  return niz

print( fibonacci(2000))
# Funkcija se poziva sa podrazumevanom vrednosti argumenta
# i stampa se rezultat.
print( fibonacci() )
# Ispisujemo objekat tipa funkcija koji odgovara nasoj funkciji.
print( fibonacci )
```

Rešenje 3.9: Funkcija fibonacci().

Primer 3.10 Primer ilustruje rad sa datotekama i argumentima komandne linije, čitajući datoteku koja je prosleđena preko argumenata komandne linije i prepisujući njen sadržaj u drugu datoteku.

Rešenje.

```
# Napisati Python skript koji ce preko argumenata komandne
# linije moci da primi naziv datoteke koju ce prepisati u
# drugu datoteku. Ako je naziv datoteke koja se prepisuje
# na primer ulaz.txt prepisujemo njen sadrzaj u datoteku sa
# nazivom kopija_ulaz.txt.

# Za pristup argumentima komandne linije iz skripta neophodno
# je da ukljucimo zaglavlje sys
import sys

# Stampamo listu argumenata, koja nista drugo nego lista niski.
# Prvi argument je uvek naziv skripta.
print( sys.argv, "\n")

# Ako imamo vise od 1 argumenta komandne linije, mozda imamo
# naziv datoteke koju bismo citali. Ukoliko nemamo prekidamo rad
# programa uz navedenu poruku.
if len(sys.argv) == 1:
  exit('Nedovoljan broj argumenata komandne linije!\n')

# Imamo argument i pokusavamo da otvorimo datoteku za citanje.
# Neophodno je da proverimo da li je uspesno otvorena datoteka
# sa datim nazivom. Datoteka sa tim nazivom moze da ne postoji
```

```
# ili da nemamo pravo da je citamo. U tom slucaju bi pokusaj
# citanja takve datoteke rezultovao izuzetkom IOError.
try:
  # Otvaramo datoteku sa nazivom koji nam je prosledjen kao
  # argument, sa namerom da je citamo.
  f = open(sys.argv[1], "r")
except IOError:
  exit("Neuspesno otvaranje datoteke " + sys.argv[1] )

# Datoteku koju smo uspesno otvorili mozemo uspesno citati
# na vise nacina:
# - liniju po liniju
#     f.readline() vraca narednu liniju datoteke. Uzastopnim
#       pozivanjem metoda procitacemo sve linije.
#     f.readlines() vraca listu svih linija iz datoteke

# - karakter po karakter ili u blokovima od n karaktera ili
#   u celosti
#     f.read() vraca nisku koja je celokupni sadrzaj datoteke
#     f.read(n) vraca narednih n karaktera sadrzaja datoteke

# Citamo ceo sadrzaj
sadrzaj = f.read()

# Datoteka nije vise potrebna i zatvaramo je.
f.close()

# Otvaramo datoteku sa nazivom koji nam je prosledjen kao
# argument, sa namerom da pisemo u nju. Ukoliko datoteka postoji
# prepisacemo njen sadrzaj, ukoliko ne postoji kreiracemo je.
# Posto smo prethodnu datoteku vezanu za f zatvorili mozemo
# promenljivu f ponovo da koristimo.
try:
  f = open("kopija_"+sys.argv[1], "w")
except IOError:
  exit("Neuspesno otvaranje datoteke " + "kopija_"
       + sys.argv[1] )

# Upisujemo sadrzaj.
f.write(sadrzaj)

# Zatvaramo datoteku.
f.close()
```

Rešenje 3.10: Obrada datoteka i argumenata komandne linije.

Primer 3.11 Primer predstavlja modifikaciju prethodnog primera, tako da se obrada datoteke vrši liniju po liniju.

Rešenje.

```
# Napisati Python skript koji ce preko argumenata komandne
# linije moci da primi naziv datoteke koju ce prepisati u
# drugu datoteku. Ako je naziv datoteke koja se prepisuje,
# na primer ulaz.txt prepisujemo njen sadrzaj liniju po liniju
# numerisane u datoteku linije_ulaz.txt

# Za pristup argumentima komandne linije iz skripta neophodno
# je da ukljucimo zaglavlje sys
import sys

# Stampamo listu argumenata, koja nista drugo nego lista niski.
# Prvi argument je uvek naziv skripta.
print( sys.argv, "\n")

# Ako imamo vise od 1 argumenta komandne linije, mozda imamo
# naziv datoteke koju bismo citali. Ukoliko nemamo prekidamo rad
# programa uz navedenu poruku.
if len(sys.argv) == 1:
  exit('Nedovoljan broj argumenata komandne linije!\n')

# Imamo argument i pokusavamo da otvorimo datoteku za citanje.
# Neophodno je da proverimo da li je uspesno otvorena datoteka
# sa datim nazivom. Datoteka sa tim nazivom moze da ne postoji
# ili da nemamo pravo da je citamo. U tom slucaju bi pokusaj
# citanja takve datoteke rezultovao izuzetkom IOError.
try:
  # Otvaramo datoteku sa nazivom koji nam je prosledjen kao
  # argument, sa namerom da je citamo. Ako cemo mali broj
  # operacija obaviti sa datotekom i zatvoriti, onda taj
  # deo mozemo objediniti u with blok.

  # f je dostupno samo u bloku i odnosi se na otvorenu datoteku.
  # Prilikom napustanja with bloka datoteka ce biti automatski
  # zatvorena i nije neophodno da je ekspicitno zatvorimo.
  with  open(sys.argv[1], "r") as f:
    # citamo ceo sadrzaj po linijama
    sadrzaj = f.readlines()
except IOError:
  exit("Neuspesno otvaranje datoteke " + sys.argv[1] )

# Otvaramo datoteku sa nazivom koji nam je prosledjen kao
# argument, sa namerom da pisemo u nju. Ukoliko datoteka postoji
# prepisacemo njen sadrzaj, ukoliko ne postoji kreiracemo je.
# Posto je prethodna datoteku vezanu za f zatvorena napustanjem
# bloka mozemo promenljivu f ponovo da koristimo.
```

```
try:
  f = open("linije_"+sys.argv[1], "w")
except IOError:
  exit("Neuspesno otvaranje datoteke " + "linije_"
       + sys.argv[1] )

# Upisujemo sadrzaj
for i  in range( 0, len(sadrzaj)) :
  # Ako zelimo da koristimo + za konkatenaciji niski onda i ceo
  # broj moramo prevesti u njegovu reprezentaciju u vidu niske.
  f.write( str(i) + ": " + sadrzaj[i])

# Zatvaramo datoteku
f.close()
```

Rešenje 3.11: Obrada datoteka liniju po liniju.

Primer 3.12 Primer ilustruje nekoliko načina formatiranja niski.

Rešenje.

```
pitanja=['ime','zanimanje','mesto u kom zivim']
odgovori=['Pera','programer', 'Beograd']

print("\n\nI nacin\n")
for a,b in zip(pitanja,odgovori):
  print( 'Tvoje' + a + ' je: Moje' +a +' je '+b+'.' )

print("\n\nII nacin\n")
for a,b in zip(pitanja,odgovori):
  print( 'Tvoje {0} je: Moje {0} je {1:10}.'.format(a,b) )

print("\n\nIII nacin\n")
# drugi nacin za formatiranje izlaza
for a,b in zip(pitanja,odgovori):
  print( 'Moje %s je: Moje %s je %10s.' %(a, a, b) )

print("\n\nIV nacin\n")
for a,b in zip(pitanja,odgovori):
  print( 'Moje %(pit)s je: Moje %(pit)s je %(odg)10s.'
       % {"pit":a, "odg":b} )

print("\n\nV nacin\n")
# drugi nacin za formatiranje izlaza
for a,b in zip(pitanja,odgovori):
  print( 'Moje {pitanje} je: Moje {pitanje} je {odgovor:20s}.'
       .format(pitanje = a, odgovor = b) )
```

Rešenje 3.12: Formatiranje niski.

3.2 Regularni izrazi

Kroz primere u ovom delu biće ilustrovane metode `re` modula. Najpre na trivijalnom primeru, a kasnije kroz konkretne upotrebe prilikom obrade datoteka.

Primer 3.13 Primer ilustruje elementranu pretragu i obradu teksta uz pomoć regularnih izraza.

Rešenje.

```
# neophodno je ukljuciti sve module koje planiramo da koristimo
import sys, re

# Kreiramo neku poruku.
poruka = "Danas je divan dan"
# Zelimo da poruku poklopimo sve reci u ignore case modu.
matcher = re.match(r"(?i)([a-z]+\s*)+", poruka);

# BITNO:
# re.match() metod radi tako sto pokusava da poklopi sablon sa
# sadrzajem niske i to uvek radi iskljucivo od pocetka niske.
# Najcesce, match se koristi samo onda kada treba poklopiti
# sadrzaj niske u potpunosti
# (na primer provera ekstenzije, doslovnog naziva, itd).

# Ako je poklapanje uspesno, prikazujemo poklapanje korisniku
if matcher is not None:
  print(matcher.group())
else:
  # inace stampamo poruku o neuspehu
  print("Sablon se ne nalazi u tekst")

# BITNO:
# re.search metod radi tako sto pokusava da pronadje prvo
# poklapanje sablona sa sadrzajem niske i uvek trazi samo prvo
# poklapanje. Za sledece poklapanje, neophodno je ponovo pozvati
# search metod, ali treba voditi racuna da pretraga treba da se
# restartuje od prvog neobradjenog karaktera, a ne ponovo od
# pocetka niske.

# Trazimo rec koja pocinje da slovo d u poruci
# i to u single line modu i ignore case modu.
m = re.search(r"\bd([a-z]+)", poruka, re.S | re.I );

# BITNO:
# single line mod tumaci ceo sadrzaj poruke kao jedan jedini
# red. Posledica ovoga je da operator . u regularnom izrazu
# uzima i nove redove.
# ignore case mod ignorise razliku izmedju velikih i malih slova

```

```
# Ako je poklapanje uspelo
if m is not None:
  # stampamo kompletno poklapanje
  print(m.group())
  # sto mozemo i na ovaj nacin
  print(poruka[m.start():m.end()])
  # stampamo 1. grupu u regex-u, tj. ono sto odgovora \1
    bekreferenci
  print(m.group(1))
  # stampamo listu svih grupa u regex-u
  print(m.groups())

# Trazimo sledecu rec koja pocinje na d.
# BITNO:
# Obratiti paznju na pomeranje offseta u poruci. Pretragu
# restartujemo od prvog neobradjenog karaktera.
m = re.search(r"\bd([a-z]+)", poruka[m.end():], re.S);

# Ako je poklapanje uspelo, ponovo stampamo isto
if m is not None:
  print(m.group())
  print(poruka[m.start():m.end()])
  print(m.group(1))
  print(m.groups())

# Za pronalazenje svih poklapanja koristi se metod findall
# Metod vraca listu svih poklapanja. Ukoliko regularni izraz
# ima podizraze u zagradama, vratice listu n-torki niski
# koje su prepoznale sve grupe pri svakom poklapanju.
reci = re.findall(r'(?i)\b(d([a-z]+))', poruka);
print(reci)

# Metod sub iz re modula menja svako poklapanje sa regularnim
# izrazom tekstom zamene, u kom se moze navesti konkretan tekst
# zamene, a moze se referisati i na bekreference.
print(re.sub(r'(?i)\b(d([a-z]+))', r'\2-\1', poruka));

# Zamena prepoznatog teksta je moguca i pomocu replace metoda
# definisanog nad tipom podataka string.
# replace metod je znatno manje fleksibilan u odnosu na re.sub.
print(poruka.replace("Danas", "Sutra"))
```

Rešenje 3.13: Metodi re modula.

Metodi iz `re` modula se na predstavljen način koriste samo onda kada nam treba jednokratno poklapanje regularnih izraza. Za bilo koje složenije ili češće poklapanje ili komplikovaniju logiku u programu koriste se kompajlirani regularni izrazi. Oni će biti korišćeni u sledećem primeru.

Primer 3.14 Korisnik kao argument komandne linije prosleđuje putanju do direktorijuma u kome se nalaze seminarski radovi studenata. Svakom studentu je dodeljen direktorijum čije ime prati alas format. Program treba da izlista zadatke koje su uradili samo oni studenti koji su kreirali direktorijum sa ispravnim imenom. Zadaci su opisani jednom cifrom i ekstenzijom koja može biti `.pas`, `.java`, `.c` ili `.cpp`.

Rešenje.

```
# Ukoliko nam u programu treba bilo koja usluga operativnog
# sistema, moramo ukljuciti modul os
import sys, re, os

# Proveravamo argumente komandne linije.
if len(sys.argv) > 1:
  homedir = sys.argv[1]
else:
  homedir = '.'

# Kompajliramo regex koji opisuje pravilan naziv direktorijuma.
# U ovom slucaju radi se o alas nalogu
re_dir = re.compile(r'^m[mnvlri]\d{5}$')
# Kompajliramo regex koji opisuje pravilno ime datoteke. U ovom
# slucaju radi se zadacima imenovanim brojevima 0-9, pri cemu
# je moguca ekstenzija java, pas, cpp ili c.
re_file = re.compile(r'^(\d)\.(pas|java|cpp|c)$')

# Listamo polazni direktorijum
for f in os.listdir(homedir):
  # Cim nadjemo datoteku u direktorijumu, kreiramo njenu putanju
  dirPath = os.path.join(homedir, f);
  # Ispitujemo da li naziv datoteke odgovara alas nalogu
  m = re_dir.match(f)
  # Ako je u pitanju direktorijum i ime je ispravno
  if os.path.isdir(dirPath) and m is not None:
    # stampamo naziv direktorijuma
    print("\n" + m.group())
    # i listamo direktorijum.
    for sf in os.listdir(dirPath):
      # Kreiramo putanju do datoteke u studentskom direktorijumu
      path = os.path.join(dirPath, sf)
      # Ispitujemo da li naziv datoteke odgovara broju zadatka
      # pracenom ispravnom ekstenzijom.
      m = re_file.match(sf)
      # Ako je u pitanju regularni datoteke ispravnog imena
      if os.path.isfile(path) and m is not None:
        # stampamo naziv datoteke.
        print("\t" + m.group())
```

Rešenje 3.14: Obrada seminarskih radova

Primer 3.15 Ovaj primer je proširenje prethodnog primera. Informacije o studentima koji su radili seminarski zadatak se nalaze u datoteci indeksi.txt. U svakoj liniji datoteke zapisan je broj indeksa, praćen imenom i prezimenom studenta. Za svakog studenta u datoteci indeksi.txt, potrebno je prikazati koje je sve zadatke uradio na osnovu sadržaja njegovog direktorijuma. Prikaz treba da bude tabelarni, poput ovog:

```
Pera Peric      - pas - c
Ivan Ivanovic   c pas - java
```

Prva kolona odgovara imenu studenta, naredna prezimenu, potom redom za zadatke počevši od zadatka pod rednim brojem 1.

Rešenje.

```
import sys, re, os

# Ako imamo argumente komendne linije uzecemo ga za polazni
# direktorijum, inace cemo koristiti tekuci direktorijum.
if len(sys.argv) > 1:
  homedir = sys.argv[1]
else:
  homedir = '.'

# Recnik koji mapira alas nalog sa imenom i prezimenom studenata
studenti = {}

# Kreiramo putanju do datoteke indeksi
putanja = os.path.join(homedir, 'indeksi')
# Kompajliramo regularni izraz koji opisuje ispravnu liniju sa
# podacima studenta u datoteci indeksi
re_ime =  re.compile(r"m[mnvlri]\d{5},\s*[A-Za-z ]+")

try:
  # Otvaramo datoteku indeksi
  with open(putanja, 'r') as f:
    # i citamo liniju po liniju.
    for linija in f.readlines():
      # Ako linija ima sadrzaj oblika 'indeks, Ime Prezime'
      if re_ime.match(linija) is not None:
        # razbijamo liniju oko zareza pracenog belinama ili
        # novog reda, jer neke linije ce imati nov red na kraju.
        info = re.split(r',\s*|\n', linija)
        # U recnik ubacujemo alas nalog kao kljuc,
        # a ime i prezime studenta kao vrednost.
        studenti[info[0]] = info[1]
  # Nakon napustanja with bloka automatski se izvrsava f.close()
except IOError:
    # Obrada greske
    sys.exit("Neuspesno otvaranje datoteke indeksi")

# Kompajliramo regularni izraz koji opisuje pravilan naziv
```

```
# direktorijuma. U ovom slucaju radi se o alas nalogu.
re_dir = re.compile(r'^m[mnvlri]\d{5}$');
# Kompajliramo regularni izraz koji opisuje pravilno ime
# datoteka. U ovom slucaju radi se o zadacima imenovanim rednim
# brojem, pri cemu je moguca ekstenzija java, pas, cpp ili c.
re_file = re.compile(r'^(\d)\.(java|pas|c(pp)?)$');

# Najveci redni broj zadatka koji su studenti uradili
max_br_zadatka = 0
# Recnik koji mapira studenta i broj zadatka sa ekstenzijom
# uradjenog zadatka
zadaci = {}

# Listamo polazni direktorijum
for f in os.listdir(homedir):
  # Kreiramo putanju do izlistanog podatka, potencijalno
  # direktorijuma sa studentskim zadacima
  stud_dir_path = os.path.join(homedir,f)
  # Ispitujemo da li mu ime odgovara alas nalogu.
  m = re_dir.match(f)
  # Ako je u pitanju direktorijum i ime je ispravno i student
  # se nalazi u recniku studenti, tj. datoteci indeksi
  if os.path.isdir(stud_dir_path) and m is not None and
   m.group() in studenti:
    # Pamtimo indeks studenta
    student_indeks = m.group()
    # i listamo studentski direktorijum
    for sf in os.listdir(stud_dir_path):
      # pravimo putanju do zadatka
      zadatak_path = os.path.join(stud_dir_path, sf)
      # Ispitujemo da li zadatak ima odgovorajuce ime, tj. da li
      # se radi o broju zadatka pracenim ispravnom ekstenzijom.
      m = re_file.match(sf)
      # Ako je u pitanju regularni fajl ispravnog imena
      if os.path.isfile(zadatak_path) and  m is not None:
        # izdvajamo redni broj zadatka,
        zadatak = int(m.group(1))
        # u recnik zapisujemo tu informaciju
        # kljuc je uredjeni par (indeks, zadatak)
        # vrednost je ekstenzija u kojoj je uradjen zadatak.
        zadaci[(student_indeks, zadatak)] = m.group(2)
        # Pamtimo najveci broj zadatka.
        if zadatak > max_br_zadatka:
          max_br_zadatka = zadatak

# Stampamo izdvojene podatke
for indeks, ime in studenti.items():
  # Prvo ime i prezime studenta
  print( ime + "  ", end =' ')
```

```
  # zatim iteriramo kroz zadatake [1,max_br_zadatka]
87  for i in range(1, max_br_zadatka+1):
      # Ako je student uradio zadatak, nalazi se u recniku zadaci
89    if (indeks, i) in zadaci:
        # stampamo ekstenziju u kojoj je resen zadatak,
91      print( "\t" + zadaci[(indeks, i)], end =' ')
      else:
93      # inace, stampamo -, kao znak da zadatak nije resio.
        print( "\t-", end =' ')
95  print()
```

Rešenje 3.15: Obrada seminarskih radova. Proširenje.

Primer 3.16 Korisnik programu prosleđuje naziv početne HTML datoteke preko argumenta komandne linije. Program pronalazi i prati sve linkove počevši od početne HTML datoteke.

Rešenje.

```
1  import sys, re

3  # Funkcija rekurzivno obradjuje datoteku i upisuje ime datoteke
   # koju je obradila u listu obradjene_datoteke
5  def obradi_datoteku(ime_datoteke):
     # Ukoliko je datoteka već evidentirana kao obrađena,
7    # prekidamo funkciju, da ne bismo imali beskonacnu rekurziju.
     if(ime_datoteke in obradjene_datoteke):
9      return
     # Inace, evidentirano datoteku u obradjene_datoteke
11   # obradjene_datoteke je globalna lista i ovo je potpuno
     # validno i vidljivo svima nakon zavrsetka funkcije.
13   obradjene_datoteke.append(ime_datoteke)

15   # Ucitavamo ceo sadrzaj datoteke u svoj program.
     try:
17     with open(ime_datoteke,"r") as f:
         datoteka = f.read()
19   except IOError:
       exit("Neuspelo otvaranje '" + ime_datoteke + "'")
21   # BITNO:
     # Ukoliko treba da tumacimo ceo sadrzaj datoteke kao jednu
23   # liniju, potrebno je da ga celog ucitamo u memoriju programa.
     # U ovom slucaju trazimo linkove. U HTML-u je potpuno validno
25   # da se tag moze prostirati u vise redova. Kao resenje ove
     # situacije namece se mogucnost da ucitamo ceo sadrzaj i
27   # koristimo single line mode pri poklapanju regularnog izraza.

29   # Kompajliramo regularni izraz koji opisuje linkove u HTML-u i
     # ukljucujemo ignore case i single line modove
```

```
  ri=re.compile(r'<a\s+href\s*=\s*"(.*?)">.*?</a>', re.I | re.S)

  # re.finditer() metod vraca iterator nad svim poklapanjima
  # u nisci koja mu je prosledjen kao argument.
  # U petlji iteriramo kroz sva poklapanja.
  for m in ri.finditer(datoteka):
    # Citamo link na koji pokazuje a tag.
    url = m.group(1)
    obradi_datoteku(url)


# Proveravamo argumente komandne linije.
if len(sys.argv) > 1:
  pocetna = sys.argv[1]
else:
  pocetna = "index.html"

# Inicijalizujemo globalnu listu i pocinjemo obradu od pocetne.
obradjene_datoteke = []
obradi_datoteku(pocetna)

# Na kraju samo stampamo sadrzaj liste obradjene_datoteke.
for datoteka in obradjene_datoteke:
  print(datoteka)
```

Rešenje 3.16: Praćenje HTML linkova.

Primer 3.17 Naredni program je proširenje programa iz prethodnog primera. Proširenje se ogleda u tome što se od početne stranice vodi evidencija o broju posećivanja veb stranica. Nakon obrade prikazuje se sortiran spisak posećenih stranica od najposećenije ka najređe posećivanoj.

Rešenje.

```
import re, sys

def obradi_datoteku(ime_datoteke):
  # Ukoliko je datoteka već evidentirana kao obrađena,
  # prekidamo funkciju, da ne bismo imali beskonacnu rekurziju.
  if (ime_datoteke in obradjene_datoteke):
    return
  # Inace, evidentiramo je.
  obradjene_datoteke.append(ime_datoteke)

  # Ucitavamo ceo sadrzaj datoteke u svoj program.
  try:
    with open(ime_datoteke,'r') as f:
      dat = f.read()
  except IOError:
```

```
16      exit('Neuspelo otvaranje datoteke ' + ime_datoteke)

18    # Kompajliramo regularni izraz koji opisuje linkove u HTML-u i
      # ukljucujemo ignore case i single line modove.
20    ri = re.compile(r'<a\s+href\s*=\s*"(.*?)">(.*?)</a>', re.I |
       re.S)
      # U petlji iteriramo kroz sva poklapanja
22    for m in ri.finditer(dat):
        # Izdvajamo tekst linka
24      tekst = m.group(2)
        # Izdvajamo link ka stranici
26      url = m.group(1)

28      # Proveravamo da li je url u recniku
        if url in statistika:
30        # Ako jeste, uvecavamo broj poseta
          statistika[url] += 1
32      else:
          # Inace ubacujemo link u recnik
34        # i inicijalizujemo broj poseta
          statistika[url] = 1
36
        obradi_datoteku(url)
38


40  # Proveravamo argumente komandne linije.
    if len(sys.argv) > 1:
42    pocetna = sys.argv[1]
    else:
44    pocetna = "index.html"

46  # Inicijalizujemo globalnu listu.
    obradjene_datoteke = []
48  # Inicijalizujemo recnik koji ce nam  mapirati linkove za brojem
    # pojavljivanja tog linka.
50  # Kljuc je link, vrednost je broj pojava.
    statistika = {}
52  # Pocinjemo obradu od pocetne.
    obradi_datoteku(pocetna)
54
    # Da bismo sortirali linkove po vrednosti, prvo treba da
56  # napravimo listu parova oblika (kljuc, vrednost) sto postizemo
    # sa recnikovim metodom items().
58  popd = list(statistika.items())
    # Potrebno je da prethodnu listu parova sortiramo opadajuce
60  # prema drugom elementu para.
    popd.sort(reverse=True, key=lambda t: t[1])
62
    # Na kraju prikazujemo sortirani izvestaj
```

```
64 for no, url in popd:
     print( url, "\t" , no)
```

Rešenje 3.17: Praćenje HTML linkova. Proširenje.

Primer 3.18 Programu se kao prvi argument komandne linije navodi ime HTML datoteke koja sadrži tabelu u kojoj se nalaze rezultati ispita. Tabela ima tri kolone. U prvoj se nalazi ime i prezime studenta u drugoj broj poena na teorijskom delu ispita i u trećoj broj poena na praktičnom delu ispita. Broj poena na svakom delu treba da bude ceo broj iz intervala $[0, 100]$. Ne izdvajaju se studenti sa ne ispravnim podacima. Program treba da pročita sadržaj tabele u datoteku *sortirano.txt* da ispiše ime i prezime studenata i ukupan broj poena na ispitu. Ukupan broj poena se računa kao aritmetička sredina brojeva poena sa oba dela ispita. Izveštaj treba da bude uređen u opadajućem poretku prema ukupnom broju poena.

Na primer, ulazna HTML datoteka bi mogla da sadrži fragment poput ovog:

```
<table width="220">
  <tr>
   <td>Savic Marko </td>
   <td> 35 </td> <td> 0 </td>
</tr>  <tr>
<td>Bogdanovic Vesna</td>
<td> 68 </td>
<td> 98 </td>
</tr>
<tr>
<td>Filipovic Vukasin</td><td>96</td> <td> 70 </td>
</tr>
<tr> <td>Pavlovic Sofija</td><td>100</td><td> 89 </td> </tr>
</table>
```

Tada bi rezultat programa bio:

```
1. Pavlovic Sofija 94.5
2. Filipovic Vukasin 83.0
3. Bogdanovic Vesna 83.0
4. Savic Marko     17.5
```

Rešenje.

```
import sys, re
2
# Proveravamo da li je navedeno ime ulazne html datoteke
4 # Metod re.fullmatch() ispituje da li se sadržaj niske poklapa
# u celosti sa regularnim izrazom.
6 if len(sys.argv) == 1 or
    (re.fullmatch('.*\.html',sys.argv[1],re.I) is None):
  sys.exit('Prvi argument mora biti html datoteka!')
8
# Ucitavamo ceo sadrzaj datoteke u memoriju.
```

```
try:
  with open(sys.argv[1],"r") as ff:
    dat = ff.read()
except IOError:
  exit( "Neuspesno otvaranje i citanje datoteke: '%s'"
       % sys.argv[1])

# Lista studenti ce nam mapirati redni broj studenta (tj.indeks
# u listi) sa njegovim imenom i prezimenom. Smatramo da se mogu
# pojaviti studenti sa istim imenom i prezimenom u rezultatima.
# Svako pojavljivanje treba da predstavlja novog studenta. Zato
# ih razlikujemo rednim brojem, tj. indeksom u listi.
studenti = []
# Lista poeni ce nam mapirati redni broj studenta (tj.indeks
# u listi) sa ukupnim poenima sa ispita.
poeni = []

# Kompajliramo regularan izraz koji ćemo izdvajati samo ispravne
# podatke studenata.
# Koristimo imenovane grupe, da ne bismo brojali zagrade.
ri = re.compile(r"<tr>"
+r"\s*<td>\s*(?P<ime>[A-Za-z]+(?: +[A-Za-z]+)+)\s*</td>"
+r"\s*<td>\s*(?P<zadaci>0|[1-9]\d|100)\s*</td>"
+r"\s*<td>\s*(?P<teorija>0|[1-9]\d|100)\s*</td>"
+r"\s*</tr>")

# Iteriramo kroz sva poklapanja.
for m in ri.finditer(dat):
  # Prepoznat tekst grupama zadaci i teorija odgovara zapisu
  # celog broja. Da bismo ih sabrali konvertujemo ih u
  # cele brojeve. Delimo realnom konstantom da bismo izbegli
  # celobrojno deljenje.
  ukupno = int(m.group('zadaci')) + int(m.group('teorija'))
  ukupno /= 2.0
  # Izdvajamo ime studenta prepoznato grupom ime i dodajemo
  # u listu studenti
  studenti.append( m.group('ime') )
  # Dodajemo studentove poene u listu poeni. Na taj nacin na
  # istom indeksu u prvom listi su nam podaci studenta, a u
  # listi poeni nalaze poeni odgovarajuceg studenta.
  poeni.append(ukupno)

# Da bismo sortirali studente, prvo treba da napravimo listu
# parova oblika (poeni, student) sto postizemo sa funkcijom
# zip. Ona ce u parove spajati elemente na istim indeksima iz
# obe liste, sto je nama i potrebno. Konvertujemo u listu.
sortirani_poeni = list(zip(poeni, studenti)  )

# sort ce parove sortirati, podrazumevano po prvom elementu
```

```
# para, pa zatim po drugom, u rastucem poretku. Zadajemo da
# nam treba opadajuci poredak.
sortirani_poeni.sort(reverse=True)

# Otvaramo datoteku u koju cemo upisati izvestaj.
try:
  with open('rezultati.txt','w') as f:
    # Zelimo numerisan spisak studenta i koristimo enumerate.
    # Enumerate ce dodati redni broj ispred svakog para iz
    # liste sortirani_poeni. Redni brojevi pocinju od 0.
    for j,(bodovi,ime) in enumerate(sortirani_poeni):
      # Cele brojeve konvertujemo u niske zbog konkatenacije.
      f.write(str(j+1) + ". " + ime + "\t\t" + str(bodovi) +
   "\n")
    # Alternativno se moglo pristupati elementima preko
    # indeksne notacije i indeks koristiti u ispisu.
except IOError:
  sys.stderr.write("Neuspelo otvaranje file 'rezultati.txt'\n")
```

Rešenje 3.18: Podaci iz HTML tabele. Rezultati ispita.

Zadatak 3.1 Napisati *python*-skript koji štampa na standardni izraz autora (−a), cenu (−c), izdavača (−i) ili godinu izdanja (−g) knjige koja se navodi kao argument komande linije, u zavisnosti od prisutne opcije komandne linije (u slučaju da nema opcija, ispisati sve podatke o traženoj knjizi). Informacije o knjigama se nalaze u fajlu *knjige.xml* koji ima sledeći format:

```
<?xml version="1.0" encoding="utf-8" ?>
<lista_knjiga>
<knjiga rbr="1" >
<naslov> Yacc </naslov>
<autor> Filip Maric </autor>
<godina_izdanja> 2004 </godina_izdanja>
<izdavac> Matematicki fakultet </izdavac>
<cena valuta="rsd"> 1000 </cena>
</knjiga>
<knjiga rbr="2" >
<autor> Fredrik Lundh </autor>
<cena valuta="eur"> 50 </cena>
<izdavac> O'Reilly & Associates </izdavac>
<godina_izdanja> 2001 </godina_izdanja>
<naslov> Python Standard Library </naslov>
</knjiga>
</lista_knjiga>
```

Primeri pozivanja programa i ispis:

```
./knjiga -a Yacc
Filip Maric
```

```
$ ./knjiga -c "Python Standard Library"
50eur
```

■

Rešenje.

```
# Ukljucivanje potrebnih modula
import re
import sys

# Promenljive koji definisu korisnikov izbor
autor = False
godina = False
izdavac = False
cena = False

# Dok korisnik ne unese, naslov je None
naslov = None

# Provera broja argumenata
if len(sys.argv) < 2:
  sys.exit('Program se poziva sa %s -[acig] naslov_knjige'
           % sys.argv[0])
#Provera opcija koje su zadate i preuzimanje naslova
if re.match(r'-[aicg]+', sys.argv[1] ) :
  if 'a' in sys.argv[1]:
    autor = True
  if 'c' in sys.argv[1]:
    cena = True
  if 'g' in sys.argv[1]:
    godina = True
  if 'i' in sys.argv[1]:
    izdavac = True

  if len(sys.argv) >2:
    naslov = sys.argv[2]
  else:
    sys.exit("Jedan od argumenata mora biti naslov knjige")
elif sys.argv[1][0] == '-' :
    sys.exit('Nepodrzana opcija')
else:      # nije opcija
  naslov = sys.argv[1]

# Ocigledno da se tekst koji nas sablon treba da prepoznaje
# nalazi u vise redova, pa ucitavamo ceo sadrazaj datoteke
try:
  with open("knjige.xml","r") as f:
    sadrzaj = f.read()
except IOError:
```

```
    sys.exit('Neuspelo otvaranje datoteke knjige.xml')

# Iz primera je ocigledno da redosled tagova nije fiksiran.
# Jedno moguce resenje takvog problema je upotreba preduvida
# Zbog nacina na koji preduvidi rade (binarno, tj. ima ili nema,
# neucestvovanja u samom poklapanju i vracanju na pocetni offset
# nakon evaluacije izraza), oni omogucavaju da prepoznamo
# sablone koji mogu da imaju permutovani sadrzaj.
# Takav pristup je u ovom resenju.

# Preduvidi ne ucestvuju u samom poklapanju vec samo vrse
# proveru da li nesto postoji ili ne. Zbog takvog ponasanja,
# mi rucno moramo da poklopimo ono sto su preduvidi prepoznali
# i upravo to radi deo .*? u 81 liniji koda. Kada bismo to
# zaboravili, ne bismo mogli da poklopimo ništa, jer se
# preduvidima ne bismo pomakli od taga <knjiga rbr="1">, a na
# red bi doslo da se poklopi zatvaracki tag </knjiga>.

# Za primenu ovog resenja jako je bitno da imamo garancije da
# ce svaka  knjiga imati sve podatke, nebitno u kom redosledu.
# Ukoliko bi se desilo da jedna knjiga nema neki podatak, na
# primer, autora, preduvid bi pronasao prvog sledeceg autora i
# dodelio ga knjizi cije podatke citamo. To bi naravno bilo
# pogresno, i obe knjige bi imale istog autora, greškom.

# Iako preduvidi ne vrse poklapanje, sva imenovana grupisanja
# mozemo da koristimo kasnije bez ikakvih problema.
re_knjiga = re.compile(
  r'<knjiga\s+rbr\s*=\s*"(?P<rbr>[1-9]\d*)"\s*>\s*'
  + r"(?=.*?<naslov>\s*(?P<naslov>.*?)\s*</naslov>)"
  + r"(?=.*?<autor>\s*"
  +   r"(?P<autor>[A-Z][a-z]+(?:\s+[A-Z][a-z]+)+)\s*</autor>)"
  + r"(?=.*?<izdavac>\s*(?P<izdavac>.*?)\s*</izdavac>)"
  + r"(?=.*?<godina_izdanja>\s*"
  +   r"(?P<godina>[12]\d{3})\s*</godina_izdanja>)"
  + r'(?=.*?<cena\s+valuta\s*=\s*"(?P<valuta>rsd|eur)">'
  +   r'\s*(?P<cena>0|[1-9]\d*)\s*</cena>)'
  + r".*?"
  + r'</knjiga>',re.S)

# Ovde je ocigledno da je svaka knjiga opisana svojim rednim
# brojem kojem su pridruzene neke informacije. Mapiranje ovog
# tipa, nas navodi da koristimo rečnik kao kolekciju podataka.
# Kljuc ce biti redni broj, a vrednost ce biti lista informacija
# Svaki unos u recniku je sledeceg oblika
# redni_broj -> naslov, autor, godina, izdavac, cena, valuta
katalog = {}

# Iteriramo kroz sva uspesna poklapanja
```

```
93 for m in re_knjiga.finditer(sadrzaj):
     # i ubacujemo podatke u recnik u ranije pomenutom formatu
95   katalog[m.group('rbr')] = [m.group('naslov'),m.group('autor'),
                                m.group('izdavac'),m.group('godina'),
97                              m.group('cena'),m.group('valuta')]

99 # da se uverimo da smo lepo pokupili podatke
   #for (k,v) in katalog.items():
101  #print(k,v)

103 podaci = None
   # pretrazujemo recnik po naslovu i izdvajamo podatke
105 for k,v in katalog.items():
     if v[0] == naslov:
107    podaci = v[1:] # [autor, izdavac, godina, cena, valuta]
       break
109
   # Stampamo sve ono sto zeli korisnik
111 if autor :
     print( "Autor:", podaci[0])
113 if izdavac:
     print( "Izdavac:",podaci[1])
115 if godina:
     print( "Godina izdanja:", podaci[2])
117 if cena:
     print( "Cena:", podaci[3],podaci[4])
119
   if not(autor or izdavac or godina or cena):
121   for info in podaci:
       print( info, end=" ")
123  print( '\n')
```

Rešenje 3.19: Biblioteka knjiga. Preduvidima.

Rešenje. na drugi način

```
1  import re, sys

3  if len(sys.argv) < 2:
     sys.exit('Program se poziva sa %s -[acig] naslov_knjige'
5             % sys.argv[0])

7  autor   = False
   godina  = False
9  izdavac = False
   cena    = False
11 naslov  = None

13 if re.match(r'-[aicg]+', sys.argv[1] ) :
     if 'a' in sys.argv[1]:
```

```
15      autor = True
    if 'c' in sys.argv[1]:
17      cena = True
    if 'g' in sys.argv[1]:
19      godina = True
    if 'i' in sys.argv[1]:
21      izdavac = True

23  if len(sys.argv) >2:
      naslov = sys.argv[2]
25  else:
      sys.exit("Jedan od argumenata mora biti naslov knjige")
27elif sys.argv[1][0] == '-' :
      sys.exit('Nepodrzana opcija')
29else: # nije opcija
    naslov = sys.argv[1]
31
try:
33  with open("knjige.xml","r") as f:
      sadrzaj = f.read()
35except IOError:
    exit('Neuspesno otvaranje datoteke knjige.xml')
37
# U ovom resenju izdvajamo prvo tekst koji odgovara jednoj
39# knjizi, tj. nalazi se izmedju otvorenog i zatvorenog taga
# <knjiga>. Zatim  taj tekst parsiramo za pojedinacne
41# informacije o knjizi. Ovaj način je robustniji na greske
# i lakse se eliminisu knjige koje imaju nepotpune informacije.
43# U resenju sa preduvidom moze se dogoditi da u slucaju
# izostanka informacije o npr. autoru, povuku se informacije o
45# autoru naredne knjige u xml-u.
re_knjiga =re.compile(
47  r'<knjiga\s+rbr\s*=\s*"(?P<rbr>[1-9]\d*)"\s*>\s+'
    + r'(?P<podaci>.*?)</knjiga>', re.DOTALL)  # isto sto i re.S
49
re_naslov = re.compile(
51  r"<naslov>\s*(?P<naslov>.+?)\s*</naslov>")
re_autor = re.compile(
53  r"<autor>\s*(?P<autor>[A-Z][a-z]+\s+[A-Z][a-z]+)\s*</autor>")
re_izdavac = re.compile(
55  r"<izdavac>\s*(?P<izdavac>.+?)\s*</izdavac>")
re_cena = re.compile(
57  r'<cena\s+valuta\s*=\s*"(?P<valuta>eur|rsd)">'
    + r'\s*(?P<cena>0|[1-9]\d*)\s*</cena>')
59re_godina = re.compile(

    r'<godina_izdanja>\s*(?P<godina>[12]\d{3})\s*</godina_izdanja>')
61
katalog = {}
```

```
63 #rbr -> [naslov, autor, izdavac, godina, cena, valuta]

65 # Iteriramo kroz sva poklapanja knjiga
   for m in re_knjiga.finditer(sadrzaj):
67   rbr = m.group('rbr')
     # Izdvajamo tekst unutar tagova <knjiga>
69   knjiga_podaci = m.group('podaci')
     # podaci je lista vrednosti koju cemo cuvati u recnik
71   # ukoliko budemo nasli sve podatke
     podaci = []
73   # Proveravamo ima li knjiga naslov.
     mm = re_naslov.search(knjiga_podaci)
75   if mm is not None:
       podaci.append(mm.group('naslov'))
77   else:
       continue
79   # Proveravamo ima li knjiga autora.
     mm = re_autor.search(knjiga_podaci)
81   if mm is not None:
       podaci.append(mm.group('autor'))
83   else:
       continue
85   # Proveravamo ima li knjiga izdavaca
     mm = re_izdavac.search(knjiga_podaci)
87   if mm is not None:
       podaci.append(mm.group('izdavac'))
89   else:
       continue
91   # Proveravamo ima li knjiga godinu izdanja.
     mm = re_godina.search(knjiga_podaci)
93   if mm is not None:
       podaci.append(mm.group('godina'))
95   else:
       continue
97   # Proveravamo ima li knjiga cenu i valutu.
     mm = re_cena.search(knjiga_podaci)
99   if mm is not None:
       podaci.append(mm.group('cena'))
101    podaci.append(mm.group('valuta'))
     else:
103    continue
     # Svaka ranija provera u slucaju greske bi nas
105  # prebacila na sledece poklapanje.
     # Dakle, imamo sve podatke. Cuvamo ih u recnik.
107  katalog[int(rbr)] = podaci

109 podaci = None
    # Izdvajamo podatke o knjizi sa unetim naslovom
111 for k,v in katalog.items():
```

```
    if v[0] == naslov:
113   podaci = v[1:] # [autor, izdavac, godina, cena, valuta]

115 # Prikazujemo zeljene informacije
    if autor :
117   print( "Autor:", podaci[0])
    if izdavac:
119   print( "Izdavac:",podaci[1])
    if godina:
121   print("Godina izdanja:", podaci[2])
    if cena:
123   print( "Cena:", podaci[3],podaci[4])

125 if not(autor or izdavac or godina or cena):
    for info in podaci:
127   print( info, end=" ")

129 print( '\n')
```

Rešenje 3.20: Biblioteka knjiga. Kombinacija regularnih izraza.

4. Konačni automati

Konačni automat (skraćeno KA) je idealizovana mašina kojom se prepoznaju šabloni u ulaznom nizu simbola. Svaki konačni automat sačinjen je od sledećeg:

- Q - konačan skup stanja automata;
- Σ - konačan skup simbola (alfabet);
- q_0 - početno stanje;
- F - skup završnih stanja;
- δ - funkcija prelaza.

Formalno, konačni automat se definiše kao uređena petorka $(Q, \Sigma, q_0, F, \delta)$. Konačni automati mogu da se podele u dve kategorije: deterministički konačni automati (DKA) i nedeterministički konačni automati (NDKA). Kada su u pitanju DKA, za svaki ulazni simbol mašina može da pređe u tačno jedno stanje. U ovom slučaju funkcija prelaza se definiše kao: $\delta : Q \times \Sigma \to Q$ Ako je funkcija prelaza definisana za svako stanje po svakom simbolu, tada se radi o potpunom determinističkom konačnom automatu (PDKA). Pored ovoga, važna karakteristika determinističkog konačnog autotama je ta da nije dozvoljen epsilon (ε) prelaz. To znači da DKA ne može da promeni stanje, a da sa ulaza ne pročita nijedan karakter. Takođe, važno je primetiti da postoji veliki broj različitih DKA za isti šablon. Zbog ovoga se prilikom implementacije bira deterministički konačni automat sa minimalnim brojem stanja, tj. minimalni deterministički konačni automat (MDKA).

U poređenju sa DKA, nedeterministički konačni automati dodaju dva svojstva:

- ε prelaz je dozvoljen, tj. automat može da promeni stanje bez čitanja simbola sa ulaza.
- Automat ima mogućnost da pređe u jedno od više različitih stanja za isti ulaz.

Iako su dodata ova dva svojstva, ona ne doprinose nikakvoj većoj moći NDKA u poređenju sa DKA. U tom smislu, obe kategorije konačnih automata su ekvivalentne. U odnosu na DKA, jedino po čemu se NDKA razlikuju je definicija funkcije prelaza: $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to$

$D, D \in \mathscr{P}(Q)$ Iz definicije funkcije prelaza za NDKA, može se primetiti da za svaki ulazni simbol, pa i, automat može da pređe u bilo koji podskup stanja automata, tj. može da prođe kroz više od jednog stanja. U slučaju NDKA, potrebno je primetiti da se ulazni string prihvata ako postoji neka putanja, tj. niz prelaza koji vodi do završnog stanja.

4.1 Primeri implementacije

Kada je u pitanju implementacija konačnih automata, potrebno je izvršiti sledeće korake:

1. Za dati jezik smisliti regularni izraz koji ga opisuje.
2. Na osnovu regularnog izraza konstruirati NDKA Tompsonovom ili Gluškovljevom konstrukcijom.
3. Transformisati NDKA u DKA.
4. Minimalizovati dobijeni DKA.
5. Na osnovu dobijenog MDKA u koraku 4, konstruisati funkciju prelaza koja je osnov za implementaciju.

Primer 4.1 Konstruisati MDKA koji prepoznaje samo one binarne reči koje imaju paran broj nula.

Rešenje. Zadatak je dovoljno jednostavan da možemo iz glave da konstruišemo automat. Na početku, uvek treba definisati alfabet, tj. $\Sigma = \{0,1\}$. Zatim, treba definisati skupove stanja automata. Očigledno je da su nam dovoljna samo dva stanja. Stanje `P` koje označava paran broj nula na ulazu i stanje `N` koje označava neparan broj nula na ulazu. Dakle, skup $Q = \{P,N\}$. Na početku, nema nijedne nule na ulazu, pa je početno stanje stanje `P` , dakle $q = P$. S obzirom da tražimo paran broj nula u ulaznoj reči, stanje `P` će biti i završno stanje, pa je skup $F = \{P\}$. Na kraju ostaje samo da definišemo funkciju prelaza , koju prikazujemo sledećom tablicom:

stanje \ simbol	0	1
P	N	P
N	P	N

Tabela 4.1: Matrica prelaza za MDKA koji prepoznaje reči sa parnim brojem 0 u zapisu.

Najjednostavniji način da kodiramo tablicu u programskom jeziku pajton je uz pomoć rečnika, tj. mape. Ključ u rečniku treba da bude uređeni par $(stanje, simbol)$, a vrednost pridružena ključu treba da bude novo stanje u koje automat treba da pređe. U kodu, to se postiže na sledeći način:

```
#!/usr/bin/python3
import sys

# proglasavamo pocetno i zavrsno stanje
stanje  = 'P'
zavrsno = 'P'

# funkciju prelaza predstavljamo kao rečnik
prelaz = {
```

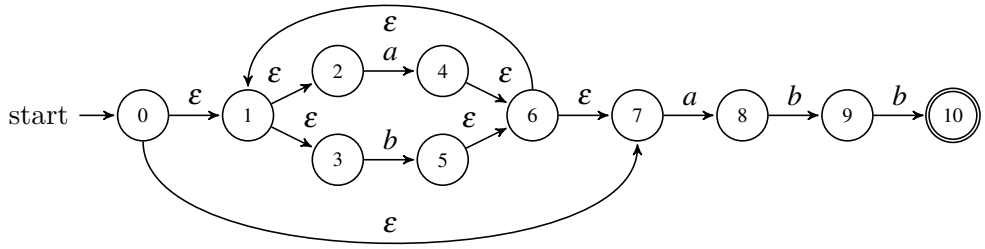
```
          ('P','0'):'N', ('P','1'):'P',
          ('N','0'):'P', ('N','1'):'N'
        }

# u petlji
while True:
  # pokusavamo da ucitamo karakter
  try:
    c = input('Unesite 0 ili 1: ')
    # ako procitani karakter nije iz alfabeta
    if (c != '0' and c !='1' ):
      # ispaljujemo izuzetak
      raise ValueError('Nije uneta ni 0 ni 1')
    # ako smo stigli do kraja ulaza, zaustavljamo petlju
  except EOFError:
    break
  # ako karakter nije iz alfabeta, prekidamo program i
  # prijavljujemo gresku
  except ValueError as e:
    print (e)
    sys.exit()

  # ako je sve u redu, prelazimo u novo stanje
  stanje = prelaz[(stanje,c)]
  print ("\t" + stanje)

  # Smemo ovako da menjamo stanja jer je automat potpun
  # i imamo prelaze iz svakog stanja po svakom slovu
  # Da je nepotpun bilo bi neophodna provera da li
  # postoji prelaz po unetom slovu iz trenutnog stanja:
  # if (stanje,c) in prelaz:
  #   stanje = prelaz[(stanje,c)]
  # else:
  #   exit("Ne postoji prelaz iz " + stanje + " po slovu " + c)

# Na kraju, proveravamo da li se automat nalazi u
# zavrsnom stanju
if stanje == zavrsno :
  print ('Rec ima paran broj nula')
else:
  print ('Rec nije prihvacena automatom')
```

Rešenje 4.1: Automat za reči sa parnim brojem 0

Primer 4.2 Konstruisati MDKA koji prepoznaje reči iz jezika definisanog regularnim izrazom `(a|b)*abb`.

Rešenje. Da bismo konstruisali MDKA, prvo je potrebno da konstruišemo NDKA uz

pomoć Tompsonovog ili Gluškovljevog algoritma. Na primer, nakon konstrukcije Tompsonovim algoritmom dobija se sledeći NDKA:



Nakon konstrukcije automata, dobijeni NDKA potrebno je determinizovati i prevesti ga u DKA. Da bismo to uradili, naći ćemo epsilon zatvorenje početnog stanja. Epsilon zatvorenje stanja 0 predstavlja skup stanja do kojih možemo da dođemo samo po ε prelazima polazeći od stanja 0. Proglašavamo ga novim stanjem i dajemo mu ime A. Dobijamo sledeće stanje:

$$\varepsilon(0) = \{0,1,2,4,7\} = A$$

Sada je potrebno da nađemo epsilion zatvorenja stanja A po prelazima po svim simbolima:

$$\varepsilon(A,a) = \{3,6,7,1,2,4,8\} = B$$
$$\varepsilon(A,b) = \{5,6,7,1,2,4\} = C$$

U nastavku, potrebno je da uradimo isto za svako novo stanje koje smo uveli. Postupak zaustavljamo onda kada nađemo prelaze za sva stanja.

$$\varepsilon(B,a) = \{3,6,7,1,2,4,8\} = B$$
$$\varepsilon(B,b) = \{3,6,7,1,2,4,9\} = D$$
$$\varepsilon(C,a) = \{3,6,7,1,2,4,8\} = B$$
$$\varepsilon(C,b) = \{5,6,7,1,2,4\} = C$$
$$\varepsilon(D,a) = \{3,6,7,1,2,4,8\} = B$$
$$\varepsilon(D,b) = \{5,6,7,1,2,4,10\} = E$$
$$\varepsilon(E,a) = \{3,6,7,1,2,4,8\} = B$$
$$\varepsilon(E,b) = \{5,6,7,1,2,4\} = C$$

Kada smo našli ε zatvorenja svih stanja, preveli smo NDKA u DKA predstavljen sledećom tablicom:

stanje \ simbol	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Nakon što smo zapisali funkciju prelaza pomoću tablice, potrebno je da odredimo skup početnih i skup završnih stanja. Početna stanja su ona koja u sebi sadrže polazno početno

stanje, tj. stanje 0. Jedino takvo stanje je stanje A. Završna stanja su ona koja u sebi sadrže polazno završno stanje, tj. stanje 10. Jedino takvo stanje je stanje E. Bitno je primetiti da dobijeni automat jeste DKA, ali nije nužno MDKA koji je neophodan da bi se implementirao konačni automat koji prepoznaje jezik definisan regularnim izrazom `(a|b)*abb`.

Na kraju, neophodno je minimalizovati polazni automat. Da bismo to uradili potrebno je da podelimo stanja na dva podskupa. Skup nezavršnih stanja $N = \{A,B,C,D\}$ i skup završnih stanja $F = \{E\}$. Navedene skupove posmatramo kao klase ekvivalencije i potrebno je da proverimo da li se sva stanja u skupovima ponašaju na isti način za svaki simbol iz alfabeta, tj. da proverimo da li skup treba razbiti na jedan ili više podskupova. Postupak se završava onda kada ne dobijemo nijednu novu particiju skupova. Jednočlane skupove ne uključujemo u razmatranje. U našem slučaju, treba da proverimo samo skup N. Proveravamo sva stanja i sva slova iz alfabeta i dobijamo sledeći niz preslikavanja:

$$(A,a) \to B \in N$$
$$(A,b) \to C \in N$$
$$(B,a) \to B \in N$$
$$(B,b) \to D \in N$$
$$(C,a) \to B \in N$$
$$(C,b) \to C \in N$$
$$(D,a) \to B \in N$$
$$(D,b) \to E \in F$$

Na osnovu prethodnog razmatranja, vidimo da se stanje D ne ponaša isto kao i stanja A,B,C i da ne pripada istoj klasi ekvivalencije. Zbog toga, potrebno je da razbijemo skup N na dva podskupa: $N_1 = \{A,B,C\}, N_2 = \{D\}, F = E$ S obzirom da skup N_1 nije jednočlani, potrebno je da ponovimo isti postupak:

$$(A,a) \to B \in N_1$$
$$(A,b) \to C \in N_1$$
$$(B,a) \to B \in N_1$$
$$(B,b) \to D \in N_2$$
$$(C,a) \to B \in N_1$$
$$(C,b) \to C \in N_1$$

Na osnovu prethodnog razmatranja, vidimo da se stanje B ne ponaša isto kao i stanja $\{A,C\}$ i da ne pripada istoj klasi ekvivalencije. Zbog toga, potrebno je da razbijemo skup N_1 na dva podskupa: $N_1 1 = \{A,C\}, N_1 2 = \{B\}, N_2 = \{D\}, F = E$ S obzirom da skup $N_1 1$ nije jednočlani, potrebno je da ponovimo isti postupak:

$$(A,a) \to B \in N_1 2$$
$$(A,b) \to C \in N_1 1$$
$$(C,a) \to B \in N_1 2$$
$$(C,b) \to C \in N_1 1$$

Stanja A i C se ponašaju isto po svim simbolima, pa nema potrebe da razbijemo skup $N_1 1$ na podskupove. S obzirom da nismo dobili nijedan novi podskup, potrebno je da spojimo

stanja A i C u jedno stanje da bi dobijeni automat postao MDKA, tj. predstavljeno tablicom:

stanje \ simbol	a	b
AC	B	AC
B	B	D
D	B	E
E	B	AC

U dobijenom automatu potrebno je odrediti početna i završna stanja. Početna stanja su ona koja u sebi sadrže stanje A, a završna su ona koja u sebi sadrže stanje E. U našem slučaju početno stanje će biti stanje AC, a završno stanje će biti stanje E.

Implementacija automata data je u sledećem kodu.

```
#!/usr/bin/python3
import sys

# proglasavamo pocetno i zavrsno stanje
stanje  = 'AC'
zavrsno = 'E'

# funkciju prelaza predstavljamo kao rečnik
prelaz ={
          ('AC','a'): 'B', ('AC','b'): 'AC',
          ('B' ,'a'): 'B', ('B' ,'b'): 'D'
          ('D' ,'a'): 'B', ('D' ,'b'): 'E'
          ('E' ,'a'): 'B', ('E' ,'b'): 'AC'
        }

# u petlji
while True:
  # pokusavamo da ucitamo karakter
  try:
    c = input('Unesite a ili b: ')
    # ako procitani karakter nije iz alfabeta
    if (c != 'a' and c !='b' ):
      # ispaljujemo izuzetak
      raise ValueError('Nije uneto ni a ni b')
  # ako smo stigli do kraja ulaza, zaustavljamo petlju
  except EOFError:
    break
  # ako karakter nije iz alfabeta, prekidamo program i
  # prijavljujemo gresku
  except ValueError as e:
    print (e)
    sys.exit()

  # ako je sve u redu, prelazimo u novo stanje
```

```
35    stanje = prelaz[(stanje,c)]
      print ("\t" + stanje)
37
  # na kraju proveravamo da li se automat nalazi u
39 # zavrsnom stanju
  if stanje == zavrsno :
41    print ('Rec je iz jezika')
  else:
43    print ('Rec nije prihvacena automatom')
```

Rešenje 4.2: Automat za primer 4.2

4.2 Zadaci za vežbu

Zadatak 4.1 Konstruisati MDKA koji prepoznaje reči iz jezika definisanog regularnim izrazom `(ab?)+(b?a+b)*`. ■

Rešenje. Nakon primene algoritma Gluškova da dobijemo automat, potom postupka determinizacije i minimizacije dobija se MDKA implementiran u sledećem kodu.

```
  # U mapi se cuva matrica prelaza, pri cemu se za kljuc mape
      uzima
2 # stanje, a vrednost je ponovo mapa oblika slovo->novo_stanje


4
  prelaz = { ('A','a'): 'BC',
6            ('BC','a'): 'BC', ('BC','b'): 'DE',
             ('DE','a'): 'BC', ('DE','b'): 'F',
8            ('F' ,'a'): 'G' ,
             ('G' ,'a'): 'G' , ('G' ,'b'): 'H',
10           ('H' ,'a'): 'G' , ('H' ,'b'): 'F'
           }
12
  # Pocetno stanje
14 pocetno = 'A'
  # Lista zavrsnih stanja
16 zavrsna = ('BC','DE','H')

18 stanje = pocetno

20 rec = input('Unesite celu rec:')

22 for c in rec:
     try:
24     # Proveravamo da li je uneto slovo azbuke
       if c != 'a' and c != 'b':
26       raise ValueError('Podrzana slova su a, b')
```

```
28      # Ukoliko imamo prelaz po unetom slovu iz trenutnog stanja
        automata
        # automat prelazi u novo stanje, inace imamo gresku
30      if (stanje,c) in prelaz:
          stanje = prelaz[(stanje,c)]
32      else:
          exit('Rec ne moze biti prihvacena automatom')
34
        # Ispisujemo trenutno stanje automata
36      print('\tStanje: ', stanje)
    except EOFError:
38      break
    except ValueError as e:
40      print(e)
        sys.exit()
42
# Ukoliko je automat u zavrsnom stanju na kraju ulaza, rec je
    prihvacena
44 if stanje in zavrsna:
    print('Automat prihvatio rec')
46 else:
    print('Rec nije prihvacena')
```

Rešenje 4.3: Implementacija MDKA za datim regularni izraz

Zadatak 4.2 Konstruisati MDKA koji prepoznaje reči iz jezika definisanog regularnim izrazom `((ab*)?c(b|ab*c)*c)*(a(b|cb*a)*|(ab*)?c(b|ab*c)*(ab*)?)`. ■

Rešenje. Nakon konstrukcije složenog automata od čak 23 stanja, nakon minimizacije dobija se jednostavan automat za implementaciju od 3 stanja.

```
1  # Pocetno stanje
   pocetno = 0
3  # Lista zavrsnih stanja
   zavrsna = (1,2)
5
   # Matrica prelaza automata
7  prelaz = { (0,'a'):1, (0,'c'):2,
              (1,'b'):1, (1,'c'):2,
9             (2,'a'):1, (2,'b'):2, (2,'c'):0
           }
11
   # ako zelimo da imamo celu unetu rec
13 rec = ""

15 stanje = pocetno
   # Ucitavamo karaktere sa ulaza
17 while True:
     try:
```

```
19      c = input()
        # Proveravamo da li je uneto slovo azbuke
21      if c != 'a' and c != 'b' and c !='c' :
          raise ValueError('Podrzana slova su a b c')
23
        # Ukoliko imamo prelaz po unetom slovu iz trenutnog
25      # stanja automata automat prelazi u novo stanje,
        # inace imamo gresku.
27      if (stanje,c) in prelaz:
          stanje = prelaz[(stanje, c)]
29      else:
          sys.exit('Rec ne moze biti prihvacena automatom')
31
        # Nadovezujemo novo slovo na kraj do sad procitanje reci
33      rec += c
        # Ispisujemo trenutno stanje automata
35      print('\tStanje: ', stanje)
    except EOFError:
37      break
    except ValueError as e:
39      print(e)
        sys.exit()
41
# Ukoliko je automat u zavrsnom stanju na kraju ulaza, rec je
    prihvacena
43 if stanje in zavrsna:
    print('Automat prihvatio rec ' + rec)
45 else:
    print('Rec nije prihvacena')
```

Rešenje 4.4: Implementacija MDKA za datim regularni izraz

5. Leksička analiza

Leksička analiza je prva faza kompilacije i predstavlja pretvaranje niza karaktera (npr. izvorni kod, veb stranica) u niz tokena. Token je niz karaktera koji u gramatici jezika predstavlja jedinstvenu celinu. Drugim rečima, token je string koji ima dodeljeno značenje pomoću koga se identifikuje u gramatici jezika. Najčešće, tokeni se struktuiraju kao uređeni parovi koji čine naziv tokena i opciono vrednost tokena. Naziv tokena predstavlja jednu leksičku kategoriju. Česti nazivi tokena su: identifikator, separator, terminator, ključna reč, operator, literal itd.

Program koji izvršava leksičku analizu se zove leksički analizator, skraćeno *lekser*. Ulaz za lekser je neki niz karaktera, na primer ranije pomenuti izvorni kod nekog programa. Rezultat izvršavanja leksera je niz tokena koji se može slati dalje parseru na sintaksnu analizu. Leksički analizatori se najčešće implementiraju uz pomoć determinističkih konačnih automata.

Prilikom dizajniranja leksera, potrebno je unapred definisati skup tokena i svaki token opisati regularnim izrazom. Na ovaj način omogućava se automatska detekcija leksičkih grešaka. Greška je sve ono što nije u polaznom skupu tokena. Ovakvim pristupom moguće je otkriti tačnu poziciju u ulaznom nizu karaktera na kojoj se desila leksička greška i to prijaviti korisniku.

S obzirom da broj tokena može biti vrlo veliki, kao i složenost samih regularnih izraza, dizajniranje DKA koji prepoznaje zadati jezik predstavlja popriličan izazov. Upravo zbog toga postoje alati koji mogu automatski da generišu DKA koji prepoznaje zadati jezik. Jedan takav alat koji generiše C kod leksičkog analazatora naziva se *flex*.

5.1 Generator leksičkih analizatora – *flex*

Flex je računarski program koji generiše leksičke analizatore. Dostupan je besplatno na Linux operativnim sistemima. U slučaju da koristimo neku od Ubuntu distribucija, flex

se instalira uz pomoć sledeće dve komande:

```
sudo apt-get update sudo apt-get install flex
```

Komande je neophodno da izvršimo ovim redom da bismo instalirali poslednju dostupnu verziju paketa za našu verziju operativnog sistema.

Da bismo generisali leksički analizator za naš problem, potrebno je prvo da napišemo njegovu specifikaciju u sintaksi *flex*-a. Nakon toga, uz pomoć *flex*-a možemo da generišemo C kod programa koji predstavlja naš leksički analizator. Specifikacija je podeljena na tri dela razdvojena sa parom karaktera %%:

1. Segment definicija
2. Segment akcija
3. Segment korisničkog koda

```
// segment definicija
%%
// segment akcija
%%
// segment korisničkog koda
```

Bitno je primetiti da %% mora da bude na početku reda i da mora da bude jedini sadržaj tog reda. Između dva procenta ne sme da se nađe razmak ili bilo kakav drugi znak. Skript mora da bude podeljen na ova tri segmenta, inače ne predstavlja validan ulaz za *flex*.

Segment definicija sadrži komande *flex*-u i C-kod koji se direktno kopira u izvršni program i opciono imenovane regularne izraze. C kod mora biti ograđen parovima karaktera %{ i %}. Između % i { ili } ne sme da se nađe nijedan znak, jer tada naredba kojom se definiše C blok neće biti validna.

Segment akcija u svakoj liniji sadrži parove oblika `regularni_izraz akcija`. Opsta struktura akcije je oblika

```
regularni_izraz {  /* c kod koji predstavlja akciju koja treba da se
                      desi nakon uparivanja regularnog izraza */
                }
```

Regularan izraz mora da se od samog početka linije. Akcija pridruzena regularnom izrazu mora da počne u istom redu u kojem je i regularni izraz i može da se prostire u više redova.

Segment korisničkog koda sadrži C-kod koji se direktno kopira na sam kraj generisanog C koda leksičkog analizatora. Segment korisničkog koda je opcioni, tj. može biti prazan.

Kreiranje leksičkog analizatora se odvija u dva koraka:

1. Pisanje flex skripta na osnovu koga se generiše C kod koji predstavlja leksički analizator.
2. Prevođenje dobijenog C koda do izvršnog koda koji predstavlja leksički analizator.

Pretpostavimo da je opis leksičkog analizatora zapisan u datoteci koji se zove `lexer.l`. Da bismo generisali C kod leksičkog analizatora, potrebno je da uz pomoć *flex*-a kreiramo `lex.yy.c` datoteku. To postižemo sledećom komandom:

```
flex lexer.l
```

Nakon što se komanda izvrši, biće generisana datoteka `lex.yy.c` koju treba prevesti *gcc* kompajlerom da bismo dobili izvršnu verziju:

```
gcc lex.yy.c -o lexer
```

U okviru *gcc* poziva, preimenovali smo rezultujuću izvršnu verziju u `lexer`. Da bismo izvršili leksičku analizu, dovoljno je samo da pokrenemo izvršnu verziju programa, `lexer`. Radi jednostavnijeg i fleksibilnijeg održavanja koda, objedinićemo ove pozive u *Makefile*:

```
lexer: lex.yy.c
    gcc lex.yy.c -o lexer

lex.yy.c: lexer.l
    flex lexer.l
```

Nakon toga možemo prevoditi leksički analizator jednostavnim pozivom komande `make` iz konzole.

U nastavku teksta biće prikazani razni primeri upotrebe alata *flex* u svrhu leksičke analize.

5.2 Primeri leksičke analize

Zadatak 5.1 Napisati program koji broji linije i karaktere sa standardnog ulaza. Rezultat analize ispisuje na standaradni izlaz. ■

Rešenje.

```
/*  Prvi deo datoteke predstavlja segment definicija.
 *  Ovo je obavezno da bi se pri nailasku na EOF zaustavilo
 *  dalje citanje neke druge ulazne datoteke.
 */
%option noyywrap

%{
  /* Ovaj deo koda se doslovno prenosi u lex.yy.c na sam
   * pocetak. Ovde mozemo ukljucivati zaglavlja koja zelimo
   * da koristimo, i/ili globalne promenljive koje
   * predstavljaju brojace.
   */
  int num_lines = 0;
  int num_chars = 0;
%}

/* U ovom delu lex datoteke se stavljaju neke regularne
 * definicije koje nam koriste da uprostimo regularne izraze
 * koje koristimo u drugom delu datoteke.
 */

/* Sledeci deo, tj. segment akcija je glavni deo datoteke i u
 * njemu se definisu regularni izrazi koje prepoznajemo zajedno
 * sa akcijama koje zelimo da se dese kad se pronadje tekst
```

```
 * koji se uparuje sa regularnim izrazom sa leve strane akcije.
 * U njemu ne smemo da imamo C-ovske komentare na pocetku
 * linije, dok u akcijama smemo.
 */
%%

\n {num_lines++; num_chars++;}
.  {num_chars++;}

%%

/* Treci deo datoteke, tj. segment korisnickog koda se doslovno
 * prenosi u lex.yy.c na sami kraj. U njemu cemo napisati main
 * funkciju.
 */

int main () {
  /* U kojoj samo pozivamo funkciju yylex() */
  yylex();

  /* Ispisujemo izlaz */
  printf("Ukupan broj karaktera je %d, a broj redova je %d.\n",
   num_chars, num_lines);
  return 0;
}
```

Rešenje 5.1: Specifikacija leksičkog analizatora koji broji linije i karaktere sa standardnog ulaza.

Zadatak 5.2 Napisati program koji proverava da li su zagrade ispravno uparene. ■

Rešenje.

```
%option noyywrap
%{
  /* Postavljamo broj otvorenih zagrada na 0 */
  int broj_otvorenih = 0;
%}

%%
"{" { /* Na standardni izlaz stampamo ono sto je prepoznato
       * regularnim izrazom. Prepoznati tekst se nalazi u
       * promenljivoj yytext.
       */
      printf("%s", yytext);
      broj_otvorenih++;
    }
"}" { /* ECHO je isto sto i printf("%s", yytext); */
      ECHO;
```

```
      /* Ne smemo zatvarati neotvorenu zagradu */
      if (broj_otvorenih==0) {
        fprintf(stderr, "Zagrade nisu korektno uparene!\n");
        /* Ovaj return se odnosi na funkciju yylex(),
         * tj. prekidamo njeno izvrsavanje.
         */
        return(-1);
      }
      else broj_otvorenih--;
    }
.   { /* Sve ostale karaktere samo ispisujemo na izlaz. */
      ECHO;
    }
\n  { ECHO; }
%%

/* Treci deo datoteke se doslovno prenosi u lex.yy.c na sam
 * kraj. U njemu cemo napisati main funkciju.
 */

int main () {
  /* Da bismo pokrenuli leksicku analizu potrebno je da pozovemo
   * funkciju yylex()
   */

  /* Ako nam yylex() vrati -1 imamo nekorektno uparene zagrade.
   */
  if (yylex() == -1)
    exit(EXIT_FAILURE);

  /* Ispitujemo brojac i stampamo poruku korisniku. */
  if (broj_otvorenih == 0)
    printf("Zagrade su korektno uparene.\n");
  else
    printf("Nekorektno uparene zagrade!\n");

  return 0;
}
```

Rešenje 5.2: Specifikacija leksičkog analizatora koji proverava uparenost zagrada.

Zadatak 5.3 Napisati program koji čisti paskal datoteku od komentara.

■

Rešenje. Primer ulazne datoteke bi mogao biti sledeći:

```
(*Ovo je komentar*) Izmedju (**** *)
(* ***** *****)
```

```
(**********)
(*********)

(**)
begin
{ jednolinijski komentar }
Prvi
{Ovo je
glavni blok}  Drugi { I
jos jedan komentar}
Treci
(* **)
end
```

```
%option noyywrap
%option nounput
%option noinput

%%
"{"[^}]*"}"     {
  /* Paskal komentari pocinju sa { a zavrsavaju se sa }
    * Kada prepoznamo neki paskal komentar, stavljamo praznu
    * akciju, jer ne zelimo da se ista desi, tj. ignorisemo ih.
    * Zagrade { i } imaju specijalno znacenje u flex skriptu,
    * pa smo morali da ih navedemo pod navodnicima
    */
  }
"(*"([^*]|"*"+[^)])*\*+")"      {
  /* Paskal komentari mogu da budu i u ovom obliku (* rec *),
    * pa i taj slucaj moramo da obradimo.
    * Kako su i zagrade ( i ), i * specijalni simboli i oni
    * moraju da se navedu pod navodnicima ili sa \ ispred
    * ispred svakog specijalnog karaktera.
    */
  }
.   {
  /* Sve sto nije komentar, prepisujemo nepromenjeno
   * na standardni izlaz.
   */
  ECHO;
  }
%%

/* Treci deo datoteke se doslovno prenosi u lec.yy.c na sam
 * kraj. U njemu cemo napisati main funkciju
 */
int main () {
  /* U kojoj samo pozivamo funkciju yylex() */
  yylex();
```

```
  return 0;
}
```

Rešenje 5.3: Specifikacija leksičkog analizatora koji uklanja komentare u paskal datoteci.

Zadatak 5.4 Napisati program koji čisti C datoteku od komentara. Pored toga, program treba da prebroji linije u originalnoj C datoteci. ■

Rešenje. Primer ulazne datoteke bi mogao biti sledeći:

```
#include <stdio.h>
/*** Ovo je test primer ***/
int main() {
  /* I ovo je komentar */
  printf{"Proba\n"};
  /* Komentar u dva reda ***
  i dalje je komentar */
  return 0;
/*   ab
  abbbb
  aba */  /* **/
  int *Prvi;
  /* * */
  float Drugi;
  /* *** */
  double Treci;
  return 0;
}
```

```
%option noyywrap

/* U dosadesnjem pristupu, kompletan tekst smo posmatrali kao
 * jednu klasu ekvivalencije i trudili smo se da sa bar jednim
 * regularnim izrazom pokrijemo ono sto se trazi od nas. U
 * zavisnosti od slozenosti problema koji resavamo ovakav
 * pristup ponekad nije dobar. Nekada je lakse podeliti tekst
 * na disjunktne celine i tako ga posmatrati. U ovom slucaju
 * dve celine koje imamo su komentar i sve ono sto nije
 * komentar.
 * Da bismo definisali celine na ovaj nacin potrebno je da
 * proglasimo stanja u kojima se leksicki analizator moze naci.
 * Podrazumevano, leksicki analizator je u stanju INITIAL, pa
 * treba da dodamo samo jedno stanje kojim cemo opisati da
 * se trenutno nalazimo u stanju komentar. To stanje cemo
 * nazvati comment. Sintaksa kojom se to postize je sledeca:
 */
%x comment
```

```
%{
  /* globalna promenljiva koja broji linije */
  int broj_linija = 0;
%}

%%
"/*"  {
        /* Kada se naidje na /*, prelazimo u stanje comment */
        BEGIN(comment);
      }
<comment>[^*\n]*    {
        /* U stanju comment ignorisemo sve osim \n i *, akcija
         * nam je prazna, jer ignorisemo i sadrzaj komentara.
         */
      }
<comment>"*"*  {
        /*  u komentarima ignorisemo * iza kojih nije / i \n */
      }
<comment>\n    {
        /* Ako naidjemo na novi red, povecavamo broj linija */
        ++broj_linija;
      }
<comment>\*+"/" {
        /* Kada naidjemo na kraj komentara, vracamo se u
         * pocetno stanje
         */
        BEGIN(INITIAL);
      }
\n    {
        /* Ako naidjemo na nov red van komentara,
         * povecavamo broj redova, i stampamo nov red
         */
        ++broj_linija;
        ECHO;
      }
.     {
        /* Ostale karaktere prepisujemo na izlaz,
         * Regularni izraz i akciju mozemo da izostavimo, jer
         * flex podrazumevano prepisuje neuparene karektere.
         */
      }
%%

int main () {
  /* Pozivamo leksicki analizator */
  yylex();

  /* Stampamo broj linija */
  printf("Broj linija je: %d\n", broj_linija);
```

```
70    return 0;
}
```

Rešenje 5.4: Specifikacija leksičkog analizatora

Dosadašnji primeri vrše transformaciju ulaznog niza karaktera uz pomoć leksičkih analizatora, što je jedan od primera upotrebe leksičke analize. U konstrukciji interpretera i kompilatora, češća primena leksičkih analizatora je tokenizacija ulaza. Preciznije, leksički analizator skenira ulazni niz karaktera i kada uspe da prepozna neki token, tada se zaustavlja i saopštava glavnom programu koji je token našao na ulazu. Nakon toga, glavni program treba da obradi token i da eventualno ponovo pokrene tokenizaciju ulaza ako za time ima potrebe. Leksički analizator nastavlja skeniranje ulaza od poslednjeg pročitanog karaktera i prijavljuje sledeći token glavnom programu. Ovaj postupak se najčešće ponavlja sve do kraja ulaza. Ovakav način upotrebe leksičkog analizatora se naziva leksička analiza na zahtev (*eng. on-demand*). U nastavku teksta, biće prikazano više primera leksičkog analiziranja ulaznog niza karaktera na zahtev.

Zadatak 5.5 Napisati leksički analizator koji u tekstu pronalazi celobrojne i realne konstante i vraća različite tokene. Nakon što se prepozna neki token, potrebno je saopštiti glavnom programu o kom tokenu se radi. Glavni program prijavljuje korisniku koji token je prepoznat i šta je prepoznato. Na kraju rada, glavni program treba da prikaže korisniku koliko kojih tokena je prepoznato. ■

Rešenje. Test datoteka može biti sledeća:

```
#include <stdio.h>
2
int main () {
4   int a = -56;
    double b = +34.56e-12;
6   return 0;
}
```

```
1 %option noyywrap

3 %{
    /* Vrednost 0 je rezervisana za EOF i ne smemo je koristiti
5    * Ovo mozemo izvuci u neko zaglavlje koje bismo ovde samo
     ukljucili
     */
7   #define F_CONST 1
    #define I_CONST 2
9 %}

11 /* Regularna definicija cifre */
   DIGIT [0-9]
13

   %%
```

```
15
[+\-]?{DIGIT}+  {
17                  /* Celobrojne konstante */
                    return I_CONST;
19                }

21 [+\-]?{DIGIT}+\.{DIGIT}*([Ee][+\-]?{DIGIT}+)?   {
                    /* Realne konstante */
23                  return F_CONST;
                  }
25 .              {
                    /* Prazna akcija da nam se nista ne bi
   ispisivalo */
27                }
\n                { /* Ignorisemo nove redove */ }
29

31 %%

33 int main () {
  /* Definisemo brojace prepoznatih konstanti */
35  int celi_brojevi = 0;
  int realni_brojevi = 0;
37
  /* Tokeni se predstavljaju celobrojnim konstantama */
39  int token;

41  /* Pozivamo funkciju sve dok ima tokena na ulazu. Kada
    * leksicki analizator dodje do EOF, vraca vrednost 0.
43   */
  while ( (token = yylex())) {
45    /* Prijavljujemo koji je token pronadjen, brojimo ih
       * i stampamo odgovarajucu leksemu
47     */
      switch(token) {
49      case I_CONST:
          celi_brojevi++;
51        printf("Pronadjena celobrojna konstanta: %s\n", yytext);
          break;
53      case F_CONST:
          realni_brojevi++;
55        printf("Pronadjena realna konstanta: %s\n", yytext);
          break;
57    }
  }
59
  /* Prikazujemo koliko kojih konstanti je prepoznato */
61  printf("Celih brojeva: %d\n", celi_brojevi);
  printf("Realnih brojeva: %d\n", realni_brojevi);
```

```
63
   return 0;
65 }
```

Rešenje 5.5: Specifikacija leksičkog analizatora na zahtev.

Zadatak 5.6 Naredni program je modifikacija prethodnog primera. U ovom primeru korisnik kroz argumente komandne linije može da prosledi putanju do ulazne i izlazne datoteke. ■

Rešenje.

```
1  %option noyywrap
2
3  %{
4    #define F_CONST 1
5    #define I_CONST 2
6
7    int celi_brojevi = 0;
8    int realni_brojevi = 0;
9  %}
10
11 DIGIT[0 - 9]
12
13 %%
   [+\-]?{DIGIT}+  {
15                   return I_CONST;
                   }
17
   [+\-]?{DIGIT}+\.{DIGIT}*([Ee][+\-]?{DIGIT}+)? {
19                   return F_CONST;
                   }
21 .      { }
   \n     { }
23 %%
24
25 int main(int argc, char *argv[])
   {
27   int token;
28
29   /* Ako su zadati argumenti komandne linije iz kojih se
30    * cita ulaz, odnosno pise izlaz
31    */
32   if (argc > 1 && (yyin = fopen(argv[1], "r")) != NULL)
33     ;
34   else
35     yyin = stdin;
36   if (argc > 2 && (yyout = fopen(argv[2], "w")) != NULL)
37     ;
```

```
  else
    yyout = stdout;

  while (token = yylex()) {
    switch(token) {
      case I_CONST:
        celi_brojevi++;
        printf("Pronadjena celobrojna konstanta: %s\n", yytext);
        break;
      case F_CONST:
        realni_brojevi++;
        printf("Pronadjena realna konstanta: %s\n", yytext);
        break;
    }
  }

  printf("Celih brojeva: %d\n", celi_brojevi);
  printf("Realnih brojeva: %d\n", realni_brojevi);

  return 0;
}
```

Rešenje 5.6: Specifikacija leksičkog analizatora na zahtev.

Zadatak 5.7 Program prepisuje sadržaj datoteke na standardni izlaz tako što sve rimske brojeve zamenjuje njihovim dekadnim zapisima. ■

Rešenje.

```
%option noyywrap
%option nounput
%option noinput

%{
  /* promenljiva sadrzi dekadnu vrednost rimskog broja */
  int vrednost = 0;

  /* Promenljiva ce nam sadrzati prepoynat rimski broja.
   * Zbog situacija kada se u deo reci prepozna kao rimski
   * broj, na primer u recima Matematicki i Veliko, da bismo
   * ih mogli ipak ispisati na ekran.
   */
  char zapis_broja[1000];
  zapis_broja[0] = '\0';
%}

/* Regularna definicija cifre */
hiljade M+
stotine C+|CD|DC*|CM
```

```
desetice X+|XL|LX*|XC
jedinice I+|IV|VI*|IX

%%
{hiljade} {
      /* duzina prepoznate lekseme se nalazi u promenljivoj
       * yyleng, preciznije
       * yyleng = strlen(yytext);
       */
      vrednost += 1000*yyleng;
      strcat(zapis_broja, yytext);
    }
{stotine} {
      if (strcmp(yytext, "CM") == 0)
        vrednost += 900;
      else if (strcmp(yytext, "CD") == 0)
        vrednost += 400;
      else
        if( yytext[0] == 'D') {
          vrednost += 500;
        if (yyleng <=4){  //DCCC
          vrednost += 100* (yyleng - 1);
        }
        else {
          fprintf(stderr,
                  "\nNeispravan rimski broj %s\n",yytext);
          exit(EXIT_FAILURE);
        }
      }
      else
        if (yyleng <= 3){
          vrednost += 100* yyleng;
        }
        else {
          fprintf(stderr,
                  "\nNeispravan rimski broj %s\n",yytext);
          exit(EXIT_FAILURE);
        }

      strcat(zapis_broja, yytext);
    }

{desetice} {
      if (strcmp(yytext, "XC") == 0)
        vrednost += 90;
      else
        if (strcmp(yytext, "XL") == 0)
          vrednost += 40;
      else
```

```
70          if( yytext[0] == 'L') {
              vrednost += 50;
72          if (yyleng <= 4){
              vrednost += 10* (yyleng - 1);
74          }
            else { //LXXXXX
76            fprintf(stderr,
                    "\nNeispravan rimski broj %s\n",yytext);
78            exit(EXIT_FAILURE);
            }
80        }
          else
82          if (yyleng <= 3){
              vrednost += 10* yyleng;
84          }
            else {
86            fprintf(stderr,
                    "\nNeispravan rimski broj %s\n",yytext);
88            exit(EXIT_FAILURE);
            }
90
          strcat(zapis_broja, yytext);
92      }

94  {jedinice} {
          if (strcmp(yytext, "IX") == 0)
96          vrednost += 9;
          else if (strcmp(yytext, "IV") == 0)
98          vrednost += 4;
          else if( yytext[0] == 'V') {
100         vrednost += 5;
            if (yyleng <= 4){
102           vrednost += (yyleng - 1);
            }
104         else {  // VIIIII
              fprintf(stderr,
106                   "\nNeispravan rimski broj %s\n",yytext);
              exit(EXIT_FAILURE);
108         }
          }
110       else
            if (yyleng <= 3){
112           vrednost +=  yyleng;
            }
114           else {
              fprintf(stderr,
116                   "\nNeispravan rimski broj %s\n",yytext);
              exit(EXIT_FAILURE);
118         }
```

```
120       strcat(zapis_broja, yytext);
      }
122
[ \n\t.!?;"':]  { /* Karakter koji nije deo reci. */
124       /* Ako imamo procitanu rimski broj prikazujemo
           * dekadnu vrednost broja.
126        */
       if(vrednost > 0){
128         printf("%d", vrednost);
           vrednost = 0;
130         zapis_broja[0]='\0';
       }
132       ECHO;
      }
134 .     {
         /* Prepoznat je neki drugi karakter,
136         * karakter reci, koji ne cini zapis rimskog broja.
           * Ukoliko smo prepoznali neki rimski broj,
138         * odustajemo od njegove dekadne vrednosti, vec
           * prikazujemo prepoznat zapis rimskog broja.
140         */
       if(vrednost > 0){
142         vrednost = 0;
           printf("%s", zapis_broja);
144         zapis_broja[0]='\0';
       }
146
       ECHO;
148   }
%%
150
int main() {
152   yylex();

154   return 0;
}
```

Rešenje 5.7: Specifikacija leksičkog analizatora koji zamenjuje rimske brojeve njihovim dekadnim zapisima.

Zadatak 5.8 Napisati leksički analizator za mini paskal. ■

Rešenje. Test primer može biti sledeći:

```
1 var
    i, j : integer;
3 begin
    i:=3345 + 5;
```

```
  if (i>0) then j:=7;
end.
```

```
%option noyywrap

%{
  /* Ukljucujmo zagalavlje zbog funkcije atof() */
  #include <stdlib.h>

  /* definisemo tokene. 0 je rezervisana za EOF */
  #define KLJUCNA_REC 1
  #define ID  2
  #define I_CONST 3
  #define F_CONST 4
  #define AOP 5
  #define ROP 6
  #define INTERPUNKCIJA   7
  #define DODELA  8
  #define Z   9
  #define NN  10
%}

/* Regularne definicije koje nam olaksavaju zapis regularnih
   izraza */
DIGIT   [0-9]
ID      [a-z][a-z0-9]*

%%
{DIGIT}+            { return I_CONST; }
{DIGIT}+\.{DIGIT}*  { return F_CONST; }
if|then|begin|end|var|function|integer  { return KLJUCNA_REC; }
{ID}            { return ID; }
[+*/-]          { return AOP; }
[<>=]|"<="|">=" { return ROP; }
:=              { return DODELA; }
[.,:;]          { return INTERPUNKCIJA; }
"("|")"         { return Z; }
"{"[^}\n]*"}"   {
            /* Ignorisemo komentare. Dozvoljeni su samo
             * jednolinijski komentari radi jednostavnosti
             */
          }
[ \t\n]   { /* I beline ignorisemo */ }
.         {
            /* Sve ostalo su neprepoznati karakteri */
            return NN;
          }

%%
```

```
46
int main(int argc, char* argv[]) {
48   int token;
   /* Ako imamo argumente komandne linije */
50   if ( argc > 1){
     if (( yyin = fopen(argv[1],"r"))==NULL )
52       yyin = stdin;
   }
54   else
     yyin = stdin;
56
   /* Vrsimo leksicku analizu   */
58   while ((token = yylex())!=0) {
     switch(token) {
60       case I_CONST:
             printf("Ceo broj: %d\n",atoi(yytext));
62           break;
         case F_CONST:
64           printf("Realan broj: %f\n",atof(yytext));
             break;
66       case ID:
             printf("Identifikator: %s\n",yytext);
68           break;
         case KLJUCNA_REC:
70           printf("Kljucna rec: %s\n",yytext);
             break;
72       case AOP:
             printf("Operator: %s\n",yytext);
74           break;
         case ROP:
76           printf("Relacioni operator: %s\n",yytext);
             break;
78       case DODELA:
             printf("Operator dodele: %s\n",yytext);
80           break;
         case INTERPUNKCIJA:
82           printf("Ingerpunkcija: %s\n",yytext);
             break;
84       case Z:
             printf("Zagrada: %s\n",yytext);
86           break;
         case NN:
88           fprintf(stderr, "Neprepoznata leksema: %s\n",yytext);
             exit(EXIT_FAILURE);
90           break;
     }
92   }

94   return 0;
```

```
}
```

Rešenje 5.8: Specifikacija leksičkog analizatora za mini paskal.

U ovom zadatku treba primetiti kako se definiše prioritet prepoznavanja tokena u slučaju konflikata. Leksički posmatrano, ključne reči u programu nisu ništa drugo nego kombinacije slova, što je podskup definicije za identifikatore. Na primer, reč `var` je rezervisana reč u *Paskal*-u i određuje deo koda u kome se definišu promenljive. Leksički gledano, reč `var` može da bude i identifikator, jer se kao kombinacija slova uklapa u definiciju identifikatora. Jedino zbog čega reč `var` nije identifikator jeste semantika jezika.

Potrebno je da na neki način razrešimo ovaj konflikt i da prilikom definisanja leksičkog analizatora uzmemo u obzir semantiku jezika. Da bismo naterali leksički analizator da ne prepoznaje ključne reči kao identifikatore, potrebno je da definicije ključnih reči navedemo prve u listi regularnih izraza, tj. da definišemo prioritet poklapanja. Prioritet poklapanja se određuje redosledom navođenja pravila u opisu leksičkog analizatora. Na ovaj način, ključne reči će uvek biti prepoznate kao ključne reči, a svi oni identifikatori koji nisu ključne reči biće tako i prepoznati. Definisanjem prioriteta i uzimanjem u obzir semantike jezika prilikom opisivanja leksičkog analizatora, korisniku je onemogućeno da kreira identifikatore sa rezervisanim imenima.

Zadatak 5.9 Napisati leksicki analizator koji na osnovu ulazne C datoteke kreira lepo formatiranu HTML stranicu. ■

Rešenje. Test primer može biti sledeći:

```
#include <stdio.h>
#define P 0
#define N 1

  /* Glavni program */
int main()
{
  char c;
  int stanje = P;
  int prelaz[2][2] = { {N, P}, {P, N} };

  while ((c = getchar()) != EOF && c != '\n') {
    if (c != '0' && c != '1') {
      printf("Greska:%c.\n", c);
      return -1;
    }
    stanje = prelaz[stanje][c - '0'];
  }

  if (stanje == P)
    printf("Prepoznata parna rec.\n\\");
  else
    printf("Nije prepoznata parna rec.\n");
  return 0;
```

```
}
```

```
%option noyywrap
%option noinput
%option nounput

%{
  /* Pomocne funkcije koje obradjuju
   * specijalne karaktere u HTML */
  void stampaj_karakter(char c);
  void stampaj_nisku(char *s);
%}

/* Regularne definicije  */
DIGIT  [0-9]
ID  [a-zA-Z_][a-zA-Z0-9_]*
WHITE  [ \t\n]

%%
{DIGIT}+    {
      /* Ako se prepozna celobrojna konstanta,
       * boji se u plavo */
      fprintf(yyout,
              "<span style=\"color:blue;\">%s</span>", yytext);
    }
{DIGIT}+\.{DIGIT}*  {
      /* Ako se prepozna realna konstanta, boji se u plavo */
      fprintf(yyout,
              "<span style=\"color:blue;\">%s</span>", yytext);
    }
if|else|switch|case|while|for|do|break|continue {
      /* Kljucne reci se isticu podebljanim slovima */
      fprintf(yyout, "<B>%s</B>", yytext);
    }
int|char|double|float|short|unsigned|long|void {
      /* Kljucne reci se isticu podebljanim slovima */
      fprintf(yyout, "<B>%s</B>", yytext);
    }
struct|union|static|extern|typedef|return|register {
      /* Kljucne reci se isticu podebljanim slovima */
      fprintf(yyout, "<B>%s</B>", yytext);
    }
^{WHITE}*#(.|\\\n)*\n  {
      /* Predprocesorske direktive boje se u zeleno */
      fprintf(yyout, "<span style=\"color:green;\">");
      stampaj_nisku(yytext);
      fprintf(yyout, "</span>");
    }
{ID}  {
```

```
      /* Identifikatori boje se u crno */
49    fprintf(yyout,
              "<span style=\"color:black;\">%s</span>", yytext);
51    }
"/*"([^*]|"*"*[^/*])*"*"*"/"  {
53    /* Komentari se boje u sivo */
      fprintf(yyout, "<span style=\"color:grey;\">");
55    stampaj_nisku(yytext);
      fprintf(yyout, "</span>\n");
57    }
\"([^\\\"]|\\.)*\"  {
59    /* Niske se boje u crveno */
      fprintf(yyout, "<span style=\"color:red;\">");
61    stampaj_nisku(yytext);
      fprintf(yyout, "</span>");
63    }
'([^\\']|\\.)' {
65    /* Karakteri se boje u ljubicasto */
      fprintf(yyout , "<span style=\"color:violet;\"> ");
67    stampaj_nisku(yytext);
      fprintf(yyout, "</span>");
69    }
.|\n  {
71    /* Sve ostalo se prepisuje kakvo jeste na yyout */
      ECHO;
73    }
%%
75
/* specijalne karaktere treba prilagoditi, radi ispravnog
77    prikaza u okviru HTML datoteke */
void stampaj_karakter(char c) {
79  if (c=='<')
      fprintf(yyout, "&lt;");
81  else if (c=='>')
      fprintf(yyout,"&gt;");
83  else if (c=='&')
      fprintf(yyout, "&amp;");
85  else
      fprintf(yyout, "%c", c);
87}

89void stampaj_nisku(char *s) {
  int i;
91  for(i=0;s[i]!='\0';i++)
      stampaj_karakter(s[i]);
93}

95int main(int argc, char* argv[]) {
  /* U zavisnosti od argumenata komandne linije menja se
```

```
97        ulaz i izlaz leksickog analizatora */
    if (argc>1) {
99      if ((yyin = fopen(argv[1],"r"))==NULL) {
          fprintf(stderr,
101               "Neuspesno otvaranje datoteke %s.\n",argv[1]);
          exit(EXIT_FAILURE);
103     }
    }
105 else
      yyin = stdin;
107
    if (argc>2) {
109     if ((yyout = fopen(argv[2],"w"))==NULL) {
          fprintf(stderr,
111               "Neuspesno otvaranje datoteke %s.\n",argv[2]);
          exit(EXIT_FAILURE);
113     }
    }
115 else
      yyout = stdout;
117
    /* Stampa se zaglavlje html dokumenta */
119 fprintf(yyout, "<html>\n");
    if (argc>1)
121   fprintf(yyout, "<head><title>%s</title></head>\n", argv[1]);
    fprintf(yyout, "<body>\n<pre>\n");
123
    /* Boji se tekst */
125 yylex();

127 /* Zavrsava se html dokument */
    fprintf(yyout,"</pre>\n</body>\n</html>\n");
129
    /* Zatvaraju se datoteke */
131 fclose(yyin);
    fclose(yyout);
133
    /* Zavrsavamo program */
135 return 0;
}
```

Rešenje 5.9: Specifikacija leksičkog analizatora kojim se C kod prevodi u lepo obojenu HTML stranicu.

6. Kontekstno slobodne gramatike

Sintaksna analiza ili parsiranje predstavlja drugu fazu kompilacije koja dolazi nakon leksičke analize. U prethodnom poglavlju o leksičkoj analizi videli smo da lekser može da prepozna tokene na osnovu regularnih izraza kojima su opisani. Međutim, leksički analizator ne može da ispita da li je zadata rečenica sintaksno ispravna zbog ograničenja samih regularnih izraza. Zbog toga, ova faza je zasnovana na kontekstno slobodnim gramatikama, skraćeno KSG. Regularni izrazi su dovoljni da se opiše leksička struktura regularnog jezika, a KSG su dovoljne da se opiše sintaksna struktura većine programskih jezika.

Ukratko, podela posla između leksičkog i sintaksnog analizatora može se prikazati sledećom tablicom:

Sintaksni analizator	Leksički analizator
Kao ulaz prihvata tokene od leksičkog analizatora.	Prepoznaje tokene u ulaznom nizu karaktera.
Obrađuje tokene u ulaznom nizu karaktera da bi prepoznao smislene strukture samog jezika.	Odgovoran je za ispravnost tokena koji se prosleđuje sintaksnom analizatoru.
Obrađuje rekurzivne konstrukte jezika.	Olakšava posao sintaksnom analizatoru.

U nastavku teksta biće reči o kontekstno slobodnim gramatikama, transformacijama gramatika i tehnikama kojima se može implementirati sintaksni analizator.

6.1 Kontekstno slobodne gramatike

Definicija 6.1.1 — Kontekstno slobodna gramatika. G je uređena četvorka (Σ, N, S, P), za koju važi:

- Σ je skup završnih simbola,
- N je skup nezavršnih simbola,
- S je aksioma (početni simbol), za koju važi $S \in N$, i
- P je skup pravila gramatike, za koji važi $P \subseteq N \times (N \cup \Sigma)^*$. Dakle, formalno rečeno, svako pravilo je uređeni par gde je prva komponenta uređenog para tačno jedan neterminal, a druga komponenta je bilo kakva kombinacija završnih i nezavršnih simbola.

Primer 6.1 Dat je jezik $L = \{a^n b^n \mid n \geq 0\}$. Napisati gramatiku koja opisuje dati jezik.

Rešenje. Prvo je potrebno definisati skup završnih simbola $\Sigma = \{a, b\}$. Zatim, definišemo gramatiku navođenjem pravila preslikavanja. S obzirom da može biti 0, možemo da izvedemo praznu reč ε, pa moramo da uključimo takvo pravilo u skup pravila preslikavanja. Dakle, skup $P = \{S \to \varepsilon, S \to aSb\}$. Na kraju, definišemo skup nezavršnih simbola $N = \{S\}$.

Definicija 6.1.2 — Rečenična forma. je bilo koja reč iz $\Sigma \cup N$.

1. S je rečenična forma.
2. Ako je $\alpha X \beta$ rečenična forma i postoji pravilo $X \to \gamma \in P$, tada je i $\alpha\gamma\beta$ rečenična forma. U tom slučaju kažemo da $\alpha X \beta$ izvodi $\alpha\gamma\beta$ i da su ove dve rečenične forme u relaciji izvođenja.

Kraće, ovo zapisujemo kao $\alpha X \beta \Rightarrow \alpha\gamma\beta$.

Radi jednostavnijeg i kraćeg zapisa uvodimo dva nova simbola za relaciju izvođenja rečeničnih formi:

- Tranzitivno zatvorenje relacije izvođenja, tj. izvođenje u jednom ili više koraka ($\Rightarrow^+$).
- Tranzitivno i refleksivno zatvorenje relacije izvođenja, tj. izvođenje u nula ili više koraka ($\Rightarrow^*$).

Jezik gramatike G označavamo sa $L(G)$. Jezik gramatike G je skup svih reči koje možemo da izvedemo u nula ili više koraka polazeći od startnog simbola S gramatike G, tj. $L(G) = \{\omega \in \Sigma^* \mid S \Rightarrow^* \omega\}$.

Primer 6.2 Primer izvodjenja u gramatici iz primera 6.1.

Rešenje. $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb \in L(G)$

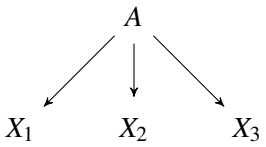
Definicija 6.1.3 Izvođenje u gramatici G je niz rečeničnih formi takvih da je prva aksioma, tj. startni simbol S, a poslednja je reč koja pripada jeziku $L(G)$ i svake dve rečenične forme su u relaciji izvođenja.

Primer 6.3 Kontekstno slobodnom gramatikom opisati liste cifri.

Rešenje. Prvo navedimo nekoliko primera ispravnih listi cifara:

```
5          1,2          4,1,9,8,5
```

Očigledno, lista cifara može biti sačinjena od samo jedne cifre ili od većeg broja cifara međusobno razdvojenih zarezima. S obzirom da je cifra najniža nedeljiva celina u sintaksi,



Slika 6.1: Razgranavanje stabla izvođenja na osnovu pravila gramatike.

upravo će ona biti završni simbol u gramatici. Pored cifre, zarez je takođe nedeljiva celina, pa i on pripada skupu tokena. Pažljivo posmatrajući ovaj opis, lako je uočiti nekoliko mogućih opisa listi, tj. mogućih grupisanja cifara:

Levo grupisanje	Desno grupisanje	Višeznačno grupisanje
$lista \rightarrow lista, CIFRA$	$lista \rightarrow CIFRA, lista$	$lista \rightarrow lista, lista$
$lista \rightarrow CIFRA$	$lista \rightarrow CIFRA$	$lista \rightarrow CIFRA$

U rekurzivnom pravilu, ako elemente grupišemo sa leve strane, dobićemo levo rekurzivno pravilo i samim tim levo rekurzivnu gramatiku. Ako elemente grupišemo sa desno strane, dobićemo desno rekurzivno pravilo i tako desno rekurzivnu gramatiku. Ako elemente grupišemo i sa leve i sa desne strane, dobićemo višeznačno pravilo i tako vešeznačnu gramatiku. Ne postoji algoritam za utvrđivanje da li je gramatika višeznačna. Jasan pokazatelj da je pravilo višeznačno ako je ono istovremeno i levo i desno rekurzivno. Gramatika koja sadži takvo pravilo je višeznačna.

Definicija 6.1.4 Drvo izvođenja je n-arno stablo sa sledećom strukturom:

- Koren stabla je početni simbol gramatike G.
- Listovi stabla su završni simboli.
- Unutrašnji čvorovi stabla su neterminali.

Ako u gramatici postoji pravilo oblika $A \rightarrow X_1X_2X_3 \in P$, onda se grananje drveta izvođenja šematski može prikazati kao na slici 6.1:

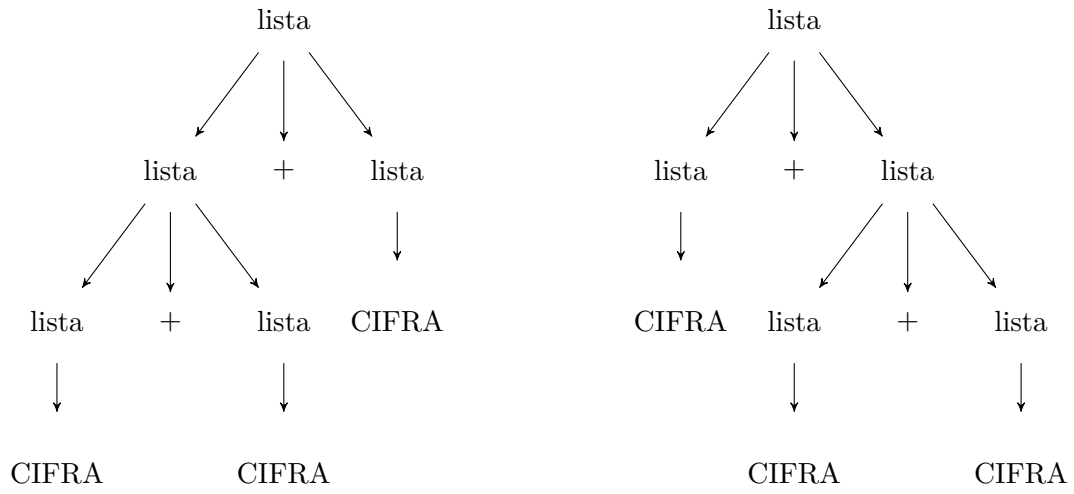
Višeznačnost se ogleda u tome da je za isti ulazni skup tokena moguće napraviti barem dva različita drveta izvođenja. U slučaju levo i desno rekurzivnih gramatika tako nešto nije moguće, ali u slučaju višeznačne gramatike jeste. Pretpostavimo da imamo sledeću gramatiku:

$$lista \rightarrow lista\ ,\ lista \mid CIFRA$$

i da je na ulazu sledeća lista cifara: 4,1,9. Na slici 6.2 prikazana su dva moguća drveta izvođenja u navedenoj gramatici za zadati ulaz, koja očigledno nisu ekvivalentna.

Ispitivanje da li je gramatika višeznačna ili ne je neodlučiv problem, tj. ne postoji algoritam kojim se za bilo koju gramatiku može utvrditi da li je višeznačna ili ne. Najlakše uočljiv indikator višeznačnosti neke gramatike je postojanje barem jednog pravila koje je istovremeno i levo i desno rekurzivno.

Zabluda:
Gramatika je višeznačna ako za istu nisku na ulazu postoje bar dva različita izvođenja.



Slika 6.2: Dva različita stabla izvođenja za istu listu $4,1,9$

Istina:
Gramatika je višeznačna ako za istu nisku na ulazu postoje bar dva različita drveta izvođenja. Dva različita izvođenja mogu imati isto drvo izvođenja, što ih čini ekvivalentnim, pa gramatika nije višeznačna.

U daljem tekstu, pažnju ćemo posvetiti isključivo levo i desno rekurzivnim gramatikama. Višeznačne gramatike i njihova implementacija su izvan opsega ovog teksta.

Primer 6.4 Kontekstno slobodnom gramatikom opisati aritmetičke izraze.

Rešenje.

Na početku, fokusiraćemo se samo na opisivanje sabiranja. Neka je dat izraz $3+5+6+7$. Lako se uočava analogija sa listama cifara. Umesto cifre, token će biti broj i umesto zareza, token će biti plus, pa imamo dve moguće gramatike koje opisuju sabiranje:

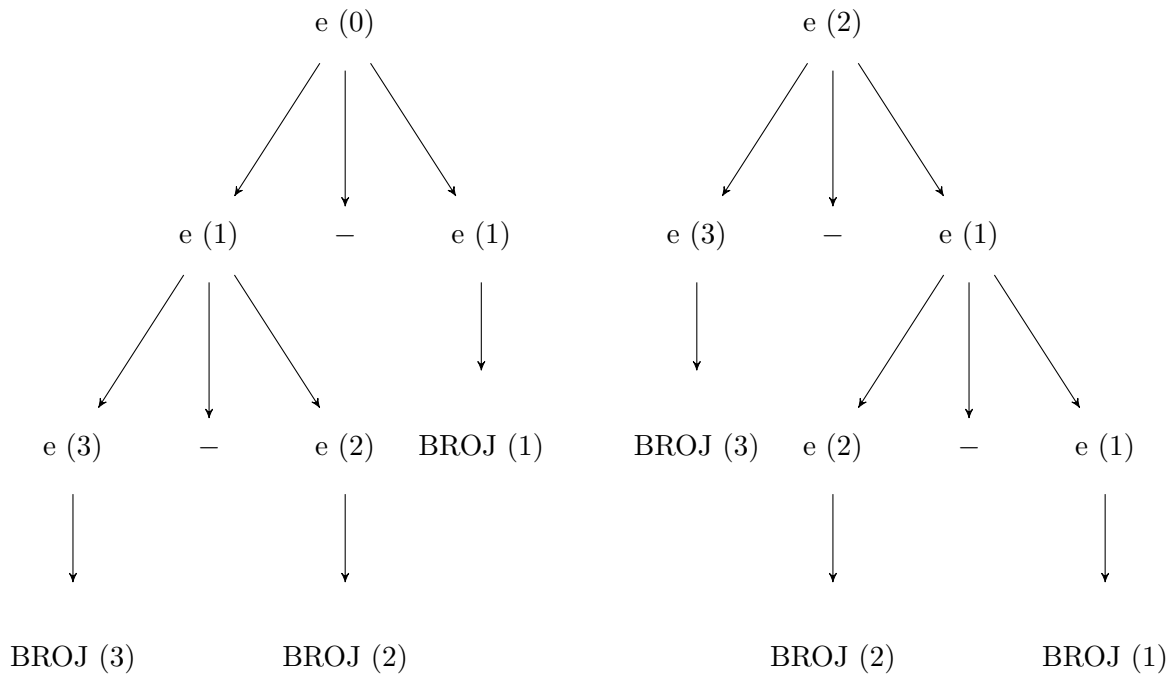
Levo rekurzivna gramatika	Desno rekurzivna gramatika
$e \to e + BROJ$	$e \to BROJ + e$
$e \to BROJ$	$e \to BROJ$

S obzirom da je operacija sabiranja komutativna i asocijativna operacija, nije važno da li grupisanje vršimo sa leve ili desne strane, pa su i jedna i druga gramatika validne.

Pored sabiranja, aritmetički izrazi obuhvataju i oduzimanje, pa u nastavku razmatramo operaciju oduzimanja. Neka je dat izraz $3-2-1$ Ako primenimo istu analogiju kao i u slučaju sabiranja, možemo da konstruišemo sledeće dve gramatike:

Levo rekurzivna gramatika	Desno rekurzivna gramatika
$e \to e - BROJ$	$e \to BROJ - e$
$e \to BROJ$	$e \to BROJ$

Obe gramatike će uspešno proveriti sintaksnu ispravnost izraza, ali samo jedna od njih je u stanju da ispravno izračuna vrednost izraza. Prilikom konstruisanja gramatike potrebno



Slika 6.3: Dva različita stabla izvođenja za isti izraz $3-2-1$

je uzmemo u obzir i značenje toga što gramatikom opisujemo. Preciznije, pored sintaksne strukture jezika gramatika bi trebalo da prati i semantiku samog jezika. Da bismo se u ovo uverili, nacrtaćemo stabla izvođenja koja odgovaraju ovim gramatikama. Stabla su prikazana na slici 6.3.

Vrednosti u zagradama su rezultati izračunavanja u gramatici. Očigledno, levo rekurzivna gramatika je izračunala vrednost izraza na pravi način, a desno rekurzivna nije. Na osnovu ovog primera bitno je izvući sledeći zaključak:

- Levoj rekurziji odgovara levo asocijativna operacija.
- Desnoj rekurziji odgovara desno asocijativna operacija.

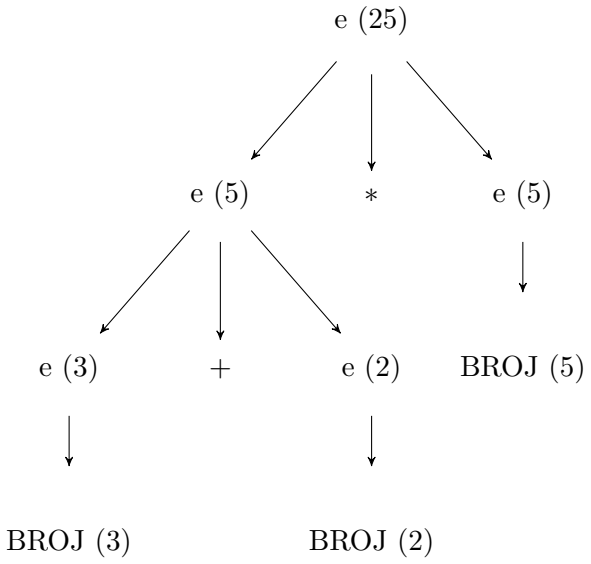
Jednostavnije rečeno, prilikom konstruisanja gramatike potrebno je da obratimo pažnju na asocijativnost operacija koje želimo da opišemo. Operacija oduzimanja je levo asocijativna operacija, pa joj odgovara levo rekurzivna gramatika. Uzimajući ovo razmatranje u obzir, kompletna KSG koja opisuje sabiranje i oduzimanje je sledeća:

$$\begin{aligned} e \rightarrow\ & e + \mathit{BROJ} \\ \mid\ & e - \mathit{BROJ} \\ \mid\ & \mathit{BROJ} \end{aligned}$$

Napišimo izvođenje za izraz $3+2-1$ u upravo definisanoj gramatici:

$$e \Rightarrow e - \mathit{BROJ} \Rightarrow e + \mathit{BROJ} - \mathit{BROJ} \Rightarrow \mathit{BROJ} + \mathit{BROJ} - \mathit{BROJ}$$

Primetimo da smo ovde naveli izvođenje u gramatici, a ne drvo izvođenja. Drvo izvođenja je jedinstveno, a izvođenje nije, jer zavisi od toga kojim redom primenjujemo pravila izvođenja, tj. na koji način obilazimo drvo izvođenja.



Slika 6.4: Ilustracija semantički neispravne gramatike izraza na izrazu $3+2*5$

U nastavku, našoj gramatici dodaćemo podršku i za operaciju množenja. Radi jednostavnosti, prvo ćemo posmatrati izraze koji sadrže samo sabiranje i množenje. Ako primenimo isto razmišljanje kao malopre, dobićemo gramatiku sledećeg oblika:

$$\begin{aligned} e &\rightarrow e + BROJ \\ &\mid e * BROJ \\ &\mid BROJ \end{aligned}$$

Gramatika je jednoznačna i sintaksno je ispravna. Da bismo to pokazali, možemo da pokušamo da napišemo izvođenje za izraz $3+2*5$. Izvođenje je sledeće:

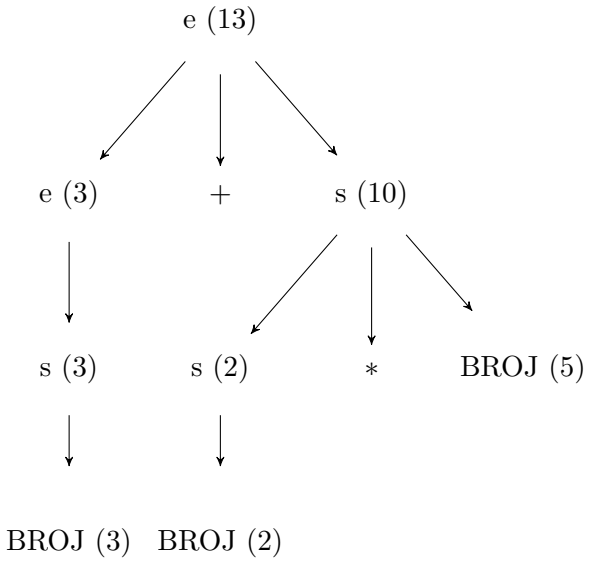
$$e \Rightarrow e * BROJ \Rightarrow e + BROJ * BROJ \Rightarrow BROJ + BROJ * BROJ$$

Iako sintaksno ispravna, gramatika nije ispravna semantički. Ono što je ovde pogrešno jeste to što ove dve operacije imaju isti prioritet. Ukoliko pokušamo da izračunamo vrednost ovog izraza uz pomoć konstruisane gramatike, dobićemo pogrešan rezultat što je ilustrovano drvetom izvođenja na slici 6.4.

Vrednosti u zagradama su rezultati izračunavanja u gramatici. Očigledno, dobijeni rezultat izvršavanja ne valja, jer prioriteti nisu lepo definisani. Kod levo rekurzivnih pravila u gramatici, operacije se uvek izvode sa leva na desno, pa treba da nađemo neki mehanizam da opišemo prioritete. Da bismo to izveli potrebno je da posmatramo problem sa višeg nivoa i zbog toga krećemo od složenijeg primera:

$$1+3*5+4*8+2*4+3$$

Da bismo konstruisali gramatiku, potrebno je da što jednostavnije opišemo navedeni primer. Najlakši opis bi mogao biti da je to zbir sabiraka. Pri čemu sabirak može biti



Slika 6.5: Semantički ispravna gramatika izraza $3+2*5$

jednostavno, broj, ili složenije, proizvod. Upravo u ovom opisu rečima se krije način kako da definišemo prioritete u gramatici:

$$
\begin{aligned}
e &\rightarrow e+s \\
&\mid s \\
s &\rightarrow s*BROJ \\
&\mid BROJ
\end{aligned}
$$

Rečima opisano, pravila nezavršnog simbola e definišu da se svaki izraz može predstaviti kao zbir sabiraka ili samo jedan sabirak. Pravila nezavršnog simbola s definišu da se svaki sabirak može predstavit kao proizvod više brojeva ili kao jedan broj. Da bismo se uverili da je ovako definisana gramatika sintaksno ispravna, nacrtaćemo drvo izvođenja za primer $3+2*5$. Drvo izvođenja je prikazano na slici 6.5.

S obzirom da u svakom koraku izvođenja možemo da primenimo tačno jedno pravilo, stablo za ovaj izraz je jednoznačno određeno. Imajući u vidu prioritete operacija, gramatiku lako možemo da proširimo sa operacijama deljenja i oduzimanja. U skladu sa dosadašnjim zaključcima dobijamo sledeću gramatiku:

$$
\begin{aligned}
e &\rightarrow e+s \\
&\mid e-s \\
&\mid s \\
s &\rightarrow s*BROJ \\
&\mid s/BROJ \\
&\mid BROJ
\end{aligned}
$$

Potpuna gramatika izraza bi trebalo da podržava i grupisanje članova, tj. zagrade, kojima korisnik može da promeni prioritet izvršavanja operacija. Da bismo to postigli potrebno

je opišemo sledeći primer izraza:

$$(2+3)*5+1*4+2*(5+3)$$

Da bismo opisali ovaj izraz treba pažljivo da ga raščlanimo. Očigledno, izraz je zbir sabiraka ili samo jedan sabirak. Svaki sabirak može biti ili proizvod više činilaca ili samo jedan činilac. Dok je činilac ili zagrada u kojoj se nalazi izraz ili samo jedan broj. Zapis gramatike izgleda ovako:

$$\begin{aligned} e &\to e+t \\ &\mid t \\ t &\to t*f \\ &\mid f \\ f &\to (e) \\ &\mid \mathit{BROJ} \end{aligned}$$

Navedena gramatika se naziva *gramatika algebarskih izraza* i predstavlja osnovni okvir za izučavanje i ilustrovanje svih važnih koncepata sintaksne analize. Označavanje neterminala u ovoj gramatici je promenjeno da bi se uskladilo sa usvojenim pravilama u klasičnoj literaturi. Pravilo kaže da se neterminali označavaju malim slovima, a terminali velikim. Preimenovan je neterminal s u t. Ukratko, neterminal e predstavlja čitav izraz (*eng. expression*), t predstavlja sabirak (*eng. term*), a f predstavlja činilac (*eng. factor*). Azbuku tokena čini skup $\Sigma = \{\mathit{BROJ}, +, *, (,)\}$. Pretpostavimo da nam je dat izraz $(2+3)*5$ i da želimo da nacrtamo drvo izvođenja u gramatici izraza. Drvo izvođenja je prikazano na slici 6.6.

U svakom koraku izvođenja, postoji samo jedno pravilo koje možemo da primenimo, pa je drvo izvođenja jednoznačno određeno. Iako je drvo izvođenja jednoznačno određeno, možemo da napišemo barem dva izvođenja za ovaj izraz u gramatici. Dva očigledna izvođenja su najlevlje (*eng. left-most*) izvođenje i najdešnje (*eng. right-most*) izvođenje, koja redom odgovaraju obilasku drveta sa leve na desnu, odnosno sa desne na levu stranu. Jednostavnijim rečnikom, najlevlje izvođenje znači da u svakom koraku izvođenja uvek izvodimo najlevlji neterminal, a u najdešnjem izvođenju uvek izvodimo najdešnji neterminal.

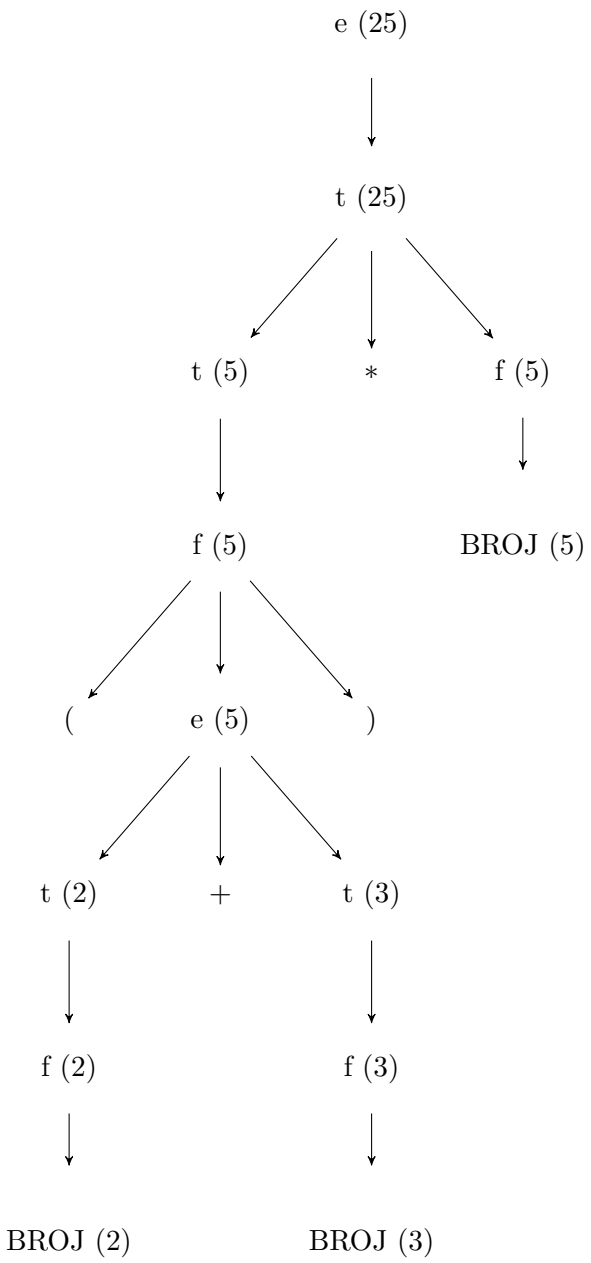
U narednim primerima, podvučen je neterminal koji se koristi za izvođenje u svakom koraku. Izvođenje na levo za naš primer u gramatici izraza je:

$$\begin{aligned} &\underline{e} \Rightarrow \underline{t} \Rightarrow \underline{t}*f \Rightarrow \underline{f}*f \Rightarrow (\underline{e})*f \Rightarrow (\underline{e}+t)*f \Rightarrow (\underline{t}+t)*f \Rightarrow (\underline{f}+t)*f \Rightarrow (\mathit{BROJ}+\underline{t})*f \\ &\Rightarrow (\mathit{BROJ}+\underline{f})*f \Rightarrow (\mathit{BROJ}+\mathit{BROJ})*\underline{f} \Rightarrow (\mathit{BROJ}+\mathit{BROJ})*\mathit{BROJ} \end{aligned}$$

Izvođenje na desno za naš primer u gramatici izraza je:

$$\begin{aligned} &\underline{e} \Rightarrow \underline{t} \Rightarrow t*\underline{f} \Rightarrow \underline{t}*\mathit{BROJ} \Rightarrow (\underline{e})*\mathit{BROJ} \Rightarrow (e+\underline{t})*\mathit{BROJ} \Rightarrow (e+\underline{f})*\mathit{BROJ} \Rightarrow (\underline{e}+\mathit{BROJ})*\mathit{BROJ} \\ &\Rightarrow (\underline{t}+\mathit{BROJ})*\mathit{BROJ} \Rightarrow (\underline{f}+\mathit{BROJ})*\mathit{BROJ} \Rightarrow (\mathit{BROJ}+\mathit{BROJ})*\mathit{BROJ} \end{aligned}$$

Ako neki operator treba da ima veći prioritet od ostalih, onda se pravila koja ga sadrže moraju navesti dublje u gramatici. Na osnovu stabla izvođenja vidite da će se prvo primeniti operacija koja je dublje u stablu, tj. bliže listovima, pa kasnije one operacije bliže korenu stabla.



Slika 6.6: Drvo izvođenja u gramatici izraza za izraz $(2+3)*5$

6.2 Transformacija gramatika

Da bismo mogli da implementiramo sintaksni analizator na osnovu kontekstno slobodne gramatike, ona mora da ispunjava nekoliko uslova. U nastavku teksta naveščemo na šta sve treba da obratimo pažnju prilikom definisanja gramatike i kako da osmišljenu gramatiku dovedemo do oblika koji može da se iskoristi za implementaciju. Na gramatiku možemo primeniti neku od navedenih transformacija:

- Eliminacija nedostižnih simbola
- Eliminacija neproduktivnih simbola
- Eliminacija anulirajućih simbola
- Eliminacija leve rekurzije

Primer 6.5 Iz date gramatike eliminisati nedostižne simbole.

$$A \to aB$$
$$B \to bB \mid b$$
$$C \to d$$

Rešenje. Prvi pojam koji uvodimo je *nedostižan simbol*. Iz samog primera je očigledno da je C nedostižan simbol, jer se ne nalazi sa leve strane nijednog drugog pravila u gramatici, pa ne postoji način da se pojavi u izvođenju. Takođe, očigledno je da samo neterminali mogu biti dostižni i nedostižni simboli. Startni simbol gramatike (aksiom) je uvek dostižni simbol.

Da bismo odredili koji simboli su nedostižni potrebno je da odredimo skup dostižnih simbola. Svi simboli koji se ne nađu u skupu dostižnih simbola su nedostižni. Algoritam za određivanje dostižnih simbola ilustrovaćemo na primeru. Prvo je potrebno da odredimo skup simbola koji su dostižni u nula koraka. Jedini simbol dostižan u nula koraka je startni simbol gramatike. Skup simbola dostižnih u nula koraka obeležavamo na sledeći način:

$$D_0 = \{A\}$$

Nakon što smo odredili skup simbola dostižnih u nula koraka, treba da odredimo skup simbola dostižnih u jednom koraku. Skup simbola dostižnih u jednom koraku čine svi oni neterminali koje možemo direktno da izvedemo iz simbola koji se nalaze u skupu D_0. Iz gramatike očigledno je da će skup simbola dostižnih u jednom koraku biti:

$$D_1 = \{A, B\}$$

Postupak iterativno nastavljamo sve dok možemo da dodamo novi simbol u skup. Skup simbola dostižnih u dva koraka biće:

$$D_2 = \{A, B\}$$

S obzirom da nismo dodali nijedan novi simbol u skup D_2, tj. $D_1 = D_2$, zaustavljamo algoritam. Svi neterminali koji se ne nalaze u skupu D_2 su nedostižni i možemo da ih obrišemo njihova pravila iz gramatike, pa naša gramatika postaje:

$$A \to aB$$
$$B \to bB \mid b$$

Formalnije, algoritam za nalaženje nedostižnih simbola u gramatici G možemo da zapišemo kao:

1. Startni simbol S se obeležava kao dostižan.
2. Simbol Y obeležavamo kao dostižan ako postoji pravilo $X \to \omega$ u skupu pravila P za neki dostižni simbol X i rečenična forma ω sadrži simbol Y.
3. Korak 2 ponavljamo sve dok postoji neki neobeleženi simbol koji možemo da obeležimo kao dostižan.
4. Svi simboli koji nisu obeleženi kao dostižni su nedostižni.

Primer 6.6 Iz date gramatike ukloniti neproduktivne simbole.

$$A \to BAa \mid b$$
$$B \to BC \mid C$$
$$C \to CC \mid C$$

Rešenje. Ako pogledamo sledeće izvođenje:

$$A \Rightarrow BAa \Rightarrow CAa \Rightarrow CCAa \Rightarrow CCCAa \Rightarrow ...$$

jasno je da se izvođenje nikada neće završiti, zbog pravila simbola C. Za simbol C kažemo da je *neproduktivan simbol*. Zbog pravila $A \to b$ možemo reći da je aksioma produktivan simbol. Tako nešto ne možemo zaključiti za simbole B i C. Međutim pomenuti simbole iako nemaju pravila koja izdvajaju samo tokene, oni nemaju ni pravila takva da se sa desne strane nalazi rečenična forma sastavljena od tokena i simbola A, za koji smo se uverili da je produktivan. Odatle možemo zaključiti da su simboli B i C neproduktivni. Uklonićemo njihova pojavljivanja u gramatici i ona postaje:

$$A \to b$$

Primetimo da je gramatika produktivna ako i samo je startni simbol gramatike produktivan.

Formalnije, algoritam za određivanje neproduktivnih stanja gramatike G se može zapisati kao:

1. Neterminal X obeležavamo kao produktivan ako postoji izvođenje $X \to \omega$, pri čemu je rečenična forma ω sačinjena isključivo od terminala i neterminala koji su već obeleženi kao produktivni.
2. Ponavljamo korak 1 sve dok postoji neki neobeleženi simbol koji možemo da obeležimo kao produktivan.
3. Svi simboli koji nisu obeleženi kao produktivni su neproduktivni.

Jednim imenom nedostižni i neproduktivni simboli se nazivaju *beskorisni simboli*. Očigledno, prvi korak prilikom transformacije je da uklonimo beskorisne simbole. Da bismo to uradili potrebno je da izvršimo sledeći postupak:

1. Odrediti skup neproduktivnih simbola i ukloniti ih zajedno sa svim pravilima koja ih uključuju.
2. Odrediti skup nedostižnih simbola u rezultujućoj gramatici i ukloniti ih zajedno sa svim pravilima koja ih uključuju.

Bitno je primetiti da korak 1 neće učiniti produktivne simbole neproduktivnim, kao što ni korak dva neće učiniti korisne simbole beskorisnim, tj. rezultujuća gramatika će biti sastavljena samo od dostižnih i produktivnih simbola.

Za neterminal X kažemo je anulirajući ako postoji izvođenje koje u nula ili više koraka izvodi praznu reč, tj. $X \Rightarrow^* \varepsilon$. Formalnije, algoritam za određivanje anulirajućih simbola u gramatici možemo da zapišemo kao:

1. Simbol X je anulirajući ako u skupu pravila postoji pravilo $X \to \varepsilon$.
2. Simbol Y obeležavamo kao anulirajući, ako postoji pravilo $Y \to \omega$ u skupu pravila P i ω je rečenična forma sačinjena samo od simbola koji su već obeleženi kao anulirajući.

Opisani algoritam ćemo ilustrovati na sledećoj gramatici.

Primer 6.7 Iz date gramatike ukloniti anulirajuće simbole.

$$A \to BC \mid Da$$
$$B \to CC \mid b$$
$$C \to BC \mid \varepsilon$$
$$D \to AD \mid \varepsilon$$

Rešenje. Očigledno, simboli C i D su anulirajući u jednom koraku, jer postoje pravila $C \to \varepsilon$ i $D \to \varepsilon$, pa definišemo skup anulirajućih u jednom koraku kao:

$$A_1 = \{C, D\}$$

Postupak nastavljamo dalje da bismo otkrili da li postoje simboli koji su anulirajući u dva koraka. Simbol B je anulirajući u dva koraka, jer postoji pravilo $B \to CC$, tj. izvodi rečeničnu formu koja je sastavljana samo od simbola iz skupa A_1 . Zato, proširujemo skup i dobijamo:

$$A_2 = \{B, C, D\}$$

Nastavljamo dalje da bismo proverili da li postoje simboli koji su anulirajući u tri koraka, tj. da li postoji neki neobeleženi simbol sa bar jednim pravilom koje izvodi reč sačinjenu isključivo od simbola u skupaA_2 . Očigledno, postoji pravilo $A \to BC$, pa je simbol A takođe anulirajući i dodajemo ga u skup:

$$A_3 = \{A, B, C, D\}$$

Obeležili smo sve simbole, pa zaustavljamo postupak.

S obzirom da je i startni simbol anulirajući, jezik sarži praznu reč, pa je eliminacija ε-pokreta nemoguća, tj. ne može biti potpuna. Potrebno je modifikovati gramatiku tako što se ubaci nova aksioma S i pravilo $S \to \varepsilon$ koje izvodi epsilon direktno ε iz novog startnog simbola i pravilo da nova aksioma izvodi staru aksiomu $S \to A$. Na taj način se onemogući vraćanje u startni simbol S, tj. ne može biti sa desne strane nijednog pravila.

Uz pomoć skupa anulirajućih simbola, transformišemo polaznu gramatiku. Sva pravila koja sadrze anulirajuće simbole se transformišu. Uklanjaju se ε pravila , poput $D \to \varepsilon$. Ako smo u polaznoj gramatici imali pravilo $A \to BC$, a B i C jesu anulirajući, dodaju se nova pravila za isti simbol sa novom rečeničnom formom sa desne strane dobijenom tako što se u rečeničnoj formi polaznog pravila svaki od anulirajućih zamenjuje sa ε. Dakle, dobije se $A \to BC \mid B \mid C$.

Tada dobijamo gramatiku koja je ε-slobodna.

$$
\begin{aligned}
&S \to A \mid \varepsilon \\
&A \to BC \mid B \mid C \mid Da \mid a \\
&B \to CC \mid C \mid b \\
&C \to BC \mid B \mid C \mid \varepsilon \\
&D \to AD \mid A \mid D \mid \varepsilon
\end{aligned}
$$

Dobijena gramatika je ε-slobodna. Međutim sadrži nekorisna pravila poput $D \to D$. Uklanjamo takva pravila iz gramatike.

$$
\begin{aligned}
&S \to A \mid \varepsilon \\
&A \to BC \mid B \mid C \mid Da \mid a \\
&B \to CC \mid C \mid b \\
&C \to BC \mid B \mid \varepsilon \\
&D \to AD \mid A \mid \varepsilon
\end{aligned}
$$

Do sada smo utvrdili da levo rekurzivna pravila odgovaraju levo asocijativnim operacijama, a desno rekurzivna pravila desno asocijativnim operacijama. Međutim, da bismo implementirali sintaksni analizator određenim tehnikama, potrebno je da iz gramatike eliminišemo levu rekurziju i levu faktorizaciju. Nije moguće eliminisati rekurziju u potpunosti. Ona je u pravilima, zbog potrebe za ponavljanjem. Međutim možemo takva pravila transformisati tako da ne budu levo rekurzivna, već desno rekurzivna.

Primer 6.8 Ukloniti levu rekurziju iz sledeće gramatike:

$$A \to Aa \mid b$$

Rešenje. Gramatika je očigledno levo rekurzivna.Potrebno je da napravimo gramatiku koja je ekvivalentna ovoj, tj. zadaje isti jezik, ali nema levu rekurziju. Posmatramo proizvoljno izvođenje u gramatici:

$$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow Aaaa \Rightarrow baaa$$

Primećujemo da gramatika opisuje jezik $L = \{ba^n | n \geq 0\}$. Primećujemo da je svrha levog rekurzivnog pravila da generiše više pojavljivanja slova a, a da pravilo $A \to b$ predstavlja izlaz iz rekurzije i b će prekinuti rekurziju i predstavljati sam početak reči jezika. Dakle gramatiku možemo transformisati na sledeći način. Stavljamo izlaz iz rekurzije na početak rečenične forme koja se izvodi iz A, i uvodimo novi nezavršni simbol A' koji će biti zadužen za izvođenja višestrukog pojavljivanja slova a.

$$
\begin{aligned}
&A \to bA' \\
&A' \to aA' \mid \varepsilon
\end{aligned}
$$

Kod pravila za neterminal A' treba voditi računa da pravilo bude desno rekurzivno. Dakle, sve što se u levom rakurzivnom pravilu nalazilo posle simbola A stavljamo na početak. Simbol A' mora da ima mogućnost da izvede ε jer jezik obuhvata i reč b koja ne sadrži slova a.

Dobijena gramatika nije levo rekurzivna, ali nije ni ε-slobodna, pa treba da je transformisemo u ε-slobodnu. Primenom algoritma za određivanje anulirajućih simbola dobijamo da je samo simbol A' anulirajući, ali da simbol A nije. Transformisana gramatika će biti:

$$A \to bA' \mid b$$
$$A' \to aA' \mid a$$

Primer 6.9 Transforimisati sledeću gramatiku tako da ne sadrži levo rekurzivna pravila.

$$A \to Aa_1 \mid Aa_2 \mid b_1 \mid b_2$$

Rešenje. Očigledno, gramatika je levo rekurzivna i zadaje jezik $L(G) = (b_1|b_2)(a_1|a_2)^*$, što je možda manje očigledno. Potrebno je da eliminišemo levu rekurziju primenjujući isti postupak, kao i u slučaju prethodne gramatike:

$$A \to b_1A' \mid b_2A'$$
$$A' \to a_1A' \mid a_2A' \mid \varepsilon$$

Dobijena gramatika je ekvivalentan polaznoj, ali nije levo rekurzivna. Transformaciju dobijene gramatike u ε-slobodnu pokušajte da uradite sami za vežbu.

Primeri 6.8 i 6.9 ilustruju eliminaciju *direktne* leve rekurzije. U pravilima može se javiti i posredna rekurzija. Algoritam za njenu eliminaciju možete videti na predavanjima.

Pored levo rekurzivnih pravila, javlja se potreba i za eliminacijom *levo faktorisana pravila*. Pravila vezana za isti nezavršni simbol su levo faktorisana, ako njihove rečenične forme sa desnih strana imaju isti prefiks.

Primer 6.10 Ukloniti levu faktorizaciju iz sledeće gramatike:

$$A \to \alpha\beta \mid \alpha\gamma \mid abc$$

Rešenje. Uočavamo da simbol A ima dva pravila kojima je zajednički prefiks α. Eliminišemo levu faktorizaciju, tako što uvodimo novi nezavršni simbol koji će izvoditi sufikse posmatranih pravila.

$$A \to \alpha B \mid abc$$
$$B \to \beta \mid \gamma$$

6.3 Zadaci za vežbu

Zadatak 6.1 Kontekstno slobodnom gramatikom opisati iskazne formule. ■

Rešenje. Ono što je bio *BROJ* u gramatici izraza, u gramatici iskaznih formula biće $\top$ i $\bot$. Operacije koje postoje u iskaznoj logici su sledeće: negacija ($\neg$), konjunkcija ($\wedge$), disjunkcija ($\vee$), implikacija ($\Rightarrow$) i ekvivalencija ($\Leftrightarrow$). Prioritet operacija odgovara redosledu navođenja operacija, tj. negacije je najvišeg, a ekvivalencija najnižeg prioriteta. Imajući u vidu prethodnu diskusiju, skup tokena će biti $\Sigma+ = \{\top, \bot, (,), \neq, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$, a

gramatika će biti:

$$
\begin{aligned}
e &\to e \Leftrightarrow t \mid t \\
t &\to t \Rightarrow f \mid f \\
f &\to f \vee g \mid g \\
g &\to g \wedge h \mid h \\
h &\to \neg k \mid k \\
k &\to (e) \mid \top \mid \bot
\end{aligned}
$$

Nismo vodili računa o konkretnim brojevima u primeru sa gramatikom algebarskih izraza. Primenimo isti princip i ovde. Prepustimo leksičkom analizatoru da prepoznaje logičke konstante $\top$ i $\bot$ i da za obe vraća token KONST. Time pojednostavljujemo skup pravila sa neterminal k.

$$k \to (e) \mid KONST$$

Zadatak 6.2 Kontekstno slobodnom gramatikom opisati deklaraciju promenljivih u programskom jeziku Java. ■

Rešenje. Da bismo lakše napravili gramatiku koja opisuje deklaraciju promenljivih, pogledaćemo sledeći primer:

```
int a;
int a,b;
Point p,q;
double[] niz;
```

Postmatrajući primer, prvo što možemo da uočimo jeste da deklaracije možemo da nižemo jednu za drugom. Kada god imamo nizanje nečega, potrebno je da se setimo primera sa listom brojeva. Da bismo pravilno obradili nizanje potrebno je da uočimo koji simbol razdvaja dve naredbe, tj. separator. U našem slučaju terminator naredbe je ;, dakle nemamo separator. Nakon što smo sveli naš problem na nizanje naredbi potrebno je da opišemo strukturu naredbe.

Prilikom opisivanja naredbi, ne treba da težimo što detaljnijem opisu, već treba da se potrudimo da napravimo dovoljno opštu definiciju koja nam omogućava lako proširivanje gramatike ako se za to javi potreba u budućnosti. Imajući to u vidu, najopštiji opis naredbe, tj. deklaracije može biti :

$$deklaracija \to tip\ niz\_promenljivih\ ;$$

S obzirom da smo uveli dva nova neterminala, potrebno je da i njih opišemo. Tip podataka može biti neki osnovni tip, klasni tip ili neki nizovski tip. Niz promenljivih je ponovo nizanje, pa je to analogno nizanju naredbi, samo imamo separator zarez. Pre definisanja same gramatike, potrebno je da utvrdimo kako su opisani naziv promenljive i ime klase. I jedno i drugo se može posmatrati kao kombinacija slova i brojeva. Upravo ta kombinaciju karaktera predstavlja nedeljivu sintaksnu celinu, tj. token koji ćemo zvati *ID*. Ako se setimo priče o prioritetu prepoznavanja tokena u flex-u, jasno je da kao tokene treba da definišemo i ključne reči poput `int`, `float`, itd, da ne bi bile prepoznate kao identifikatori.

Pored ovih, tokeni treba da budu i ranije pomenuti , i ;. Ako ovo razmatranje iskoristimo prilikom definisanja gramatike dobićemo:

$$
\begin{aligned}
\textit{niz\_deklaracija} &\rightarrow \textit{niz\_deklaracija deklaracija} \\
&\mid \textit{deklaracija} \\
\textit{deklaracija} &\rightarrow \textit{tip niz\_promenljivih} \ ; \\
\textit{tip} &\rightarrow \textit{osnovni\_tip} \mid \textit{klasni\_tip} \mid \textit{tip}[\,] \\
\textit{osnovni\_tip} &\rightarrow \textit{INT} \mid \textit{FLOAT} \mid \textit{DOUBLE} \mid \textit{BOOLEAN} \\
\textit{klasni\_tip} &\rightarrow \textit{ID} \\
\textit{niz\_promenljivih} &\rightarrow \textit{niz\_promenljivih} \ , \ \textit{promenljiva} \\
&\mid \textit{promenljiva} \\
\textit{promenljiva} &\rightarrow \textit{ID}
\end{aligned}
$$

Važno je primetiti da smo prilikom definisanja gramatike uzeli u obzir samo test primer, a ne celokupnu sintaksu jezika Java. Međutim, opštost definicija za naredbe i tipove podataka omogućava nam da gramatiku lako proširimo naredbama sa drugačijom strukturom ili drugim tipovima podataka.

Pored ovoga potrebno je primetiti pravila za izvođenje deklaracije, promenljive i izvođenje klasnog tipa. Takva pravila nazivaju se jednostruka pravila i ona nepotrebno produžuju izvođenje i produbljuju stablo izvođenja. Činjenica da se svode na jednostavno preimenovanje omogućava nam da izbacimo jednostruka pravila iz gramatike i time ubrzamo sintaksnu analizu. Izbacivanje jednostrukih pravila iz gramatike svodi se na jednostavnu zamenu neterminala odgovarajućim rečeničnom formom sa desne strane pravila, pa gramatika postaje:

$$
\begin{aligned}
\textit{niz\_deklaracija} &\rightarrow \textit{niz\_deklaracija tip niz\_promenljivih} \ ; \\
&\mid \textit{tip niz\_promenljivih} \ ; \\
\textit{tip} &\rightarrow \textit{osnovni\_tip} \mid \textit{klasni\_tip} \mid \textit{tip}[\,] \\
\textit{osnovni\_tip} &\rightarrow \textit{INT} \mid \textit{FLOAT} \mid \textit{DOUBLE} \mid \textit{BOOLEAN} \\
\textit{niz\_promenljivih} &\rightarrow \textit{niz\_promenljivih} \ , \ \textit{ID} \\
&\mid \textit{ID}
\end{aligned}
$$

Ukoliko prepustimo leksičkom analizatoru da nam prepoznaje osnovne tipove i za sve njih nam vraća isti token, recimo OSNOVNI, još ćemo uprostiti gramatiku. Takođe klasni tip nije ništa drugo nego naziv klase, pa možemo ga zameni tokenom ID.

$$
\begin{aligned}
\textit{niz\_deklaracija} &\rightarrow \textit{niz\_deklaracija tip niz\_promenljivih} \ ; \\
&\mid \textit{tip niz\_promenljivih} \ ; \\
\textit{tip} &\rightarrow \textit{OSNOVNI} \mid \textit{ID} \mid \textit{tip}[\,] \\
\textit{niz\_promenljivih} &\rightarrow \textit{niz\_promenljivih} \ , \ \textit{ID} \\
&\mid \textit{ID}
\end{aligned}
$$

7. Sintaksna analiza naniže

Sintaksni analizator ili *parser* prima tokone od leksičkog analizatora i proverava njihovu saglasnost sa sintaksnim pravilima jezika koji su obično zadati kontekstno slobodnom gramatikom. Zadatak sintaksne analize se može opisati kao process utvrđivanja da se data reč može generisati gramatikom ili ne. Tokom analize, ulaz se skenira sa levo na desno, najčešće jedan po jedan token. Takođe, najčešće je dovoljan samo jedan token na ulazu da bi se odredio naredni korak sintaksne analize. Taj token na ulazu u sintaksni analizator se naziva *preduvidni simbol*.

Sintaksna analiza se obično vrši raznim metodama složenosti $\mathcal{O}(n)$, pri čemu je n dužina ulaza. Za sintaksnu analizu velikog broja programskih jezika, sasvim su dovoljni algoritmi linearne složenosti. Ovakve metode postoje za pojedine klase KSG o kojima će biti reči u nastavku.

U sintaksnoj analizi naniže, počinje se od startnog simbola i tokom analize se generiše izvođenje na levo niske tokena u gramatici koja se sa leva poredi sa niskom na ulazu. Na osnovu preduvidnih simbola u niski tokena određuje se sledeće pravilo iz skupa P koje se primenjuje.

Klasa $LL(1)$ kontekstno slobodnih gramatika dopušta da se prilikom analize naniže koristi samo jedan token sa ulaza da bi se jednoznačno odredilo pravilo koje treba primeniti u datom trenutku. Gramatika koja ima levo rekurzivna, levo faktorisana ili višeznačna pravila ne može biti $LL(1)$.

Ukoliko imamo levu faktorizaciju, onda za isti nezavršni simbol imamo više pravila sa istim prefiksom. Tada nije moguće na osnovu samo jednog tokena odrediti koje od pravila treba primeniti u izvođenju, jer isti token predstavlja početak svih pravila. Neophodno je da imamo još jedan ili više tokena sa ulaza da bismo doneli sigurno odluku pri izboru pravila ili detektovali grešku. Takvo odlučivanje nije svojstveno $LL(1)$ gramatikama.

Ako prvo eliminišemo levu rekurziju izgubiće se i leva faktorizacija, ako je postojala. U

slučaju da gramatika nije levo rekurzivna, uvek treba da proverimo da slučajno ne postoji leva faktorizacija da bismo je eliminisali.

7.1 Konstrukcija LL(1) gramatika i skupova izbora

Primer 7.1 Transformisati gramatiku algebarskih izraza u $LL(1)$ gramatiku.

$$
\begin{aligned}
e &\to e + t \\
&\mid t \\
t &\to t * f \\
&\mid f \\
f &\to (e) \\
&\mid BROJ
\end{aligned}
$$

Rešenje.

Očigledno, gramatika ima levo rekurzivna pravila, pa sigurno nije $LL(1)$. Prvi korak je eliminacija leve rekurzije.

$$
\begin{aligned}
e &\to t\ ep \\
ep &\to +\ t\ ep \\
&\mid \varepsilon \\
t &\to f\ tp \\
tp &\to f\ tp \\
&\mid \varepsilon \\
f &\to (\ e\ ) \\
&\mid BROJ
\end{aligned}
$$

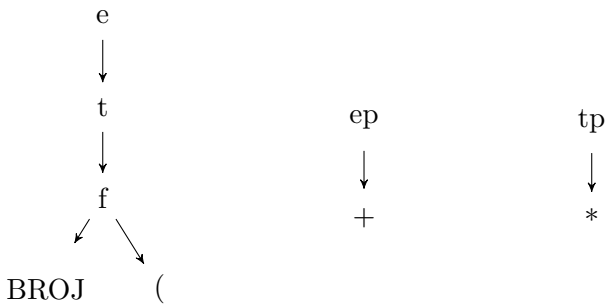
Transformisana gramatika više nije levo rekurzivna, ali to ne znači da je automatski iz klase $LL(1)$. Da bismo utvrdili da li je gramatika $LL(1)$, potrebno je da izračunamo skupove izbora za svako pravilo.

Skupovi izbora predstavljaju niske tokena koje određuju šta treba da se pojavi na ulazu, tj. koju vrednost treba da ima preduvidni simbol, da bismo primenili određeno pravilo. Za primenu određenog pravila, dovoljno je da se na ulazu pojavi bilo koji element iz njemu pridruženog skupa izbora. Da bismo odredili skupove izbora, potrebno je da najpre za svaki nezavršni simbol odredimo skupove *Prvi* (*eng. First*) i *Sledi* (*eng. Follow*).

Pre nego što počnemo sa njihovim određivanjem, potrebno je utvrditi koji neterminali su anulirajući. Ukoliko je neki od anulirajućih simbola prefiks rečenične forme sa desne strane pravila, onda se u razmatranje mora uzimati i činjenica da ti simboli mogu izvesti i konkretnu reč, ali i samo ε. To može uticati na određivanje skupova *Prvi* i *Sledi*.

U našem primeru anulirajući simboli u prvom koraku izvođenja su: $A_1 = \{ep, tp\}$ i nijedan više. Ovo su novouvedeni simboli prilikom oslobađanja od leve rekurzije. Tako da ne postoji pravilo koje izvodi rečeničnu formu sastavljenu samo od ova dva neterminala.

Skup $Prvi(X)$ nekog nezavršnog simbola X predstavlja skup terminala kojima počinju niske koje se izvode iz nezavršnog simbola X. Prilikom određivanja skupova $Prvi(X)$ gledaju se



Slika 7.1: Graf za određivanje skupova *Prvi*

samo ona pravila neterminala X koja ne sadrže ε pravila. Da bismo ih odredili potrebno je da gledamo kojim tokenom može da počne rečenična forma izvedena iz svakog neterminala i da prateći pravila dođemo do tokena koji čine skup. Grafički to možemo da prikažemo kao na slici 7.1.

Grana grafa predstavlja da neterminal iz početnog čvora grane ima iste početne tokene kao i neterminal iz čvora sa kraja grane.

Zapisujemo sve do sada u tablicu 7.1.

Neterminal	ε	Prvi
e	/	BROJ, (
ep	Da	+
t	/	BROJ, (
tp	Da	*
f	/	BROJ, (

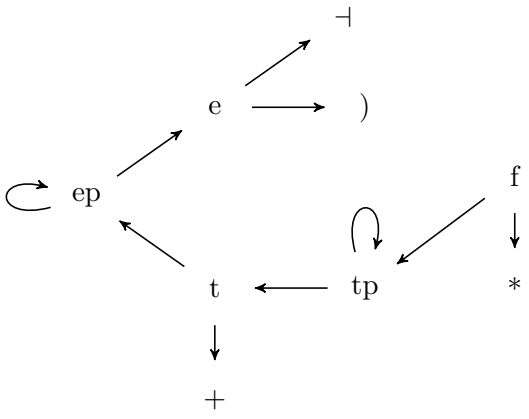
Tabela 7.1: Utvrđeni skupovi *Prvi* i anulirajući neterminali

Nakon što smo odredili skupove *Prvi*, potrebno je da odredimo i skupove *Sledi*. Skup *Sledi*(X) obuhvata sve završne simbole koji se mogu pojaviti u izvođenju neposredno nakon rečenične forme koju je izveo neterminal X.

Startni simbol uvek sadrži kraj ulaza ($\dashv$) u svom skupu *Sledi*. Da bismo odredili skup *Sledi*(X) neterminala X potrebno je da pogledamo sva pojavljivanja neterminala X sa desne strane pravila gramatike. Nakon što smo uočili takvo pojavljivanje, treba samo da pogledamo šta sve može da se pojavi neposredno nakon tog neterminala. Da bismo ih odredili ponovo ćemo da koristimo grafički prikaz. Grafički prikaz određivanja skupova je na slici 7.2.

Na osnovu podataka u grafu 7.2 možemo dopuniti tablicu 7.2.

Na kraju, potrebno je da odredimo skupove izbora uz pomoć skupova *Prvi* i *Sledi*. Za sva pravila koja nisu anulirajuća skup izbora je skup *Prvi* nezavršnog simbola sa leve strane pravila ili njegov podskup ako nezavršni simbol ima više pravila. Za sva anulirajuća ε pravila, skup izbora je skup *Sledi*. Ukoliko nezavršni simbol X ima pravilo koje nije direktno anulirajuće, ali može izvesti ε, skup izbora pored već određenih konkretnih tokena iz skupa *Prvi*(X) obuhvata i tokene iz skupa *Sledi*(X). Primenimo to na gramatiku algebarskih izraza.



Slika 7.2: Graf za određivanje skupova *Sledi* za gramatiku algebarskih izraza

Neterminal	ε	Prvi	Sledi
e	/	BROJ, (	*dashv*,)
ep	Da	+	⊣,)
t	/	BROJ, (	+,⊣,)
tp	Da	*	+,⊣,)
f	/	BROJ, (	*,+,⊣,)

Tabela 7.2: Utvrđeni skupovi *Prvi Sledi* i anulirajući neterminali

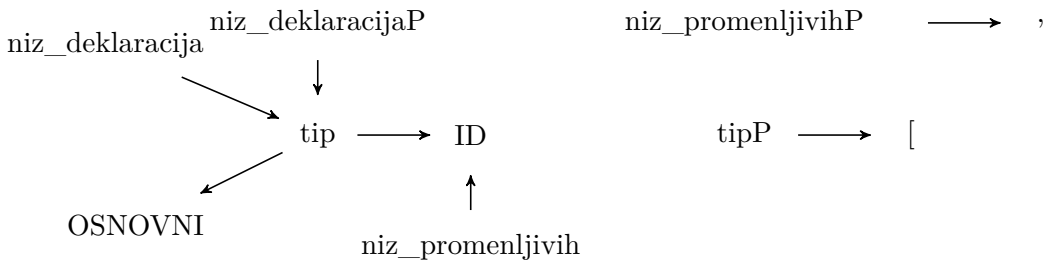
$$
\begin{aligned}
e &\to t\ ep && \{BROJ, (\} \\
ep &\to +\ t\ ep && \{+\} \\
&\mid \varepsilon && \{\dashv,)\} \\
t &\to f\ tp && \{BROJ, (\} \\
tp &\to *\ f\ tp && \{*\} \\
&\mid \varepsilon && \{+, \dashv,)\} \\
f &\to (\ e\ ) && \{(\} \\
&\mid BROJ && \{BROJ\}
\end{aligned}
$$

Nakon što smo odredili skupove izbora svih neterminala moramo da proverimo da li postoji neterminal za čija pravila skupovi izbora nisu disjunktni. Ako takav neterminal postoji, gramatika nije $LL(1)$, tj. ne može se na osnovu jednog preduvidnog simbola jednoznačno odrediti naredni korak izvođenja u gramatici.

7.1.1 Zadaci za vežbu

Zadatak 7.1 Opisati deklaraciju promenljivih u programskom jeziku Java $LL(1)$ gramatikom. ∎

Rešenje. Da bismo lakše napravili gramatiku koja opisuje deklaraciju promenljivih, pogledaćemo sledeći primer:



Slika 7.3: Graf za određivanje skupova *Prvi*

```
int a;
int a,b;
Point p,q;
double[] niz;
```

Iskoristićemo gramatiku koju smo već konstruisali za isti problem u zadatku 6.2.

$$
\begin{aligned}
\mathit{niz\_deklaracija} &\rightarrow \mathit{niz\_deklaracija}\ \mathit{tip}\ \mathit{niz\_promenljivih}\ ; \\
&\mid \mathit{tip}\ \mathit{niz\_promenljivih}\ ; \\
\mathit{tip} &\rightarrow \mathit{OSNOVNI} \mid \mathit{ID} \mid \mathit{tip}[\,] \\
\mathit{niz\_promenljivih} &\rightarrow \mathit{niz\_promenljivih}\ ,\ \mathit{ID} \\
&\mid \mathit{ID}
\end{aligned}
$$

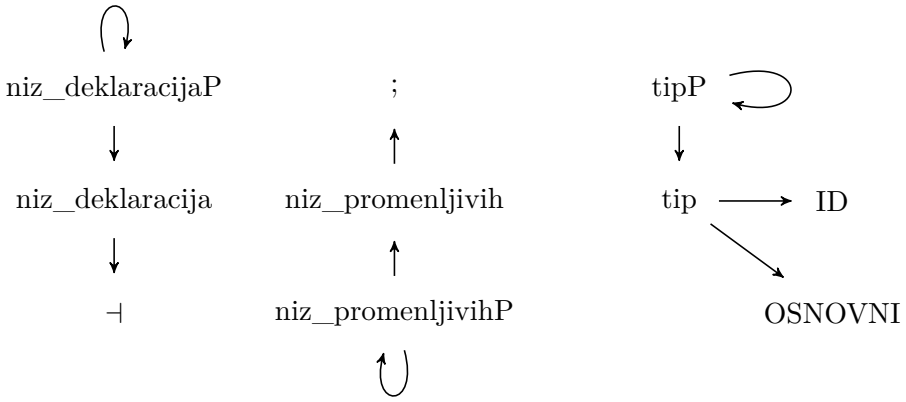
Gramatika nije $LL(1)$, jer ima levo rekurzivna pravila, pa moramo da je transformišemo prema ranije prikazanom postupku. Nakon eliminacije leve rekurzije dobićemo sledeću gramatiku:

$$
\begin{aligned}
\mathit{niz\_deklaracija} &\rightarrow \mathit{tip}\ \mathit{niz\_promenljivih}\ ;\ \mathit{niz\_deklaracijaP} \\
\mathit{niz\_deklaracijaP} &\rightarrow \mathit{tip}\ \mathit{niz\_promenljivih}\ ;\ \mathit{niz\_deklaracijaP} \\
&\mid \varepsilon \\
\mathit{tip} &\rightarrow \mathit{OSNOVNI}\ \mathit{tipP} \mid \mathit{ID}\ \mathit{tipP} \\
\mathit{tipP} &\rightarrow [\,]\ \mathit{tipP} \\
&\mid \mathit{eps} \\
\mathit{niz\_promenljivih} &\rightarrow \mathit{ID}\ \mathit{niz\_promenljivihP} \\
\mathit{niz\_promenljivihP} &\rightarrow ,\ \mathit{ID}\ \mathit{niz\_promenljivihP} \\
&\mid \varepsilon
\end{aligned}
$$

Transformisana gramatika nema levo rekurzivna, levo faktorisana i višeznačna pravila. Potrebno je da nađemo skupove izbora da bismo se uverili da je gramatika $LL(1)$. Skupove izbora ćemo odrediti pomoću skupova *Prvi* i *Sledi*. Pomenute skupove ćemo odrediti crtanjem odgovarajućih grafova. Grafovi su prikazani na slikama 7.3 i 7.4.

Nakon što smo odredili skupove *Prvi* i *Sledi* , zapisaćemo ih u sledećoj tablici 7.3:

Na osnovu tablice, možemo da odredimo skupove izbora za pravila iz gramatike.



Slika 7.4: Graf za određivanje skupova *Sledi*

Neterminal	ε	Prvi	Sledi
niz_deklaracija	/	OSNOVNI, ID	⊣
niz_deklaracijaP	Da	OSNOVNI, ID	⊣
tip	/	OSNOVNI, ID	ID
tipP	Da	[	ID
niz_promenljivih	/	ID	;
niz_promenljivihP	Da	,	;

Tabela 7.3: Utvrđeni skupovi *Prvi Sledi* i anulirajući neterminali

$$
\begin{aligned}
niz\_deklaracija &\rightarrow tip\ niz\_promenljivih\ ;\ niz\_deklaracijaP && \{OSNOVNI, ID\}\\
niz\_deklaracijaP &\rightarrow tip\ niz\_promenljivih\ ;\ niz\_deklaracijaP && \{OSNOVNI, ID\}\\
&\mid \varepsilon && \{\dashv\}\\
tip &\rightarrow OSNOVNI\ tipP \mid ID\ tipP && \{OSNOVNI, ID\}\\
tipP &\rightarrow [\ ]\ tipP && \{[\}\\
&\mid eps && \{ID\}\\
niz\_promenljivih &\rightarrow ID\ niz\_promenljivihP && \{ID\}\\
niz\_promenljivihP &\rightarrow ,\ ID\ niz\_promenljivihP && \{,\}\\
&\mid \varepsilon && \{;\}
\end{aligned}
$$

Na kraju, potrebno je da proverimo da li su skupovi izbora za sve grane svih neterminala sa više grana disjunktne. U našem slučaju, svi skupovi izbora za sve izbore su disjunktni, pa možemo jednoznačno da odredimo naredni korak u izvođenju na osnovu jednog predu-vidnog simbola, odakle zaključujemo da je naša gramatika $LL(1)$.

7.2 Rekurzivni spust

Rekurzivni spust je jedna od tehnika koja se koristi da bi se implentirao parser koji koristi sintaksnu analizu naniže. Parser koji koristi rekurzivni spust je izgrađen od skupa uzajamno rekurzivnih procedura ili funkcija dodeljenih neterminalima u gramatici. Zbog toga, struktura rezultujućeg programa je gotovo identična gramatici čiji jezik prepoznaje.

U nastavku teksta bavićemo se isključivo prediktivnim parserima, tj. onima koji ne zahtevaju bektreking da bi odredili koji je naredni korak u izvođenju. Prediktivni parser je moguće napisati isključivo za kontekstno slobodno gramatike koje pripadaju klasi $LL(k)$. $LL(k)$ gramatike su one za koje postoji neki pozitivan ceo broj k koji omogućava parseru koji koristi rekurzivni spust da na osnovu sledećih k tokena na ulazu jednoznačno odredi koje pravilo treba primeniti u sledećem koraku izvođenja. Da bi gramatika bila $LL(k)$ ne sme da sadrži višeznačna, levo rekurzivna i levo faktorisana pravila. Svaka kontekstno slobodna gramatika se može transformisati u ekvivalentnu gramatiku koja nije levo rekurzivna. Eliminacija leve rekurzije ne mora nužno da dovede do gramatike koja pripada klasi $LL(k)$. Prediktivni parser zasnovan na rekurzivnom spustu se izvršava u linearnom vremenu.

Ograničićemo se isključivo na gramatike klase $LL(1)$ prilikom implementacije rekurzivnog spusta. Gramatike koje pripadaju ovoj klasi mogu da odrede koje se naredno pravilo izvođenja primenjuje samo na osnovu jednog preduvidnog simbola. Da bismo uopšte mogli da implementiramo parser zasnovan na rekurzivnom spustu potrebno je da izvršimo sledeće korake:

1. Smisliti gramatiku koja opisuje zadati jezik.
2. Eliminisati leva rekurzivna i levo faktorisana pravila.
3. Odrediti skupove *Prvi* i *Sledi*.
4. Na osnovu skupova i odrediti skupove izbora.
5. Proveriti da li su disjunktni skupovi izbora za pravila koja su vezana za isti nezavršni simbol. Ako jesu, onda možemo jednoznačno da utvrdimo koje pravilo izvođenja treba da primenimo samo na osnovu jednog preduvidnog simbola.

Ovaj postupak ćemo ilustrovati na gramatici izraza koju znamo od ranije i za koju smo već utvrdili da je $LL(1)$.

Primer 7.2 Napisati sintaksni analizator za gramatiku algebarskih izraza koristeći tehniku rekurzivnog spusta.

Rešenje. Polazna gramatika je:

$$
\begin{aligned}
e &\to e + t \\
&\mid t \\
t &\to t * f \\
&\mid f \\
f &\to (e) \\
&\mid \mathit{BROJ}
\end{aligned}
$$

U skladu sa opisanim postupkom prvo je potrebno da eliminišemo levu rekurziju, zatim da odredimo skupove *Prvi* i *Sledi*, da uz pomoć njih odredimo skupove izbora i na kraju da se uverimo da je $LL(1)$. Na kraju tog postupka transformisana gramatika sa određenim skupovima izbora je već definisana u 7.1.

Sada kada imamo transformisanu gramatiku, potrebno je da implementiramo parser tehnikom rekurzivnog spusta. Da bismo to uradili, potrebno je da prvo implementiramo lekser koji će tokenizovati ulazni niz karaktera i prosleđivati parseru koji je sledeći token na ulazu. Zbog toga, prvo treba da odredimo azbuku tokena $\Sigma = \{\mathit{BROJ}, *, +, \dashv, (,)\}$. Prilikom određivanja azbuke tokena ne smemo da zaboravimo da uključimo kraj ulaza u

skup, tj.⊣. Na osnovu azbuke tokena, treba da definišemo tokene preko makroa i treba da definišemo posebnu globalnu promenljivu u kojoj ćemo čuvati koji je sledeći token na ulazu, tj. preduvidni simbol. Svakom neterminalu potrebno je da dodelimo jednu funkciju. Dakle, imaćemo posebnu datoteku `tokeni.h` u kojoj se nalazi spisak tokena, zatim posebnu globalnu promenljivu `preduvid` i spisak funkcija kojima opisujemo neterminale:

```
int preduvid;
void E(void);
void EP(void);
void T(void);
void TP(void);
void F(void);
```

U svakoj funkciji treba da proverimo da li preduvudni simbol ima neku od vrednosti iz skupa izbora dodeljenom tom neterminalu i da primenimo odgovarajuće pravilo. Ako preduvidni simbol nema očekivanu vrednost, tada se desila sintaksna greška. Da bismo započeli parsiranje, potrebno je da učitamo prvi preduvidni simbol i da pozovemo funkciju koja odgovara startnom simbolu gramatike. Na kraju izvršavanja programa, ako preduvidni simbol ima vrednost tokena *EOI* koji odgovara tokenu ⊣, onda je ulaz sintaksno ispravan. Sintaksni analizator za gramatiku izraza zasnovan na rekurzivnom spustu prikazan je u nastavku:

```
#ifndef TOKENI_H
#define TOKENI_H

/* tokene iz azbuke zapisujemo kao makroe
 * 0 je rezervisana vrednost koja uvek mora
 * da predstavlja kraj ulaza, tj. EOI
 */
#define EOI     0
#define BROJ    1
#define OZ      2
#define ZZ      3
#define PLUS    4
#define PUTA    5

#endif
```

Rešenje 7.1: Specifikacija tokena

```
%option noyywrap

%{
/* ukljucujemo dokument u kome se nalazi spisak tokena */
#include "tokeni.h"
%}

%%
[0-9]+      { /* parseru vratimo token koji odgovara broju */
              return BROJ;
            }
```

```
")"          { return ZZ; }
"("          { return OZ; }
"+"          { return PLUS; }
"*"          { return PUTA; }
[ \t\n]      { /* beline ignorisemo */}
.            {
               /* ako prepoznamo bilo sta sto nije iz skupa
                * tokenaprijavljujemo leksicku gresku */
               fprintf(stderr,
                 "Leksicka greska: %s\n", yytext);
               exit(EXIT_FAILURE);
             }
%%
/* Treci deo dokumenta je prazan, jer main program pripada
 * parseru. Uz pomoc flex-a generisemo leksicki analizator koji
 * tokenizuje ulazni niz karaktera, tj. samo funkciju yylex()
 * koju linkujemo sa parsersom.
 */
```

Rešenje 7.2: Specifikacija leksičkog analizatora

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "tokeni.h"
/* makro kojim omogucavamo ispisivanje koraka u izvodjenja */
#define DEBUG (1)
/* preduvidni simbol */
int preduvid = 0;
/* deklarisemo funkciju koja tokenizuje ulaz */
extern int yylex();

/* funkcije pridruzene neterminalima */
void e(void);
void ep(void);
void t(void);
void tp(void);
void f(void);

void greska(char* s){
  fprintf(stderr,"Sintaksna greska: %s\n",s);
  exit(EXIT_FAILURE);
}

/* glavni program */
int main() {
    /* ucitavamo prvi preduvidni simbol */
    preduvid = yylex();
```

```
29      /* startujemo sintaksnu analizu */
        e();
31
        /* ako smo stigli do kraja ulaza, sve je ok */
33      if (preduvid == EOI) {
            printf("Recenica je u jeziku\n");
35      }
        else {
37          /* inace prijavljujemo gresku */
            printf("Recenica nije u jeziku\n");
39      }

41      exit(EXIT_SUCCESS);
}
43
void e(void) {
45    /* ako je preduvidni simbol iz skupa izbora */
    if(preduvid == BROJ || preduvid == OZ){
47  #ifndef DEBUG
    printf("e --> t ep\n");
49  #endif
        /* pozivamo funkcije koje odgovaraju pravilu izvodjenja */
51      t();
        ep();
53    }
    /* ako preduvidni simbol nije iz skupa izbora
55       prijavljujemo gresku */
    else
57      greska("(e) Ocekivan je token BROJ ili (.");
}
59
void ep(void){
61    if (preduvid == PLUS){
  #ifndef DEBUG
63    printf("ep --> + t ep\n");
  #endif
65      /* kada naidjemo na token u pravilu izvodjenja, potrebno
         * je da proverimo da li se isti nalazi na ulazu. Ako se
67       * nalazi treba ga "pojedemo", tj. da ucitamo sledeci
         * preduvidni simbol koji nam treba da navodi analizu
69       * za sledeci neterminal koji je na redu.
         */
71      preduvid = yylex();
        /* pozivamo funkcije koje odgovaraju pravilu izvodjenja */
73      t();
        ep();
75    }
    /* ako je preduvidni simbol iz skupa izbora koji odgovara
77   * anulirajucem pravilu
```

```
    */
  else if(preduvid == ZZ || preduvid == EOI){
#ifndef DEBUG
    printf("ep --> eps\n");
#endif
    /* samo prekinemo izvrsavanje funkcije */
    return;
  }
  /* ako preduvidni simbol nije iz skupa izbora
     prijavljujemo gresku */
  else
    greska("(ep) Ocekivan je token + ili ) ili EOI");
}

void t(void){
  if(preduvid == OZ || preduvid == BROJ){
#ifndef DEBUG
  printf("t --> f tp\n");
#endif
    /* pozivamo funkcije koje odgovaraju pravilu izvodjenja */
    f();
    tp();
  }
  else
    greska("(t) Ocekivano je ( ili BROJ.");
}

void tp(void) {
  if(preduvid == PUTA){
#ifndef DEBUG
  printf("tp --> * f tp\n");
#endif
    /* ucitavamo sledeci preduvidni simbol */
    preduvid= yylex();
    f();
    tp();
  }
  else if(preduvid == PLUS || preduvid == ZZ || preduvid ==
   EOI){
#ifndef DEBUG
  printf("tp --> eps\n");
#endif
    /* samo prekinemo izvrsavanje funkcije */
    return;
  }
  else greska("(tp) Ocekivano na ulazu PUTA, PLUS, ZZ ili EOI");
}

void f(void) {
```

```
  if(preduvid == OZ){
#ifndef DEBUG
  printf("f --> ( e )\n");
#endif
    /* ucitavamo sledeci preduvidni simbol */
    preduvid = yylex();
    /* pozivamo funkciju neterminala iz pravila izvodjenja */
    e();
    /* proveravamo da li je preduvidni simbol jednak ocekivanom
     * nakon izvrsavanja funkcije neterminala
     */
  if(preduvid == ZZ){
    /* Jeste, ucitavamo sledeci preduvidni simbol */
    preduvid = yylex();
  }
  else
    greska("(f) Ocekivano je ).");
  }
  else if(preduvid == BROJ){
#ifndef DEBUG
  printf("f --> BROJ\n");
#endif
    /* ucitavamo sledeci preduvidni simbol */
    preduvid = yylex();
  return;
  }
   /* ako preduvidni simbol nije iz skupa izbora */
  else
    greska("(f) Ocekivano je na ulazu OZ ili BROJ.");
}
```

Rešenje 7.3: Specifikacija sintaksnog analizatora

```
parser: parser.o lex.yy.o
  gcc -Wall parser.o lex.yy.o -o parser
parser.o: parser.c tokeni.h
  gcc -Wall -c parser.c -o parser.o
lex.yy.o: lex.yy.c
  gcc -Wall -c lex.yy.c -o lex.yy.o
lex.yy.c: lexer.l tokeni.h
  flex  lexer.l

.PHONY: clean

clean:
  rm  lex.yy.* parser
  rm *.o
```

Rešenje 7.4: Makefile

Makefile je napisan samo u okviru ovog zadatka čisto kao ilustracija. U svim drugim zadacima, Makefile se podrazumeva, ali neće biti eksplicitno navođenjen kao deo rešenja.

Primer 7.3 Napisati sintaksni analizator za gramatiku algebarskih izraza koristeći tehniku rekurzivnog spusta koji će pored provere sintaksne ispravnosti računati i vrednost izraza.

Rešenje. Prethodno implementirani parser može da da odgovor na pitanje da li je izraz sintaksno ispravan ili ne. Vrlo jednostavno možemo da unapredimo parser, tako da može i da izračuna vrednost izraza. Neophodno je da određenim tokenima i neterminalima dodelimo atribute. Vrednosti koje će im se pridruživati tokom leksičke, odnosno sintaksne analize. Pored dodeljivanja atributa, neophodno je i da dodelimo pravila na osnovu kojih će se računati vrednosti atributa kroz gramatiku. Na taj način dobiće se atributska gramatika. Polazeći od gramatike algebarskih izraza 7.1. , dodelićemo atribut `vrednost` tokenu *BROJ* i svim neterminalima. Token će vrednost atributa dobiti prilikom prepoznavanja u lekseru. Neterminali će svoj atribut dobijati i računati kroz sintaksnu analizu. Zato svakom pravilu gramatike pridružujemo atributski deo, akciju koja se odnosi na vrednosti atributa. Da bismo dobro odredili atributske akcije treba da shvatimo da neterminal e izvodi jedan sabirak, a ostatak zbira izračunava ep. Ako ep izvodi ε, dakle ostatak zbira ne postoji, atribut od ep mora biti 0 jer je neutral za sabiranje. Slično je i sa t koji izvodi jedan činilac i ostatak proizvoda izvodi tp. Ako tp izvodi ε, dakle ostatak proizvoda ne postoji, atribut od tp mora biti 1 jer je neutral za množenje. Što se neterminala f tiče, ako izvodi *BROJ* njegov atribut će imati vrednost atributa tokena *BROJ*, a ako izvodi izraz u zagradi, vrednost atributa za f biće ista vrednosti atributa za izraz u zagradi.

$$
\begin{aligned}
e &\to t\ ep && \{e.vrednost = t.vrednost + ep.vrednost\}\\
ep &\to +\ t\ ep && \{ep.vrednost = t.vrednost + ep.vrednost\}\\
&\mid \varepsilon && \{ep.vrednost = 0)\}\\
t &\to f\ tp && \{t.vrednost = f.vrednost + tp.vrednost\}\\
tp &\to *\ f\ tp && \{tp.vrednost = f.vrednost + tp.vrednost\}\\
&\mid \varepsilon && \{tp.vrednost = 1\}\\
f &\to (\ e\ ) && \{f.vrednost + e.vrednost\}\\
&\mid BROJ && \{BROJ.vrednost\}
\end{aligned}
$$

Primetimo da smo ovde koristili desnu asovijativnost zbog desno rekurzivnih pravila. To nam nije problem, jer imamo samo sabiranje i mnozenje. Da smo imali i oduzimanje ili deljenje, morali bi da obezbedimo levu asocijativnost iako imamo desno rekurzivna pravila.

Potrebno je da napravimo minimalne izmene u okviru leksera i parsera iz prethodnog primera. Naime, potrebno je da dodamo globalnu promenljivu , na primer, `yylval` pomoću koje ćemo moći da prenesemo vrednost pridruženu tokenu iz funkcije `yylex` do parsera. Modifikovaćemo funkcije pridružene neterminalima, tako da će po završetku svakak vraćati vrednsot svog atributa funkciji koja ju je pozvala. Vrednosti netreminala ćemo računati korišenjem akcija atributske gramatike. Takvi atributi se nazivaju *sintetizovani*. Izmenjeni leksički analizator i parser slede u nastavku:

```
%option noyywrap
%{
```

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

#include "tokeni.h"
/* navodimo promenljivu pomocu koje mozemo
 * da prenesemo parseru vrednost pridruzenu tokenu
 */
extern int yyval;
%}

%%
"+" { return PLUS;}
"*" { return PUTA;}
"(" { return OZ;}
")" { return ZZ;}
[0-9]+ {
        /* vrednost broja upisemo u promenljivu */
        yyval = atoi (yytext);
        /* parseru vratimo token koji odgovara broju */
        return BROJ;
       }
[ \t\n]   {  /* beline ignorisemo */}

.   { fprintf(stderr,"Leksicka greska: %s\n", yytext);
      exit(EXIT_FAILURE);}
%%
```

Rešenje 7.5: Specifikacija leksičkog analizatora

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "tokeni.h"
/* makro kojim omogucavamo ispisivanje koraka u izvodjenja */
#define DEBUG  1
/* preduvidni simbol */
int preduvid;
/* funkcije pridruzene neterminalima */
int e();
int ep();
int t();
int tp();
int f();

void greska(char* s){
  fprintf(stderr,"Sintaksna greska: %s\n",s);
  exit(EXIT_FAILURE);
}
```

```
/* promenljiva kojom prenosimo vrednost pridruzenu tokenu */
int yyval;

extern int yylex();

/* glavni program */
int main() {
    /* ucitavamo prvi preduvidni simbol */
    preduvid = yylex();

    /* startujemo sintaksnu analizu */
    int rezultat = e();

    /* ako smo stigli do kraja ulaza, sve je ok */
    if (preduvid == EOI) {
        printf("Recenica je u jeziku\n");
        printf("Vrednost izraza je: %d\n", rezultat);
    }
    else {
        /* inace prijavljujemo gresku */
        printf("Recenica nije u jeziku\n");
        greska("Izveden je samo prefiks.");
    }

    exit(EXIT_SUCCESS);
}

int e(){
  if(preduvid == BROJ || preduvid == OZ){
#ifdef DEBUG
  printf("e --> t ep\n");
#endif
    return t() + ep();
  }
  else
    greska("(e) Ocekivan je token BROJ ili OZ.");

  /* Nedostizna naredba, ali je tu zbog upozorenja gcc-a. */
  return -1;
}

int ep(){
  if (preduvid== PLUS){
#ifdef DEBUG
  printf("ep --> + t ep\n");
#endif
    preduvid = yylex();
    return t()+ ep();
  }
```

```
  else if(preduvid == ZZ || preduvid == EOI){
#ifdef DEBUG
    printf("ep --> eps\n");
#endif
      return 0;
  }
  else
    greska("(ep) Ocekivan je token PLUS ili ZZ ili EOI");

  return -1;
}

int t(){
  if(preduvid == OZ || preduvid == BROJ){
#ifdef DEBUG
  printf("t --> f tp\n");
#endif
    return f() *tp();
  }
  else
    greska("(t) Ocekivano je OZ ili BROJ.");

  return -1;
}

int tp(){
  if(preduvid == PUTA){
#ifdef DEBUG
  printf("tp --> * f tp\n");
#endif
    preduvid= yylex();
    return f() * tp();
  }
  else if( preduvid == PLUS || preduvid == ZZ
          || preduvid == EOI ){
#ifdef DEBUG
  printf("tp --> eps\n");
#endif
    return 1;
  }
  else
    greska("(tp) Ocekivano je +, *, ), EOI");
  return -1;
}

int f(){
  if(preduvid == OZ){
  int pom;
#ifdef DEBUG
```

```
119   printf("f --> (e)\n");
#endif
121     preduvid = yylex();
      pom = e();
123     if(preduvid == ZZ)
        preduvid = yylex();
125     else
        greska("(f) Ocekivano je ZZ.");
127
      return pom;
129   }
    else if(preduvid == BROJ){
131 #ifdef DEBUG
    printf("f --> BROJ\n");
133 #endif
      preduvid = yylex();
135     return yyval;
    }
137   else
      greska("(f) Ocekivano je ZZ.");
139
    return -1;
141 }
```

Rešenje 7.6: Specifikacija sintaksnog analizatora

Primer 7.4 Napisati sintaksni analizator za gramatiku algebarskih izraza koristeći tehniku rekurzivnog spusta koji će pored provere sintaksne ispravnosti računati i vrednost izraza. Od podržanih operacija podržano je sabiranje, oduzimanje, množenje i deljenje.

Rešenje. Proširimo prethodnu gramatiku izraza tako da podržava i oduzimanje i deljenje pored sabiranja i množenja. Sabiranje i oduzimanje su istog prioriteta i za njihovo izvođenje biće zadužen isti neterminal ep, slično, tp će izvoditi i deljenje. Uz izmenu gramatike, moramo ažurirati i skupove izbora, da bismo mogli da implementiramo parser tehnikom rekurzivnog spusta.

$$
\begin{array}{rll}
e & \to & t\ ep & \{BROJ, (\} \\
ep & \to & +\ t\ ep & \{+\} \\
 & | & -\ t\ ep & \{-\} \\
 & | & \varepsilon & \{\dashv,)\} \\
t & \to & f\ tp & \{BROJ, (\} \\
tp & \to & *\ f\ tp & \{*\} \\
 & | & /\ f\ tp & \{/\} \\
 & | & \varepsilon & \{+, -, \dashv,)\} \\
f & \to & (\ e\ ) & \{(\} \\
 & | & BROJ & \{BROJ\}
\end{array}
$$

Da bismo računali vrednost potrebne su nam akcije za atributsku gramatiku. Kako su oduzimanje i deljenje, levo asocijativne operacije, ne možemo se računati kao u prethodnom primeru. Za te operacije je jako bitna vrednost izraza pre primene operacije da bismo imali dobar rezultat.

Počnimo od pravila $e \rightarrow\ t\ ep$. Neterminal t je samo član zbira i razlike, to ostaje neterminalu ep da proveri. Dakle, vrednost atributa neterminala t je samo međurezultat, a vrednost celokupnog izraza može samo ep da utvrdi. Jasno je da vrednost atributa neterminala ep zavisi od vrednosti atributa neterminala t i da tu vrednost treba da mu prosledimo. Zbog toga, prilikom pozivanja funkcije koja odgovara neterminalu ep potrebno je da joj prosledimo vrednost atributa neterminala t. Ta prosleđena vrednost se naziva *nasleđeni atribut*.

Posmatrajmo pravilo $ep_1 \rightarrow\ -\ t\ ep_2$. Da bismo lakše razlikovali pravila, označićemo indeksima svaku pojavu neterminala ep. Vrednost atributa neterminala t treba da se oduzme od vrednosti izraza koji je prethodio tokenu $-$. Stoga neterminal ep_1 pre nego što izvede ovo pravilo mora znati vrednost međurezultata koji prethodi ovoj razlici. Vrednost međurezultata će dobijati preko nasleđenog atributa i oduzimaće vrednost atributa neterminala t. Vrednost među rezultata će biti $ep_1.nasledjen - t.vrednost$. To treba bude nasleđen atribut za neterminal ep_2. Dalje računanje će nastaviti ep_2. Kada završi prepoznavanje i izračuna vrednost celokupnog izraza i vratiće ga preko povratne vrednosti, tj. sintetizovanog atributa. Ta vrednost će biti i vrednost atributa neterminala ep_1.

Kada se primenjuje pravilo $ep \rightarrow \varepsilon$, funkcija koja odgovara neterminalu ep dobiće vrednost prethodno prepoznatog dela izraza preko nasleđenog atributa, a ne izvodi ništa. Dakle, vrednost njenog atributa biće nasledjena vrednost. Vratiće tu vrednost, funkciji koja ju je pozvala.

Slično rezonovanje se može primeniti na neterminale t i tp. Nakon što to učinimo, dobićemo gramatiku sa sledećim pridruženim akcijama:

$$
\begin{aligned}
e &\rightarrow t\ ep && \left\{ \begin{array}{c} e.vrednost = ep.vrednost \\ ep.nasledjen = t.vrednost \end{array} \right\} \\
ep &\rightarrow\ +\ t\ ep && \left\{ \begin{array}{c} ep_2.nasledjen = ep_1.nasledjen + t.vrednost \\ ep_1.vrednost = ep_2.vrednost \end{array} \right\} \\
&\ \ |\ -\ t\ ep && \left\{ \begin{array}{c} ep_2.nasledjen = ep_1.nasledjen - t.vrednost \\ ep_1.vrednost = ep_2.vrednost \end{array} \right\} \\
&\ \ |\ \varepsilon && \{ep_1.vrednost = ep_1.nasledjen\} \\
t &\rightarrow f\ tp && \left\{ \begin{array}{c} t.vrednost = tp.vrednost \\ tp.nasledjen = f.vrednost \end{array} \right\} \\
tp &\rightarrow\ *\ f\ tp && \left\{ \begin{array}{c} tp_2.nasledjen = tp_1.nasledjen * f.vrednost \\ tp_1.vrednost = tp_2.vrednost \end{array} \right\} \\
&\ \ |\ /\ f\ tp && \left\{ \begin{array}{c} tp_2.nasledjen = tp_1.nasledjen / f.vrednost \\ tp_1.vrednost = tp_2.vrednost \end{array} \right\} \\
&\ \ |\ \varepsilon && \{tp_1.vrednost = tp_1.nasledjen\} \\
f &\rightarrow (\ e\ ) && \{f.vrednost + e.vrednost\} \\
&\ \ |\ BROJ && \{BROJ.vrednost\}
\end{aligned}
$$

Nasleđen atribut se propagira od korena prema listovima sintaksnog stabla, a sintetizovani od listova prema korenu. Nasleđene atribute prosleđujemo funkcijama preko argumenta,

a sintetizovani funkcije vraćaju preko povratnih vrednosti. Kako se sintaksna analiza radi sa leva na desno tako se i vrednost izraza računa. Vrednost početnog dela izraza propagira se preko argumenta funkcijama kojima je potreban.

Na kraju, ostaje nam samo da implementiramo parser. Da bismo to uradili potrebno je da definišemo tokene za nove operacije, da modifikujemo lekser i da pridružimo akcije funkcijama koje implementiraju neterminale u parseru. Kompletna implementacija je u nastavku:

```
%option noyywrap
%option noinput
%option nounput

%{
  #include "tokeni.h"
  extern int yylval;
%}

%%
"+" return PLUS;
"-" return MINUS;
"*" return PUTA;
"/" return PODELJENO;
"("  return OZ;
")" return ZZ;
[0-9]+ {
  yylval = atoi(yytext);
  return BROJ;
}
"\n" return EOI;
. {fprintf(stderr,"Leksicka greska: Nepoznata leksema %s!\n",
   yytext);
  exit(EXIT_FAILURE);
}
%%
```

Rešenje 7.7: Specifikacija leksičkog analizatora

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "tokeni.h"

#define ISPIS 1

int preduvid;
int e();
int ep(int nasledjen );
int t();
int tp(int nasledjen);
```

```
13 int f();

15 int yylval;
extern int yylex();
17
void greska(char* s){
19   fprintf(stderr,"Sintaksna greska: %s\n",s);
//fprintf(stderr,"Preduvidni simbol: %d\n", preduvid);
21   exit(EXIT_FAILURE);
}
23
int main() {
25   preduvid = yylex();
   int vrednost = e();
27
   if( preduvid == EOI ){
29     printf("\n\nRecenica odgovara jeziku zadatim
    gramatikom!\n");

31     printf("\nVrednost izraza je %d\n", vrednost);
   }
33   else
     printf("\nPrevise tokena na ulazu!\n");
35
   return 0;
37 }

39 int e() {
   if (preduvid == OZ || preduvid == BROJ) {
41 #ifdef ISPIS
   printf("e -> t ep\n");
43 #endif
     /* pamtimo vrednost atributa neterminala t */
45     int medjurezultat = t();
     /* i prosledjujemo je neterminalu ep */
47     return ep(medjurezultat);
   }
49   else
     greska("(e) Ocekivano na ulazu OZ ili BROJ");
51
   return 0;
53 }

55 int ep(int nasledjen) {
   if( preduvid == PLUS) {
57 #ifdef ISPIS
     printf("ep -> + t ep\n");
59 #endif
     /* ucitavamo naredni token */
```

```
61      preduvid = yylex();
        /* racunamo medjurezultat */
63      int rezultat = nasledjen + t();
        /* i prosledjujemo dalje */
65      return  ep(rezultat);
    }
67  else if( preduvid == MINUS) {
#ifdef ISPIS
69      printf("ep -> - t ep\n");
#endif
71      preduvid = yylex();
        /* izracunavamo medjurezultat */
73      int rezultat = nasledjen - t();
        /* i prosledjujemo ga neterminalu ep */
75      return  ep(rezultat);
    }
77  else if (preduvid == ZZ || preduvid == EOI) {
#ifdef ISPIS
79      printf("ep -> eps\n");
#endif
81        /* nasledjeni atribut je vrednost izraza
           * i u isto vreme vrednost atributa neterminala */
83        return nasledjen;
      }
85      else
        greska("(ep) Ocekivano +,-, BROJ ili OZ");
87
  return 0;
89}

91int t() {
  if(preduvid == BROJ || preduvid == OZ) {
93#ifdef ISPIS
      printf("t -> f tp\n");
95#endif
      int rezultat = f();
97    return  tp(rezultat);
  }
99  else
      greska("(t) Ocekivano je (, BROJ");
101
  return 0;
103}

105int tp(int nasledjen) {
  if (preduvid == PUTA) {
107#ifdef ISPIS
      printf("tp -> * f tp\n");
109#endif
```

```
    /* ucitavamo naredni token */
    preduvid = yylex();
    /* izracunavamo medjurezultat */
    int medjurezultat = nasledjen*f();
    /* i prosledjujemo ga neterminalu tp */
    return tp(medjurezultat);
  }
  else if (preduvid == PODELJENO) {
#ifdef ISPIS
    printf("tp -> / f tp\n");
#endif
    /* ucitavamo naredni token */
    preduvid = yylex();
    int medjurezultat = f();
    if(medjurezultat == 0)
      greska("Semanticka greska: Deljenje nulom");

    /* izracunavamo medjurezultat */
    medjurezultat = nasledjen / medjurezultat;
    /* i prosledjujemo ga neterminalu tp */
    return tp(medjurezultat);
  }
  else if (preduvid == PLUS || preduvid == MINUS
            || preduvid == ZZ || preduvid == EOI){
#ifdef ISPIS
      printf("tp -> eps\n");
#endif
      /* nasledjeni atribut je vrednost atributa neterminala */
      return nasledjen;
    }
    else
      greska("(tp) Ocekivano je *, /, +, -, ZZ ili EOI");

  return 0;
}

int f() {
  if (preduvid == OZ) {
#ifdef ISPIS
    printf("f -> ( e )\n");
#endif
    /* ucitavamo naredni token */
    preduvid = yylex();
    /* pamtimo medjurezultat */
    int medjurezultat = e();
    /* proveravamo da li je preduvidni simbol
     * onaj koji ocekujemo
     */
    if (preduvid != ZZ)
```

```
159       greska("(f) Ocekivano je ZZ");

161     /* ucitavamo naredni token */
        preduvid = yylex();
163     /* medjurezultat je vrednost atributa neterminala */
        return medjurezultat;
165   }
      else if (preduvid == BROJ) {
167 #ifdef ISPIS
        printf("f -> BROJ\n");
169 #endif
        /* pamtimo vrednost broja */
171     int tmp = yylval;
        /* ucitavamo naredni token */
173     preduvid = yylex();
        /* vrednost broja je vrednost atributa neterminala */
175     return tmp;
      }
177   else
        greska("(f) Ocekivano je BROJ ili OZ");
179
      return 0;
181 }
```

Rešenje 7.8: Specifikacija sintaksnog analizatora

Primer 7.5 Nadogradimo prethodni sintaksni analizator tako da nam pored, računanja vrednosti ispisuje aritmetički izraz u postfiksnoj notaciji.
Na primer, za `2+3` ispisuje `2 3 +`, a za `2-(3+2*4)/2` ispisuje `2 3 2 4 * + 2 / -` .

Rešenje. Dodajemo pravilima akcije tako što ih pišemo unutar pravila. Tako preciziramo kada se akcije izvrsavaju. Ispis broja treba da se desi kada se broj prepozna unutar funkcije za neterminal f. Ispis operatora treba da se desi nakon prepoznavanja drugog operanda. Zato akciju za ispis $+$ dodajemo unutar pravila $ep \rightarrow + \ t \ ep$ posle završene analize za t, a pre početka analiza za ep. Analogno za ostale operatore. Nadogradjena gramatika izgleda

ovako:

$$
\begin{aligned}
e &\rightarrow t\ ep && \left\{\begin{array}{c} e.vrednost = ep.vrednost \\ ep.nasledjen = t.vrednost \end{array}\right\} \\
ep &\rightarrow +\ t\ \{print(+)\}\ ep && \left\{\begin{array}{c} ep_2.nasledjen = ep_1.nasledjen + t.vrednost \\ ep_1.vrednost = ep_2.vrednost \end{array}\right\} \\
&\mid -\ t\ \{print(-)\}\ ep && \left\{\begin{array}{c} ep_2.nasledjen = ep_1.nasledjen - t.vrednost \\ ep_1.vrednost = ep_2.vrednost \end{array}\right\} \\
&\mid \varepsilon && \{ep_1.vrednost = ep_1.nasledjen\} \\
t &\rightarrow f\ tp && \left\{\begin{array}{c} t.vrednost = tp.vrednost \\ tp.nasledjen = f.vrednost \end{array}\right\} \\
tp &\rightarrow *\ f\ \{print(*)\}\ tp && \left\{\begin{array}{c} tp_2.nasledjen = tp_1.nasledjen * f.vrednost \\ tp_1.vrednost = tp_2.vrednost \end{array}\right\} \\
&\mid /\ f\ \{print(/)\}\ tp && \left\{\begin{array}{c} tp_2.nasledjen = tp_1.nasledjen / f.vrednost \\ tp_1.vrednost = tp_2.vrednost \end{array}\right\} \\
&\mid \varepsilon && \{tp_1.vrednost = tp_1.nasledjen\} \\
f &\rightarrow (\ e\ ) && \{f.vrednost + e.vrednost\} \\
&\mid BROJ\{print(BROJ.vrednost)\} && \{BROJ.vrednost\}
\end{aligned}
$$

Sve izmene su napravljene u datoteci `parser.c`, implementacija u nastavku:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "tokeni.h"

// #define ISPIS 1

int preduvid;
int e();
int ep(int nasledjen );
int t();
int tp(int nasledjen);
int f();

int yylval;
extern int yylex();

void greska(char* s){
  fprintf(stderr,"Sintaksna greska: %s\n",s);
//fprintf(stderr,"Preduvidni simbol: %d\n", preduvid);
  exit(EXIT_FAILURE);
}

int main() {
  preduvid = yylex();
  int vrednost = e();

```

```
  if( preduvid == EOI ){
    printf("\n\nRecenica odgovara jeziku zadatim
     gramatikom!\n");
    printf("\nVrednost izraza je %d\n", vrednost);
  }
  else
    printf("\nPrevise tokena na ulazu!\n");

  return 0;
}

int e() {
  if (preduvid == OZ || preduvid == BROJ) {
#ifdef ISPIS
  printf("e -> t epu\n");
#endif
    int medjurezultat = t();
    return ep(medjurezultat);
  }
  else
    greska("(e) Ocekivano na ulazu OZ ili BROJ");

  return 0;
}

int ep(int nasledjen) {
  if( preduvid == PLUS) {
#ifdef ISPIS
    printf("ep -> + t ep\n");
#endif
    preduvid = yylex();
    int rezultat = nasledjen + t();
    printf(" +");

    return  ep(rezultat);
  }
  else if( preduvid == MINUS) {
#ifdef ISPIS
    printf("ep -> - t ep\n");
#endif
    preduvid = yylex();
    int rezultat = nasledjen - t();
    printf(" -");

    return  ep(rezultat);
  }
  else if (preduvid == ZZ || preduvid == EOI) {
#ifdef ISPIS
    printf("ep -> eps\n");
```

```
#endif
77        return nasledjen;
      }
79      else
        greska("(ep) Ocekivano +,-, BROJ ili OZ");
81
    return 0;
83  }

85  int t() {
    if(preduvid == BROJ || preduvid == OZ) {
87  #ifdef ISPIS
      printf("t -> f tp\n");
89  #endif
      int rezultat = f();
91      return  tp(rezultat);
    }
93    else
      greska("(t) Ocekivano je (, BROJ");
95
    return 0;
97  }

99  int tp(int nasledjen) {
    if (preduvid == PUTA) {
101 #ifdef ISPIS
      printf("tp -> * f tp\n");
103 #endif
      preduvid = yylex();
105     int medjurezultat = nasledjen * f();
      printf(" *");
107
      return tp(medjurezultat);
109   }
    else if (preduvid == PODELJENO) {
111 #ifdef ISPIS
      printf("tp -> / f tp\n");
113 #endif
      preduvid = yylex();
115     int medjurezultat = f();
      if(medjurezultat == 0)
117       greska("Semanticka greska: Deljenje nulom");

119     printf(" /");
      medjurezultat = nasledjen / medjurezultat;
121
      return tp(medjurezultat);
123   }
    else if (preduvid == PLUS || preduvid == MINUS
```

```
125             || preduvid == ZZ || preduvid == EOI){
#ifdef ISPIS
127       printf("tp -> eps\n");
#endif
129
        return nasledjen;
131     }
      else
133       greska("(tp) Ocekivano je *, /, +, -, ZZ ili EOI");

135   return 0;
}
137
int f() {
139   if (preduvid == OZ) {
#ifdef ISPIS
141     printf("f -> ( e )\n");
#endif
143
      preduvid = yylex();
145     int medjurezultat = e();

147     if (preduvid != ZZ)
        greska("(f) Ocekivano je ZZ");
149
      preduvid = yylex();
151
      return medjurezultat;
153   }
    else if (preduvid == BROJ) {
155 #ifdef ISPIS
      printf("f -> BROJ\n");
157 #endif

159     int tmp = yylval;
      printf(" %d", yylval);
161     preduvid = yylex();

163     return tmp;
    }
165   else
      greska("(f) Ocekivano je BROJ ili OZ");
167
    return 0;
169 }
```

Rešenje 7.9: Specifikacija sintaksnog analizatora

Primer 7.6 Naredni primer na kojem ćemo ilustrovati sintaksnu analizu naniže uz pomod rekurzivnog spusta je izmišljeni programskim jezik sličan Paskalu, *mini paskal*. Primer koda koji bi trebao da prođe našu sintaksnu analizu.

```
while ( a > 5 ) do
begin
  if ( b < 3) then
    i := 1;
  z := 34;
end
```

Rešenje. Uočavamo da je naš program sastavljen iz niza naredbi razdvojenih karakterom ;. Nakon toga, potrebno je da opišemo naredbe. Naredba može da bude dodela, petlja, grananje, neki blok ili prazna naredba. Prazna naredba je u primeru između naredbe dodele za z i ključne reči *end*. Pre pisanja same gramatike, potrebno je odrediti azbuku tokena. Azbuku tokena čine ključne reči, identifikator, sepratori, terminatori, relacioni i aritmetički operatori, zagrade, znak dodele i brojevi. Kada smo sve ovo definisali, gramatika može da bude sledeća:

$$
\begin{aligned}
\mathit{nizNaredbi} &\rightarrow \mathit{nizNaredbi\ TZ\ naredba} \\
&\mid \mathit{naredba} \\
\mathit{naredba} &\rightarrow \mathit{WHILE\ uslov\ DO\ naredba} \\
&\mid \mathit{IF\ uslov\ THEN\ naredba} \\
&\mid \mathit{ID\ DODELA\ izraz} \\
&\mid \mathit{BEGIN\ nizNaredbi\ END} \\
&\mid \varepsilon \\
\mathit{uslov} &\rightarrow \mathit{OZ\ izraz\ ROP\ izraz\ ZZ} \\
\mathit{izraz} &\rightarrow \mathit{ID} \mid \mathit{BROJ}
\end{aligned}
$$

Očigledno, gramatiku prvo treba da transformišemo tako da bude $LL(1)$, da bismo mogli da implementiramo parser. Prvi korak je eliminacija leve rekurzije, zatim određivanje anulirajućih simbola, računanje skupova *Prvi* i *Sledi*, određivanje skupova izbora i konačno, implementacija parsera. Transformisana gramatika je sledeća:

$$
\begin{aligned}
\mathit{nizNaredbi} &\rightarrow \mathit{naredba\ nizNaredbiP} \\
\mathit{nizNaredbiP} &\rightarrow \mathit{TZ\ naredba\ nizNaredbiP} \\
&\mid \varepsilon \\
\mathit{naredba} &\rightarrow \mathit{WHILE\ uslov\ DO\ naredba} \\
&\mid \mathit{IF\ uslov\ THEN\ naredba} \\
&\mid \mathit{ID\ DODELA\ izraz} \\
&\mid \mathit{BEGIN\ nizNaredbi\ END} \\
&\mid \varepsilon \\
\mathit{uslov} &\rightarrow \mathit{OZ\ izraz\ ROP\ izraz\ ZZ} \\
\mathit{izraz} &\rightarrow \mathit{ID} \mid \mathit{BROJ}
\end{aligned}
$$

Kada odredimo anulirajuće simbole, vidimo da su svi neterminali osim izraza i uslova anulirajuci. Imamo mogućnost da gramatiku transformisemo i oslobodimo anulirajucih

simbola. Ako bismo hteli za ovu gramatiku da se oslobodimo ε pravila, dobili bismo levo-faktorisana pravila. Njih moramo ukloniti jer nam treba $LL(1)$ gramatika. Oslobađanjem od leve faktorizacije, uvešćemo ponovo ε pravila. Zato to ovde nećemo raditi. Kako zadražavamo sve anulirajuće simbole, onda prilikom određivanja skupova *Prvi* i *Sledi* moramo voditi računa da li je neki od neterminala u pravilu anulirajući.

Neterminal	ε	Prvi	Sledi
nizNaredbi	Da	WHILE, IF, ID, BEGIN, TZ	⊣, END
nizNaredbiP	Da	TZ	⊣, END
naredba	Da	WHILE, IF, ID, BEGIN	TZ, ⊣, END
uslov	/	OZ	DO, THEN
izraz	/	ID, BROJ	ROP, ZZ, TZ, ⊣, END

Tabela 7.4: Utvrđeni skupovi *Prvi*, *Sledi* i anulirajući neterminali

Određujemo skupove izbora. Skup izbora za pravilo neterminala *nizNaredbi* mora da objedini skupove *Prvi* i *Sledi* tog neterminala jer je to jedno pravilo, zapravo anulirajuće u dva koraka izdvođenja.

$$
\begin{aligned}
nizNaredbi &\rightarrow naredba\ nizNaredbiP && \{WHILE, IF, ID, BEGIN, TZ, EOI, END\}\\
nizNaredbiP &\rightarrow TZ\ naredba\ nizNaredbiP && \{TZ\}\\
&\mid \varepsilon && \{\dashv, END\}\\
naredba &\rightarrow WHILE\ uslov\ DO\ naredba && \{WHILE\}\\
&\mid IF\ uslov\ THEN\ naredba && \{IF\}\\
&\mid ID\ DODELA\ izraz && \{ID\}\\
&\mid BEGIN\ nizNaredbi\ END && \{BEGIN\}\\
&\mid \varepsilon && \{TZ, \dashv, END\}\\
uslov &\rightarrow OZ\ izraz\ ROP\ izraz\ ZZ && \{OZ\}\\
izraz &\rightarrow ID && \{ID\}\\
&\mid BROJ && \{BROJ\}
\end{aligned}
$$

Sintaksni analizator za gramatiku *mini paskala* zasnovan na rekurzivnom spustu prikazan je u nastavku:

```
#ifndef _TOKENI_H
#define _TOKENI_H

/* yylex vraca 0 kada dodje od kraja ulazne datoteke
 * Marker za kraj ulaza uvek mora imati vrednost 0. */
#define EOI 0
#define ID 1
#define BROJ 2
#define WHILE_TOKEN 3
#define IF_TOKEN 4
#define TZ 5
#define OZ 6
#define ZZ 7
#define DODELA 8
```

```
#define BEGIN_TOKEN 9
#define END_TOKEN 10
#define ROP 11
#define DO_TOKEN 12
#define THEN_TOKEN 13

#endif
```

Rešenje 7.10: Specifikacija tokena

```
%option noyywrap
%option nounput
%option noinput

%{
#include "tokeni.h"
%}

%%
while   return WHILE_TOKEN;
if    return IF_TOKEN;
begin   return BEGIN_TOKEN;
end   return END_TOKEN;
do    return DO_TOKEN;
then  return THEN_TOKEN;

"("   return OZ;
")"   return ZZ;
":="  return DODELA;
[<>]=?|=|"<>"   return ROP;

[a-z]+  return ID;
[0-9]+  return BROJ;
;     return TZ;

[\n\t ] {}
.     { fprintf(stderr, "Nepoznata leksema %s.\n",yytext);
      exit(EXIT_FAILURE);}
%%
```

Rešenje 7.11: Specifikacija leksičkog analizatora

```
#include <stdio.h>
#include <stdlib.h>
#include "tokeni.h"

#define _DEBUG

int preduvid;
```

```
/* F-ja ispisuje gresku i prekida program */
void greska(char* s){
  fprintf(stderr,"Sintaksna greska: %s\n",s);
  exit(EXIT_FAILURE);
}

extern int yylex();

void nizNaredbi();
void naredba();
void nizNaredbiP();
void uslov();
void izraz();

void nizNaredbi(){
    if(preduvid == WHILE_TOKEN || preduvid == IF_TOKEN
      || preduvid == ID || preduvid == BEGIN_TOKEN
      || preduvid == TZ || preduvid == EOI
      || preduvid == END_TOKEN){
#ifdef _DEBUG
    printf("nizNaredbi-->naredba nizNaredbiP\n");
#endif
    naredba();
    nizNaredbiP();
  }
  else
    greska("(nizNaredbi) Ocekivano je nesto od:
   WHILE,IF,ID,BEGIN,TZ,EOI,END");
}

void naredba(){
  if(preduvid == WHILE_TOKEN){
#ifdef _DEBUG
    printf("naredba--> WHILE uslov DO naredba\n");
#endif
    preduvid = yylex();
    uslov();
    if(preduvid != DO_TOKEN)
      greska("(naredba) Ocekivano DO!");

    preduvid = yylex();
    naredba();
  }
  else if (preduvid == IF_TOKEN){
#ifdef _DEBUG
    printf("naredba--> IF uslov THEN naredba\n");
#endif
    preduvid = yylex();
```

```
    uslov();
    if(preduvid != THEN_TOKEN)
      greska("(naredba) Ocekivano THEN!");
    preduvid = yylex();
    naredba();
  }
  else if(preduvid == ID){
#ifdef _DEBUG
    printf("naredba--> ID DODELA izraz\n");
#endif
    preduvid = yylex();
    if(preduvid != DODELA)
      greska("Ocekivano :=!");
    preduvid = yylex();
    izraz();
  }
  else if (preduvid == BEGIN_TOKEN){
#ifdef _DEBUG
    printf("naredba--> BEGIN nizNaredbi END\n");
#endif
    preduvid = yylex();
    nizNaredbi();
    if(preduvid != END_TOKEN)
      greska("(naredba) Ocekivano END!");
    preduvid =yylex();
  }
  else if(preduvid == TZ || preduvid == EOI
          || preduvid == END_TOKEN){
#ifdef _DEBUG
    printf("naredba --> eps\n");
#endif
    return;
  }
  else{
    char pom[30];
    sprintf(pom,"(naredba) Neocekivan token %d!", preduvid);
    greska(pom);
  }
}

void nizNaredbiP(){
  if(preduvid == TZ){
#ifdef _DEBUG
    printf("nizNaredbiP --> TZ naredba nizNaredbiP\n");
#endif
    preduvid = yylex();
    naredba();
    nizNaredbiP();
  }
```

```
105   else if(preduvid == EOI || preduvid == END_TOKEN){
#ifdef _DEBUG
107     printf("nizNaredbiP--> eps\n");
#endif
109     return;
    }else
111     greska("(nizNaredbiP) Ocekivano je: TZ ili EOI,ili END!");
}
113
void uslov(){
115   if(preduvid == OZ){
#ifdef _DEBUG
117     printf("uslov--> OZ izraz ROP izraz ZZ\n");
#endif
119     preduvid = yylex();
        izraz();
121     if(preduvid != ROP)
          greska("(uslov) Ocekivano ROP!");
123     preduvid = yylex();
        izraz();
125     if(preduvid != ZZ)
          greska("(uslov) Ocekivano ZZ!");
127     preduvid = yylex();
    }
129   else
        greska("(uslov) Ocekivana je OZ!");
131 }

133 void izraz(){
    if(preduvid == ID ){
135 #ifdef _DEBUG
        printf("izraz--> ID\n");
137 #endif
        preduvid =yylex();
139   }
    else if(preduvid ==BROJ){
141 #ifdef _DEBUG
        printf("izraz--> BROJ\n");
143 #endif
        preduvid =yylex();
145   }
    else
147     greska("(izraz) Ocekivano je ID ili BROJ!");
}
149
int main(){
151   preduvid = yylex();

153   nizNaredbi();
```

```
155   if(preduvid != EOI)
        greska("\nVisak tokena na ulazu!\n\n");
157
      printf("\nSintaksno ispravan kod!\n\n");
159   return 0;
}
```

Rešenje 7.12: Specifikacija sintaksnog analizatora

7.3 Potisni automati

Do sada smo prilikom implementiranja parsera zasnovanom na rekurzivnom spustu implicitno koristili sistemski stek, tj. simultanim rekurzivnim pozivima funkcija koje smo dodelili neterminalima smo simulirali izvođenje u gramatici. Sistemski stek nam je služio da pamtimo neterminale tokom izvođenja, a tokene smo jednostavno čitali sa ulaza i obrađivali ih tako što učitamo sledeći kada nam trenutni više ne treba. Ovakav pristup je vrlo jednostavan za implementiranje, jer gotovo u potpunosti prati strukturu gramatike, ali zbog velikog broja rekurzivnih poziva, tj. otvaranje stek okvira funkcija nije baš efikasan.

Drugi pristup kojim možemo da implementiramo sintaksni analizator naniže jeste da izbegnemo korišćenje sistemskog steka, tj. rekurzivne pozive funkcija i da umesto toga kompletnu strukturu podataka održavamo u okviru svog programa. U tom slučaju, potrebno je da napravimo svoju implementaciju strukture podataka stek na kojem ćemo pamtiti sve korake u izvođenju.

Umesto funkcija, neterminalima pridružujemo stanja koja ćemo čuvati na steku. Pored neterminala na steku moramo da čuvamo i tokene, pa treba da vodimo računa da su skup vrednosti koje pridružujemo stanjima i skup vrednosti koje pridružujemo tokenima disjunktni, jer ih stavljamo na isti stek. Parsiranje pomoću potisnog automata svodi se na proveravanje vrha steka. Ako je vrh steka prazan, stigli smo da kraja izvršavanja, tj. ulaz je sintaksno ispravan. U suprotnom treba da utvrdimo da li se radi o neterminalu ili tokenu. Ako se radi o neterminalu, skidamo neterminal sa vrha steka i u zavisnosti od preduvidnog simbola, zamenjujemo neterminal odgovarajućom desnom stranom pravila izvođenja i nastavljamo parsiranje. Ako se radi o tokenu, upoređujemo vrh steka sa preduvidnim simbolom i ako su identični skidamo token sa vrha steka i učitavamo naredni preduvidni simbol, a ako nisu identični prijavljujemo sintaksnu grešku. Na početku rada, potrebno je da učitamo preduvidni simbol i da na vrh steka postavimo startni simbol gramatike. Parser implementiran na ovaj način se naziva potisni automat. Da bismo mogli da implementiramo potisni automat potrebno je da izvršimo sve one korake koji su bili neophodni i za rekurzivni spust:

1. Smisliti gramatiku koja opisuje zadati jezik.
2. Eliminisati leva rekurzivna i levo faktorisana pravila.
3. Odrediti skupove *Prvi* i *Sledi*.
4. Na osnovu skupova *Prvi* i *Sledi* i odrediti skupove izbora. Skupovi izbora se određuju tako što svim anulirajućim granama neterminala pridružimo tokene iz skupa *Sledi* tog neterminala, a svim produktivnim granama neterminala dodelimo tokene iz skupa

tog neterminala, osim ako pravilo nije anulirajuće u više od jednog koraka, kada treba objediniti skup *Sledi* sa celim ili podskupom skupa *Prvi*.

5. Nakon određivanja skupova izbora potrebno je da proverimo da li je gramatika $LL(1)$, tj. da li su disjunktni skupovi izbora za pravila istog neterminala. Ako jesu, onda možemo jednoznačno da utvrdimo koje pravilo izvođenja treba da primenimo samo na osnovu jednog preduvidnog simbola.

Tek kada izvršimo svih pet koraka možemo da implementiramo potisni automat. Bitno je primetiti da su potisni automat i rekruzivni spust samo dve različite tehnike implementiranja prediktivnih parsera za klase gramatika $LL(k)$, tj. razlikuju se samo u tome da li koriste stek implicitno ili eksplicitno.

Primer 7.7 Implementaciju potisnog automata ćemo ilustrovati na gramatici algebarskih izraza.

Rešenje. Transformisana gramatika izraza sa skupovima izbora je već ranije određena u 7.1. Prilikom implementacije potisnog automata, potrebno je uvek imati na umu da se radi o steku, tj. *LIFO* strukturi podataka. Zbog toga, kada neterminal zamenjujemo njegovom desnom stranom pravila, potrebno je da simbole stavljamo na stek s desna na levo.

```
#ifndef TOKENI_H
#define TOKENI_H

/* definicija tokena */
#define EOI     0
#define PLUS    1
#define PUTA    2
#define BROJ    3
#define OZ      4
#define ZZ      5

/* definicija neterminala */
#define E       100
#define EP      101
#define T       102
#define TP      103
#define F       104

#endif
```

Rešenje 7.13: Specifikacija tokena

```
%option noyywrap
%option noinput
%option nounput

%{
    #include "tokeni.h"
%}
```

```
%%

[0-9]+    { return BROJ; }
"+"       { return PLUS; }
"*"       { return PUTA; }
"("       { return OZ; }
")"       { return ZZ; }
[ \t\n]   { }
.      {
         fprintf(stderr, "Leksicka greska: %s\n", yytext);
         exit(EXIT_FAILURE);
       }
%%
```

Rešenje 7.14: Specifikacija leksičkog analizatora

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include "tokeni.h"
#define _DEBUG 1

/* maksimalna dubina steka*/
#define MAX_SIZE 256
int preduvid = 0;

extern int yylex();

void greska(char *s) {
  fprintf(stderr, "%s\n", s);
  exit(EXIT_FAILURE);
}

int stek[MAX_SIZE];
/* vrh steka -> prva slobodna pozicija,
 * raste ka visim adresama */
int sp = 0;

/* funkcija proverava da li je stek prazan */
int empty() {
  return sp == 0 ? 1 : 0;
}
/* funkcija proverava da li je stek pun */
int full() {
  return sp == MAX_SIZE ? 1 : 0;
}
/* funkcija skida simbol sa vrha steka */
int pop() {
```

```
33    if (empty()) {
        greska("Stek je prazan");
35    }
      return stek[--sp];
37  }

39  /* funkcija stavlja simbol s na stek */
    void push(int s) {
41    if (full()) {
        greska("Stek je pun");
43    }
      stek[sp++] = s;
45  }
    /* funkcija vraca simbol sa vrha steka */
47  int top() {
      if (empty()) {
49      greska("Stek je prazan");
      }
51    return stek[sp - 1];
    }
53

    /* funkcija proverava da li su preduvidni simbol i x jednaki */
55  int match(int x) {
      return preduvid == x;
57  }

59  /* funkcija cita sledeci token sa ulaza */
    void advance(){
61    preduvid = yylex();
    }
63

    /* funkcija ispisuje vrednosti na steku prilikom debugovanja*/
65  void print_stek() {
      int i = sp - 1;
67    for (; i >= 0; i--)
        printf("%d ", stek[i]);
69    printf("\n");
    }
71

    /* glavni program */
73  int main(){
      /* ucitavamo preduvidni simbol i stavljamo ga na vrh steka */
75    advance();
      push(E);
77

      /* sve dok stek nije prazan */
79    while (!empty()) {
        /* u zavisnosti od simbola na vrhu steka */
81      switch (top()) {
```

```
      /* ako se radi o neterminalu E */
      case E:
        /* proveramo preduvidni simbol */
        if (match(BROJ) || match(OZ)) {
#ifdef _DEBUG
          printf("E -->T EP\n");
          print_stek();
#endif
          /* skidamo neterminal sa vrha steka */
          pop();
          /* i zamenjujemo ga desnom stranom pravila
           * i to u smeru s desna na levo
           */
          push(EP);
          push(T);
        }
        /* ako preduvidni simbol ne pripada skupu izbora */
        else {
          /* prijavljujemo gresku */
          greska("(E) Ocekivano je BROJ ili (");
        }
        break;
      /* ako se radi o neterminalu EP */
      case EP:
        /* proveramo preduvidni simbol */
        if (match(PLUS)) {
#ifdef _DEBUG
          printf("EP --> + T EP\n");
          print_stek();
#endif
          /* skidamo neterminal sa vrha steka */
          pop();
          /* i zamenjujemo ga desnom stranom pravila
           * i to u smeru s desna na levo
           */
          push(EP);
          push(T);
          push(PLUS);
        }
        /* proveramo preduvidni simbol */
        else if (match(ZZ) || match(EOI)){
#ifdef _DEBUG
          printf("EP --> eps\n");
          print_stek();
#endif
          /* skidamo neterminal sa vrha steka */
          pop();
          /* u epsilon grani, ne stavljamo nista na stek */
        }
```

```
131           /* ako preduvidni simbol ne pripada skupu izbora */
              else {
133             /* prijavljujemo gresku */
                greska("(EP) Ocekivano +, ) ili EOI");
135           }
              break;
137         /* ako se radi o neterminalu T */
            case T:
139           /* proveramo preduvidni simbol */
              if (match(BROJ) || match(OZ)) {
141 #ifdef _DEBUG
                printf("T --> F TP\n");
143             print_stek();
    #endif
145             /* skidamo neterminal sa vrha steka */
                pop();
147             /* i zamenjujemo ga desnom stranom pravila
                 * i to u smeru s desna na levo
149              */
                push(TP);
151             push(F);
              }
153           /* ako preduvidni simbol ne pripada skupu izbora */
              else {
155             /* prijavljujemo gresku */
                greska("(T) Ocekivano je BROJ ili )");
157           }
              break;
159         /* ako se radi o neterminalu TP */
            case TP:
161           /* proveramo preduvidni simbol */
              if (match(PUTA)) {
163 #ifdef _DEBUG
                printf("TP --> * F TP\n");
165             print_stek();
    #endif
167             /* skidamo neterminal sa vrha steka */
                pop();
169             /* i zamenjujemo ga desnom stranom pravila
                  * i to u smeru s desna na levo
171               */
                push(TP);
173             push(F);
                push(PUTA);
175           }
              /* proveramo preduvidni simbol */
177           else if (match(PLUS) || match(ZZ) || match(EOI)){
    #ifdef _DEBUG
179             printf("TP -->eps\n");
```

```
            print_stek();
#endif
            /* skidamo neterminal sa vrha steka */
            pop();
            /* u epsilon grani, ne stavljamo nista na stek */
          }
            /* ako preduvidni simbol ne pripada skupu izbora */
          else {
            /* prijavljujemo gresku */
            greska("(TP) Ocekivano je *, +, ) ili EOI");
          }
            break;
        /* ako se radi o neterminalu F */
        case F:
          /* proveramo preduvidni simbol */
          if (match(BROJ)) {
#ifdef _DEBUG
            printf("F --> BROJ\n");
            print_stek();
#endif
          /* skidamo neterminal sa vrha steka */
            pop();
            push(BROJ);
          }
          /* proveramo preduvidni simbol */
          else if (match(OZ)) {
#ifdef _DEBUG
            printf("F --> ( E )\n");
            print_stek();
#endif
            /* skidamo neterminal sa vrha steka */
            pop();
            /* i zamenjujemo ga desnom stranom pravila
              * i to u smeru s desna na levo
              */
            push(ZZ);
            push(E);
            push(OZ);
          }
          /* ako preduvidni simbol ne pripada skupu izbora */
          else {
            /* prijavljujemo gresku */
            greska("(F) Ocekivano je broj ili (");
          }
          break;
        /* ako se radi o tokenu */
        default:
          /* proveravamo da li se token sa vrha steka
          * poklapa sa preduvidnim simbolom
```

```
229           */
              if (match(top())){
231             /* skidamo neterminal sa vrha steka */
                pop();
233             /* ucitavamo sledeci preduvidni simbol */
                advance();
235           }
              /* ako se tokeni ne poklapaju */
237           else {
                /* prijavljujemo gresku */
239             greska("Pogresan token na ulazu");
              }
241           break;
            }
243     }

245     /* ako je stek prazan, analiza je gotova, a da li je uspesna
         * zavisi od toga da li na ulazu ima jo[ tokena ili ne */
247     if(preduvid != EOI){
          greska("Pprepoznat samo prefiks izraza!\n");
249     }

251     printf("Prepoznat je aritmeticki izraz!\n");
        exit(EXIT_SUCCESS);
253 }
```

Rešenje 7.15: Sintaksni analizator immplementiran simulacijom rada potisnog autotama

Primer 7.8 Proširiti funkcionalnost sintaksnog analizatora iz prethodnog primera, tako da računa i vrednost izraza.

Rešenje. Zarad racunanja vrednosti, izraz se prevodi u postfiksnu notaciju. Potom se korišćenjem steka vrednosti računa vrednost izraza na isti način kao što bi se to radilo sa izrazom u postfiksnoj notaciji. Na primer, za izraz $2+3*5$ postfiksna notacija bi glasila `2 3 5 * +`. Kada se prepozna bilo koji broj stavljaće se na stek vrednosti, kada se prepozna operator $*$, oba potrebna operanda biće već na steku vrednosti. Potrebno je da ih skinemo sa steka, primenimo operaciju množenje i rezultat vratimo na stek. Taj rezultat će biti drugi operand kada se bude pročitalo $+$ sa ulaza. Bitno je da vodimo računa da nam je prvi operand skinut sa steka vrednosti zapravo drugi operand u operaciji. Stoga sa minimalno truda možemo implementirati i podršku za levo asocijativne operatore oduzimanja i deljenja.

Prevođenje u postfiksnu notaciju se postiže sledećom shemom.

$$
\begin{aligned}
e &\to t\ ep \\
ep &\to +\ t\ \{\ print(+)\ \}\ ep \\
&\quad |\ -\ t\ \{\ print(-)\ \}\ ep \\
&\quad |\ \varepsilon \\
t &\to f\ tp \\
tp &\to *\ f\ \{\ print(*)\ \}\ tp \\
&\quad |\ /\ f\ \{\ print(/)\ \}\ tp \\
&\quad |\ \varepsilon \\
f &\to (\ e\ ) \\
&\quad |\ BROJ\ \{print(BROJ.vrednost)\}
\end{aligned}
$$

Akcija u sredini pravila treba da se dogodi nakon što neterminal t obavi svoj deo analize, a pre nego što analizu nastavi ep. Zato uvodimo pomoćne neterminale. Nisu bili neophodni za sintaksnu analizu, ali su neophodni za semantički deo.

$$
\begin{aligned}
e &\to t\ ep \\
ep &\to +\ t\ a1\ ep \\
&\quad |\ -\ t\ a2\ ep \\
&\quad |\ \varepsilon \\
t &\to f\ tp \\
tp &\to *\ f\ a3\ tp \\
&\quad |\ /\ f\ a4\ tp \\
&\quad |\ \varepsilon \\
f &\to (\ e\ ) \\
&\quad |\ BROJ \qquad \{\ print(BROJ.vrednost)\ \} \\
a1 &\to \varepsilon \qquad\qquad\qquad \{\ print(+)\ \} \\
a2 &\to \varepsilon \qquad\qquad\qquad \{\ print(-)\ \} \\
a3 &\to \varepsilon \qquad\qquad\qquad \{\ print(*)\ \} \\
a4 &\to \varepsilon \qquad\qquad\qquad \{\ print(/)\ \}
\end{aligned}
$$

Ukoliko zelimo i efektivno da se izracunava vrednost uvodi se pored steka za parsiranje u stek vrednost i akcije za neterminale $a1$ do $a4$ se menjaju.

$$
\begin{aligned}
a1 &\to \varepsilon \quad \{\ a = pop();\ b = pop();\ push(a+b);\ print(+)\ \} \\
a2 &\to \varepsilon \quad \{\ a = pop();\ b = pop();\ push(b-a);\ print(-)\ \} \\
a3 &\to \varepsilon \quad \{\ a = pop();\ b = pop();\ push(a*b);\ print(*)\ \} \\
a4 &\to \varepsilon \quad \{\ a = pop();\ b = pop();\ push(b/a);\ print(/)\ \}
\end{aligned}
$$

```
#ifndef TOKENI_H
#define TOKENI_H 1

#define EOI        0
#define BROJ       1
```

```
#define PLUS      2
#define PUTA      3
#define OZ        4
#define ZZ        5
#define MINUS     6
#define PODELJENO 7

#define E   101
#define EP  102
#define T   103
#define TP  104
#define F   105

#define A1  106
#define A2  107
#define A3  108
#define A4  109

#endif
```

Rešenje 7.16: Specifikacija tokena

```
%option noyywrap
%option nounput
%option noinput

%{
#include "tokeni.h"

extern int yylval;
%}

%%
[0-9]+  { yylval = atoi(yytext);
          return BROJ; }
"+"     return PLUS;
-       return MINUS;
"*"     return PUTA;
"/"     return PODELJENO;
"("     return OZ;
")"     return ZZ;
[ \t]   {}
\n      return EOI;
.   {
      fprintf(stderr,"Nepoznata leksema (%s)\n",yytext);
      exit(EXIT_FAILURE);
    }
%%
```

Rešenje 7.17: Specifikacija leksičkog analizatora

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

#include "tokeni.h"

#define MAX_DEPTH 256
// #define ISPIS 1
#define ISPIS_POSTFIX 2
extern int yylex();


int preduvid;
int yylval;

void greska(char * s){
  fprintf(stderr,"Greska: %s",s);
  printf("preduvid %d\n",preduvid);

  exit(EXIT_FAILURE);
}

/* Koristicemo istu strukturu jer i stek za parsiranje
 * i stek vrednosti treba da sadrze cele brojeve.
 */
typedef struct{
  int array[MAX_DEPTH];
  int sp ;
} Stack;

/* funkcija proverava da li je stek prazan */
int empty(Stack stack) {
  return stack.sp <= 0 ? 1 : 0;
}
/* funkcija proverava da li je stek pun */
int full(Stack stack) {
  return stack.sp >= MAX_DEPTH ? 1 : 0;
}
/* funkcija skida simbol sa vrha steka
 * dobija pokazivac jer treba da ga menja */
int pop(Stack *stack) {
  if (empty(*stack)) {
    greska("Stek je prazan");
  }
  return stack->array[--stack->sp];
}

/* funkcija stavlja simbol s na stek */
void push(Stack *stack, int s) {
  if (full(*stack)) {
```

```
50      greska("Stek je pun");
    }
52    stack->array[stack->sp++] = s;
}
54  /* funkcija vraca simbol sa vrha steka */
int top(Stack stack) {
56    if (empty(stack)) {
        greska("Stek je prazan");
58    }
    return stack.array[stack.sp - 1];
60  }

62  /* funkcija proverava da li je preduvidni simbol jednak simbolu
      x */
int match(int x) {
64    return preduvid == x;
}
66  /* funkcija cita sledeci token sa ulaza */
void advance(){
68    preduvid = yylex();
}
70
/* funkcija ispisuje vrednosti na steku prilikom debugovanja*/
72  void print_stack(Stack stack) {
    int i = stack.sp - 1;
74    printf("< ");
    for (; i >= 0; i--)
76      printf("%d ", stack.array[i]);
    printf(" >\n");
78  }

80  int main(){
    Stack parse_stack;
82    parse_stack.sp = 0;
    Stack value_stack;
84    value_stack.sp = 0;

86    advance();
    push(&parse_stack,E);
88
    print_stack(parse_stack);
90    while(!empty(parse_stack)){
      switch(top(parse_stack)){
92      case E:
            if(preduvid == BROJ || preduvid == OZ){
94            pop(&parse_stack);
              push(&parse_stack,EP);
96            push(&parse_stack,T);
#ifdef ISPIS
```

```
98              printf("E -> T EP\n");
#endif
100           }
              else
102             greska("E: Ocekivano BROJ ili (\n");
              break;
104         case EP:
            if(preduvid == PLUS){
106           pop(&parse_stack);
              push(&parse_stack,EP);
108           push(&parse_stack,A1);
              push(&parse_stack,T);
110           push(&parse_stack,PLUS);
#ifdef ISPIS
112           printf("EP -> + T A1 EP\n");
#endif
114         }
            else if(preduvid == MINUS){
116           pop(&parse_stack);
              push(&parse_stack,EP);
118           push(&parse_stack,A2);
              push(&parse_stack,T);
120           push(&parse_stack,MINUS);
#ifdef ISPIS
122           printf("EP -> - T A2 EP\n");
#endif
124         }
            else if(preduvid == EOI || preduvid == ZZ){
126           pop(&parse_stack);
#ifdef ISPIS
128           printf("EP -> eps\n");
#endif
130         }
            else
132           greska("EP: Ocekivano +, -, ), EOI\n");
            break;
134         case T:
            if(preduvid == BROJ || preduvid == OZ){
136           pop(&parse_stack);
              push(&parse_stack,TP);
138           push(&parse_stack,F);
#ifdef ISPIS
140           printf("T -> F TP\n");
#endif
142         }
            else
144           greska("T(): Ocekivano BROJ, (\n");
            break;
146         case TP:
```

```
        if(preduvid == PUTA){
148       pop(&parse_stack);
          push(&parse_stack,TP);
150       push(&parse_stack, A3);
          push(&parse_stack,F);
152       push(&parse_stack,PUTA);
#ifdef ISPIS
154       printf("TP -> * F A3 TP\n");
#endif
156     }
        else if(preduvid == PODELJENO){
158       pop(&parse_stack);
          push(&parse_stack,TP);
160       push(&parse_stack, A4);
          push(&parse_stack,F);
162       push(&parse_stack,PODELJENO);
#ifdef ISPIS
164       printf("TP -> / F A4 TP\n");
#endif
166     }
        else if(preduvid == PLUS || preduvid == MINUS
168       || preduvid == ZZ || preduvid == EOI){
          pop(&parse_stack);
170 #ifdef ISPIS
          printf("TP -> eps\n");
172 #endif
        }
174     else greska("TP(): Ocekivano +,*,-,/,),EOI\n");
        break;
176   case F:
        if(preduvid == BROJ){
178       pop(&parse_stack);
          push(&parse_stack,BROJ);
180 #ifdef ISPIS
          printf("F -> BROJ\n");
182 #endif
    #ifdef ISPIS_POSTFIX
184       printf("%d ",yylval);
    #endif
186       push(&value_stack,yylval);
        }
188     else if (preduvid == OZ){
          pop(&parse_stack);
190       push(&parse_stack,ZZ);
          push(&parse_stack,E);
192       push(&parse_stack,OZ);
#ifdef ISPIS
194       printf("F -> ( E )\n");
#endif
```

```
196           }
              else
198             greska("F: ocekuje broj ili (");
              break;
200         case A1:{
              int a, b;
202           pop(&parse_stack);
#ifdef ISPIS
204           printf("A1 -> eps");
#endif
206 #ifdef ISPIS_POSTFIX
              printf("+ ");
208 #endif
              b = pop(&value_stack);
210           a = pop(&value_stack);
              push(&value_stack,a+b);
212           break;
            }
214         case A2:{
              int a,b;
216           pop(&parse_stack);
#ifdef ISPIS
218           printf("A2 -> eps");
#endif
220 #ifdef ISPIS_POSTFIX
              printf("- ");
222 #endif
              b = pop(&value_stack);
224           a = pop(&value_stack);
              push(&value_stack,a-b);
226           break;
            }
228         case A3:{
              int a,b;
230           pop(&parse_stack);
#ifdef ISPIS
232           printf("A3 -> eps");
#endif
234 #ifdef ISPIS_POSTFIX
              printf("* ");
236 #endif
              b = pop(&value_stack);
238           a = pop(&value_stack);
              push(&value_stack,a*b);
240           break;
            }
242         case A4:{
              int a,b;
244           pop(&parse_stack);
```

```
#ifdef ISPIS
246         printf("A4 -> eps");
#endif
248 #ifdef ISPIS_POSTFIX
         printf("/ ");
250 #endif
         b = pop(&value_stack);
252      if(b==0)
           greska("Semanticka greska: Deljenje nulom.\n");
254      a = pop(&value_stack);
         push(&value_stack,a/b);
256      break;
       }
258    default:
         if(match(top(parse_stack))){
260        pop(&parse_stack);
           advance();
262      }
         else switch(top(parse_stack)){
264        case PUTA:  greska("Ocekivano je *\n");
           case PLUS:  greska("Ocekivano je +\n");
266        case MINUS:  greska("Ocekivano je -\n");
           case PODELJENO:  greska("Ocekivano je /\n");
268        case BROJ:  greska("Ocekivano je BROJ\n");
           case OZ:    greska("Ocekivano je (\n");
270        case ZZ:    greska("Ocekivano je )\n");
         }
272    }
  }
274
  if(preduvid != EOI){
276   greska("\nPrepoznat samo prefiks izraza!\n");
  }
278
  printf("\nVrednost izraza je  %d\n",top(value_stack));
280
  return 0;
282 }
```

Rešenje 7.18: Sintaksni analizator immplementiran simulacijom rada potisnog autotama

8. Sintaksna analiza naviše

U prethodnom poglavlju je bilo reči o sintaksnoj analizi naniže. Videli smo da se u toj analizi polazi od startnog simbola i pokušava da generiše drvo izvođenja počevši od korena ka listovima, tj. ka niski tokena. Pored ovakvog pristupa, postoji i pristup koji kreće od listova i pokušava da generiše drvo izvođenja prema korenu, tj. kreće od niske tokena i pokušava da izvede startni simbol. Takav pristup naziva se *sintaksna analiza naviše*.

8.1 Konstrukcija potisnog automata

Sintaksni analizator koji sintaksnu analizu vrši analizom naviše se implementira pomoću potisnog automata. Stek je na početku prazan. Pre nego što navedemo prvi primer, uvešćemo dve akcije koje svaki potisni automat može da izvrši:

1. Prebacivanje (*eng. shift*) – prebacivanje tokena sa ulaza na stek.
2. Svođenje (*eng. reduce*) – kada automat na vrhu potisne liste prepozna neku nisku tokena i neterminala koja odgovara desnoj strani nekog pravila, tada automat kompletnu potisnu listu zamenjuje levom stranom prepoznatog pravila.

Postupak izgradnje sintaksnog stabla na ovaj način generiše izvođenje na desno, pa levo rekurzivna pravila ne predstavljaju nikakav problem, čak su i poželjna. U sintaksnoj analizi naviše proces sintaksne analize teče u suprotnom smeru, tj. od niske tokena ka početnom simbolu gramatike.

Da bismo ilustrovali postupak analize naviše iskoristićemo gramatiku izraza.

Primer 8.1 Opisati rad potisnog automata koji odgovara gramatici algebarskih izraza koji bi se koristio prilikom sintaksne analize naviše. Koristiti gramatiku algebarskih

izraza o kojoj je bilo reči u prethodnim poglavljima.

$$
\begin{aligned}
e &\to e + t \\
&\mid t \\
t &\to t * f \\
&\mid f \\
f &\to (e) \\
&\mid \textit{BROJ}
\end{aligned}
$$

Nećemo eliminisati levu rekurziju. Neka je na ulazu, na primer, izraz $2+3*5$. Tablica 8.1 predstavlja simulaciju rada potisnog automata:

Stek	**Ulaz**	**Akcija**
	2 + 3 * 5	Prebacivanje tokena BROJ na stek
BROJ	+ 3 * 5	Svođenje po pravilu $f \to \textit{BROJ}$
f	+ 3 * 5	Svođenje po pravilu $t \to f$
t	+ 3 * 5	Svođenje po pravilu $e \to t$
e	+ 3 * 5	Prebacivanje tokena + na stek
e +	3 * 5	Prebacivanje tokena BROJ na stek
e + BROJ	* 5	Svođenje po pravilu $f \to \textit{BROJ}$
e + f	* 5	Svođenje po pravilu $t \to f$
e + t	* 5	Prebacivanje tokena * na stek
e + t*	5	Prebacivanje tokena BROJ na stek
e + t * BROJ	⊣	Svođenje po pravilu $f \to \textit{BROJ}$
e + t * f	⊣	Svođenje po pravilu $t \to t * f$
e + t	⊣	Svođenje po pravilu $e \to e + t$
e	⊣	Automat je prihvatio izraz sa ulaza.

Tabela 8.1: Simulacija rada potisnog automata tokom sintaksne analize naviše

Stigli smo do kraja ulaza i startni simbol je jedini simbol na steku, što znači da je niska uspešno prihvaćena. Ako spojimo čitav stek i ulaz u jednu rečeničnu formu, gledano odozdo na gore, dobićemo izvođenje na desno:

$$
\begin{aligned}
&e \Rightarrow e + t \Rightarrow e + t * f \Rightarrow e + t * \textit{BROJ} \Rightarrow e + f * \textit{BROJ} \Rightarrow e + \textit{BROJ} * \textit{BROJ} \\
&\Rightarrow t + \textit{BROJ} * \textit{BROJ} \Rightarrow f + \textit{BROJ} * \textit{BROJ} \Rightarrow \textit{BROJ} + \textit{BROJ} * \textit{BROJ}
\end{aligned}
$$

Međutim, potrebno je da pažljivo pogledamo korak u prethodnoj analizi koji je uokviren linijama. Radi se o situaciji u kojoj na steku imamo $e+t$ i na ulazu nisku tokena $*5$. U ovom koraku, mogli smo da izaberemo i operaciju svođenja po pravilu $e \to e+t$ umesto prebacivanja tokena $*$ sa ulaza na stek. U tom slučaju, na steku bismo neterminal e i token $*$, a na ulazu bismo imali token *BROJ*. Ništa drugo ne bismo mogli nego i njega da prebacimo na stek. Tada bi smo jedino mogli da primenimo svođenje po prvilu $f \to \textit{BROJ}$ i potom po pravilu $t \to f$. Nakon toga bismo na steku imali $e*t$. Kako ne postoji pravilo sa takvom desnom stranom i dobili bismo sintaksnu grešku, iako je ulaz validan. Dakle, time u što smo u označenom koraku u tabeli 8.1 izabrali da uradimo operaciju prebacivanja umesto svođenja, faktički smo namestili izvođenje koje nam odgovara.

Ovakva situacija se naziva *Shift-Reduce konfilkt*. Automat koji ima konflikte nije deterministički. Zbog toga ćemo suziti skup na gramatike koje pripadaju klasi $LR(1)$, za koje može da se konstruiše deterministički potisni automat. Gramatike koje pripadaju klasi $LR(1)$ su one koje omogućavaju jednoznačno određivanje koju akciju treba primeniti u sledećem koraku izvođenja na osnovu jednog preduvidnog simbola.

Primer 8.2 Objasniti postupak konstrukcije determinističkog potisnog automata koji sintaksnom analizom naviše može da utvrdi da li niska pripada jeziku opisanom gramatikom na primeru uprošćene gramatike izraza:

$$\begin{aligned} e &\rightarrow \quad e \ + \ t \\ &\mid t \\ t &\rightarrow BROJ \end{aligned}$$

Prvi korak je da uvedemo novi startni simbol e_0, čime omogućavamo izvođenje startnog simbola samo u slučaju kada prihvatamo ulaz. Takođe, neophodno je da izvršimo numeraciju pravila da bismo znali po kom pravilu iz gramatike vršimo svođenje. Transformisana gramatika biće:

$$\begin{aligned} e_0 &\rightarrow e && (0) \\ e &\rightarrow e \ + \ t && (1) \\ &\mid t && (2) \\ t &\rightarrow BROJ && (3) \end{aligned}$$

Pored uvođenja novog startnog simbola i numeracije pravila, potrebno je da uvedemo koncept *ajtema* (*eng. item*). Ajtem predstavlja mesto do kog se stiglo sa analizom ulaza. U gramatici ajtem se označava pravilom koje u sebi sadrži simbol $\cdot$. Na primeru pravila (0) iz gramatike, ajtem $e_0 \rightarrow \ \cdot \ e$ označava da smo na početku prepoznavanja izraza koji treba neterminal e da izvede. Dok, ajtem $e_0 \rightarrow e \ \cdot$ označava da je prepoznat ceo izraz sa ulaza, koji je netermnial e već izveo. Stanja nedeterminističkog automata biće ajtemi. Na slici 8.1 prikazan je nedeterministički potisni automat koji odgovara uprošćenoj gramatici izraza.

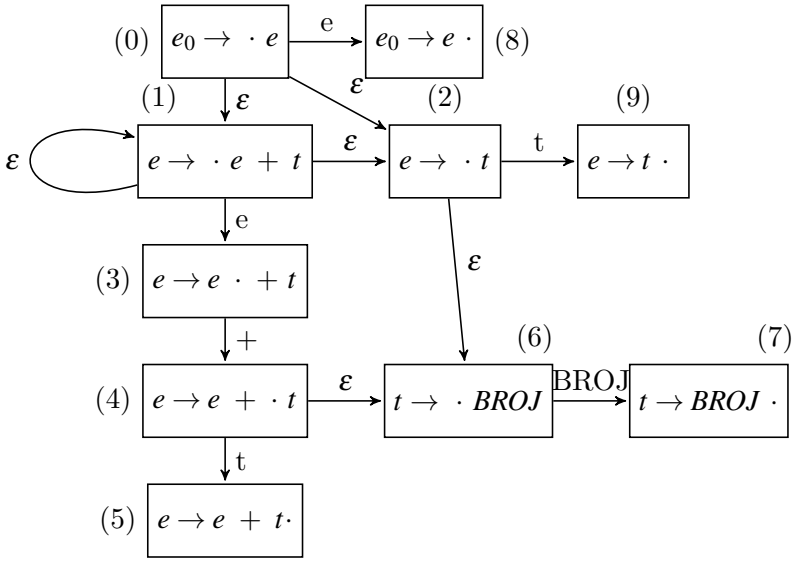
Brojevi u zagradama predstavljuju numeraciju stanja potisnog automata i ne treba ih mešati sa numeracijom pravila u gramatici. U nastavku možemo da probamo da simuliramo rad potisnog automata sa slike.

Ilustrovaćemo rad automata sa slike 8.1 na ulaznom niz tokena $BROJ + BROJ$. Simulacija rada automata prikazana je u tabeli 8.2. Neposredno pre svođenja, podvučen je deo koji će prilikom svođenja biti skinut sa steka i zamenjen.

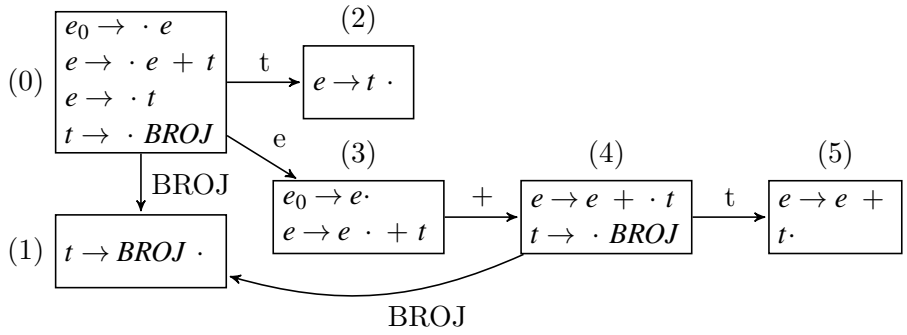
Očigledno, nedeterminizam nije nešto što želimo da imamo u svom programu, pa je automat potrebno determinizovati. Srećom, automat od starta možemo da pravimo deterministički. Da bismo to uradili potrebno je da posmatramo ε-zatvorenja stanja. Preciznije, ako je stanju odgovara ajtem u kom se simbol $\cdot$ nalazi ispred nekog neterminala, tada u to stanje treba dodati ajteme za sva pravila tog neterminala takva da je simbol $\cdot$ na početku pravila. Konstrukcija determinističkog potisnog automata je na prikazan na slici 8.2.

Ispratimo rad ovog automata na istom izrazu $BROJ + BROJ$, prikazanog u tabeli 8.3.

Konstruisani potisni automat ima manje stanja nego prethodni automat. Nema epsilon prelaza, ali i dalje postoji konflikt u stanju 3. U tabeli 8.3 uokvirena je situacija kada je



Slika 8.1: Nedeterministički potisni automat za sintaksnu analizu naviše



Slika 8.2: Potisni automat za sintaksnu analizu naviše bez ε prelaza

Stek	**Ulaz**	**Akcija**
0	BROJ + BROJ	Prelaz po ε
0 ε 1	BROJ + BROJ	Prelaz po ε
0 ε 1 ε 2	BROJ + BROJ	Prelaz po ε
0 ε 1 ε 2 ε 6	BROJ + BROJ	Prebacivanje tokena BROJ na stek
0 ε 1 ε 2 $\underline{\varepsilon\ 6\ \text{BROJ}\ 7}$	+ BROJ	Svođenje po pravilu 3
0 ε 1 $\underline{\varepsilon\ 2\ \text{t}\ 9}$	+ BROJ	Svođenje po pravilu 2
0 ε 1 e 3	+ BROJ	Prebacivanje tokena + na stek
0 ε 1 e 3 + 4	BROJ	Prelaz po ε
0 ε 1 e 3 + 4 ε 6	BROJ	Prebacivanje tokena BROJ na stek
0 ε 1 e 3 + 4 $\underline{\varepsilon\ 6\ \text{BROJ}\ 7}$	⊣	Svođenje po pravilu 3
0 $\underline{\varepsilon\ 1\ \text{e}\ 3\ +\ 4\ \text{t}\ 5}$	⊣	Svođenje po pravilu 1
$\underline{0\ \text{e}\ 8}$	⊣	Svođenje po pravilu 0 (*Accept*)
e_0	⊣	Automat je prihvatio izraz

Tabela 8.2: Simulacija rada nedeterminističkog potisnog automata tokom sintaksne analize naviše

Stek	**Ulaz**	**Akcija**
0	BROJ + BROJ	Prebacivanje tokena BROJ na stek
0 $\underline{\text{BROJ}\ 1}$	+ BROJ	Svođenje po pravilu 3
0 $\underline{\text{t}\ 2}$	+ BROJ	Svođenje po pravilu 2
0 e 3	+ BROJ	Prebacivanje tokena + na stek
0 e 3 + 4	BROJ	Prebacivanje tokena BROJ na stek
0 e 3 + 4 $\underline{\text{BROJ}\ 1}$	⊣	Svođenje po pravilu 3
0 $\underline{\text{e}\ 3\ +\ 4\ \text{t}\ 5}$	⊣	Svođenje po pravilu 1
$\underline{0\ \text{e}\ 3}$	⊣	Svođenje po pravilu 0 (Accept)
e_0	⊣	Automat je prihvatio izraz

Tabela 8.3: Simulacija rada nedeterminističkog potisnog automata tokom sintaksne analize naviše

automat u stanju 3. Do tada je prepoznat izraz koji je izveo e. To znamo na osnovu ajtema i činjenice da imamo e na steku. Moglo bi da se izvrši svođenje po pravilu $e_0 \to e$ i da se tako prihvati prepoznat izraz ili da se izvrši prebacivanje tokena + na stek. Ukoliko bismo izabrali svođenje, pogrešili bismo jer bi automat izveo prefiks izraza sa ulaza, a ne ceo ulaz, i imali bismo sintaksnu grešku za ispravan algebarski izraz. Dok, videli smo u tabeli 8.3, prebacivanje + na stek vodi ka uspešnoj sintaksnoj analizi. Trenutno, automat ne sadrži mehanizam kojim bi jednoznačno odredio da li u stanju 3 da izvrši svođenje po pravilu 0 iz gramatike ili učita naredni token. Da bismo automat proglasili determinističkim potrebno je da razrešimo sve konflikte. Pod razrešavanjem konflikata podrazumevamo da su akcije svođenja i prebacivanja na stek jednoznačno određene u svakom stanju automata.

Najjednostavniji način kako možemo da pokušamo da razrešimo konflikte jeste da uzmemo u obzir preduvidni simbol. Uz pomoć preduvidnog simbola, možemo da definišemo za stanje kada radimo akciju svođenja, a kada akciju prebacivanja tokena sa ulaza na stek. Jedan od načina pridruživanja akcija stanju može da bude sledeći:

- Akciju prebacivanja izvodimo onda kada se na ulazu nađe token kojim automat prelazi u novo stanje.

- Ukoliko stanju odgovara ajtem $X \to \alpha \cdot$, akciju svođenja po pravilu $X \to \alpha$, izvodimo onda kada se na ulazu nađe token koji pripada skupu $Sledi(X)$.

Da bismo pokušali ovako da razrešimo konflikt u našem automatu, potrebno je da prvo izračunamo skupove *Sledi* za neterminale iz gramatike. Prikazani su u tablici 8.4.

Neterminal	Sledi
e_0	$\dashv$
e	+, $\dashv$
t	+,$\dashv$

Tabela 8.4: Utvrđeni skupovi *Prvi Sledi* i anulirajući neterminali

Pridružujući akcije na ovaj način, razrešićemo konflikt u stanju 3. Naime, akcija svođenja se izvršava onda kada se na ulazu nađe kraj ulaza (tj. $\dashv$), a akcija prebacivanja tokena na stek se radi onda kada se na ulazu nađe token +. S obzirom da su skupovi tokena za ove dve akcije disjunktni, ponašenje automata u stanju 3 je jednoznačno određeno, pa je automat deterministički.

Osim grafički, automat možemo da predstavimo i tablično. Tablica kojom se predstavlja automat se naziva *Action-GoTo* tablica. Action deo se pridružuje tokenima, a GoTo deo se pridružuje neterminalima. Potrebno je da zapišemo šta se dešava u svakom stanju automata po svakom tokenu i neterminalu. U Action delu tablice nalaze se novi simboli :

s_i - označava prelazak u stanje i ako se na ulazu nađe odgovarajući token;
r_i - označava svođenje po pravilu gramatike sa rednim brojem i, ako se na ulazu nađe odgovarajući token.

U GoTo delu tablice se nalaze prelasci po neterminalima u nova stanja automata i taj deo tablice se gleda nakon svake akcije svođenja, da bi se prešlo u novo stanje automata. Action/GoTo tablica za naš automat data je u tabeli 8.5. Primetimo da je svođenje po pravilu sa rednim brojem 0 r_0, zapisano kao *acc* da označi akciju prihvatanja prihvatanja ulaza.

Stanje	Action			GoTo	
	BROJ	+	$\dashv$	e	t
0	s_1			3	2
1		r_3	r_3		
2		r_1	r_2		
3		s_4	*acc*		
4	s_1				5
5		r_1	r_1		

Tabela 8.5: Action/GoTo tablica za potisni automat

Da ponovo napomenemo, izuzetno je važno da se ne mešaju numeracija stanja automata i pravila u gramatici. S obzirom da smo utvrdili da automat nema konflikte nakon što smo pridružili akcije, prikazaćemo simulaciju rada konstruisanog automata, koristeći podatke iz Action/GoTo tablice 8.5.

Stek	Ulaz	Akcija
0	BROJ + BROJ	Prebacivanje tokena BROJ na stek i prelazak u stanje 1
0 BROJ 1	+ BROJ	Svođenje po pravilu 3 i prelazak u stanje 2 (iz Goto dela)
0 t 2	+ BROJ	Svođenje po pravilu 2 i prelazak u stanje 3 (iz Goto dela)
0 e 3	+ BROJ	Prebacivanje tokena + na stek i prelazak u stanje 4
0 e 3 + 4	BROJ	Prebacivanje tokena BROJ na stek i prelazak u stanje 1
0 e 3 + 4 BROJ 1	⊣	Svođenje po pravilu 3 i prelazak u stanje 5 (iz Goto dela)
0 e 3 + 4 t 5	⊣	Svođenje po pravilu 1 i prelazak u stanje 3 (iz Goto dela)
0 e 3	⊣	Svođenje po pravilu 0 (Accept)
e_0	⊣	Automat je prihvatio izraz

Tabela 8.6: Simulacija rada determinističkog potisnog automata tokom sintaksne analize naviše

Primer 8.3 Pojasniti prednost leve rekurzije za sintaksnu analizu naviše na primeru potisnog automata za jezik $L = b^*$.

Jezik $L = b^*$ bismo mogli opisati sledećim gramatikama:

Levo rekurzivna	Desno rekurzivna
$A \to A\ b$	$A \to b\ A$
$A \to \varepsilon$	$A \to \varepsilon$

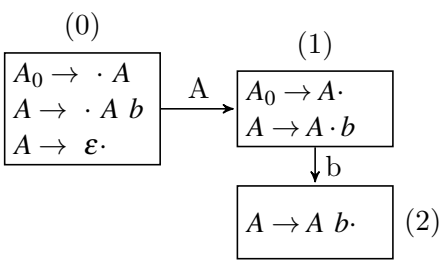
Da bismo mogli da implementiramo odgovarajuće potisne automate, potrebno je da uvedemo novi startni simbol u koji ne možemo da se vratimo tokom izvođenja gramatika. Potom treba da transformišemo polazne gramatike i numerišemo njihova pravila. Transformisane gramatike su date u tabeli 8.7.

Proširene gramatike			
Levo rekurzivna		Desno rekurzivna	
$A_0 \to A$	(0)	$A_0 \to A$	(0)
$A \to A\ b$	(1)	$A \to b\ A$	(1)
$A \to \varepsilon$	(2)	$A \to \varepsilon$	(2)

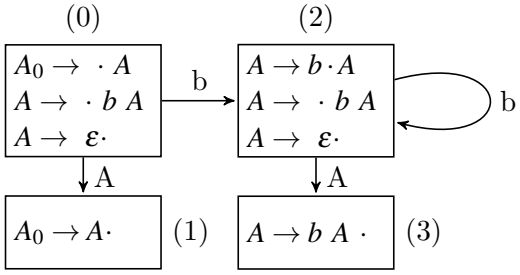
Tabela 8.7: Proširene gramatike i numerisana pravila za izgradnju potisnog automata

Nakon transformacije gramatika, nacrtaćemo determinstčke konačne automate koji im odgovaraju. Na slici 8.3 prikazan je automat koji odgovara levo rekurzivnoj gramatici, a na slici 8.4 prikazan je automat koji odgovara desno rekurzivnoj gramatici.

Ni jedan ni drugi automat nisu deterministički, jer oba imaju shift-reduce konflikte. Automat sa slike 8.3 ima konflikt u stanju 1, a automat sa slike 8.4 ima konflikt u stanju



Slika 8.3: Levo rekurzivna gramatika



Slika 8.4: Desno rekurzivna gramatika

Slika 8.5: Automati za gramatike jezika $b*$

0. Pokušaćemo da ih razrešimo tako što akcijama prebacivanja dodelimo tokene po kojima automat prelazi u novo stanje, a akcijama svođenja pridružimo tokene iz skupa *Sledi* neterminala čije je pravilo po kom se vrši svođenje. Na ovaj način dobijamo sledeće Action/GoTo tablice:

Stanje	Action		GoTo
	b	⊣	A
0	r_2	r_2	1
1	s_2	r_0	
2	r_1	r_1	1

Tabela 8.8: Levo rekurzivne gramatike

Stanje	Action		GoTo
	b	⊣	A
0	s_2	r_2	1
1		r_0	
2	s_2	r_2	3
3		r_1	1

Tabela 8.9: Desno rekurzivne gramatike

Tabela 8.10: Action/GoTo tablice za potisne automate

Uz pomoć tablica možemo lako da izvršimo simulaciju rada potisnog automata za ulaz Tabele 8.11 i 8.12 prikazuju simulaciju izvršavanja.

Stek	**Ulaz**	**Akcija**
0	bbbb	Svođenje po pravilu 2
0 A 1	bbbb	Prebacivanje b na stek
0 A 1 b 2	bbb	Svođenje po pravilu 1
0 A 1	bbb	Prebacivanje b na stek
0 A 1 b 2	bb	Svođenje po pravilu 1
0 A 1	bb	Prebacivanje b na stek
0 A 1 b 2	b	Svođenje po pravilu 1
0 A 1	b	Prebacivanje b na stek
0 A 1 b 2	⊣	Svođenje po pravilu 1
0 A 1	⊣	Svođenje po pravilu 0
A_0	⊣	Automat prihvata ulaz

Tabela 8.11: Simulacije rada automata levo rekurzivne gramatike za ulaz *bbbb*

Upoređujući ova dva izvršavanja, očigledno je da je stek značajno dublji kada se izvršava desno rekurzivna gramatika. Razlog tome leži u činjenica da kod desno rekurzivng pravila prvo moramo da prebacimo čitavu nisku na stek da bismo mogli da izvršimo operaciju svođenja. U slučaju levo rekurzivne gramatike postupak izvršavanja se svodi na sukcesivnu

Stek	**Ulaz**	**Akcija**
0	bbbb	Prebacivanje b na stek
0 b 2	bbb	Prebacivanje b na stek
0 b 2 b 2	bb	Prebacivanje b na stek
0 b 2 b 2 b 2	b	Prebacivanje b na stek
0 b 2 b 2 b 2 b 2	⊣	Svodjenje po pravilu 2
0 b 2 b 2 b 2 b 2 A 3	⊣	Svodjenje po pravilu 1
0 b 2 b 2 b 2 A 3	⊣	Svodjenje po pravilu 1
0 b 2 b 2 A 3	⊣	Svodjenje po pravilu 1
0 b 2 A 3	⊣	Svodjenje po pravilu 1
0 A 1	⊣	Svođenje po pravilu 0
A_0	⊣	Automat prihvata ulaz

Tabela 8.12: Simulacije rada automata desno rekurzivne gramatike za ulaz *bbbb*

primenu operacija prebacivanja i svođenja, pa stek nije nikada bio dublji od 2.

Kod gramatika vidimo celu desnu stranu pravila i još jedan simbol, preduvidni token. Na osnovu toga, donosimo odluku koji je sledeći korak u izvođenju. Kod gramatika vidimo samo jedan simbol sa ulaza i na osnovu toga treba da donesemo zaključak koji je sledeći korak analize. Jasno je da je sintaksna analiza na više mnogo moćnija. U nastavku teksta, nećemo se baviti teorijskim implementacijama potisnih automata i analizom naviše, već ćemo koristiti generator sintaksnih analizatora koji se naziva *bison*.

8.2 Generator sintaksnih analizatora – *yacc/bison*

Bison je računarski program koji generiše sintaksni analizator. Dostupan je besplatno na *Linux* operativnim sistemima. U slučaju da koristimo neku od *Ubuntu* distribucija, instalira se uz pomoć sledeće dve komande:

```
sudo apt-get update
sudo apt-get install bison
```

Komande je neophodno da izvršimo ovim redom da bismo instalirali poslednju dostupnu verziju paketa za našu verziju operativnog sistema.

Da bismo generisali sintaksni analizator za naš problem, potrebno je prvo da napišemo specifikaciju kontekstno slobodne gramatike u *bison* sintaksi. Nakon toga, uz pomoć *bison*-a možemo da generišemo C/C++ kod programa koji predstavlja naš sintaksni analizator. Sintaksni analizator čita ulazni niz tokena i odlučuje da je taj niz saglasan sa definisanom gramatikom. Najčešće, *bison* se uparuje sa leksičkim analizatorima generisanim od strane *flex*-a. Leksički analizator tokenizuje ulazni niz karaktera i prosleđuje ih kao ulaz za sintaksni analizator generisan uz pomoć *bison*-a.

Specifikacija sintaksnog analizatora se *bison*-u zadaje u vidu skripa koji je podeljen na tri dela razdvojena sa %%:

Segment definicija - sadrži komande *bison*-u, C/C++ kod koji se direktno kopira u izvršni program i opciono defincije tokena, neterminala i njima pridruženih tipova podataka.

Segment akcija - predstavlja opis kontekstno slobodne gramatike i akcija dodeljenih svakom neterminalu.

Segment korisničkog koda - sadrži C/C++ kod koji se direktno kopira u izvršni program.

```
// segment definicija
%%
// segment akcija
%%
// segment korisničkog koda
```

Bitno je primetiti da par karaktera %% mora da bude na početku reda i da mora da bude jedini sadržaj tog reda. Između dva procenta ne sme da se nađe razmak ili bilo kakav drugi znak. Skript mora da bude podeljen na ova tri segmenta, inače ne predstavlja validan ulaz za *bison*.

Pre prelaska na primere, zadržaćemo se na segmentu definicija i segmentu akcija, jer su oni najvažniji za pravilnu implementaciju parsera, jer u njima zapravo pišemo opis sintaksnog analizatora. Opis sintaksnog analizatora čine definicije tokena i neterminala, kao i njima pridruženih vrednosti, zatim definicija prioriteta operatora i na kraju definicija gramatičkih pravila i akcija pridruženih svakom pravilu. U segmentu definicija navode se tokeni, neterminali, njima pridružene vrednosti navođenjem tipova podataka i opisuju se prioriteti operatora. Segment akcija služi za definisanje gramatičkih pravila i njima pridruženih akcija.

```
%{
    /* C/C++ kod koji se direktno prenosi u parser */
%}

/* definicije sintaksnog analizatora */

%%
```

Segment definicija možemo da podelimo na dva dela. Prvi deo obuhvata C/C++ kod koji se direktno prenosi u parser, a drugi deo obuhvata segment definicija samog sintaksnog analizatora.

Definicije sintaksnog analizatora u opštem slučaju sadrže sledeće:

- Definicije tokena i neterminala i njima pridruženih vrednosti.
- Povezivanje promenljivih koje će čuvati vrednosti atributa tokena i neterminala.
- Opis prioriteta operatora.
- Proglašavanje startnog simbola gramatike.

Ukoliko imamo atributske akcije u gramatici, pre definisanja tokena i neterminala potrebno je da odredimo tipove podataka i promenljive koje možemo da dodelimo tokenima i neterminalima. Podrazumevan tip atributa svih tokena i neterminala je `int`. Ukoliko imamo potrebe za drugim tipovima podataka za atribute potrebno je da definišemo uniju koja će objediniti sve potrebne tipove podataka. Postižemo to koristeći sledeću komandu *bison*-a:

```
%union{
  tip1 promenljiva1;
  tip2 promenljiva2;
```

```
  ...
  tipN promenljivaN;
}
```

Tipovi u uniji mogu biti samo promenljive osnovnih tipova i/ili pokazivači na klase, korisničke tipove. Na primer:

```
%union{
  int ceoBroj;
  char* naziv;
  string* ime
  double realanBroj;
  Lista* lista;
}
```

Nakon što smo definisali moguće tipove podataka i njima pridržene promenjive, možemo da definišemo azbuku tokena koju ćemo koristiti u parseru. Jednokarakterski tokeni se mogu koristiti u gramatici, bez potrebe da se pre toga proglase tokenima. Za njih će *bison* koristiti njihove ASCII vrednosti, tj. svim korisnički definisanim tokenim će dodeljivati celobrojeve veće od 257. Za njih lex treba samo da parseru vrati ASCII vrednost prepoznatog karaktera. Ne moraju svi složeniji tokeni da imaju pridružene vrednosti, ali je neophodno da ih sve proglasimo tokenima. Tokene definišemo koristeći komandu `%token`. Ukoliko tokeni nemaju atrubute dovoljno ih je samo nabrojati u nastavku linije započete sa `%token`, na primer:

```
%token ID BROJ PRINT
```

Ukoliko neki tokeni imaju atribute oni se navode u zasebnoj liniji koja ponovo počinje sa `%token`, ali se navodi i promenljiva iz pomenute unije koja će preuzimati vrednost tog tokena. Na primer:

```
%token<ime> ID
%token<ceoBroj> BROJ
%token PRINT
```

Svaki simbol koji se javi u gramatici, a nije jednokarakterski token niti proglašen tokenom, smatra se neterminalom. Očekuje se da ima odgovarajuća pravila. Biće prijavljena greška ukoliko se nađe simbol koji nije token, a nema svoja pravila u gramatici.

Ne moraju ni svi neterminali imati atribute. Ukoliko treba da imaju, neterminalima možemo dodeliti tip podataka atributa, sledećom naredbom:

```
%type<ime promenljive iz unije> lista neterminala
```

Na primer, za neterminali koji izvode izraz ili pod izraz celih brojeva, celobrojni atribut je odgovarajući i to bismo naznačili naredbom :

```
%type<ceoBroj> e t f
```

Startni simbol gramatike proglašavamo komandom `%start` na sledeći način:

```
%start pocetniSimbol
```

Ukoliko se izostavi ova komanda, za startni simbol gramatike uzima se simbol čije pravilo je prvo navedeno u specifikaciji gramatike.

Bison može da otkrije postojanje *shift-reduce konflikata*. Kada se jave nekada je neophodno prepraviti gramatiku, a nekada je dovoljno definisati prioritete operatora koje koristimo u pravilima. Da bismo to izbegli potrebno je da definišemo prioritete tako što navedemo asocijativnost i nastavku linije spisak operatora sa istom asocijativnošću koji su istog prioriteta. Asocijativnost može biti jedna od sledećih komandi: `%nonassoc` za neasocijativne operatore, `%left` za levo asocijativne operatore i `%right` za desno asocijativne operatore. Operatore istog prioriteta navodimo u istoj liniji razdvojene blanko karakterom. U liniji ispod mogu se navesti drugi operatori, po istom principu, navođenjem asocijativnosti pre navođenja samih operatora. *bison* će smatrati da što je kasnija linija sa navedenim simbolima to im je prioritet veći. Razrešavanje prioriteta među operatorima istog prioriteta vrši se na osnovu asocijativnosti. Na primer, za operatore algebarskih operacija, navođenje njihovih prioriteta, izgledalo bi ovako:

```
%left '+' '-'
%left '*' '/'
%left '^'
```

Redosledom komandi navedeno je da su operacije sabiranja i oduzimanja levo asocijativne i istog prioriteta. Operacije množenja i deljenja su takođe levo asocijativne i međusobno istog prioriteta, ali su višeg prioriteta od sabiranja i oduzimanja. Na kraju, operacija stepenovanja je najvišeg prioriteta i levo asocijativna.

Ukoliko mi ne razrešimo pronađene konflikte, *bison* će ih razrešavati prema podrazumevanom ponašanju, tako što će uvek favorizovati akciju prebacivanja tokena na stek, tj *shift* više od svođenja po pravili, tj *reduce*. Moguće je naći ulaz koji bi opravdao prihvatanje *bison*-ovog načina razrešavanja, ali se često može vrlo lako naći i ulaz za koji bi podrazumevano rešenje konflikta bilo potpuno semantički pogrešno.

Gramatička pravila se u opštem slučaju zapisuju kao:

neterminal : *pravilo*1 { *akcija*1 }
|*pravilo*2 { *akcija*2 }
...
|*praviloN* { *akcijaN* }
;

Akcija pridružena pravilu predstavlja C/C++ kod koji će biti izvršen kada u toku sintaksne analize bude izvršena akcija svođenja po tom pravilu. Akcija pridružena pravilu gramatike mora da počne u istom redu u kojem se nalazi samo pravilo. Ovako grupisana pravila istog neterminala moraju da se završe eksplicitnim navođenjem karaktera `;`.

Bison može da prijavi i *reduce-reduce konflikte*. Oni se javljaju kada pronađe više pravila sa identičnim desnim stranama, a različitim neterminalom sa desne strane. Kao i za *shift-reduce konflikte*, i u ovom slučaju postoji podrazumevan način razrešanja koji se svodi na favorizovanje pravila koje je ranije navedeno u gramatici i po njemu će biti izvršeno svođenje. Na taj način, drugo pravilo sa identičnom desnom stanom, postaje suvišno, tj. nikada neće biti upotrebljeno. Postavlja se pitanje, da li zaista sme tek tako da se zanemari. Ako sme, onda ga treba jednostavno obrisati, ali češće će biti da je potrebno jer je gramatika loše napisana. Rešenje u tom slučaju se svodi na revidiranje gramatike i pronalaženju druge gramatike za opis jezika.

Ilustrujmo način pristupa atributima neterminala i tokena u okviru akcije pravila. Neka

je dato sledeće pravilo:

$$a : b\ c\ d\ \{\ akcija\ \}$$

pri čemu važi $a \in N$ i $b, c, d \in (\Sigma \cup N)$. Pretpostavimo da smo svim simbolima u pravilu pridružili vrednosti na ranije opisani način i da želimo da ih koristimo u okviru akcije. *Bison* jasno razlikuje levu i desnu stranu pravila. Leva strana pravila se dobija svođenjem i nije inicijalizovana, tj. možemo eksplicitno da je inicijalizujemo u okviru akcije ukoliko treba. Atributu neterminala sa leve strane pravila, pristupamo navođenjem `$$`. Neterminali sa desne strane pravile imaju atribute sa već dodeljene vrednosti koje možemo da koristimo prilikom izračunavanja vrednost za `$$`. Da bi se jasno pristupalo atributima neterminala sa desne strane pravila, *bison* numeriše sve simbole s leva na desno počevši od broja jedan. Na našem primeru, simbolima će biti dodeljene promenljive za atribut na sledeći način: $b \leftarrow \$1$, $c \leftarrow \$2$ i $d \leftarrow \$3$. Kako se vrši sintaksna analiza naviše i akcija se izvršava tek pri akciji svođenja po posmatranom pravilu, znamo da su svi simboli sa desne strane pravila prepoznati i nalaze se na steku analize. Vrednosti njihovih atributa su na steku vrednosti. Svaki element na steku vrednosti je tipa unije koju smo definisali sa `%union`. *Bison* interno razrešava koje polje unije treba da pročita sa steka vrednosti kada referišemo na `$$` ili `$N` na osnovu gore definisane unije i pridruživanja polja unije neterminalu.

Svaki simbol sa desne strane pravila će dobiti promenljivu bilo da smo mi odrediti polje unije koje odgovara atributu ili ne. Ako smo im dodelili polje unije, onda možemo koristiti promenljivu $\$N$ da pristupimo atributu, u suprotnom ne možemo, ali one u svakom slučaju postoje i o tome uvek treba da vodimo računa.

Treba imati u vidu da neka pravila ne moraju imati akcije. U tom slučaju je bolje navesti samo `{ }` i tako ekspicitno naglasiti da pravilo ima praznu akciju. Ukoliko to izostavimo, a ni ne napišemo konkretnu akciju, *bison* će izvršavati svoju podrazumevanu akciju

```
$$ = $1;
```

Nekada nam to možda i odgovara, a nekada može napraviti problem. Svako je nešto čega treba biti svesan i na šta treba obratiti posebno pažnju.

Kreiranje sintaksnog analizatora se odvija u više koraka:

1. Definisanje azbuke tokena i kontekstno slobodne gramatike. Prilikom definisanja gramatike treba da vodimo računa da bude dovoljno opšta i da omogućava lako proširivanje novim pravilima.
2. Pisanje bison skripta na osnovu koga se generiše C/C++ kod sintaksnog analizatora i fajl koji sadrži opise svih tokena.
3. Pisanje flex skripta na osnovu koga se generiše C/C++ kod koji predstavlja leksički analizator.
4. Prevođenje dobijenog C/C++ koda do izvršnog koda koji predstavlja sintaksni analizator
5. Prevođenje dobijenog C/C++ koda do izvršnog koda koji predstavlja leksički analizator.
6. Linkovanje sintaksnog i leksičkog analizatora u konačni izvršni program.

Opis sintaksnog analizatora zapisan u datoteci ekstenzije `.y` za C kod, odnosno `.ypp` za C++ kod, na primer `parser.ypp`. Da bismo generisali C++ kod sintaksnog analizatora, potrebno je da uz pomoć `bison`-a kreiramo `parser.tab.cpp` i `parser.tab.hpp` datoteke. Datoteka `parser.tab.cpp` sadrži implementaciju sintaksnog analizatora, a datoteka

parser.tab.hpp sadrži spisak definicija tokena i unije za atribute simbola. To postižemo sledećom komandom:

```
bison -d parser.y
```

Ukoliko želimo da nam se pored navedenog generiše i datoteka u kojoj je čitljivo opisan potisni automat, sa svim stanjima, ajtemima i skupovima dodeljenih tokena akcijama, potrebno je da prilikom poziva *bison*-a zadamo i opciju `-v`. Tada bi poziv glasio ovako

```
bison -d -v parser.y
```

Pretpostavimo da je opis leksičkog analizatora zapisan u datoteci koji se zove `lexer.l`. Da bismo generisali C kod leksičkog analizatora, potrebno je da uz pomoć *flex*-a kreiramo datoteku `lex.yy.c` u kojoj će biti implemetacija opisanog leksičkog analizatora. To postižemo sledećom komandom:

```
flex lexer.l
```

Kada imamo implementacije i sintaksnog i leksičkog analizatora potrebno je da ih prevedemo do objektnog koda sledećim komandama:

```
g++ -c -Wall lex.yy.c -o lex.yy.o
g++ -c -Wall parser.tab.cpp -o parser.tab.o
```

Na kraju, potrebno je da obe objektne datoteke linkujemo u jednu izvršnu koji predstavlja naš sintaksni analizator:

```
g++ -Wall lex.yy.o parser.tab.o -o parser
```

U okviru `g++` poziva, preimenovali smo rezultujući izvršnu verziju u `parser`. Da bismo izvršili sintaksnu analizu, dovoljno je samo da pokrenemo izvršni program parser. Radi jednostavnijeg i fleksibilnijeg održavanja koda, objedinićemo ove pozive u `Makefile`:

```
parser: parser.tab.o lex.yy.o
  g++ -Wall -std=c++11 -o parser parser.tab.o lex.yy.o

lex.yy.o: lex.yy.c parser.tab.hpp
  g++ -Wall -c -o lex.yy.o lex.yy.c

lex.yy.c: lexer.l parser.tab.hpp
  flex lexer.l

parser.tab.o: parser.tab.cpp parser.tab.hpp
  g++ -Wall -std=c++11 -c -o parser.tab.o parser.tab.cpp

parser.tab.cpp parser.tab.hpp: parser.ypp
  bison -d -v parser.ypp
```

Rešenje 8.1: Makefile

Podrazumeva se da je Makefile obavezni deo rešenja, ali u nastavku neće biti navođen osim u slučaju da postoji neka promena koju treba ilustrovati.

8.3 Primeri sintaksne analize naviše

Primer sintaksnih analizatora generisanih uz pomoć *bison*-a biće napisani u C++ uz upotrebu objektne paradigme koja nam omogućava mnogo fleksibilnije i jednostavnije pisanje rešenja. To nije neophodno u prvih nekoliko primera, ali radi lakše nadogadnje rešenja, krenućemo odmah sa C++.

Prilikom pisanja parsera u *bison*-u, nema potrebe da računamo skupove izbora, jer bison to radi sam. Jedino o čemu treba da vodimo računa je da gramatika nema shift-reduce konflikte. U slučaju da ih ima, rešavamo ih definisanjem prioriteta operatora. Ako to ne pomogne, potrebno je da smislimo drugačiju gramatiku koja neće imati shift-reduce ili reduce-reduce konflikte koji ne mogu da se reše definicijom prioriteta operatora.

Primer 8.4 Prvi sintaksni analizator naviše implementiraćemo za ranije korišćenu gramatiku algebarskih izraza 8.1, tj. :

$$
\begin{aligned}
e &\rightarrow e + t \\
&\quad \mid t \\
t &\rightarrow t * f \\
&\quad \mid f \\
f &\rightarrow (e) \\
&\quad \mid BROJ
\end{aligned}
$$

Parser treba samo da ispita sintaksnu ispravnost algebarskog izraza.

Rešenje. Navedene su specifikacije leksičkog analizatora za alat *flex* i specifikacija sintaksnog analizatora za alat *bison*.

```
%option noyywrap
%option noinput
%option nounput

%{
#include <iostream>
#include <cstdlib>

/* Bison vodi racuna o tokenima i njihove definicije smesta u
   datoteku parser.tab.hpp
   Posto su nam potrebni za komunikaciju izmedju sintasnog i
   leksickog analizatora ukljucujemo zaglavlje parser.tab.hpp u
   kom su tokeni definisani. */

#include "parser.tab.hpp"

using namespace std;
%}

%%
[0-9]+    { return BROJ; }
```

```
[+*()]    { return yytext[0]; }
[\n \t]   { }
.   {
      cerr << "Nepoznata leksema \"" << yytext << "\""<< endl;
      exit(EXIT_FAILURE);
    }
%%
```

Rešenje 8.2: Specifikacija leksičkog analizatora

```
%{
/* C++ kod koji se direktno prepisuje na pocetak datoteke
   parser.tab.cpp */
/* zaglavlja koja su nam potrebna zbog leksickog analizatora i
   funkcije za obradu greske */

#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <string>

#define YYDEBUG 1

using namespace std;

/* Leksicki analizator - lex ce nam ga izgraditi */
extern int yylex();
/* Funkcija koja ispisuje poruku o gresci i prekida program  */
void yyerror(string s) ;
%}

/* Deklarisemo BROJ kao token. */
%token BROJ
/* Deklarisemo aksiomu (startni simbol) gramatike */
%start e

%%
e : e '+' t
  | ˜t
  ;
t : t '*' f
  | f
  ;
f : '(' e ')'
  | BROJ
  ;
%%

int main(){
```

```
  if(yyparse() == 0)
39   cout <<"Uparen je aritmeticki izraz!" << endl;

41  return 0;
}
43

void yyerror(string s) {
45   cerr << s << endl;
   exit(EXIT_FAILURE);
47 }
```

Rešenje 8.3: Specifikacija sintaksnog analizatora

Primer 8.5 Proširiti funkcionalnost sintaksnog analizatora iz primera 8.4 tako da računa vrednost svakog ispravnog izraza koji korisnik napiše. Primeri naredbi parseru su:

```
2 + 3 * 5
6 * 3 + (12 * 3 +11)
```

Rešenje. U leksičkom analizatoru treba izvršiti minimalne promene tako da uz token BROJ sada vraća i vrednost.

```
18 [0-9]+ { /* yylval je tipa unije definisane u parser.ypp
             Atributu tokena BROJ odgovara polje vrednost
20           */
           yylval.vrednost = atoi(yytext);
22         return BROJ;
         }
24

[+*()]  return *yytext;
```

Rešenje 8.4: Specifikacija leksičkog analizatora

U sintaksnom analizatoru treba definisati promenljivu `yylval` preko koje će lekser parseru prosleđivati vrednosti atributa tokena. Ona će biti tipa unije koju smo definisali sa komandom `%union`. Potom treba dodeliti svakom pravilu gramatike kod semantičkih akcija.

Da bismo prepoznali kada je kraj prepoznavanja celog izraza, uvešćemo novu aksiomu, *pocetak*, koja će izvoditi staru, *e*. Za razliku od stare aksiome ona se nikada ne javlja sa desne strane pravila.

```
1 %{
/* C++ kod koji se direktno prepisuje na pocetak datoteke
   parser.tab.cpp i zaglavlja koja su nam potrebna zbog
   leksickog analizatora i funkcije za obradu greske */
3

#include <iostream>
5 #include <cstdlib>
#include <string>
```

```
7
  /* Za ispis tok analize kroz stek i stanja automata. */
9 #define YYDEBUG 1

11 using namespace std;

13 /* Leksicki analizator - lex ce nam ga izgraditi */
  extern int yylex();
15 /* Funkcija za obradu greške */
  void yyerror(string s) ;
17 %}

19 /* Sa %union redefinisemo tip vrednosti koje atributi simbola
     gramatike mogu imati.
     Podrazumevani tip je int. Ako zelimo neki drugi tip, ovo je
     nacin da postignemo, potrebno je jos da navedemo polja
     odgovarajucih tipova unutar unije. */
21 %union{
    /* i dalje prepoznajemo celobrojne izraze. Svi simboli ce
     imati celobrojni atribut tako da je unutar unije potrebno da
     imamo samo celobrojno polje. */
23   int vrednost;
  }
25

  /* unutar definicije tokena BROJ navodimo polje unije koje
     odgovara tokenu.
27   Pridruzivanjem polja unije simbolu se zapravo samo zadaje
     tip atributa. */
  %token <vrednost> BROJ
29 /* Isto se može uraditi i sa neterminalima, na sledeci nacin:*/
  %type <vrednost> e t f
31 /* Simboli koji imaju atribut istog tipa navode se u istom redu
     */

33 /* Definisemo aksiomu gramatike*/
  %start pocetak
35

  %%
37 /* Pravila KSG sa pridruzenim semantickim akcijama
    Dobra praksa je da se prvo svim pravilima pridruze prazne
     akcije, a da se potom zamenjuju sa konkretnim akcijama. Tako
     smo sigurni da se nigde nece izvrsavati podrazumevana
     akcija, a da tako nismo hteli. */
39

  pocetak: e     {
41                   cout << "Vrednost izraza je " << $1 << endl;
                   }
43   ;
  e : e '+' t    { $$ = $1 + $3; }
```

```
45    | t          { $$ = $1; }
      ;
47  t : t '*' f    { $$ = $1 * $3; }
      | f          { $$ = $1; }
49    ;
    f : '(' e ')'  { $$ = $2; }
51    | BROJ       { $$ = $1; }
      ;
53  %%

55  int main(){
    /* Za ispis tok analize kroz stek i stanja automata potrebno je
       da se yydebug postavi na 1 */
57    yydebug = 0;

59    if(yyparse() == 0)
        cout << "Uparen je aritmeticki izraz!" << endl;
61    return 0;
    }
63
    void yyerror(string s) {
65     cerr << s << endl;
       exit(EXIT_FAILURE);
67  }
```

Rešenje 8.5: Specifikacija sintaksnog analizatora

Ukoliko nam je potrebno da pratimo tok analize kroz stek i stanja automata, neophodno je da predprocesorskom direktivom definišemo `YYDEBUG`.

```
#define YYDEBUG 1
```

i u main funkciji postavimo promenljivoj yydebug vrednost različitu od nule.

```
yydebug = 1
```

Kada nam debagovanje ne bude više potrebno promenljivoj `yydebug` u `main` funkciji postavimo vrednost na 0 ili celu dodelu stavimo pod komentar.

Primetimo da prilikom implementacije u prethodnom primeru nismo vodili računa o tome da li je naša gramatika levo rekurzivna ili ne, jedino što smo gramatikom rešili su bili prioriteti operacija, uvođenjem neterminala t i f i njihovih pravila. Međutim, ni to ne moramo da radimo u samoj konstrukciji gramatike, već možemo da koristimo višeznačne gramatike.

Primer 8.6 Implementirati sintaksni analizator algebarskih izraza nad realnim brojevima tako da računa vrednost svakog ispravnog izraza koji korisnik napiše. Podržane su operacije sabiranja, oduzimanja, množenja, deljenja, deljenja sa ostatkom, unarni minus i grupisanje izraza. Omogućiti prihvatanje i računanje vrednosti više izraza po pozivu programa. Izraz se završava karakterom `;`. Primeri naredbi parseru su:

```
2.2 + 3 * 5 ;
6.21 / 3 + (1.2 * 3 - 11)  ;
(-5) + -7 * (11 + 3 * 9 / 4) ;
```

Rešenje. Za implementaciju ovo parsera možemo koristiti gramatiku korišćenu u primeru 8.1. Ipak ovaj put ćemo koristiti višeznačnu gramatiku algebarskih izraza.

$$
\begin{aligned}
e \rightarrow\ & e + e \\
\mid\ & e * e \\
\mid\ & e - e \\
\mid\ & e / e \\
\mid\ & e \qquad \mid -e \\
\mid\ & (e) \\
\mid\ & \mathit{BROJ}
\end{aligned}
$$

U leksičkom analizatoru treba izvršiti minimalne promene da vraća tokene koji odgovaraju novim operacijama.

```
[+\-*/();]      return *yytext;
```

Rešenje 8.6: Specifikacija leksičkog analizatora

Bison će nam prijaviti shift-reduce konflikte koje ćemo razrešiti definisanjem prioriteta operatora. Takođe, potrebno je da obratimo pažnju na način kako smo definisali prioritet unarnog minusa. Leksički gledano, razlika između unarnog i binarnog minusa ne postoji, jer su i jedan i drugi predstavljeni istim karakterom `-`, pa time i istim tokenom u gramatici. Da bismo napravili razliku između unarnog i binarnog minusa uvešćemo nov token `UMINUS` navodeći ga u delu gde smo definisali prioritete operacija. Prioritet tog tokena ćemo iskoristiti u gramatici tako što ćemo ga dodeliti pravilu za unarni minus. To postižemo tako što na kraju pravila stavljamo naredbu `%prec UMINUS`. Na taj način smo parseru rekli da taj minus u pravilu $e \rightarrow -e$ zapravo predstavlja operaciju najvišeg prioriteta koja se zove `UMINUS`. I pravilo će biti korišćeno za svođenje što ranije moguće, tj. najprioritetnije, kao da se u njemu nalazi token najvišeg prioritete, `UMINUS`, a ne token vrlo niskog prioriteta `-`.

Koristeći debagovanje ispratiti tok analize i redosled primenjivanja pravila kada nije zadat prioritet i kada je prioritet dodeljen. Na primer, za ulaz `-2*5;`.

```
%{
#include <iostream>
#include <cstdlib>
#include <string>

/* Za ispis tok analize kroz stek i stanja automata */
#define YYDEBUG 1

using namespace std;

/* Leksicki analizator - lex ce nam ga izgraditi */
extern int yylex();
```

```
/* Funkcija koja ispisuje poruku o gresci i prekida program  */
void yyerror(string s) ;
%}

/* Tip atributa simbola ce biti tip nekog od polja ove unije. */
%union{
  float vrednost;
}

/* Definisemo asocijativnost i prioritete operatora. */
%left '+' '-'
%left '*' '/'
%right UMINUS

/* Pridruzujemo konkretno polje unije tokenu i neterminalu e */
%type <vrednost> e

/* tokenu dodeljujemo atribut */
%token <vrednost> BROJ

/* zadajemo aksiomu gramatike */
%start niz_naredbi

%%
niz_naredbi: niz_naredbi naredba ';'     {  }
    | naredba ';'                        {  }
    ;
naredba: e     { cout << "Vrednost izraza je " << $1 << endl; }
  ;
e : e '+' e    { $$  = $1 +$3;}
  | e '-' e    { $$ = $1 - $3;}
  | e '*' e    { $$ = $1 * $3;}
  | e '/' e    { if( $3 == 0)
                   yyerror("Deljenje nulom!");

                 $$ = $1 / $3; }
  | '-' e %prec UMINUS  { $$ = -$2; }
  | '(' e ')'     { $$ = $2; }
  | BROJ      { $$ = $1;}
  ;
%%

int main(){
/* Za ispis tok analize kroz stek i stanja automata potrebno je
   da se yydebug postavi na 1 */
//   yydebug = 1;
  yyparse();

  return 0;
```

```
61 }

63 void yyerror(string s) {
       cerr << s << endl;
65     exit(EXIT_FAILURE);
   }
```

Rešenje 8.7: Specifikacija sintaksnog analizatora

Primer 8.7 Implementirati sintaksni analizator koji će pored izračunavanja vrednosti izraza omogućiti korisniku da definiše promenljive koje mogu sačuvati vrednost izraza i mogu učestvovati u drugim izrazima. Omogućiti korisniku da štampa vrednost izraza, vrednosti svih promenljivih u memoriji i da upoređuje vrednosti izraza. Svaka od naredbi se zadaje u zasebnoj liniji. Omogućiti da se prihvate i prazni redovi između konkretnih naredbi. Primer poziva programa je dat u nastavku, sa desne strane naredbi štampe, dat je ispis na ekran.

```
a = 4

b = 5
c = (a+ 3)*b-b/-2
print(c)                                    37.5
print_all                                    a 4
                                             b 5
                                             c 37.5

b = -3*b
print(b)                                   b -15
print(d)                                   Promenljiva d nije definisana!
```

Rešenje. Pre definisanja gramatike, potrebno je da pažljivo sagledamo test primere i da na osnovu njih definišemo gramatiku. Gramatika treba da bude dovoljno opšta da omogući lako proširivanje u slučaju da se doda neki novi zahtev u zadatku. Pažljivim posmatranjem naredbi, uočavamo da naš program predstavlja nizanje naredbi koje su terminisane karakterom `;`. U nastavku treba da definišemo koje su moguće naredbe u programu. Očigledno, tipovi naredbi su sledeće:

1. Naredba za štampanje izraza.
2. Naredba štampanja svih sačuvanih promenljivih.
3. Naredba za definisanje promenljivih.
4. Naredbe za upoređivanje izraza.

Nakon definisanja naredbi potrebno je da definišemo izraze. Jedina novina u izrazima je ta što sada možemo da koristimo ranije definisane promenljive u jeziku, ravnopravno sa tokenom BROJ, pa dodajemo pravilo $e \to ID$, gde je `ID` token koji odgovara promenljivoj.

Da bismo to omogućili koristićemo strukturu podataka *map* iz C++ standardne biblioteke, u kojoj možemo da čuvamo parove `<naziv promenljive, vrednost promenljive>`.

Kada koristimo promenljive potrebno je da pažljivo definišemo logiku u programu i da vodimo računa o memoriji, tj. ne smemo da imamo curenje memorije. Kada koristimo promenljive postoje dva slučaja:

- Definisanje promenljive.
- Korišćenje promenjive u izrazu.

Kada definišemo promenljivu, potrebno je da proverimo da li se promenljiva sa tim imenom već nalazi u strukturi podataka. Ukoliko se nalazi, potrebno je da promenimo njenu vrednost. Ako se ne nalazi u strukturi podataka, treba da je dodamo u strukturu. U slučaju da je promenljiva deo izraza, potrebno je da proverimo u strukturi podataka da li imamo promenljivu sa tim imenom. Ako je imamo, treba da koristimo njenu vrednost ili njenu kopiju ako se radi o referentnim tipovima. Ako promenljiva pod tim imenom ne postoji, treba da prijavimo grešku korisniku. Nakon toga, možemo da prekinemo program, ili da umesto nedeifinisane promenljive koristimo neku unapred definisanu vrednost i nastavimo sa radom. Mi ćemo u ovom primeru prekidati program.

Leksički analizator je trebalo proširiti da vraća nove tokene za naredbe štampanja, identifikatore i operatore poređenja.

```
%option noyywrap
%option noinput
%option nounput

%{
#include <iostream>
#include <cstdlib>

#include "parser.tab.hpp"
%}

%%
[0-9]+(\.[0-9]+)?   { yylval.vrednost = atof(yytext);
                      return BROJ;
                    }
print           { return PRINT; }
print_all       { return PRINT_ALL; }

[a-zA-Z_][a-zA-Z_0-9]* {
    /* pravimo kopiju procitane promenljive. Ne smemo da
   vratimo pokazivac na yytext, jer će  u yytext biti upisan
   nov sadrzaj, kada bude pozvan leksicki analizator i
   prepoznat naredni token */
    yylval.ime = new std::string(yytext);
    return ID;
  }

"=="                { return EQ;  }
"!="                { return NEQ; }
"<="                { return LEQ; }
">="                { return GEQ; }
[+\-*/()<>=\n]      { return *yytext; }
[ \t]               { }
.   {
      std::cerr << "Nepoznata leksema: " << *yytext <<
```

```
    std::endl;
33        exit(EXIT_FAILURE);
     }
35 %%
```

Rešenje 8.8: Specifikacija leksičkog analizatora

Bitno obratiti pažnju da se operatori poređenja koji se sastoje od 2 karaktera vraćaju kao jedan token. Na primer, u 27. liniji

```
"<="                    { return LEQ; }
28 ">="                 { return GEQ; }
[+\-*/()<>=\n]          { return *yytext; }
30 [ \t]                { }
.    {
32        std::cerr << "Nepoznata leksema: " << *yytext <<
    std::endl;
        exit(EXIT_FAILURE);
34     }
%%
```

Rešenje 8.9: Prepoznavanje složenih operatora

leksički analizator vraća token `LEQ`, samo ako su se karakteri našli jedni za drugim. Na taj način leksički analizator ih prepoznaje zajedno i vraća adekvatan token. Ukoliko ne bismo napravili zaseban regularni izraz za `<=`, može nam delovati da će se isto postići ako u gramatici postoji pravilo *naredba* → *izraz*$'<'$ $'='$ *izraz*. Međutim, upotrebom razdvojenih tokena u pravilu omogućavamo da se između njih prepoznaju drugi validni tokeni. Većina tokena koja bi se pojavila između bi izazvala sintaksnu grešku, jer se posle `<` očekuje samo `=`. Ali tokeni koje u lekseru ignorišemo, poput belina, neće narušiti sintaksu, a trebalo bi. Ne možemo reći da je ispravno napisati `a <  = b`, već da treba `a <= b`. Sa pravilom *naredba* → *izraz*$'<'$ $'='$ *izraz* i jedno i drugo će biti ispravna naredba poređenja.

Zbog upotrebe klase string za atribut promenljive, unutar unije moramo napisati `std::string* ime;` nevezano od toga da li u kodu postoji naredba `using namespace std`;

Naredni kod predstavlja specifikaciju sintaksnog analizatora za kalkulator za bison.

```
1 %{
  #include <iostream>
3 #include <cstdlib>
  #include <string>
5 #include <map>

7 #define YYDEBUG 1

9 extern int yylex();
  void yyerror(std::string s);
11
  /* Mapa koja preslikava ime promenljive u njenu vrednost. */
13 std::map<std::string, float> tablica;
```

```
%}

/* Definisemo moguce tipove podataka za tokene i neterminale */
%union{
  float vrednost;
  std::string* ime;
}

/* Definisemo asocijativnost i prioritete operatora. */
%left '+' '-'
%left '*' '/'
%right UMINUS

/* definisemo tokene i pridruzujemo im odgovarajuce tipove tako
   sto navodimo ime polja unije */
%token <vrednost> BROJ
%token <ime> ID
%token PRINT PRINT_ALL LEQ NEQ EQ GEQ

/* dodeljujemo tipove neterminalima tako sto navodimo ime polja
   unije */
%type<vrednost> izraz

%%
niz_naredbi : niz_naredbi naredba '\n'
    | /* epsilon - da i program bez naredbi bude ispravan */
    ;

naredba : PRINT '(' izraz ')' {
        std::cout << $3 << std::endl;
      }
    | PRINT_ALL  {  // iteriramo kroz mapu promenljivih
        for(auto i = tablica.begin(); i!=tablica.end(); i++)
          std::cout << i->first << " " << i->second <<
   std::endl;
      }
    | ID '=' izraz {
        /*  $1 je tipa std::string*
           *$1 je tipa std::string
          Upisujemo ime promenljive i vrednost izraza u mapu.*/
        tablica[*$1] = $3;
        /* Ukoliko postoji biće prepisana stara vrednost novom.
          To je u redu jer je u pitanju primitivan tip podataka,
          float. Inace, biće dodata nova promenljiva. */

        /* Dealociramo string koji je alociran u lekseru. Pri
          upisu u mapu upisujemo vrednost tog stringa, pa nam $1
          vise nije potreban. */
        delete $1;
```

```
      }
61
    /* naredbe za uporedjivanje vrednost izraza */
63  | izraz '<' izraz        {
                                if ($1 < $3)
65                                  printf("DA\n");
                                else
67                                  printf("NE\n");
                            }
69  | izraz '>' izraz        {
                                if ($1 > $3)
71                                  printf("DA\n");
                                else
73                                  printf("NE\n");
                            }
75  | izraz EQ izraz         {
                                if ($1 == $3)
77                                  printf("DA\n");
                                else
79                                  printf("NE\n");
                            }
81  | izraz NEQ izraz        {
                                if ($1 != $3)
83                                  printf("DA\n");
                                else
85                                  printf("NE\n");
                            }
87  | izraz LEQ izraz        {
                                if ($1 <= $3)
89                                  printf("DA\n");
                                else
91                                  printf("NE\n");
                            }
93  | izraz GEQ izraz        {
                                if ($1 >= $3)
95                                  printf("DA\n");
                                else
97                                  printf("NE\n");
                            }
99  |  /* prazna naredba da bi parser prihvatao i prazne linije
    */
     ;
101
izraz : izraz '+' izraz      { $$ = $1 +$3; }
103     | izraz '*' izraz     { $$ = $1 * $3; }
        | izraz '-' izraz     { $$ = $1 - $3; }
105     | izraz '/' izraz     {
                  if ($3 == 0)
107                 yyerror("Deljenje nulom!");
```

```
109                 $$ = $1 / $3;
                  }
111       | '-' izraz %prec UMINUS     { $$ = -$2; }
          | '(' izraz ')'              { $$ = $2;  }
113       | BROJ                       { $$ = $1;  }
          | ID {
115           /* Proveravamo da li je vec definisana promenljiva sa
                tim imenom
117             Funkcija find() vraca iterator na pronadjen par
                <std::string, int> ili na tablica.end() za slucaj da
119             promenljiva nije pronadjena u mapi. */
              auto i = tablica.find(*$1);
121           if (i != tablica.end()) {
                /* Posto je pronadjena vrednost promenljive sa
123               imenom *$1 vracen nam je uredjeni par
                  <std::string, int> cijim elementima mozemo
125               pristupati sa first i second
                  std::cout << i->first << " = " << i->second ;
127               Vrednost izraza biće ona koju čuva promenljiva. */
                $$ = i->second;
129           }
              else {
131             std::string poruka = "Promenljiva " +*$1 + " nije
    definisana!";
                yyerror(poruka);
133           }
              /* String alociran u lexeru nije vise potreban. */
135           delete $1;
            }
137       ;
%%

139
int main()
141 {
//    yydebug = 1;
143   yyparse();
    /* s obzirom da smo koristili map i string iz biblioteke, sve
     iz tablice će biti oslobođeno biblioteckim destruktorima */
145   return 0;
}
147
void yyerror(std::string s)
149 {
      std::cerr << s << std::endl;
151   exit(EXIT_FAILURE);
}
```

Rešenje 8.10: Specifikacija sintaksnog analizatora

Program se prevodi uobičajenim nizom naredbi. Nakon što je program napisan potrebno je obratiti pažnju na sledeće:

1. Naredbe grupišemo u neterminale prema rezultatu izvršavanja. Sa tim u vezi očigledno je da u prethodnom primeru imamo tri moguća tipa naredbi:
   - Naredbe za štampanje kojima prikazujemo rezultate rada.
   - Naredba dodele kojom definišemo nove promenljive ili menjamo vrednost postojećih.
   - Naredbe poređenja izraza kojima korisniku dajemo odgovor na osnovu logičke vrednosti.
2. Pored ovih tipova naredbi potrebno je da opišemo i šta za nas predstavlja izraz, kao i da definišemo prioritete svih operacija koje se javljaju u izrazima.
3. Kada god imamo promenljive u zadatku potrebno je da definišemo odgovarajući tip podataka koji nam omogućava da ih opišemo na pravi način. Pored toga treba nam struktura podataka u kojoj ćemo pamtiti te promenljive i njima pridružene vrednosti i/ili stanja.
4. Potrebno je da vodimo računa o izvrašavanju i memoriji koju koristimo. U svakom trenutno moramo da znamo gde promenljivu alociramo, gde je koristimo i gde treba da je obrišemo.

Definisanje funkcionalnosti sintaksnog analizatora se radi definisanjem gramatike koja opisuje jezik i pridruživanjem odgovarajućih pravila neterminalima. U nastavku biće urađeno nekoliko složenijih zadataka koji zahtevaju upotrebu objektno orijentisane paradigme da bi se rešenje pojednostavilo.

8.4 Složeniji zadaci

Zadatak 8.1 Napisati interpreter koji izvršava naredbe nad polinomima:

a Polinomi se zadaju nizom realnih koeficijanata. Potrebno je prepoznati validno zadate polinome i ispisati ih u čitljivom obliku. Na primer:

```
<1, -3, 0, -1.2, 6>            1 - 3x - 1.2x^3 + 6x^4
```

b Podržati operacije sabiranja, oduzimanja, unarnog minusa i množenja polinoma.

```
<1, 2, 1, 2> + <0, -1, 3>      1 + x + 4x^2 + 2x^3
<1, 2, 1, 2> - <0, -1, 3>      1 + 3x - 2x^2 + 2x^3
<1, 2, 1,  2> * <0, -1, 3>     - x + x^2 + 5x^3 + x^4 + 6x^5
- <0, -1, 3>                   x - 3x^2
```

c Jezik poseduje promenljive tipa polinom:

```
p1  := <1, 2, 1, 2>
p1                             1 + 2x + x^2 + 2x^3
p2 := <0, -1, 3>
p1 * p2                        - x + x^2 + 5x^3 + x^4 + 6x^5
p3 := (p1-p2)*<1>
p3                             1 + 3x - 2x^2 + 2x^3
```

1. Omogućiti poređenje polinoma:

```
p1 == p2                       False
p1 != p2                       True
```

2. Na polinomima se mogu primeniti operatori diferenciranja i integracije. Uz ope-

rator integracije obavezno se daje konstanta koja predstavlja koeficijent uz x0
.

```
p1'                        2 + 2x + 6x^2
(p1+p2)'- <1, 1>'          8x + 6x^2
$p1|3                      3 + x + x^2 + 0.333333x^3 + 0.5x^4
$p1+p2|2.3                 2.3 + x + 0.5x^2 + 1.333333x^3 + 0.5x^4
<1, 2> + $p1+p2|2.3        3.3 + 3x + 0.5x^2 + 1.333333x^3 + 0.5x^4
```

3. Operator [ ] se koristi za računanje vrednosti polinoma u tački (koristiti Hornerovu šemu):

```
p1[1]                      6
(p1+p2)[0]                 1
```

■

Rešenje. Da bismo mogli da uradimo zadatak, potrebno je da podelimo zadatak na celine i da pažljivo pogledamo test primere. Najčešće je za dobro razumevanje problema, potrebno bar letimice pogledati sve test primere. Na taj način se kasnije štedi vreme, jer na početku imamo predstavu šta sve gramatika treba da podrži.

Kada rešavamo složene zadatke idemo od jednostavnog ka složenijem. Prilikom rešavanja svake od celina, gradićemo program i povećavati mu funkcionalnosti. Već u delovima pod a) i b) shvatamo da će nam biti potrebna struktura podataka adekvatna da čuva polinom. O polinom treba da znamo kog je stepena i koji su mu koeficijenti. Zatim daljim čitanjem test primera, shvatamo da ćemo primenjivati operacije na polinome. Adekvatno je rešenje da napišemo klasu `polinomial` kojom ćemo predstaviti polinome, i u kojoj ćemo implementirati metode za operacije nad polinomima. Deklaracija klase se piše u `.hpp` datoteci, a definicije svih metoda pišu se u odgovarajućoj `.cpp` datoteci.

Paralelno sa razvojem klase polinomial teče razvoj leksera i pisanje gramatike parsera. Potrebno je :

- definisati skup tokena i njima pridruženih vrednosti.
- odrediti šta je terminator naredbe i da li u jeziku postoji potreba za nizanjem naredbi. Utvrditi da li jezik podržava praznu naredbu.
- odrediti tipove naredbi i grupisati ih prema toj podeli (naredbe dodele, naredbe štampanja, naredbe poređenja, uslovi, petlje, izračunavanja ...).
- definisati gramatiku uvek imajući u vidu semantiku jezika koju tek treba da implementiramo.
- definisati prioritete operacija u gramatici, bilo struktuiranjem pravila gramatike, bilo eksplicitnim definisanjem prioriteta u *bison-u*
- odrediti kog sve tipa podataka mogu biti atributi simbola i dodeliti odgovarajuće tipove atributa simbolima.
- pridružiti semantičke akcije gramatičkim pravilima.

Paralelno sa svim gradimo i leksički analizator koji prepoznaje sve navedene tokene. Od stavke do stavke postaje složenija gramatika i s vremena na vreme i lekser sa njom. U lekseru, za razliku od prethodnih primera moramo imati uključivanja zaglavlja kao što je `polinomial.hpp` u datoteku `lexer.l`. Klasa polinomial tj. pokazivač na polinomial nalazi se u definiciji unije, stoga je neophodno poznavanje te klase da bi se odredila veličina unije u bajtovima. Kako leksički analizator pristupa uniji, mora znati njenu veličinu u bajtovima.Kako u zaglavlju `parser.tab.hpp` se nalazi definicija unije, uključivanje zaglavlja `polinomial.hpp` mora da bude navedeno pre uključivanja zaglavlja `parser.tab.hpp`

unutar datoteke `lexer.l`.

Specifikacija leksičkog analizatora

```
%option noyywrap
%option nounput
%option noinput

%{
#include <iostream>
#include <string>
#include "polinomial.hpp"
#include "parser.tab.hpp"

%}

%%
-?[0-9]+(\.[0-9]+)? {
    yylval.f_vrednost = atof(yytext);
    return BROJ;
  }

[a-zA-Z][a-zA-Z0-9]+ {
    yylval.str_vrednost = new std::string(yytext);
    return ID;
  }

[<>,\n+*()!$|'\[\]-]    return *yytext;

:=  {
    return DODELA;
    /* Spojeno da bi morale zajedno da se prepoznaju, tj. bez
       ignorisanih belina izmedju ili nekog drugog tokena */
  }
!=  return NEQ;
==  return EQ;

[ \t] { }
. {
    std::cerr
        << "Leksicka greska: Neprepoznat karakter: "
        << *yytext
        << std::endl;
    exit(EXIT_FAILURE);
}
%%
```

Rešenje 8.11: lexer.l

Specifikacija sintaksnog analizatora

```
%{
#include <iostream>
#include <string>
#include <cstdlib>
#include <map>
#include "polinomial.hpp"

#define YYDEBUG 1

extern int yylex();
void yyerror(std::string s);
void dealokacija();

std::map<std::string, polinomial*> promenljive;
%}

%union{
  float f_vrednost;
  polinomial * poly_vrednost;
  std::string * str_vrednost;
}

%type<poly_vrednost> NizKoeficijenata polinom
%token<f_vrednost> BROJ
%token<str_vrednost> ID
%token DODELA EQ NEQ

%left '-' '+'
%left '*'
%right UMINUS
%left '\''


%%
program : program naredba '\n'
  |
  ;
naredba: polinom  {
        std::cout << *$1 << std::endl;
        delete $1;
      }
  | ID DODELA polinom   {
        /* ako postoji promenljiva brisemo njen polinom */
        if (promenljive.find(*$1) != promenljive.end() )
          delete promenljive[*$1];

        promenljive[*$1] = $3;

        delete $1;
```

```
50        }
    | polinom  EQ polinom   {
52          if( *$1 == *$3 )
              std::cout << "True" << std::endl;
54          else
              std::cout << "False" << std::endl;
56
            delete $1;
58          delete $3;
        }
60    | polinom  NEQ polinom  {
            if( *$1 != *$3 )
62            std::cout << "True" << std::endl;
            else
64            std::cout << "False" << std::endl;

66          delete $1;
            delete $3;
68      }
    | polinom '[' BROJ ']' {
70          std::cout << $1->valueAt($3) << std::endl;
            delete $1;
72      }
    |   /* dozvolimo praznu naredbu da bi dopustili
74          prazne redove u testu*/
    ;
76
polinom :   polinom '+' polinom {
78          // ---------------------------------------------
            // | BITAN DETALJ
80          // ---------------------------------------------
            // | $$ = *$1 + *$3 nije dobro jer je rezultat
82          // | izracunavanja *$1 + *$3 lokalna promenljiva
            // | na steku dok je tip od $$ pokazuje na polinom.
84          // | Odnosno, neophodno je da dinamicki alociramo
            // | polinom. Koristimo podrazumevani konstruktor
86          // | kopije koji poziva konstruktore kopije za
            // | podatke unutar klase, sto je u nasem slucaju
88          // | konstruktor kopije za vektor.
            // ---------------------------------------------
90          $$ = new polinomial(*$1 + *$3);
            delete $1;
92          delete $3;
        }
94    | polinom '*' polinom    {
            $$ = new polinomial((*$1) * (*$3));
96          delete $1;
            delete $3;
98      }
```

```
  | polinom '-' polinom    {
100         $$ = new polinomial(*$1 - *$3);
            delete $1;
102         delete $3;
        }
104   | '-' polinom   %prec UMINUS   {
            $$ = new polinomial($2->uminus());
106         delete $2;
        }
108   | polinom '\''    {
            $$ = new polinomial($1->derivative());
110         delete $1;
        }
112   | '$' polinom '|' BROJ   {
            $$ = new polinomial($2->integral($4));
114         delete $2;
        }
116   | '(' polinom ')'     { $$ = $2; }
      | '<' NizKoeficijenata '>'  { $$ = $2; }
118   | ID          {
        /* ukoliko postoji pravimo kopiju,
120       u suprotnom prijavljujemo gresku*/
          auto trazen  = promenljive.find(*$1);
122       if ( trazen != promenljive.end() )
            $$ = new polinomial(*(trazen->second));
124       else
            yyerror("Nije definisana promenljiva "+*$1);
126
          delete $1;
128     }
  ;
130
NizKoeficijenata : NizKoeficijenata ',' BROJ {
132       $$ = $1;
          $$->add_coef($3);
134     }
  | BROJ        {
136       $$ = new polinomial();
          $$->add_coef($1);
138     }
  ;
140 %%

142 int main(){
  yydebug = 0;
144
  yyparse();
146
  dealokacija();
```

```
148
  return 0;
150 }

152 void yyerror(std::string s){
  std::cerr << s << std::endl;
154   dealokacija();

156   exit(EXIT_FAILURE);
}
158
/* oslobadja se prostor svakog polinoma */
160 void dealokacija() {
  for(auto i = promenljive.begin(); i != promenljive.end(); i++)
162     delete i->second;
}
```

Rešenje 8.12: parser.ypp

Zaglavlje za klasu `polinomial`

```
1 #ifndef ppj_POLINOMIAL_HPP
  #define ppj_POLINOMIAL_HPP
3
  #include <vector>
5 #include <iostream>

7 class polinomial {
  public:
9   polinomial();
    polinomial(const std::vector<float>& coeffs);
11
    // GET-eri
13  const std::vector<float>& get_coefs() const;
    unsigned get_power() const;
15
    void add_coef(float c);
17
    /* Prvi nacin - implementacija operatora za sabiranje kao
19     funkcije clanice (member function, tj. method) koja
       sabira  objekte *this i p. */
21  polinomial operator+(const polinomial& p) const;

23  /* Drugi nacin, i cesci za binarne operatore kako bi
       argumenti bili simetricni, odnosno a i b,
25     umesto this (pokazivac) i p (referenca).
       Ovde se funkcije deklarisu kao prijateljske funkcije
27     za klasu kako bi mogle da pristupe privatnim podacima.
       OVO NARUSAVA ENKAPSULACIJU, ali je cesto u praksi
29     usled dobijanja na performansama. */
```

```
  friend polinomial operator-(const polinomial& a, const
   polinomial& b);
  friend polinomial operator*(const polinomial& a, const
   polinomial& b);

  /*  Alternativa (treci nacin) definisanju operatora bi
    bila implementacija funkcija clanica koje implementiraju
    slicnu logiku.
    Na primer:
    polinomial add(const polinomial& p) const;
    polinomial sub(const polinomial& p) const;
    polinomial mul(const polinomial& p) const;
  */

  polinomial uminus() const;

  friend bool operator == (const polinomial& p, const
   polinomial& q);
  friend bool operator != (const polinomial& p, const
   polinomial& q);

  // Vrednost polinoma u tacki
  float valueAt( float x) const;

  // Izvod polinoma
  polinomial derivative() const;

  //Integral polinoma
  polinomial integral(float c) const;

  void print(std::ostream& s) const;

private:
  std::vector<float> _coeffs;

  /* Kako bi nakon svake operacije bili usaglaseni stepen i
   koeficijent ako ima potrebe.
     Na primer za <1,2,3> - <1,2,3>, rezultat je <0> const*/
  void normalize();
};

std::ostream& operator<<(std::ostream& out, const polinomial&
   p);

#endif
```

Rešenje 8.13: polinomial.hpp

Izvorni kod za klasu `polinomial`

```
#include "polinomial.hpp"

polinomial::polinomial()
{ }

polinomial::polinomial(const std::vector<float>& coeffs)
  : _coeffs(coeffs)
{
  normalize();
}

const std::vector<float>& polinomial::get_coefs() const
{
  return _coeffs;
}

unsigned polinomial::get_power() const
{
  if (_coeffs.empty())
    return 0;
  else
    return _coeffs.size() - 1;
}

void polinomial::add_coef(float c)
{
  _coeffs.push_back(c);
}

polinomial polinomial::operator+(const polinomial& p) const
{
  unsigned stepen = std::max(this->get_power(), p.get_power());
  std::vector<float> coefs(stepen + 1);

  for (unsigned i = 0; i <= stepen; i++) {
    if (i > this->get_power() ) {
      coefs[i] = 0;
    } else {
      coefs[i] = this->_coeffs[i];
    }

    if( i <= p.get_power() ) {
      coefs[i] += p._coeffs[i];
    }
  }

  // ------------------------------------------------------
  // Nepotrebno za kurs ali dosta lepo za znati za industriju:
  // RVO: https://stackoverflow.com/questions/12953127/
```

```
50    //      what-are-copy-elision-and-return-value-optimization
      // -----------------------------------------------------
52    return polinomial(coefs);
}
54
polinomial operator-(const polinomial& a, const polinomial& b)
56{
    unsigned stepen = std::max(a.get_power(), b.get_power());
58  std::vector<float> coefs(stepen + 1);

60  for (unsigned i = 0; i <= stepen; i++) {
      if (i > a.get_power() ) {
62      coefs[i] = 0;
      } else {
64      coefs[i] = a._coeffs[i];
      }
66
      if( i <= b.get_power() ) {
68      coefs[i] -= b._coeffs[i];
      }
70  }

72  return polinomial(coefs); // RVO
}
74
polinomial operator*(const polinomial& a, const polinomial& b)
76{
    unsigned stepen = a.get_power() + b.get_power();
78  std::vector<float> coefs(stepen + 1, 0);

80  for (unsigned i = 0; i < a._coeffs.size(); ++i) {
      for (unsigned j = 0; j < b._coeffs.size(); ++j) {
82      coefs[i + j] += a._coeffs[i] * b._coeffs[j];
      }
84  }

86  return polinomial(coefs);
}
88
void polinomial::normalize(){
90  int velicina = this->_coeffs.size();

92  for(int i = velicina-1; i>0; i--)
      if( this->_coeffs[i] == 0)
94      velicina--;

96  this->_coeffs.resize(velicina);
}
98
```

```
polinomial polinomial::uminus() const
{
    std::vector<float> coefs = _coeffs;

    for (auto i = coefs.begin(); i != coefs.end(); i++) {
        *i = - (*i);
    }

    return polinomial(coefs);
}

bool operator == (const polinomial& a, const polinomial& b)
{
  if (a.get_power() != b.get_power())
    return false;

  for (unsigned i =0; i <= a.get_power(); i++)
    if(a._coeffs[i] != b._coeffs[i] )
      return false;

  return true;
}

bool operator != (const polinomial& p, const polinomial& q)
{
  return !(p == q);
}

float polinomial::valueAt( float x) const
{
    float result = 0;
    int i = get_power();
    for (; i >= 0; i--)
      result = result * x + _coeffs[i];

    return result;
}

polinomial polinomial::derivative() const
{
  std::vector<float> coefs = _coeffs;
  if (get_power() > 0) {
    unsigned nov_stepen = get_power() - 1;
    coefs.resize(nov_stepen+1);

    for ( unsigned i = 0; i <= nov_stepen; i++)
      coefs[i] = (i + 1) * _coeffs[i + 1];
  }
  else
```

```
148     coefs[0] = 0;

150   return polinomial(coefs); //RVO
}
152
polinomial polinomial::integral( float c) const
154 {
  std::vector<float> coefs = _coeffs;
156   unsigned nov_stepen = get_power() + 1;
  coefs.resize(nov_stepen+1);
158
  for ( unsigned i = nov_stepen; i >0; i--)
160     coefs[i] = 1.0 / i * coefs[i - 1];
  coefs[0] = c;
162
  return polinomial(coefs); //RVO
164 }

166 void polinomial::print(std::ostream& out) const {
  /* first ce biti indikator da li još uvek nije ispisan
168      nijedan koeficijent*/
  bool first = true;
170   unsigned degree = get_power();

172   for(unsigned  i = 0; i<= degree; i++){
    /* nema i-tog koeficijenta */
174     if(_coeffs[i] == 0)
      continue;
176
    /* postoji koeficijent */
178     /* ako nije prvi koeficijent koji ispisujemo, a pozitivan
       je prvo treba ispisati operator + */
180     if( !first && _coeffs[i] > 0)
      out << " +";
182
    /* nije koeficijent uz x^0 */
184     if( i != 0) {
      /* specijalna ponasanja za 1 i -1 */
186       if ( _coeffs[i] == 1)
        ;                   // nista ne ispisuj
188       else if(_coeffs[i] == -1){
        out << " -";        // samo znak
190       }
      /* ostali brojevi */
192       else
        out << _coeffs[i];
194     }
    /* koeficijent uz x^0 jednostavno ispisuje */
196     else
```

```
      out << _coeffs[i];

    first = false;

    /* x iza koeficijenta i odgovarajući stepen */
    if (i > 0) {
      out << "x";
      if(i > 1)
        out << "^" << i;
    }

    out  << " " ;
  }

  if(degree == 0 && _coeffs[0] == 0)
    out << 0;

  out << std::endl;
}


std::ostream& operator<<(std::ostream& out, const polinomial& p)
{
  p.print(out);

  return out;
}
```

Rešenje 8.14: polinomial.cpp

Makefile pored uobičajenih datoteka u kojima su implementacije leksera i parsera prevodi i linkuje implementaciju klase polinomial u cilju dobijanja konačnog rešenja. Makefile

```
polinom: lex.yy.c parser.tab.cpp polinomial.cpp
  g++ -o polinom -Wall -std=c++11 lex.yy.c parser.tab.cpp
   polinomial.cpp

lex.yy.c: lexer.l parser.tab.hpp
  flex lexer.l

parser.tab.cpp parser.tab.hpp : parser.ypp
  bison -d -v parser.ypp

.PHONY: clean

clean:
  rm *.tab.* *.c polinom *.output
```

Rešenje 8.15: Makefile

U okviru ovog primera, potrebno je da pažljivo obratimo pažnju na upravljanje memorijom. U lekseru svaki put pravimo kopiju naziva promenljive i zbog toga moramo da je oslobodimo svaki put nakon korišćenja u parseru. Kada koristimo polinome u gramatici izraza, mi zapravo uvek koristimo kopije već postojećih polinoma i zato svaki put treba da ih oslobodimo nakon što ih iskoristimo prilikom izačunavanja izraza. Da bi ovo bili jasnije, objasnićemo tok izvšavanja naredbe za ispis vrednosti polinoma za različite kombinacije ulaza.

Ako je na ulazu sledeći niz karaktera `<0, 1, -2, 3>`, tada će deo izvođenja biti *naredba* $\Rightarrow$ *polinom* $\Rightarrow <$ *lista* $>$ .... Prateći akcije pridružene pravilima, jasno je da je posledica svođenja po bilo kom pravilu neterminala *lista* alokacija novog polinoma, koji se posle dodelama propagira neterminalu *polinom* i na kraju neterminalu *naredba*, koji će odštampati prepoznati polinom. Nakon što se polinom odštampa, jasno je nam polinom više ne treba i da alocirani prostor mora da se obriše, jer će u suprotnom doći do curenja memorije.

Pretpostavimo da na ulazu imamo neki izraz, na primer, `<0, 1, -2, 3> - <1, 2, -3>` . Sličnim razmatranjem kao u prethodnom primeru, jasno je da će deo izvođenja biti *naredba* $\Rightarrow$ *polinom* $\Rightarrow$ *polinom* + *polinom* $\Rightarrow <$ *lista* $>$ + *polinom* $\Rightarrow$ .... Odavde lako zaključujemo da ćemo u koraku svođenja po pravilu *polinom* $\rightarrow$ *polinom* + *polinom*, imati alocirana dva polinoma koja koristimo da izračunamo rezultat. Prilikom sabiranja polinoma alociramo novi polinom koji predstavlja rezultat. Nakon što rezultat izračunamo potrebno je da obrišemo operande i da taj rezultat postavimo kao vrednost neterminala sa leve strane pravila. Nakon svođenja po pravilu *naredba* $\rightarrow$ *polinom* i izvršavanja akcije, tj. štampanja polinoma, potrebno je da memoriju za polinom oslobodimo. Isto razmatranje važi za sve druge operacije.

Ako na ulazu imamo neku promenljivu, npr. `p1` . Tada će deo izvođenja biti *naredba* $\Rightarrow$ *polinom* $\Rightarrow$ *ID* . Potrebno je da ispratimo kompletno izvršavanje programa da bi nam bilo jasno šta se dešava sa memorijom. U lekseru, svaki put kada se prepozna token *ID*, pravimo kopiju prepoznate niske i tu kopiju prosleđujemo parseru preko adrese iz promenljive `yylval`. U koraku svođenja po pravilu *polinom* $\rightarrow$ *ID* koristimo tu kopiju naziva promenljive da bismo pretražili mapu i pronašli promenljivu. Nakon što pronađemo promenjivu u mapi, pravimo kopiju vrednosti, tj. polinoma koji joj je pridružen i dodeljujemo tu kopiju kao vrednost atributa neterminala *polinom*. Na taj način, možemo da koristimo vrednosti kako želimo bez bojazni da ćemo promeniti originalnu vrednost promenljive u kolekciji. Na kraju akcije, potrebno je da oslobodimo kopiju imena promenljive koju smo dobili iz leksera, jer nam dalje nije potrebna. Nakon što izvršimo svođenje po pravilu *naredba* $\rightarrow$ *polinom* i odštampamo vrednost polinoma koja je kopija vrednosti iz mape, potrebno je da i tu kopiju obrišemo. Ovakav postupak se sprovodi kada god je neki identifikator deo izraza.

Na kraju preostaje da oslobodimo i prostor koji zauzima mapa sa vrednostima promenljivih. Kako je mapa iz biblioteke C++, pozivaće se destruktor koji će ukloniti celu mapu, ključeve klase `string` i pokazivače na objekte klase `polinomial`. Da bi se zaista i uklonili polinomi na koje vode ti pokazivači neophodno je da svaki zasebno dealociramo. Izdvojili smo to u funkciju `void dealokacija()` i dodali poziv funkiciji na kraj `main` funkcije i pre izlaska iz funkcije `yyerror`.

Zadatak 8.2 Napisati program koji omogućava korisniku rad sa konačnim skupovima brojeva.

1. Skupovi se zadaju bilo navođenjem elemenata između `{` i `}`, bilo navođenjem intervala oblika `a..b`. Komandom `print` se ispisuju elementi skupa.
```
P = {3, 2, 2, -1};
Q = 7..10;
print P;                                    {-1, 2, 3}
print Q;                                    {7, 8, 9, 10}
```
2. Nad skupovima je moguće vršiti operacije unije (`\/`), preseka (`/\`) i razlike (`\`).
```
A = {1, 2} \/ {3, 4};
B = {3, 5, 7, 8} /\ {3, 4, 5, 7};
print A \/ B;                               {1, 2, 3, 4, 5, 7}
print A /\ B;                               {3}
print A \ B;                                {1, 2, 4}
```
3. Program treba da omogući i proveru da li je dati broj element skupa (`:`) i da li je jedan skup podskup drugog (`<`).
```
check 5 : A;                                 False
check 7 : A \/ B;                            True
check A < B;                                 False
check {5, 7} < B;                            True
```
4. Kardinalnost skupova se izračunava korišćenjem komande `card`. (3 poena)
```
card  {1, 3, 2, 7, 3, 4} \/ {6};             6
```
5. Definisati operaciju komplementiranja `~`. Komplementiranje se vrši u odnosu na univerzalni skup koji se zadaje kao `UniversalSet`. Ukoliko univerzalni skup nije prethodno definisan, program treba da prijavi grešku.
```
UniversalSet = 1..10;
C = {1, 3, 5, 7, 9};
print ~C;                                    {2, 4, 6, 8, 10}
```

■

Rešenje. Iz teksta zadatka jasno je da je potrebno da predstavimo skupove odgovarajućim tipom podataka u programskom jeziku C++. Takođe, potrebno je uočiti da u programu ne postoji podrška za multiskupove. Ovo ćemo ostvariti najlakše ako koristimo ugrađenu kolekciju podataka iz biblioteke C++ koja se zove `set`. Zatim, skupovne operacije ćemo implementirati preopterećivanjem operatora da bismo skratili zapis i učinili kod intuitivnijim.

Pored definisanja tipa klase Skup, potrebno je definisati i sam parser. Prilikom definisanja parsera neophodno je pažljivo razmotriti semantiku jezika, povesti računa o prioritetima operacija i tipovima naredbi. Takođe, prilikom definisanja akcija potrebno je voditi računo i o curenju memorije.

Specifikacija leksičkog analizatora

```
%option noyywrap
%option nounput
%{
  #include <string>
  #include "skup.hpp"
  #include "parser.tab.hpp"
```

```
%}

%%
UniversalSet  return UNIVERSE;
check         return CHECK;
card          return CARD;
print         return PRINT;
[A-Z][A-Za-z0-9]*   {
              yylval.ime= new std::string(yytext);
              return ID;  }
[={,};:<~]    return *yytext;
".."          return TT;
"\\/"         return U;
"/\\"         return P;
"\\"          return R;

-?[0-9]+   {  yylval.i_val = atoi(yytext);
              return BROJ;   }

[ \n\t]    { }
.  {
    fprintf(stderr, "Nepoznata leksema: \"%s\"\n",yytext);
    exit(1); }
%%
```

Rešenje 8.16: lexer.l

Specifikacija sintaksnog analizatora

```
%{
#include <iostream>
#include <cstdlib>
#include <string>
#include <map>
#include "skup.hpp"
#define YYDEBUG 1
using namespace std;

extern FILE *yyin;
extern int yylex();

map<string, Skup*> promenljive;
Skup* universe = nullptr;

void dealokacija();
void yyerror(string s);
%}

%union{
  std::string* ime;
```

```
22   int i_val;
     Skup *elementi;
24 }

26 %nonassoc ':' '<'
   %left U
28 %left P R
   %right '~'
30
   %token<ime> ID
32 %token<i_val> BROJ
   %token PRINT TT CHECK CARD UNIVERSE
34
   %type<elementi> skup lista
36
   %start program
38 %%
   program: program naredba    {}
40       |                     {}
         ;
42
   naredba: ID '=' skup ';' {
44         auto trazen = promenljive.find(*$1);

46         if (trazen != promenljive.end())
             delete trazen->second;
48
           promenljive[*$1] = $3;
50
           delete $1;
52       }
     | PRINT skup ';'          { std::cout << *$2;
54                               delete $2;
                                }
56   | CHECK BROJ ':' skup ';' { if( $4->nadji($2) )
                                   printf("True!\n");
58                               else
                                   printf("False!\n");
60
                                 delete $4;
62                              }
     | CHECK skup '<' skup ';' { if( $2->podskup($4) )
64                                 printf("True!\n");
                                 else
66                                 printf("False!\n");

68                               delete $2;
                                 delete $4;
70                              }
```

```
  | CARD skup ';' {
72                  cout << $2->brojElemenata() << endl;
                    delete $2 ;
74                }
  | UNIVERSE '=' skup ';' {
76                  if(universe != nullptr)
                      delete universe;
78                  else
                      universe = $3;   }
80 ;

82 skup:  '~' skup     {
            if(universe == nullptr)
84            yyerror("Univerzalni skup nije definisan.\n");

86          $$ = razlika(universe,$2);
            delete $2;
88        }
  | skup U skup   { $$ = unija($1,$3);
90                  delete $1;
                    delete $3;
92                }
  | skup P skup    { $$ = presek($1,$3);
94                  delete $1;
                    delete $3;
96                }
  |  skup R skup  { $$ = razlika($1,$3);
98                  delete $1;
                    delete $3;
100               }
  |'{' lista '}'    { $$ = $2; }
102 | BROJ TT BROJ    { $$ = new Skup($1, $3); }
  | ID         {
104           auto trazen = promenljive.find(*$1);

106           if (trazen == promenljive.end())
                yyerror(string("Nepostojeci skup ")+*$1);
108
              $$ = new Skup(*(trazen->second));
110           delete $1;
            }
112 ;

114 lista : lista ',' BROJ  {
              $$ = $1;
116           $$->dodaj($3);  }
  | BROJ    {
118           $$ = new Skup();
              if ($$ == NULL)
```

```
                yyerror("Alokacija novog skupa nije uspela");

              $$->dodaj($1);   }
  ;
%%


void yyerror(string s){
  cerr << s << endl;

  dealokacija();

  exit(EXIT_FAILURE);
}

void dealokacija(){
  for(auto i = promenljive.begin(); i!= promenljive.end(); i++)
    delete i->second;

  delete universe;
}

int main(){
  yydebug = 0;

  yyin = fopen("ulaz.txt","r");
  if(yyin == NULL)
    yyerror("Test file can't be open!\n");

  yyparse();

  fclose(yyin);

  dealokacija();

  return 0;
}
```

Rešenje 8.17: parser.ypp

Zaglavlje za klasu `Skup`

```
#ifndef _SKUP_HPP_
#define _SKUP_HPP_

#include <set>
#include <iostream>

class Skup {
  public:
```

```
    Skup() ;
    Skup(int pocetak, int kraj);

    void dodaj(int x);
    void obrisi(int x);

    void stampaj(std::ostream& out) const;

    bool nadji(int x) const;

    bool podskup( const Skup* veci  ) const;

    int brojElemenata() const;

  private:
    std::set<int> _elementi;

    friend Skup*  unija(const Skup* prvi, const Skup* drugi);
    friend Skup*  presek(const Skup* prvi, const Skup* drugi);
    friend Skup*  razlika(const Skup* prvi, const Skup* drugi);
};

std::ostream& operator << (std::ostream& out, const Skup& skup);

#endif
```

Rešenje 8.18: skup.hpp

Izvorni kod za klasu `Skup`

```
#include "skup.hpp"

Skup::Skup()
{}

Skup::Skup(int pocetak, int kraj){
  for (int i = pocetak; i <= kraj; i++)
    _elementi.insert(i);
}

void Skup::dodaj(int x) {
  _elementi.insert(x);
}

void Skup::obrisi(int x){
  if( nadji(x) )
    _elementi.erase(x);
}

void Skup::stampaj(std::ostream& out) const{
```

```
21    out << "{";
      for(auto i = _elementi.cbegin(); i != _elementi.cend(); i++) {
23      if (i != _elementi.cbegin())
          out << ", ";
25
        out << *i;
27    }
      out << "}" << std::endl;
29  }

31  bool Skup:: nadji(int x) const{
      return _elementi.find(x) != _elementi.end();
33  }

35  Skup*  unija(const Skup* prvi, const Skup* drugi){
      /* Kreiramo kopiju da bismo nju modifikovali*/
37    Skup *rezultat = new Skup(*prvi);

39    for(auto i = drugi->_elementi.cbegin();
                       i != drugi->_elementi.cend(); i++){
41      rezultat->dodaj(*i);
      }
43
      return rezultat;
45  }

47  Skup*  presek(const Skup* prvi, const Skup* drugi) {
      /* Kreiramo kopiju da bismo nju modifikovali*/
49    Skup *rezultat = new Skup(*prvi);

51    for(auto i = prvi->_elementi.cbegin();
                       i != prvi->_elementi.cend(); ) {
53      if( ! drugi->nadji(*i) ) // nije nadjen
          rezultat->obrisi(*i);  //brisemo ga
55      ++i;
      }
57
      return rezultat;
59  }

61  Skup*  razlika(const Skup* prvi, const Skup* drugi){
      /* Kreiramo kopiju da bismo nju modifikovali*/
63    Skup *rezultat = new Skup(*prvi);

65    for(auto i = prvi->_elementi.cbegin();
                     i != prvi->_elementi.cend(); i++) {
67      if( drugi->nadji(*i) ) // nadjen je
          rezultat->obrisi(*i);  //brisemo ga
69    }
```

```
71    return rezultat;
  }
73
  bool Skup::podskup( const Skup* veci  ) const {
75    for(auto i = _elementi.cbegin(); i != _elementi.cend(); i++) {
        if( ! veci->nadji(*i) )
77      return false;
      }
79    return true;
  }
81
  int Skup::brojElemenata() const{
83    return _elementi.size();
  }
85
  std::ostream& operator <<(std::ostream& out, const Skup& skup) {
87    skup.stampaj(out);
      return out;
89  }
```

Rešenje 8.19: skup.cpp

Zadatak 8.3 1. U programskom paketu *MATLAB*, svaka promenjiva predstavlja matricu, pri čemu se matrice zapisuju tako što se u uglastim zagradama navede niz vrsta razdvojenih sa `;`, dok su elementi jedne vrste brojevi razdvojeni sa belinama. Npr. matrica

$$\begin{pmatrix} 1.5 & 2 \\ 3 & 4.3 \end{pmatrix}$$

se zapisuje sa `[1.5 2; 3 4.3]`. Tekstualna datoteka sadrži niz linija oblika *imeprom=izraz* ili *imeprom=izraz;*, pri čemu su ispravni izrazi sastavljeni od matrica i prethodno definisanih promenjivih korišćenjem sledećih operatora:

`+`	sabiranje matrica
`-`	oduzimanje matrica
`*`	uobičajeno množenje matrica
`.*`	množenje dve matrice istih dimenzija "elemenat po elemenat"
`'`	transponovanje matrice

Napisati program koji ispisuje vrednost svih definisanih promenjivih, čija definicija nije terminisana sa `;`. Na primer za:

```
x = [1 2 3; 4 3 1]
y = [2 1; 3 1];
z = x' * y + [1 1; 2 2; 3 3]
```

program treba da ispiše:

```
x =
     1     2     3
     4     3     1
z =
```

```
    15     6
    15     7
    12     7
```

U slučaju da se operacija primenljuje na matrice ne saglasnih dimenzija prekinuti program uz poruku korisniku. Na primer za:

```
x = [1 2 3; 4 3 1]
y = [2 1; 3 1];
z = x + y
```

program treba da ispiše:

```
Matrice nisu saglasnih dimenzija
```

■

Rešenje. Specifikacija leksičkog analizatora

```
%option noyywrap
%option noinput
%option nounput

%{

#include <iostream>
#include <cstdlib>
#include <string>
#include <vector>

using namespace std;

#include "matrix.hpp"
#include "parser.tab.hpp"

%}

%%

[a-zA-Z_][a-zA-Z0-9_]*  {
              yylval.ime = new string(yytext);
              return ID;
            }
[+-]?[0-9]+(\.[0-9]+)?  {
              yylval.x = atof(yytext);
              return BROJ;
            }
[\[\];,*'=+\n.()-]      {return *yytext;}
[ \t]          {  }
.             {
              cerr<<"Leksicka greska: "<<yytext<<endl;
              exit(EXIT_FAILURE);
            }

%%
```

Rešenje 8.20: lexer.l

Specifikacija sintaksnog analizatora

```
%{
#include <iostream>
#include <cstdlib>
#include <string>
#include <vector>
#include <map>

#define YYDEBUG 1

using namespace std;

#include "matrix.hpp"

extern int yylex();

void yyerror(string s) {

  cerr  << endl << s << endl;
  exit(EXIT_FAILURE);
}

map<string, Matrix*> promenljive;

void deinicijalizuj(){

  for (auto it = promenljive.begin(); it != promenljive.end();
   it++)
    delete it->second;
}

%}

%union{
  double x;
  string* ime;
  Matrix* m;
  vector<double>* v;
}

%left '+' '-'
%left '*' '.'
%nonassoc '\''
%right UMINUS
```

```
%token<ime> ID
%token<x> BROJ

%type<m> izraz matrica niz_vrsta
%type<v>  vrsta

%start program

%%

program: naredba '\n' program  {}
  |              {}
  ;
naredba: ID '=' izraz  {
        auto it = promenljive.find(*$1);
        if (it != promenljive.end())
          delete it->second;

        promenljive[*$1] = $3;
        cout << (*$1) << " = " << endl << (*$3) << endl;
        delete $1;
      }
  | ID '=' izraz ';'  {
        auto it = promenljive.find(*$1);
        if (it != promenljive.end())
          delete it->second;

        promenljive[*$1] = $3;
        delete $1;
      }
  ;
izraz: izraz '+' izraz  { $$ = *$1 + *$3;
                          delete $1;
                          delete $3;
                          if( $$ == NULL)
                            yyerror("Matrice nisu saglasnih
   dimenzija");
                        }
  | izraz '-' izraz   { $$ = *$1 - *$3;
                        delete $1;
                        delete $3;
                        if( $$ == NULL)
                          yyerror("Matrice nisu saglasnih
   dimenzija");
                      }
  | izraz '*' izraz   { $$ = *$1 * *$3;
                        delete $1;
                        delete $3;
                        if( $$ == NULL)
```

```
                          yyerror("Matrice nisu saglasnih
   dimenzija");
92                      }
  | '-' izraz %prec UMINUS  { $$ = -(*$2);
94                          delete $2;
                          }
96  | izraz '\''            { $$ = $1->transpose();
                          delete $1;
98                        }
  | izraz '.''*' izraz    { $$ = $1->scalarMultiply(*$4);
100                         delete $1;
                            delete $4;
102                         if( $$ == NULL)
                              yyerror("Matrice nisu saglasnih
   dimenzija");
104                       }
  | '(' izraz ')'    { $$ = $2; }
106 | ID         {
                  auto it = promenljive.find(*$1);
108               if (it == promenljive.end()){
                    delete $1;
110                 yyerror("Ne postoji promenljiva");
                  }
112               $$ = new Matrix(*(it->second));
                  delete $1;
114             }
  | matrica                     { $$ = $1; }
116 ;
matrica: '[' niz_vrsta ']'      { $$ = $2; }
118 ;
niz_vrsta: niz_vrsta ';' vrsta  { $1->addRow(*$3);
120                               $$ = $1;
                                  delete $3;
122                             }
  | vrsta             { $$ = new Matrix();
124                     $$->addRow(*$1);
                        delete $1;
126                   }
  ;
128 vrsta: vrsta  BROJ  { $$ = $1;
                        $$->push_back($2);
130                   }
  | BROJ              { $$ = new vector<double>();
132                     $$->push_back($1);
                      }
134 ;
%%
136
int main(int argc, char** argv) {
```

```
138 //    yydebug = 1;

140   if (yyparse() != 0)
        yyerror("Previse tokena na ulazu");
142
      cout<<"Sve ok"<<endl;
144
      deinicijalizuj();
146   exit(EXIT_SUCCESS);
    }
```

Rešenje 8.21: parser.ypp

Zaglavlje za klasu `Matrix`

```
1  #ifndef MATRIX_HPP
   #define MATRIX_HPP
3
   #include <iostream>
5  #include <cstdlib>
   #include <vector>
7
   using namespace std;
9
   class Matrix{
11   public:
       Matrix() {};
13     Matrix(unsigned m, unsigned n);
       Matrix(Matrix& m);
15
       vector<double> operator [](unsigned i) const;
17     vector<double>& operator [](unsigned i);

19     Matrix* operator +(Matrix& m) const;
       Matrix* operator -(Matrix& m) const;
21     Matrix* operator *(Matrix& m) const;
       Matrix* operator -() const;
23
       Matrix* transpose() const;
25     Matrix* scalarMultiply(Matrix& m) const;

27     void show(ostream& s) const;

29     void addRow(vector<double>& v);

31   private:
       vector<vector<double>> _m;
33 };

35 ostream& operator <<(ostream& s, const Matrix& m);
```

```
37 #endif
```

Rešenje 8.22: matrix.hpp

Izvorni kod za klasu `Matrix`

```
#include <iostream>
2 #include <vector>

4 using namespace std;
#include "matrix.hpp"
6
Matrix::Matrix(unsigned m, unsigned n) {
8
  _m.resize(m);
10  for (unsigned i = 0; i < m; i++)
    _m[i].resize(n);
12 }

14 Matrix::Matrix(Matrix& m) {

16  _m = m._m;
}
18
vector<double> Matrix::operator [](unsigned i) const {
20
  return _m[i];
22 }

24 vector<double>& Matrix::operator [](unsigned i) {

26  return _m[i];
}
28
Matrix* Matrix::operator +(Matrix& m) const {
30
  if (_m.size() != m._m.size() || _m[0].size() !=
   m._m[0].size())
32    return NULL;

34  Matrix* newMatrix = new Matrix(_m.size(), _m[0].size());

36  for (unsigned i = 0; i < _m.size(); i++) {
    for (unsigned j = 0; j < _m[0].size(); j++) {
38      (*newMatrix)[i][j] = _m[i][j]+m._m[i][j];
    }
40  }

42  return newMatrix;
```

```
}

Matrix* Matrix::operator -(Matrix& m) const {

  if (_m.size() != m._m.size() || _m[0].size() !=
   m._m[0].size())
    return NULL;

  Matrix* newMatrix = new Matrix(_m.size(), _m[0].size());

  for (unsigned i = 0; i < _m.size(); i++) {
    for (unsigned j = 0; j < _m[0].size(); j++) {
      (*newMatrix)[i][j] = _m[i][j]-m._m[i][j];
    }
  }

  return newMatrix;
}

Matrix* Matrix::operator *(Matrix& m) const {

  if (_m[0].size() != m._m.size())
    return NULL;

  Matrix* newMatrix = new Matrix(_m.size(), m._m[0].size());

  for (unsigned i = 0; i < _m.size(); i++) {
    for (unsigned j = 0; j < m._m[0].size(); j++) {
      (*newMatrix)[i][j] = 0;
      for(unsigned k = 0; k < _m[0].size(); k++) {
        (*newMatrix)[i][j] += _m[i][k]*m._m[k][j];
      }
    }
  }

  return newMatrix;
}

Matrix* Matrix::operator -() const {

  Matrix* newMatrix = new Matrix(_m.size(), _m[0].size());

  for (unsigned i = 0; i < _m.size(); i++) {
    for (unsigned j = 0; j < _m[0].size(); j++) {
      (*newMatrix)[i][j] = -_m[i][j];
    }
  }

  return newMatrix;
```

```
}

Matrix* Matrix::transpose() const {

  Matrix* newMatrix = new Matrix(_m[0].size(), _m.size());

  for (unsigned i = 0; i < _m.size(); i++) {
    for (unsigned j = 0; j < _m[0].size(); j++) {
      (*newMatrix)[j][i] = _m[i][j];
    }
  }

  return newMatrix;
}

Matrix* Matrix::scalarMultiply(Matrix& m) const {

  if (_m.size() != m._m.size() || _m[0].size() !=
   m._m[0].size())
    return NULL;

  Matrix* newMatrix = new Matrix(_m.size(), _m[0].size());

  for (unsigned i = 0; i < _m.size(); i++) {
    for (unsigned j = 0; j < _m[0].size(); j++) {
      (*newMatrix)[i][j] = _m[i][j] * m._m[i][j];
    }
  }

  return newMatrix;
}

void Matrix::show(ostream& s) const {

  for (unsigned i = 0; i < _m.size(); i++) {
    for (unsigned j = 0; j < _m[0].size(); j++){
      s << _m[i][j] << " ";
    }
    s << endl;
  }
}

void Matrix::addRow(vector<double>& v) {

  _m.push_back(v);
}

ostream& operator <<(ostream& s, const Matrix& m) {

```

```
  m.show(s);
  return s;
}
```

Rešenje 8.23: matrix.cpp

9. Zadaci sa praktičnih provera

9.1 Zadaci sa kolokvijuma

9.1.1 Kolokvijum 2012. godine - Grupa I

Zadatak 9.1 Napisati *python* skript koji kao prvi argument komandne linije prihvata ime fajla koji je dobijen sistemom *UNITEX* kao rezultat prepoznavanja imenovanih entiteta. Na osnovu tog fajla treba napraviti *.xml* fajl koji je dobijen zamenom semantičkih markera odgovarajućim tagom (`+org` se menja tagom `<org>`). ■

Primer sadržaja *.STN* datoteke

```
{Kompanija "Air Serbia",.NE+org:1s} saopstila je da ce od {ponedeljka,
26. novembra,.NE+date:2s}, organizovati decembarsku promotivnu akciju
u okviru koje ce putnici moci za {99 evra,.NE+price:4p} da kupe povratne
karte do {Podgorice,.NE+city:2q}, {Tivta,.NE+city:2q},
{Sarajeva,.NE+city:2q} i {Skoplja,.NE+city:2q}.
```

Rezultat programa:

```
<xml>
<org>Kompanija "Air Serbia"</org> saopstila je da ce od <date>ponedeljka,
26. novembra</date>, organizovati decembarsku promotivnu akciju u okviru
koje ce putnici moci za <price>99 evra</price> da kupe povratne karte do
<city>Podgorice</city>, <city>Tivta</city>, <city>Sarajeva</city> i
<city>Skoplja</city>.
</xml>
```

Zadatak 9.2 Napisati program koji dobija preko komandne linije datoteku sa poenima studenata na zadacima po rokovima i od nje formira datoteku sa ocenama studenata. Pretpostavka je da je za jednog studenta dovoljan jedan red teksta koji sadrži, redom, alas nalog, ime i prezime i poene (od 0 do 100) na svakom od zadataka na ispitu.

Program treba da izdvaja studente koji su upisali fakultet od 2007 do 2012 i imaju broj indeksa manji od 350. Ime i prezime studenta su vlastita imena i kao takvao pišu se velikim početnim slovom. ■

Primer sadržaja ulazne datoteke:

```
Rezultati januarskog ispitnog roka
mm09123 Ana Marija Petrovic 80 90 55 100
mr05031 Pera Peric  34 55 87 30
mi10001 Ivan Ivanovic 12 70 100 34
Poeni sa junskog ispitnog roka
mv12244 Ana Marija Petrovic 80 90 55 100
mi07089 Stefan Stefanovic  28 100 100 30
```

Ispis programa:

```
Uspeh studenata
=====================================
mi10001   Ivan Ivanovic          6
mv12244   Ana Marija Petrovic    9
mi07089   Stefan Stefanovic      7
```

Rešenje.

```
# ukljucivanje potrebnih modula
import re, sys

# provera ulaznih argumenata
if len(sys.argv) != 2:
  exit('Program se poziva sa: python ' + sys.argv[0]
       + " STN datoteka")
if re.fullmatch(r"[\w-]+\.stn",sys.argv[1]) is None:
  exit('Prosledjena datoteka mora biti ekstenzije .stn!')

# Iz primera se vidi da ono sto treba da prepoznamo moze da se
# nalazi u vise redova, pa je ocigledno da moramo da ucitamo ceo
# sadrzaj datoteke u program
try:
  with open(sys.argv[1],"r") as f:
    sadrzaj = f.read()
except IOError:
  sys.exit('Neuspelo otvaranje i citanje datoteke '
    + sys.argv[1] )

# Uocavamo da je ono sto pretrazujemo uokvireno zagradama {},
# pa ce ivice izraza biti {}. Uocavamo da su pojedini delovi
# konstantni. To su ,.NE+ i :, koji su praceni nekim drugim
```

```
# karakterima. To sta se oko njih nalazi nama sluzi da napravimo
# xml tagove, pa treba da ih grupisemo radi lakse zamene.
# {(neki tekst),.NE+(tag):brojslovo}
#
# Sada ostaje samo da opisemo nedostajuce delove regularnog
# izraza neki tekst ocigledno moze da bude bilo sta, dakle .* Sa
# ovim treba da budemo oprezni, jer je zvezda pohlepna pa ce
# uzimati karaktere sve do poslednje }
# Da se to ne bi desilo treba da je ulenjimo sa ?
# neki tekst -> (.*?)
#
# Tag je neka kombinacija malih slova -> ([a-z]+)
# broj -> [0-9]
# slovo -> [a-z]
#
# Kada sve sklopimo dobijamo regularni izraz koji nam treba.
uzorak=re.compile(r'{(.*?),\.NE\+([a-z]+):[0-9][a-z]}',re.S)

# Sada preostaje da svako prepoznavanje sablona zamenimo
# ispravnim xml tagom. Koristimo bekreference iz regularnog
# izraza da bismo definisali tekst zamene.
# sub metod ne radi zamenu u mestu, vec kreira novi string
zamena = uzorak.sub(r"<\2>\1</\2>", sadrzaj)

# Upisujemo rezultate u datoteku, koja ima isto ime kao ulazna,
# ali ne ekstenzije .xml.
try:
  with open(sys.argv[1][:-3]+"xml","w") as fxml:
    # f.write ne dodaje nove redove kao sto to radi print,
    # vec direktno zapisuje bajtove
    fxml.write("<xml>\n")
    fxml.write(zamena)
    fxml.write("\n</xml>")
    # alternativno:
    # fxml.write("<xml>\n" + zamena + "\n</xml>\n")

except IOError:
    sys.exit('Neuspelo otvaranje i citanje datoteke jat.xml')
```

Rešenje 9.1: Rešenje prvog zadatka.

9.1.2 Kolokvijum 2012. godine - Grupa II

Zadatak 9.3 Napisati *python*-skript koji kao prvi argument komandne linije prihvata ime LaTeX fajla koji u sebi sadrži tabelu sa konačnim rezultatima ispita iz PPJ. Na osnovu ovog fajla treba generisati *.html* fajl koji sadrži tabelu sa ocenama sortiranim

opadajuće. Tabla treba imati tri kolone (*Rbr*, *ImePrezime*, *Ocena*). ■

Primer sadržaja *.TEX* datoteke

```
\begin{table}[ht]
\caption{Rezultati iz PPJ}
\begin{tabular}{c c c c c}
Rbr & Ime & Prezime & Kol & Ispit \\
1 & Marko & Markovic & 30 & 45 \\
2 & Branko & Brankovic & 20 & 25 \\
3 & Janko & Jankovic & 30 & 51 \\
\end{tabular}
\end{table}
```

Rezultat programa:

```
<html>
<body>
  <table>
    <caption>Rezultati iz PPJ</caption>
    <tr><th>Rbr</th><th>ImePrezime</th><th>Ocena</th></tr>
    <tr><th>1</th><th>Janko Jankovic</th><th>9</th></tr>
    <tr><th>2</th><th>Marko Markovic</th><th>8</th></tr>
    <tr><th>3</th><th>Branko Brankovic</th><th>5</th></tr>
  </table>
</body>
</html>
```

Zadatak 9.4 Datoteka sadrži listing korisnika mobilne mreže u Srbiji u kom se nalazi broj kome je upućen poziv ili poruka i koliko je B ili KB podataka skinuto sa mreze. Broj mobilnog telefona počinje sa 06 i moze imati najviše 10 cifara. Programu se datoteka šalje preko argumenta komandne linije. Program treba da ispiše na ekran koliko je minuta razgovora potrošeno, koliko je sms poslato i kB podataka skinuto. ■

Primer sadržaja ulazne datoteke:

```
Protok saobracaja za: Novembar 2012
-----------------------------------
063123000  12 min. 12 sec.
0651231234 3 min. 3 sec.
DATA 1204 B
060987654 SMS
0611234567 33 min.
062008765 52 sec.
DATA 23 kB
```

Ispis programa:

```
Poruka: 1 Razgovori: 49 min 7 sec. Data: 24.175781kB
```

9.1.3 Kolokvijum 2017. godine, Grupa I

Zadatak 9.5 Napisati *Python* skript koji iz datoteke ekstenzije *.csv* (npr. *igraci.csv*) čita statističke podatke o ukupnom uspehu fudbalera. Putanja do datoteke se skriptu prosleđuje kao prvi argument komandne linije. Ukoliko je prosleđena putanja do datoteke koje nema ekstenziju *.csv*, obavestiti korisnika i prekinuti rad skripta.

Datoteka sadrži sledeće informacije:

ime i prezime igrača - sastavljeno od najviše 3 reči od kojih je svaka zapisana slovima. Prva i poslednja obavezno počinju velikim slovom, praćenim malim slovima, ostale mogu biti u tom obliku ili samo od malih slova.
država - jedna reč. Prvo slovo je veliko,a ostala su mala.
broj golova - ceo broj > 0
broj utakmica - ceo broj > 0
godina1-godina2 - godina1 označava početak karijere, godina2 kraj karijere, ukoliko je nastupio, inače nije navedena. Godina se piše sa 4 cifre.
klubovi - Lista klubova u kojima je igrač igrao tokom karijere, razdvojeni karakterom `,` eventualno praćenim belinama. Ime kluba pocinje velikim slovom ali može sadržati i više reči sastavljenih od slova i brojeva.

Podaci u liniji su razdvojeni karakterom `,` i eventualnim belinama. Ukoliko neke linije ne zadovoljavaju navedenu specifikaciju, ne izdvajati ih.

Skriptu se prosleđuju i ostali argumenti, i to ako se prosledi:

- `-g` - skript na standardni izlaz opadajuće sortiran spisak igrača po broju datih golova po utakmici. Prikazuje se ime, prezime i broj datih golova po utakmici *broj golova / broj utakmica*.
- `-t KLUB` - skript izdvaja sve igrače koji su nekada igrali u klubu `KLUB`. Prikazuje se ime, prezime i početak i dužina trajanja karijere. Ako kraj karijere nije poznat, oduzeti početak karijere od 2018. godine.

■

Primer sadržaja datoteke *rezultati.csv*

```
Cristiano Ronaldo,Portugal,111,144,2003-,Manchester United,Real Madrid
Lionel Messi,Argentina,97,119,2005-,Barcelona
Raul,Spain,71,142,1995-2011,Real Madrid,Schalke 04
Ruud van Nistelrooy,Netherlands,56,73,1998-2009,PSV Eindhoven,Manchester United,Real Madrid
Karim Benzema,France,51,97,2006-,Lyon,Real Madrid
Thierry Henry,France,50,112,1997-2012,Monaco,Arsenal,Barcelona
Alfreda Di Stefano,Argentina,49,58,1955-1964,Real Madrid
```

Primeri izvršavanja programa:

```
python 1.py igraci.csv -g
----------------------------
Alfredo Di Stefano   0.84
Lionel Messi         0.82
Cristiano Ronaldo    0.77
Ruud van Nistelrooy  0.77
```

```
Karim Benzema          0.53
Raul                   0.50
Thierry Henry          0.45

python 1.py rezultati.csv -t Manchester United
-----------------------------------------------
Cristiano Ronaldo     2003  15
Ruud van Nistelrooy   1998  11
```

Zadatak 9.6 Napisati niz poziva `grep` i `sed` kojima se iz datoteke *igraci.csv* izdvajaju samo informacije o igračima koji odigrali bar 100 utakmica pre nego što su završili karijeru. Izlaz formatirani na sledeći način:

```
Ime (Država) [godina1 - godina2] : klub1 -> klub2 -> klub3
Ime (Država) [godina1 - godina2] : klub1
...
```

Smatrati da je datoteka struktuirana na isti način kao što je navedeno u 1. zadatku, ali ne treba proveravati ništa više od uslova iz ovog zadatka. ■

Primer izvršavanja programa:

```
Raul (Spain) [1995-2011]: Real Madrid -> Schalke 04
Thierry Henry (France) [1997-2012] : Monaco -> Arsenal -> Barcelona
```

Rešenje.

```
import re, sys

if len(sys.argv) <3:
  exit('Nedovoljno argumenata')

gol = False
ekipa = False
klub = ""

for a in sys.argv[2:] :
  if( a == "-g") :
    gol = True
  elif( a == "-t"):
    ekipa = True
  elif ekipa == True:
    klub = a
  else:
    exit('Neispravan argument'+a)

if( re.match(r"^.*\.csv$",sys.argv[1]) == None):
    exit(sys.argv[1]+"nije csv ")

igraci = []

try:
```

```
  with open(sys.argv[1],"r") as f:
    datoteka = f.readlines()
except IOError:
    exit('Neispravno otvaranje datoteke')

reg = re.compile(
  r"^([A-Z][a-z]+\s*(?:[A-Za-z]*\s+[A-Z][a-z]*)?)"
  + r",\s*[A-Z][a-z]+,\s*(0|[1-9]\d*)"
  + r",\s*(0|[1-9]\d*),\s*(\d{4})-(\d{4})?,"
  + r"\s*([A-Z][a-zA-Z\d]+[ a-z\dA-Z]*"
  + r"(?:,\s*[A-Z][a-z\d]+[ a-z\dA-Z]+)*)")

for linija in datoteka:
  m = reg.search(linija)
  if m is not None:
    #print( m.groups())
    igraci.append(m.groups())

if gol :
  igraci.sort(reverse = True, key= lambda x:
   int(x[1])/float(x[2]))
  for i in igraci:
    print( "%s %.2f" %(i[0], int(i[1])/float(i[2])) )

elif ekipa:
  for i in igraci:
    if klub in i[5]:
      print( i[0], i[3],end=" " )
      if(i[4] is not None):
        print( int(i[4]) - int(i[3]) )
      else:
        print( 2018 - int(i[3]) )
```

Rešenje 9.2: Rešenje prvog zadatka

```
egrep '[A-Za-z ]+, *[0-9]+, *[1-9][0-9][0-9]+,' igraci.csv |
sed -r 's/([A-Za-z ]+),\s*([A-Za-z]+),\s*[0-9]+,\s*[0-9]+,\
\s*([0-9]+)-([0-9]+)?,\s*(.*)/\1 (\2) [\3 - \4] : \5/' |
sed -r 's/,\s*/ -> /g'
```

Rešenje 9.3: Rešenje drugog zadatka

9.1.4 Kolokvijum 2017. godine, Grupa II

Zadatak 9.7 *Dejna Vajt* (eng. Dana White), vlasnik organizacije *UFC* (eng. Ultimate Fighting Championship) sporta *MMA* (eng. mixed martial arts) se sprema da organizuje 223. po redu događaj. Na raspolaganju mu je datoteka koja sadrži sve prethodne

borbe organizacije UFC. Odabrao je baš Vas da mu napišete potrebni *Python* skript koji će mu pomoći da odredi potencijalne mečeve.

Napisati *Python* skript koji iz datoteke ekstenzije *.csv* (npr. *rezultati.csv*) čita podatke o rezultatima borbi u sportu *MMA* organizacije *UFC*. Putanja do datoteke se skriptu prosleđuje kao prvi argument komandne linije. Ukoliko je prosleđena putanja do datoteke koje nema ekstenziju *.csv*, obavestiti korisnika i prekinuti rad skripte.

Datoteka sadrži sledeće informacije:

- `event_date`: Datum borbe oblika `mm/dd/gggg`, pretpostaviti da svaki mesec ima 31 dan, godina 12 meseci, a godina zapisana sa 4 cifre.
- `f1name`: Ime i prezime prvog borca
- `f2name`: Ime i prezime drugog borca
- `f1result`: Rezultat prvog borca (`win` ili `loss`)
- `f2result`: Rezultat drugog borca (`win` ili `loss`)
- `method`: Način na koji je završen meč (`Submission`, `Decision` ili `KO`)
- `ref`: Ime sudije
- `round`: Runda u kojoj je okončan meč (`1`, `2`, `3`, `4` ili `5`)
- `time`: Vreme u rundi u kojoj je okončan meč (`mm:ss`)

Pri čemu za imena važi da je prvo slovo i imena i prezimena veliko, a ostala su mala. Takođe, svi imaju 1 ime i 1 prezime. Separator u liniji je karakter `;`.

Skriptu se prosleđuju i ostali argumenti, i to ako se prosledi:

- `-b` - skript na standardni izlaz prikazuje ime, prezime i broj pobeda borca koji je pobedio u najviše mečeva
- `-m METHOD` - skript izdvaja sve borce koji su ostvarili barem jednu pobedu na način `METHOD`

Primer sadržaja *.csv* datoteke

```
2/21/2016;Donald Cerrone;Alex Oliveira;win;loss;Submission;Mario Yamasaki;1;2:33
2/21/2016;Derek Brunson;Roan Carneiro;win;loss;KO;Keith Peterson;1;2:38
1/3/2015;Jon Jones;Daniel Cormier;win;loss;Decision;Herb Dean;5;5:00
4/26/2014;Jon Jones;Glover Teixeira;win;loss;Decision;Dan Miragliotta;5;5:00
9/21/2013;Jon Jones;Alexander Gustafsson;win;loss;Decision;John McCarthy;5;5:00
4/21/2012;Jon Jones;Rashad Evans;win;loss;Decision;Herb Dean;5;5:00
11/19/2011;Wanderlei Silva;Cung Le;win;loss;KO;Dan Stell;2;4:49
2/20/2010;Mirko Filipovic;Anthony Perosh;win;loss;KO;Herb Dean;2;5:00
9/19/2009;Junior Santos;Mirko Filipovic;win;loss;Submission;Dan Miragliotta;3;2:00
```

Primeri poziva programa:

```
python 1.py rezultati.csv -m Submission
----------------------------------------
 Junior Santos
 Donald Cerrone
```

```
python 1.py rezultati.csv -b
----------------------------
 Jon Jones 4
```

Zadatak 9.8 Napisati niz poziva `grep` i `sed` kojima se iz datoteke o *MMA* borbama prikazuju sve borbe održane 2016. godine koje su završene u prvoj rundi. Izlaz formatirani na sledeći način:

```
2016: Ime1 Prezime1 (win) vs Ime2 Prezime2 (loss)
2016: Ime3 Prezime3 (loss) vs Ime3 Prezime3 (win)
...
```

■

Primer izlaza:

```
2016: Donald Cerrone (win) vs Alex Oliveira (loss)
2016: Derek Brunson (win) vs Roan Carneiro (loss)
```

9.1.5 Kolokvijum 2018. godine, Grupa I

Zadatak 9.9 Python predstavlja jedan od popularnijih programskih jezika za koji postoji veliki broj biblioteka i paketa. Kako bi se olakšao rad, Python zajednici je potreban prototip softvera koji predstavlja menadžer paketa. Upravo ste odabrani Vi da razvijete navedeni protitip čije je kodno ime `matfpip`.

Datoteka `paketi.xml` sadrži metapodatke o paketima koji su trenutno dostupni. Navedeni su sledeći podaci:

- naziv paketa - jedna reč koja predstavlja naziv paketa
- verzija paketa u formatu `X.Y.Z` gde su X, Y i Z brojevi ≥ 0
- opis - niska koja daje kratak opis paketa
- repozitorijum - jedna reč koja može biti `github`, `gitlab` ili `bitbucket`
- veb strana paketa - veb strana na kojoj se nalazi prezentacija i dokumentacija paketa, pri čemu je deo `www.` izostavljen kod nekih adresa i adrese se završavaju ili sa `org` ili `com`

Ukoliko se skript pozove sa jednom od opcija:

- `-a` - ispisuju se svi paketi sortirani leksikografski po nazivu
- `-v NAZIV` - ispisuje se verzija paketa sa nazivom `NAZIV`
- `-w NAZIV` - ispisuje se veb lokacija paketa sa nazivom `NAZIV`
- `-r NAZIV` - ispisuje se ime repozitorijuma paketa sa nazivom `NAZIV`
- `-o NAZIV` - ispisuje se opis paketa sa nazivom `NAZIV`

■

Primer sadražaja datoteke *paketi.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<paketi>
 <paket id="1">
  <naziv>pandas</naziv>
  <verzija>0.23.3</verzija>
 <opis>Biblioteka za rad sa podacima.
  </opis>
                    ...
 <paket id="4">
   <naziv>tester</naziv>
   <verzija>2.3</verzija>
<opis>Biblioteka za testiranje koda.
   </opis>
```

```
  <repo>github</repo>
  <veb>pandas.pydata.org</veb>
 </paket>
 <paket id="2">
  <naziv>pytorch</naziv>
  <repo>bitbucket</repo>
  <veb>www.pytorch.org</veb>
  <verzija>0.4.1</verzija>
  <opis>Biblioteka za duboko ucenje.
  </opis>
 </paket>
 <paket id="3">
  <naziv>matplotlib</naziv>
  <verzija>3.0.2</verzija>
  <repo>gitlab</repo>
  <opis>Vizuelizacija podataka.</opis>
  <veb>www.matplotlib.org</veb>
 </paket>
  ...
```

```
  <repo>github</repo>
  <veb>tonytester.com</veb>
 </paket>
 <paket id="5">
  <naziv>los naziv paketa</naziv>
  <verzija>1.1.1</verzija>
  <veb>lnp.org</veb>
  <opis>Highway to heaven</opis>
  <repo>bitbucket</repo>
 </paket>
 <paket id="6">
  <naziv>los primer</naziv>
  <verzija>3.0.</verzija>
  <repo>bitbucket</repo>
  <opis>Los Python paket.</opis>
  <veb>www.matplotlib.</veb>
 </paket>
</paketi>
```

Primeri poziva programa:

```
./01.py -a
------------------
[matplotlib] v3.0.2 gitlab
www.matplotlib.org
Vizuelizacija podataka.

[pandas] v0.23.3 github
pandas.pydata.org
Biblioteka za rad sa podacima.

[pytorch] v0.4.1 bitbucket
www.pytorch.org
Biblioteka za duboko ucenje.
```

```
./1.py -v pandas
---------------------
Verzija: 0.23.3

./1.py -w pytorch
---------------------
Veb: www.pytorch.org

./1.py
---------------------
Neispravni argumenti.
```

```
./1.py -v
-----------------
Neispravni argumenti.
```

Zadatak 9.10 Konstruisati i implementirati automat koji prihvata binarne brojeve koji se završavaju sa 00 ili 11. Automat implementirati u programskom jeziku Python. Za unos sa standardnog ulaza koristiti funkciju `input()` iz Python-a 3. Očekivati da korisnik unosi karakter po karakter, a ne celu reč. ■

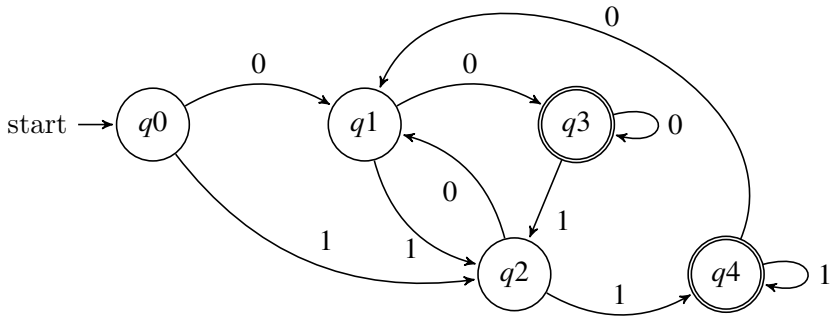
Rešenje.

```
#! /usr/bin/env python
import os, sys, re

def get_package_details(data):
  try:
    naziv = re.findall('<naziv>(.*?)</naziv>', paket_data)[0]
    repo = re.findall('<repo>(.*?)</repo>', paket_data)[0]
    veb = re.findall(
```

```
      '<veb>((www\.)?([a-z-A-Z]+\.)+(org|com))</veb>',
      paket_data)[0][0]
    opis = re.findall('<opis>(.*?)</opis>', paket_data)[0]
    verzija = re.findall(
      '<verzija>(\d+\.\d+\.\d+)</verzija>', paket_data)[0]
  except IndexError:
    # IndexError ce biti ispaljen za slucaj da regularni
    # izraz ne pronadje potrebne informacije pa se pokusava
    # indeksiranje u vracenim rezultatima.
    return None

  return {'naziv'  : naziv,
          'repo'   : repo,
          'opis'   : opis,
          'veb'    : veb,
          'verzija': verzija }

# Proveravamo argumente komandne linije
if not (len(sys.argv) >= 2 and len(sys.argv) <= 6):
  sys.exit('usage:\n\tmatfpip.py [-v -o -r -w]
   name\n\tmatfpip.py -a')

try:
  with open("paketi.xml", "r") as f:
    data = f.read()
except IOError:
  sys.exit('Failed opening paketi.xml')

# Mapa koja sadrzi indikatore o prosledjenim argumentima.
args = { 'naziv' : False,
         'repo'   : False,
         'verzija': False,
         'veb'    : False,
         'opis'   : False,
         'sve'    : False  }

package_name = None

if '-a' in sys.argv:
  args['sve'] = True

else:
  if '-o' in sys.argv:
    args['opis'] = True
  if '-r' in sys.argv:
    args['repo'] = True
  if '-v' in sys.argv:
    args['verzija'] = True
  if '-w' in sys.argv:
```

```
    args['veb'] = True
  if sys.argv[-1][0] == '-':
    exit("Missing package name")
  else:
    package_name = sys.argv[-1]

# Regularni izraz koji izvlaci jedan paket iz xml datoteke.
reg = re.compile(r'<paket\s*id="[0-9]+">.*?</paket>', re.DOTALL)

packages = []

for paket in reg.finditer(data):
  paket_data = paket.group()
  package_details = get_package_details(paket_data)

  if package_details is not None:
    #print('Adding {}'.format(package_details['naziv']))
    packages.append(package_details)

# Proveravamo prosledjene argumente.
if args['sve']:
  # Sortiramo
  packages.sort(key=lambda x: x['naziv'])
  for pack in packages:
    print('[{}] v{} {}\n{}\n{}'
          .format(pack['naziv'], pack['verzija'],
                  pack['repo'], pack['veb'], pack['opis']))
    print()
else:
  # Trazimo paket koji je uneo korisnik
  i = -1
  for k, p in enumerate(packages):
    if p['naziv'] == package_name:
      i = k
      break
  if i != -1:
    if args['opis']:
      print('Opis: {}'.format(packages[i]['opis']))
    if args['verzija']:
      print('Verzija: {}'.format(packages[i]['verzija']))
    if args['veb']:
      print('Opis: {}'.format(packages[i]['veb']))
    if args['repo']:
      print('Opis: {}'.format(packages[i]['repo']))
  else:
    print('Paket {} nije pronadjen.'.format(package_name))
```

Rešenje 9.4: Rešenje zadatka 9.9



Slika 9.1: Automat za drugi zadatak

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
import sys


def main():
  user_input = ""

  print('Unesite cifru po cifru.\nDozvoljene cifru su 0 i
   1.\nUlaz prekidate signalom EOF (Ctrl+D).')

  stanje = 1
  zavrsna = [4, 5]
  prelaz = {
    (1, 0): 2,
    (1, 1): 3,
    (2, 0): 5,
    (2, 1): 3,
    (3, 0): 2,
    (3, 1): 4,
    (4, 0): 2,
    (4, 1): 4,
    (5, 0): 5,
    (5, 1): 3,
  }

  while True:
    try:
      c = input()
      user_input += c
      c = int(c)
      if c != 0 and c !=1:
        raise ValueError('Nije uneta ni 0 ni 1')
    except EOFError:
      break
    except ValueError as e:
```

```
36        print(e)
          sys.exit()
38
      stanje = prelaz[(stanje, c)]
40    print("Trenutno stanje: {}".format(stanje))


42
  print("Korisnicki unos: ", user_input)
44  if stanje in zavrsna:
      print("Ulaz je u odgovarajucem obliku.")
46  else:
      print("Greska u ulazu.")
48


50 if __name__ == "__main__":
    main()
```

Rešenje 9.5: Rešenje zadatka 9.10

9.1.6 Kolokvijum 2018. godine, Grupa II

Zadatak 9.11 Napisati *Python* skript kojim se omogućava pretraga video igara. Datoteka `igre.xml` sadrži metapodatke o igrama svih vremena. Navedeni su sledeći podaci i obavezno u sledećem redosledu:

naziv igre - može biti od više reči i sa interpunkcijskim znakovima poput .,():‘ i ciframa
godina izdanja - četvorocifren ceo broj, takav da važi $1950 \leq ocena \leq 2018$.
izdavač - naziv izdavača može imati više reči, svaka počinje velikim slovom ili cifrom i sadrži cifre ili slova
platforme - lista podržanih platformi koje su razdvojene zarezom, čiji nazivi se sastoje od slova i cifara.
ocena - realan broj sa jednom cifrom iza decimalne tačke takav da važi $0 \leq ocena \leq 10$.

Ukoliko se skript pozove sa samo jednom od opcija:

- `-sve` - ispisuju se nazivi svih igara sortrani rastuće leksikografski.
- `-info REC` - ispisuju se nazivi i, u zagradama, ocene igara koje u svom nazivu imaju reč `REC`, bez obzira na veličinu slova
- `-i NAZIV` - ispisuju se nazivi i, u zagradama, godine objavljivanja igara izdavača sa nazivom `NAZIV`
- `-p NAZIV` - ispisuju se nazivi svih igara koje se mogu igrati na platformi sa nazivom `NAZIV`, sortirane opadajuće po oceni.

Naziv platforme može biti neki od sledećih: `PlayStation, Xbox, Nintendo, PC` ■

Primer sadražaja datoteke *igre.xml* ——————————————-

```
<?xml version="1.0" encoding="utf-8"?>                    ...
<igre>                                  <igra id="4">
 <igra id="1">                           <naziv>World of Goo</naziv>
```

```
  <naziv>The Legend of Zelda:
     Breath of the Wild </naziv>
  <godina>2017</godina>
  <izdavac>Nintendo</izdavac>
  <platforme>Nintendo</platforme>
  <ocena>8.2</ocena>
 </igra>
 <igra id="2">
  <naziv>Grand Theft Auto IV</naziv>
  <godina>2008</godina>
  <izdavac>Rockstar Games</izdavac>
  <platforme>PlayStation, Xbox, PC
  </platforme>
  <ocena>7.5</ocena>
 </igra>
 <igra id="3">
  <naziv>The Elder Scrolls V: Skyrim</naziv>
  <godina>2011</godina>
  <izdavac>Bethesda</izdavac>
  <platforme>PlayStation, Xbox, Nintendo
  </platforme>
  <ocena>8.6</ocena>
 </igra>
     ...
  <godina>2008</godina>
  <izdavac>2D Boy</izdavac>
 <platforme>PC, Nintendo</platforme>
  <ocena>8.8</ocena>
 </igra>
 <igra id="5">
<naziv>Red Dead Redemption 2</naziv>
  <godina>2018</godina>
  <izdavac>Rockstar Games</izdavac>
  <platforme>Xbox, PlayStation
  </platforme>
  <ocena>8.9</ocena>
  /igra>
</igre>
```

Primeri poziva programa:

```
./1.py -sve
------------------
Grand Theft Auto IV
Red Dead Redemption 2
The Elder Scrolls V: Skyrim
The Legend of Zelda: Breath of the Wild
World of Goo

./1.py -info the
-------------------------------------------------
Grand Theft Auto IV (7.5/10)
The Elder Scrolls V: Skyrim (8.6/10)
The Legend of Zelda: Breath of the Wild (9.2/10)

./1.py -p Nintendo
---------------------------------------
The Legend of Zelda: Breath of the Wild
The Elder Scrolls V: Skyrim
World of Goo
```

```
./1.py -info
-----------------
Neispravni argumenti.

./1.py
----------------------
Neispravni argumenti.

./1.py -i "2D Boy"
--------------------
World of Goo (2008)
```

Zadatak 9.12 Konstruisati i implementirati automat koji prihvata reči napisane slovima a, b, c tako da je početni deo reči sastavljen samo od slova a i c, a ostatak počinje bar

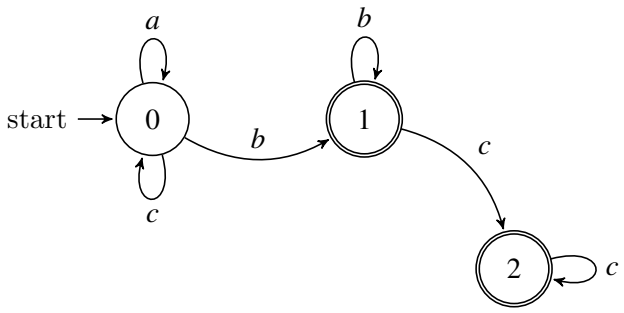
jednim slovom `b`, praćenim sa 0 ili više slova `c`. Na primer, automat treba da prihvati reči `acacab` i `bbccc`, ali ne prihvata reč `ac`.. Automat implementirati u programskom jeziku Python. Za unos sa standardnog ulaza koristiti funkciju `input()` iz *Python 3*. Očekivati da korisnik unosi karakter po karakter, a ne celu reč. ■

Rešenje.

```
import re, sys

if len(sys.argv) < 2 :
  exit('Nedovoljan broj argumenata')

sve =False
izdavac = False
platforma= False
info = False

if sys.argv[1] == "-sve":
  sve = True
else:
  if len(sys.argv) ==2:
    exit("Nedovoljno argumenata")
  else:
    rec= sys.argv[2]

  if sys.argv[1] == '-i':
    izdavac = True
  elif sys.argv[1] == '-p':
    platforma= True
  elif sys.argv[1] == '-info' :
    info = True
  else:
    exit("Nepodrzana opcija")

regex = re.compile( r'<igra\s+id="(\d+)">\s*'
  + r"<naziv>\s*(?P<naslov>([a-zA-Z0-9,.:'()]+ ?)+)\s*"
  + r"</naziv>\s*"
  + r"<godina>\s*(?P<godina>19[5-9]\d|200\d|201[0-8])\s*"
  + r"</godina>\s*"
  + r"<izdavac>\s*(?P<izdavac>([0-9A-Z][a-zA-Z0-9]*\s?)+)\s*"
  + r"</izdavac>\s*"
  + r"<platforme>\s*(?P<platform>(PlayStation|Xbox|PC|Nintendo)"
  + r"(,\s*(PlayStation|Xbox|PC|Nintendo))*)\s*</platforme>\s*"
  + r"<ocena>\s*(?P<ocena>\d\.\d|10\.0)\s*</ocena>\s*"
  + r"</igra>" )

try:
  with open("igre.xml","r") as f:
    datoteka = f.read()
except IOError:
```

```
  exit("Ne mogu da citam podaci.xml")
45
zbirka ={}
47
for m in regex.finditer(datoteka):
49  zbirka[m.group(1)] = [m.group('naslov'),m.group('godina'),
                          m.group('izdavac'), m.group('ocena'),
51                        re.split(', *',m.group('platform'))]

53if sve:
  lista = list(zbirka.values())
55  lista.sort()
  for  i in lista:
57    print(i[0])

59if izdavac :
  for v in zbirka.values():
61    if rec == v[2]:
      print(v[0], "(",v[1],")")
63
if info:
65  for v in zbirka.values():
    if re.search(r''+rec,v[0],re.I) is not None:
67      print (v[0],"(",v[3],")")

69if platforma:
  for v in zbirka.values():
71    if rec in v[-1]:
      print (v[0])
```

Rešenje 9.6: Rešenje zadatka 9.11



Slika 9.2: Automat za drugi zadatak

```
2stanje = 0
prelaz= {0:{'a':0,'b':1,'c':0}, 1:{'b':1, 'c':2} , 2:{'c':2} }
4zavrsna =[ 1,2]

6rec=""
```

```
while(True):
  try:
    c= input()
    if (c!='a' and c!='b' and c!= 'c'):
      raise ValueError('Invalid input character')

    if (c not in prelaz[stanje].keys()):
      print("Automat ne prihvata unetu rec!")
      exit()
    else:
      stanje = prelaz[stanje][c]
      rec+=c

  except EOFError:
    break
  except ValueError as v:
    exit( v)

if (stanje in zavrsna):
    print("Rec prihvacena automatom "+ rec)
else:
    print("Automat nije prihvatio rec " + rec)
```

Rešenje 9.7: Rešenje zadatka 9.12

9.2 Zadaci sa ispita

9.2.1 Februar 2016. godine

Zadatak 9.13 Napisati program koji omogućava korisniku rad sa iskaznim formulama.

1. Iskazne formule se dobijaju rekurzivno od logičkih promenljivih (čija imena se sastoje od proizvoljnog broja malih slova engleske abecede i cifara), logičkih konstanti `True` i `False`, konačnom primenom konjunkcije, disjunkcije, implikacije, ekvivalencije i negacije nad već definisanim formulama. Formule mogu da sadrže i zagrade. Napraviti interpreter koji proverava sintaksnu ispravnost formula koje se unose sa ulaza.
   ```
   p /\ q <=> True
   !p => q \/ r <=> s
   p1 /\ (p2 <=> !p3)
   p /\ /\ r                    syntax error
   ```
2. Interpreteru napravljenom u delu pod a), dodati naredbu dodele:
   ```
   p := True
   q := False
   r := p <=> q
   ```
3. Unaprediti interpreter tako da računa vrednost zadate formule:
   ```
   p := True
   q := False
   r := p <=> q
   p /\ q <=> r <=> True        True
   !p => q \/ r <=> s           Variable s is not defined.
   ```
4. Omogućiti da u slučaju korišćenja promenljive koja dotada nije definisana, program ipak sračuna vrednost formule, na taj način što će dati konkretnu vrednost (`True` ili `False`) ako vrednost formule ne zavisi od neinicijalizovane promenljive, a inače je vrednost formule nova logička konstanta `Undef`.
   ```
   p := True
   q := False
   p /\ q \/ s                  Undef
   p \/ s \/ q                  True
   !(q => s)                    False
   ```

■

Zadatak 9.14 Korišćenjem tehnike rekurzivnog spusta napisati sintaksni analizator koji bi proveravao ispravnost iskazne formule unete sa standarnog ulaza. Iskazne formule se dobijaju rekurzivno od logičkih promenljivih, logičkih konstanti `True` i `False` , konačnom primenom samo konjunkcije, disjunkcije i negacije.

Napomena: Koristiti leksički analizator iz prethodog zadatka. Za testiranje koristiti uprošćen test primer dela pod a) koji sadrži od operatora samo disjunkciju, konjunkciju i negaciju. Prioritet negacije je najveći, a disjunkcije najmanji. ■

9.2.2 Januar 2019. godine

Zadatak 9.15 Napisati interpretator za minijaturni programski jezik koji podržava rad sa označenim 32-bitnim celim brojevima. Brojevne konstante se mogu navoditi zapisane u dekadnom, oktalnom i heksadekadnom brojevnom sistemu. Dekadne konstante se zapisuju uobičajeno, heksadekadne sa prefiksom 0x, a oktalne sa prefiksom 0.

Jezik raspolaže svim aritmetičkim (+,-,*,/) i bitskim operatorima (&, |, ^, ~, «, ») i relacionim operatorom ==. Svi su po uzoru na operatore iz programskog jezika C.
Funkcija *print*(x,b) ispisuje brojne vrednosti u osnovi b. Pri čemu osnova može biti samo neka od podržanih zapisa za konstante.
Moguće je korišćenje promenljivih, ali svaka mora biti deklarisana pre korišćenja. Imena promenljivih počinju znakom _ iza koje obavezno sledi slovo, iza koga mogu i ne moraju se javiti slova ili cifre.

Programski jezik podržava i rad sa listama označenih celih brojeva. Podržane su operacije ispisa elementa liste i pristup elementima preko indeksa. Identifikator liste počinje uvek malim slovom `l` koje je praćeno sa bar jednim slovom, cifrom ili _.

Svaka greška uzrokuje ispis poruke na ekran i prekid programa. Svaka naredba programskog jezika se završava sa ;.
Test primeri sa izlazom u produžetku:

a)
```
int _a = 0xFF;
print(_a, 8);                          977
print(_a);                             255
print(_a, 16);                         FF
print(_a, 19);                         Nepodrzana osnova!
print(_nova1);                         _nova1 nije definisana!
```
b)
```
int _a = 2;
int _b = _a + 1;
_b = (_a + 1)<<(_a*2);
print(_b + 2,16);                      32
int _c ;
_c = (_a + _b)/012 + 013;
print(_c,16);                          10
_a = 7;
_b = ~(_a-1);
_b & _a == 0;                          False!
_a == _a | 0x0 & 0xFFFFFFFF;           True!
int _nova2 = _c >> (_a -1 -~_b);       Drugi operand nije veci od nule!
```
c)
```
list lx = [-1, 02, 0x3];
print( lx);                            [ -1, 2, 3 ]
print(lx[0] << lx[1]);                 -4
```

■

Zadatak 9.16 Koristeći tehniku rekurzivnog spusta implementirati sintaksni analizator koji prepoznaje izraze nad označenim brojevima korišćenjem samo bitskih operatora (&,

| ,~), naredbu dodelu vrednosti promenljivoj, kao i ispis promenljive koristeći funkciju `print` bez navođenja osnove. Nije potrebno vršiti izračunavanja tokom parsiranja ulaza već samo proveriti sintaksu. ■

Rešenje. Rešenje zadatka 9.15

Specifikacija leksičkog analizatora

```
%option noyywrap
%option noinput
%option nounput

%{
#include <iostream>
#include <cstdlib>
#include <string>
#include <vector>
#include "parser.tab.hpp"
%}

%%
print   { return PRINT; }

int     { return INT; }

list    { return LIST; }

"<<"    { return SHL; }

">>"    { return SHR; }

"=="    { return EQV; }

[_][a-zA-Z][a-zA-Z0-9]* {
          yylval.ime = new std::string(yytext);
          return ID;
        }
(0|[1-9][0-9]*) {
          yylval.broj = atoi(yytext);
          return BROJ;
        }
0x(0|[1-9A-F][0-9A-F]*) {
          sscanf( yytext, "%x",&yylval.broj);
          return BROJ;
        }

0(0|[1-7][0-7]*) {
          sscanf( yytext, "%o",&yylval.broj);
          return BROJ;
        }
l[a-zA-Z0-9_]+ {
```

```
          yylval.ime = new std::string(yytext);
          return LID;
        }
[=+\-*/&|^,();[\]~]     { return *yytext; }

[ \t\n]     { }

. { std::cerr << "Leksicka greska: " << *yytext << std::endl;
    exit(EXIT_FAILURE); }

%%
```

Rešenje 9.8: lexer.l

Specifikacija sintaksnog analizatora

```
%{

#include <iostream>
#include <string>
#include <map>
#include <vector>

#define YYDEBUG 1

std::map<std::string, int> promenljive;
std::map<std::string, std::vector<int>* > liste;

extern int yylex();

void yyerror(const std::string& poruka){
  std::cerr << poruka << std::endl;
  exit(EXIT_FAILURE);
}

void stampaj(int broj, int osnova){
  if (osnova ==  10 )
    printf("%d",broj);
  else if(osnova == 16)
    printf("%x",broj);
  else if(osnova == 8)
    printf("%o",broj);
  else {
   yyerror("Nepodrzana osnova!");
  }
  std::cout  << std::endl;
}

void stampaj_listu(std::vector<int> * l){
  std::cout << "[ ";
```

```
36   for(unsigned i = 0; i<l->size(); i++ ){
       std::cout << (*l)[i];
38     if( i!= l->size()-1)
         std::cout << ", ";
40   }

42   std::cout << "]" << std::endl;
}
44
%}
46
%union {
48   int broj;
   std::string* ime;
50   std::vector<int> * elementi;
}
52
%left '|'
54 %left '^'
%left '&'
56 %left SHR SHL
%left '+' '-'
58 %left '*' '/'
%right '~' UMINUS
60
%token PRINT INT LIST EQV
62 %token<ime> ID
%token<broj> BROJ
64 %token<ime> LID

66 %type<broj> izraz
%type<elementi> lista_vrednosti lista_izraz
68
%start program
70
%%
72
program:  program naredba ';'{}
74   | naredba ';' {}
   ;
76
naredba:  INT ID  {
78         if (promenljive.find(*$2) != promenljive.end()){
             yyerror("Vec deklarisana promenljiva!");
80         }

82         promenljive[*$2] = 0;
           delete $2;
```

```
84        }
    | INT ID '=' izraz {
86              if (promenljive.find(*$2) != promenljive.end()){
                  yyerror("Vec deklarisana promenljiva!");
88              }

90              promenljive[*$2] = $4;
                delete $2;
92             }
    | ID '=' izraz  {
94              if (promenljive.find(*$1) == promenljive.end()){
                  yyerror("Nije deklarisana promenljiva!");
96              }
                promenljive[*$1] = $3;
98              delete $1;
             }
100 | PRINT '(' izraz ')'  {
                stampaj($3, 10);
102             }
    | PRINT '(' izraz ',' BROJ ')'  {
104             if ($5 != 8 && $5 != 10 && $5 != 16) {
                   yyerror("Nepodrzana osnova!");
106               }

108             stampaj($3, $5);
             }
110 | PRINT '(' lista_izraz ')' {
                stampaj_listu($3);
112          }
    | izraz EQV izraz  {
114             if ($1 == $3)
                  std::cout << "True!" << std::endl;
116             else std::cout << "False!" << std::endl;
             }
118
    | LIST LID '=' lista_izraz {
120             if (liste.find(*$2) != liste.end()){
                  yyerror("Vec deklarisana lista!");
122             }

124             liste[*$2] = $4;
                delete $2;
126             }
    ;
128
izraz:  izraz '+' izraz { $$ = $1 + $3; }
130       | izraz '-' izraz { $$ = $1 - $3; }
          | izraz '*' izraz { $$ = $1 * $3; }
132       | izraz '/' izraz { $$ = $1 / $3; }
```

```
      | izraz '&' izraz { $$ = $1 & $3; }
134   | izraz '|' izraz { $$ = $1 | $3; }
      | izraz '^' izraz { $$ = $1 ^ $3; }
136   | izraz SHR izraz {
                        if ($3 <= 0){
138                       yyerror("Drugi operand mora biti veci
   od 0 !");
                        }
140                     $$ = $1 >> $3;
                      }
142   | izraz SHL izraz {
                        if ($3 <= 0){
144                       yyerror("Drugi operand!");
                        }
146                     $$ = $1 << $3;
                      }
148   | '~' izraz       { $$ = ~$2; }
      | '-' izraz   %prec UMINUS {$$ = -$2;}
150   | '(' izraz ')'   { $$ = $2; }
      | BROJ            {$$ = $1;}
152   | ID         {
            if (promenljive.find(*$1) == promenljive.end()){
154           yyerror(*$1 + " nije deklarisana!");
            }
156
            $$ = promenljive[*$1];
158         delete $1;
          }
160     | LID '[' BROJ ']' {
            if (liste.find(*$1) == liste.end() ){
162           yyerror(*$1 + " nije deklarisana lista!");
            }
164         $$ = (*liste[*$1])[$3];
            delete $1;  }
166   ;

168 lista_izraz:  '[' lista_vrednosti ']'  {
              $$= $2;
170         }
        | LID {
172           if (liste.find(*$1) == liste.end() ){
              yyerror(*$1 + " nije deklarisana lista!");
174           }
              $$ = new std::vector<int>(*liste[*$1]);
176           delete $1;  }
      ;
178
  lista_vrednosti : lista_vrednosti ',' izraz   { $$ = $1;
180
```