

PREVOĐENJE PROGRAMSKIH JEZIKA

AJZENHAMER NIKOLA
BUKUROV ANJA



Copyright ©2022 Ajzenhamer Nikola, Bukurov Anja

IZDATO OD STRANE „AJZENHAMER NIKOLA, BUKUROV ANJA”

[HTTPS://THEIKEOFFICIAL.GITLAB.IO/PPJ/](https://theikeofficial.gitlab.io/PPJ/)

Ovo delo je zaštićeno licencom Creative Commons Attribution-NonCommercial 3.0 Unported License („Licenca“). Ovo delo se ne može koristiti osim ako nije u skladu sa Licencom. Detalji Licence mogu se videti na veb adresi <http://creativecommons.org/licenses/by-nd/3.0>. Dozvoljeno je umnožavanje, distribucija i javno saopštavanje dela, pod uslovom da se navedu imena autora. Upotreba dela u komercijalne svrhe nije dozvoljena. Prerada, preoblikovanje i upotreba dela u sklopu nekog drugog nije dozvoljena.

Drugo izdanje, Maj 2022.

Poslednja izmena: 2022-05-15 12:15

Sadržaj

Predgovor	7
1 O prevođenju programskih jezika	9
1.1 Proces kompiliranja	9
1.2 Vrste programskih prevodilaca. Etape kompiliranja.	11
1.3 Etapa analize programskih prevodilaca	13
1.3.1 Osnovno o leksičkoj analizi	14
1.3.2 Osnovno o sintakšičkoj analizi	15
1.3.3 Osnovno o semantičkoj analizi	19
1.4 Etapa sinteze programskih prevodilaca	21
1.5 Pitanja i zadaci	24
2 Regularni jezici	27
2.1 Azbuka i jezici	27
2.2 Operacije nad rečima i jezicima	29
2.2.1 Prioritet operacija nad jezicima	32
2.3 Regularni jezici	33
2.3.1 Proširenja regularnih izraza i jezika	34
2.3.2 Pitanja i zadaci	36
3 Konačni automati	37
3.1 Konačni automati	37
3.1.1 Definicija (N)KA	40
3.1.2 Uslovi za deterministički konačni automat (DKA)	42
3.1.3 Jezik automata	42

3.2 Klinijeva teorema	43
3.2.1 Tompsonov algoritam	44
3.2.2 Algoritam oslobađanja od ε -prelaza	47
3.2.3 Gluškovljev algoritam	50
3.2.4 Algoritam konstrukcije podskupova	52
3.2.5 Murov algoritam	57
3.2.6 Metod eliminacije stanja	64
3.3 Skupovne operacije i konačni automati	67
3.3.1 Lema o razrastanju	69
3.3.2 Pitanja i zadaci	71
4 Kontekstno-slobodne gramatike	73
4.1 Definicija KS gramatike	74
4.2 Stablo izvođenja	76
4.3 Višezačne gramatike	80
4.4 O asocijativnosti, prioritetima i razrešavanju višezačnosti	81
4.5 Transformacije KS gramatika	86
4.5.1 Eliminacija neproduktivnih simbola	87
4.5.2 Eliminacija ε -pravila	88
4.5.3 Eliminacija jednostruktih pravila	90
4.5.4 Eliminacija leve rekurzije	91
4.5.5 Leva faktorizacija	95
5 Potisni automati	97
5.1 Osnovni pojmovi	97
5.2 Primeri potisnih automata	100
5.3 Odnos PA sa KS jezicima	101
5.3.1 Lema o razrastanju za KS jezike	102
5.4 Deterministički KS jezici	102
5.5 Podfamilije determinističkih KS jezika	103
5.5.1 LL jezici	103
5.5.2 LR jezici	104
5.6 Pitanja i zadaci	105
6 Sintaksička analiza	107
6.1 Sintaksna analiza naniže	109
6.1.1 Marker kraja ulaza	109
6.1.2 Rekurzivni spust	109
6.1.3 LL(1) gramatike	110
6.1.4 Tablice LL-analize	113
6.1.5 Nerekurzivna LL-analiza	114
6.2 Sintaksna analiza naviše	115
6.2.1 Ručka	116
6.2.2 Analiza naviše prebacivanjem i svođenjem	118
6.2.3 Konflikti tokom analize naviše prebacivanjem i svođenjem	119

6.3	Osnovni <i>LR</i>-analizatori	119
6.3.1	Automat <i>LR(0)</i> -analize	120
6.3.2	Opšti algoritam <i>LR</i> -analize	124
6.3.3	Tablice <i>SLR(1)</i> -analize	127
6.4	Pitanja i zadaci	130
7	Semantička analiza	135
7.1	Atributske gramatike	135

Dodatak

Literatura	143
Knjige	143

Predgovor

Ovaj tekst predstavlja skriptu za obavezni kurs „Prevođenje programskih jezika”, na 3. godini smera Informatika na Matematičkom fakultetu Univerziteta u Beogradu. Skripta je koncipirana na osnovu beleški sa časova predavanja održanih od strane Filipa Marića u akademskoj 2016/2017. godini. Skripta je prateći materijal pre svega studentima koji ovaj kurs slušaju u okviru svojih studija, ali i svima Vama koji biste želeli da se upoznate sa ovom tematikom. Ovaj materijal ne može zameniti pohađanje časova predavanja i vežbi na navedenom kursu, niti drugu preporučenu literaturu.

Kao i svaki nerecenzirani materijal ovog tipa, i ovaj tekst je podložan propustima u njegovom pripremanju. Ukoliko ste pažljivi čitalac ove skripte, i ukoliko uočite bilo kakvu grešku, možete se javiti autorima putem elektronske pošte na adresu nikola_ajzenhamer@math.rs. Potrebno je da u naslovu elektronske poruke stavite tekst „[PPJ] Skripta”. Svi komentari, sugestije, kritike, ali i pohvale vezane za ovaj materijal su dobrodošli.

Autori

1. O prevodenju programskih jezika

Uvodno poglavlje ove skripte posvećeno je upoznavanjem čitaoca sa osnovnim pojmovima i procesima pri prevodenju programskih jezika. Čitalac ima priliku da se upozna sa ovim procesom iz najšireg posmatranog ugla, kako bi se u daljim poglavljima ta znanja produživila.

Na početku poglavlja prikazujemo proces kompiliranja kroz dve etape i njihovih faza, a zatim se nešto detaljnije upoznajemo sa elementima svake od faza. Ipak, akcenat ove skripte je stavljen na prvu etapu, te se zbog toga njoj pridodaje više značaja.

1.1 Proces kompiliranja

Da bismo uspešno rešili odgovarajući problem u poslovnom domenu, potrebno je da specifikujemo problem koristeći odgovarajući jezik. Za komunikaciju sa ljudima koristimo *prirodni jezik* (ili samo *jezik*). Međutim, računari (još uvek) nisu sposobni da razumeju jezik na način na koji ga ljudi razumeju. Zbog toga, potrebno je odabrati jezik koji oni mogu da razumeju, a zatim je neophodno zapisati naš problem u terminima odabranog jezika. Računari su sposobni da razumeju *mašinski jezik*, međutim, za ljude je često teško definisati problem iz poslovnog domena u terminima mašinskog jezika. Umesto toga, osmišljeni su specijalni jezici koji se mogu koristiti u komunikaciji sa računarima.

Definicija 1.1.1 — Viši programski jezik (programski jezik). *Viši programski jezik* (nadalje samo *programski jezik*) je veštački jezik za opis konstrukcija koje mogu biti prevedene u mašinski jezik i izvršene od strane računara.

Kao što moramo da naučimo pravila, na primer, engleskog jezika da bismo uspešno komunicirali sa ljudima pomoću njega, tako isto moramo da naučimo pravila za komunikaciju korišćenjem programskih jezika. Svaki jezik definiše svoja pravila, te se prvo upuštamo u odabir programskog jezika i učenje njegovih pravila.

Nakon odabira programskog jezika sledi zapisivanje problema iz poslovnog domena u terminima odabranog programskog jezika. Programske instrukcije koje nastaju na ovaj način se najčešće čuvaju u datotekama na sistemu datoteka i čine izvorni kod.

Definicija 1.1.2 — Izvorni kod. *Izvorni kod* čini niz naredbi zadatog programskog jezika.

Međutim, napisani izvorni kod se ne može izvršiti na računaru jer, kao što smo rekli, računari razumeju samo instrukcije mašinskog jezika. Zbog toga se izvorni kod podvrgava procesu prevođenja, čime se dobijaju datoteke koje sadrže mašinske instrukcije.

Definicija 1.1.3 — Izvršni kod. *Izvršni kod* čini niz instrukcija mašinskog jezika datog procesora.

Definicija 1.1.4 — Program (aplikacija). *Program* ili *aplikacija* predstavlja datoteku na sistemu datoteka koja sadrži izvršni kod.

Definicija 1.1.5 — Programska prevodilac, kompiliranje. *Programski prevodilac* ili *jezički procesor* je program koji čita izvorni kod, obrađuje ga i prevodi u izvršni kod, čime se omogućava izvršavanje programa. Proces prevođenja izvornog koda u izvršni kod poznat je pod nazivom *kompiliranje*¹.

Nakon uspešnog kompiliranja, korisnik može odgovarajućom komandom da instruiše operativnom sistemu da pokrene prevedeni program, čime se dobija rešenje polaznog problema.

Kao što vidimo, programski jezici i programski prevodioci predstavljaju veoma moćne alate kojima možemo rešavati najrazličitije probleme iz poslovnog domena. Međutim, nisu svi programski jezici jednako moćni i upoređivanje programskih jezika radi odabira konkretnog u kojem će polazni problem biti zapisan može predstavljati mukotrpan posao.

Sa druge strane, u nekim poslovnim primenama postoji primena za definisanjem novog jezika kojim se rešavaju postavljeni problemi. Razlozi za ovo mogu biti razni, a najveći broj njih dolazi iz specifičnosti samog domena u kome se problem razmatra. Zbog toga, razumevanje procesa kompiliranja donosi sa sobom potrebna znanja za konstrukciju i implementaciju ovakvih jezika.

Definicija 1.1.6 — Jezici poslovnih domena. Programski jezici koji su konstruisani sa ciljem rešavanja problema u nekom konkretnom domenu se nazivaju *jezici poslovnih domena* (engl. *domain-specific language*).

Neki primeri jezika poslovnih domena su: Mathematica (jezik za definisanje i rešavanje matematičkih problema), UML (jezik za definisanje i rešavanje problema u razvoju softvera), makefile (jezik za definisanje i rešavanje problema kompilacije drugih programskih jezika) i dr.

Uvođenjem jezika poslovnih domena u diskusiju, pojam programskih prevodilaca ne mora nužno da se odnosi na kompiliranje programskih jezika radi dobijanja programa koji se izvršava, već može uključivati bilo kakav procesor tekstualnog sadržaja. Zbog toga je teorija koja stoji u pozadini procesa kompiliranja veoma značajna, čak i izvan konteksta viših programskih jezika. Naravno, mi ćemo više govoriti o principima prevođenja programskih jezika, ali svi koncepti o kojima budemo diskutovali se mogu primeniti i na jezike poslovnih

¹Termini engleskog jezika *compilation* i *compiler* potiču iz latinskog jezika, te zbog toga u ovoj skripti koristimo termine kompiliranje (eventualno, kompilacija) i kompilator. Nasuprot ovome, u svakodnevnoj upotrebi su termini kompajliranje i kompajler uveliko postali dominantniji.

domena, uz malo ili čak nimalo izmena.

1.2 Vrste programskih prevodilaca. Etape kompiliranja.

Kao što smo već rekli, da bismo mogli da rešimo problem iz poslovnog domena koji je zapisan u terminima nekog programskog jezika, potrebno je prvo da izvorni kod prevedemo u izvršni kod, a zatim je potrebno da učitamo izvršni kod čime se kreirani program izvršava i izračunava rešenje datog problema. U načinu na koji programski prevodioci izvršavaju ove dve faze, svi programski prevodioci mogu se podeliti u dve vrste:

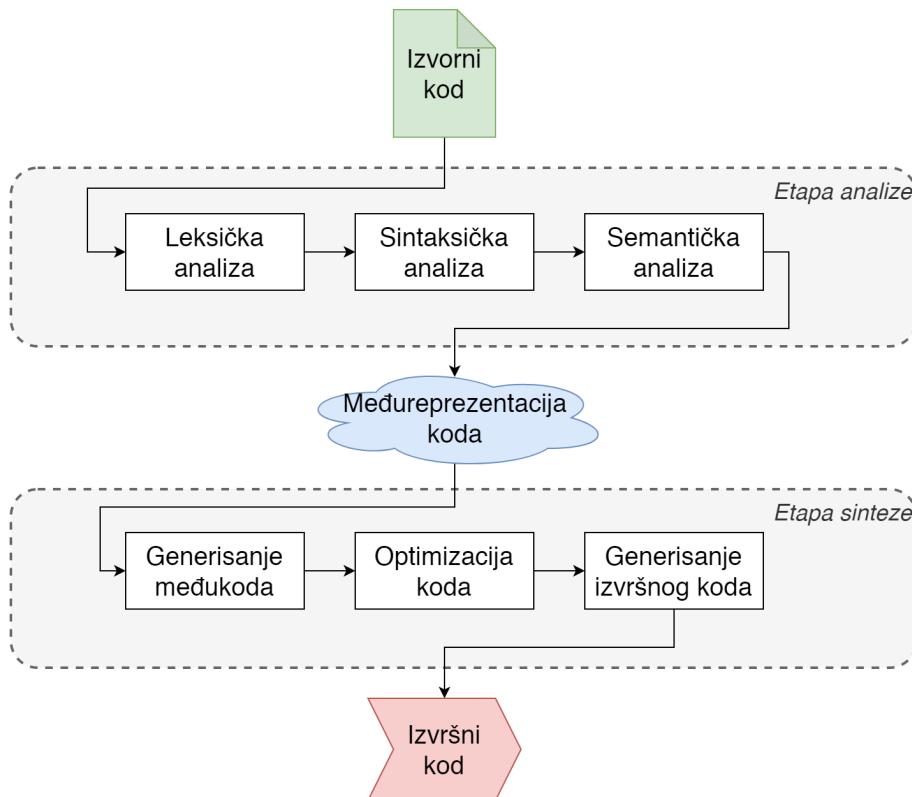
1. *Kompilatori* su programski prevodioci kod kojih je jasna razlika između faze prevođenja i faze izvršavanja. Izvorni kod se prvo prevede u izvršni kod, a zatim se dobijeni kod izvršava učitavanjem u memoriju i izvršavanjem instrukcija. Jedna od prednosti ovog pristupa jeste u optimizaciji koda pre faze izvršavanja, čime se dobija na brzini u fazi izvršavanja. Kompilatori se najčešće koriste u programskim jezicima kao što su C, Java², C++ i dr.
2. *Interpretatori* su programski prevodioci kod kojih su faza prevođenja i faza izvršavanja isprepletane, i vrlo često ne postoji fizička materijalizacija prevođenja u vidu izvršnog koda zapisanog u datoteku na fajl sistemu koji se učitava u memoriju. Umesto toga, interpretatori čitaju ceo izvorni kod instrukciju-po-instrukciju, kreiraju neku vrstu međukoda i izvršavaju instrukcije međukoda. Jasno je da je ovakav način izvršavanja programa sporiji, ali doprinosi boljom dijagnostici problema, upravo zato što izvršavaju program naredbu-po-naredbu. Interpretatori se najčešće koriste u programskim jezicima kao što su: JavaScript, Haskell, Python i dr.

Sada kada smo razumeli postojanje različitih vrsta programskih prevodilaca, upoznajmo se sa procesom kompiliranja. Kao što smo nagovestili, kompiliranje predstavlja kompleksan proces koji se sastoji od nekoliko faza. Različite literature definišu ove faze različitim granularnošću, najčešće uzimajući u obzir opširnost materijala koji se izvodi kao i koji elementi ovog složenog procesa imaju najveći fokus. Ovakav pristup će biti usvojen i u ovoj skripti, te ćemo dati najopštiji pogled na ovaj proces, koji je prikazan na slici 1.1. U opštem slučaju, faze kompiliranja mogu se podeliti u dve etape:

1. Etapa *analyze*, koja ima za cilj da od izvornog koda kreira neku vrstu međureprezentacije koda. *Međureprezentacija koda* (engl. *intermediate representation*) predstavlja veliki broj struktura podataka koji predstavljaju apstrakciju izvornog koda i služe za izvor informacija o samom kodu. Da bi se ova reprezentacija uspešno kreirala, izvorni kod se prvobitno analizira i ispituje se da li je napisan u skladu sa programskim jezikom. Ono što karakteriše etapu analize jeste što je ista i za kompilatore i za interpretatore. Elementi ove etape su veoma dobro opisani u teoriji. Ovu fazu izvršavaju elementi prevodioca koji čine tzv. *prednji deo prevodioca* (engl. *front end*). Deli se na tri faze:
 - (a) Leksička analiza.
 - (b) Sintaksička analiza.
 - (c) Semantička analiza.

²Preciznije, Java programski prevodilac kombinuje kompiliranje i interpretiranje. Java izvorni kod se prvo kompilira, ali ne u izvršni, već u međukod koji se naziva bajtkod. Zatim se taj bajtkod interpretira od strane Java virtualne mašine. Prednost ovoga se ogleda u tome da bajtkod kompiliran na jednoj mašini može biti interpretiran na nekoj drugoj mašini (naravno, druga mašina mora da podržava izvršavanje Java izvršnih programa).

2. Etapa *sinteze*, koja ima za cilj da od međureprezentacije koda kreira program. Sinteza koda predstavlja proces koji uzima u obzir izlazni jezik, koji je najčešće mašinski jezik ili asemblerSKI jezik, arhitekturu računara na kojoj se vrši prevodenje, okruženje pod kojim se izvršava i dr. Ovu fazu izvršavaju elementi prevodioca koji čine tzv. *zadnji deo prevodioca* (engl. *back end*). Deli se na tri faze:
- Generisanje međukoda.
 - Optimizacija međukoda.
 - Generisanje izvršnog koda.



Slika 1.1: Grafički prikaz procesa kompiliranja.

Iako su sve faze kompiliranja podeljene na navedene dve etape, one ipak nisu u potpunosti izolovane. U ovom trenutku samo napominjemo strukturu podataka koja se naziva *tablica simbola* (engl. *symbol table*) koja započinje svoju izgradnju već u fazi leksičke analize, a koja ima svoju ulogu u kasnijim fazama, čak i u fazi generisanja izvršnog koda.

Takođe, napomenimo da je moguće koristiti isti prednji deo prevodioca sa različitim zadnjim delovima. Na primer, analiziranje C koda se vrši na isti način bez obzira na kojoj se platformi kod kompilira, ali se koriste različiti zadnji delovi za različite operativne sisteme. Slično tome, moguće je korišćenje različitih prednjih delova sa istim zadnjim delom. Na primer, razvojno okruženje .NET dozvoljava pisanje koda u C#, F# ili Visual Basic programskim jezicima, i za njih ima različite prednje delove, ali svi oni se apstrahuju na zajednički međukod koji se dalje sintetiše na isti način korišćenjem jedinstvenog zadnjeg dela.

Diskutovali smo o procesu prevodenja programa napisanih u nekom programskom jeziku. Za ovaj proces nam je bio potreban softver koji smo nazvali programski prevodilac. Me-

đutim, validno je postaviti pitanje u kom jeziku programiramo kompilator i da li nam je ta odluka važna ili ne. Zanimljivo je razumeti da se kompilatori vrlo često pišu u onim jezicima koje oni prevode. Na primer, kompilator Clang za programski jezik C++ ima prednji deo koji je napisan u programskom jeziku C++, dok za zadnji deo koristi LLVM kompilator koji je takođe napisan u programskom jeziku C++. Ovo je moguće zbog toga što se za pisanje kompilatora koristi tehnika *podizanja* (engl. *bootstrapping*). U pitanju je tehnika koja koristi neki drugi programski jezik (ali može se koristiti i asembler) za pisanje centralnog dela kompilatora, a zatim se taj centralni deo koristi za pisanje ostalih delova kompilatora.

Međutim, ono što treba razumeti jeste da efikasnost kompilatora da kreira program koji će se brže izvršavati ne zavisi nužno od vremena koje utroši u fazi prevođenja. Drugim rečima, ne mora da znači da će jedan prevodilac kompilirati brži program ukoliko je utrošio manje vremena za proces kompiliranja od nekog drugog prevodioca. Češća je situacija da, što je prevodilac bolje konstruisan, to on mora više posla da uradi (tj. više vremena da potroši) prilikom faze kompiliranja, ali su zato programi koje on proizvodi brži. Dakle, to ne znači da je neophodno da kompilatore pišemo u brzim programskim jezicima, već da ih pažljivo konstruišemo. Naravno, savremeni kompilatori se često ipak pišu u brzim programskim jezicima, kao što su C i C++ kako bi se, pored ubrzanja izvršavanja programa dobilo i na ubrzanju u fazi prevođenja, odnosno, ubrzanju razvoja softvera.

Kao što smo rekli, u ovoj skripti fokus je stavljen na etapu analize. Etapa sinteze predstavlja proces kojim se u teoriji bavi „konstrukcija kompilatora”. Zbog toga ćemo u nastavku teksta pažnju više usmeriti na etapu analize.

1.3 Etapa analize programskih prevodilaca

Pre nego što prevodilac može da generiše izvršni kod, prednji deo se mora postarati o tome da li je izvorni kod validan. Validnost izvornog koda se ogleda u tome da li prati pravila koja su postavljena višim programskim jezikom. Ova pravila se sastoje od provera da li sve „reči” izvornog koda čine „reči” programskega jezika, da li su te „reči” struktuirane korišćenjem pravila za formiranje „rečenica” u programskom jeziku, kao i da li formirane „rečenice” imaju validno značenje. Ukoliko izvorni kod zadovoljava sve provere, onda se na osnovu njega formira međureprezentacija, koja se dalje predaje zadnjem delu prevodioca.

Da bismo ovu diskusiju približili čitaocu, uvodimo naredni primer koda koji je napisan u programskom jeziku C. Kroz ovaj primer ćemo objasniti načine na koje svaka od faza u prednjem delu prevodioca obraduje dati kod, kao i predstaviti ulaze i izlaze za svaku od faza. Pre nego što prikažemo primer, uvedimo jednu definiciju.

Definicija 1.3.1 — Fragment koda. Uređeni pravi podskup multiskupa instrukcija izvornog koda napisanog u nekom programskom jeziku koji (najčešće) ne može samostalno da se prevede nazivamo *fragment koda* (engl. *code fragment*).

Primer 1.1 U narednom tekstu koristimo naredni fragment koda napisan u programskom jeziku C.

$x = 2*y1 + 3;$

1.3.1 Osnovno o leksičkoj analizi

Prilikom analiziranja „reči” izvornog koda, prevodilac koristi definiciju programskog jezika i upoređuje da li „reči” na koje nailazi predstavljaju validne „reči” u tom programskom jeziku. U poglavlju 2 ćemo se upoznati sa formalnim matematičkim alatima kojima se ovaj proces izvodi. Ovaj proces se naziva *leksička analiza*.

Definicija 1.3.2 — Leksički analizator. Deo prevodioca koji obavlja zadatku leksičke analize naziva se *leksički analizator* ili *lekser* (engl. *lexer* ili *scanner*).

Leksikologija prirodnih jezika je deo nauke o jeziku koji proučava reči u njihovom svojstvu osnovnih jedinica imenovanja. Slično tome, leksički analizator čita karakter po karakter iz izvornog koda i identificiše celine koje nazivamo lekseme.

Definicija 1.3.3 — Leksema. *Leksema* je najmanja samostalna jedinica leksičkog sistema.

Primer 1.2 Lekseme koje možemo izdvojiti iz primera 1.1 su sledeće:

x	=	2	*	y1	+	3	;
---	---	---	---	----	---	---	---

Nakon identifikovanja leksema, svakoj od njih se pridružuje odgovarajuća leksička kategorija, kao i jedinstvena odgovarajuća oznaka kategorije koja je pridružena leksemi. Ta oznaka naziva se token.

Definicija 1.3.4 — Token. *Token* (engl. *token*) predstavlja gramatičko obeležje ili kategoriju u programskom jeziku.

Svaka leksema mora da pripada nekom tokenu da bi bila prepoznata od strane leksera. Primeri tokena u programskom jeziku C su: identifikator (imena promenljivih u izvornom kodu), operator (ugrađene operacije koje izračunavaju neke izraze), separator (specijalne oznake za kraj izraza ili naredbe), razne doslovne vrednosti (brojevi, doslovne konstantne niske, itd.) i dr.

Ne predstavljaju svi karakteri lekseme jezika. Na primer, lekseri većina programskih jezika ignoriraju karaktere belina kao što su razmaci, tabulatori i karakteri za predstavljanje prelaska u novi red, kao što je slučaj u programskim jezicima C, Java i C++. Međutim, u nekim drugim programskim jezicima razmaci igraju važnu ulogu, kao što je slučaj u programskim jezicima Python i Go.

Primer 1.3 Nakon identifikovanja leksema u primeru 1.2, lekser dodeljuje naredne tokene svakoj od prepoznatih leksema:

- Lekseme x i y1 su identifikatori.
- Lekseme =, * i + su operatori.
- Lekseme 2 i 3 su doslovne brojevne vrednosti.
- Leksema ; je separator.

Nakon procesiranja izvornog koda od strane leksičkog analizatora, izlaz iz ove faze predstavlja sekvenca tokena. Ova sekvenca zatim predstavlja ulaz u narednu fazu prevodenja. Napomenimo da iako leksički analizator operiše sa izvornim kodom direktno, on ne vodi računa o tome da li je, na primer, neki identifikator promenljiva, funkcija, struktura i sl.

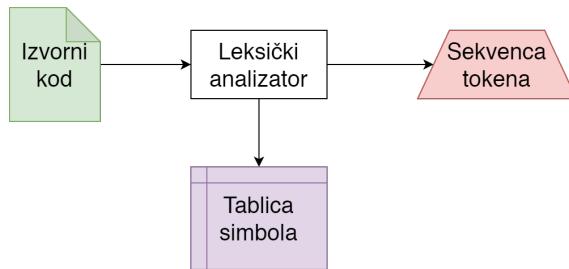
S obzirom da je ovo jedina faza u kojoj se operiše direktno nad izvornim kodom, zadatku

leksera jeste i da čuva informaciju o, na primer, nazivu identifikatora koji je prepoznao, vrednosti doslovne brojevne vrednosti koju je prepoznao i sl. Ove vrednosti se mogu čuvati kao, na primer, niske koje će u kasnijim fazama biti pretvorene u odgovarajuće vrednosti. Dodatno, struktura tokena se može obogatiti vrednostima koje pokazuju na početak i kraj reda i kolone u izvornom kodu u kojima počinje, odnosno, u kojima se završava prepoznata leksema. Ove informacije mogu biti od značaja za prikazivanje čitljive poruke u slučaju prepoznavanja greške prilikom bilo koje od faza analize.

Osim navedenih operacija, leksički analizator održava i generiše veoma važnu strukturu podataka koja se naziva tablica simbola. Ova tabela, koja se inicijalizuje tokom leksičke analize, dopunjava se i koristi i u ostalim fazama.

Definicija 1.3.5 — Tablica simbola. *Tablica simbola* (engl. *symbol table*) je struktura podataka u kojoj se, tokom etape analize, prikupljaju informacije o tipu, opsegu i memorijskoj lokaciji identifikatora.

Iako jednostavna, faza leksičke analize je veoma spora. Ovo potiče otuda što kompilator jedino u ovoj fazi neposredno radi nad karakterskim niskama izvornog programa (dok se ostale faze odvijaju nad tokenima). Prikaz procesa leksičke analize dat je na slici 1.2.



Slika 1.2: Prikaz procesa koji izvršava leksički analizator, sa ulazima i izlazima.

1.3.2 Osnovno o sintaksičkoj analizi

Slično kao što sintaksa prirodnog jezika proučava pravila koja određuju kako se reči kombinuju u rečenice u datom jeziku, sintaksa programskih jezika predstavlja niz pravila koja definišu kombinaciju tokena za koje se smatra da daju ispravno strukturiran fragment koda u traženom programskom jeziku.

Dakle, u sintaksičkoj analizi se proverava da li su tokenizovane lekseme koje se dobijaju iz faze leksičke analize sklopljene prateći prethodno definisanu gramatiku programskog jezika. U ovoj fazi, lekseme se postupno grupišu u gramatičke jedinice ili kategorije.

Definicija 1.3.6 — Sintaksički analizator. Deo kompilatora koji obavlja zadatku sintaksičke analize naziva se *sintaksički analizator* ili *parser* (engl. *parser*).

Prilikom grupisanja leksema u gramatičke jedinice, sintaksički analizator započinje izgradnju strukture koja će do kraja etape analize prerasti u jednu od struktura podataka u međureprezentaciji koda. Ova struktura podataka se naziva sintaksičko stablo.

Definicija 1.3.7 — Sintaksičko stablo. *Sintaksičko stablo* (engl. *parse tree*) predstavlja *m*-narno stablo u čijim čvorovima se nalaze tokeni. Stablo je struktuirano tako da koren svakog podstabla predstavlja osnovni element kojim se identificuje jednu gramatičku

jedinicu, a grane od korena svakog podstabla do njegovih naslednika oslikavaju preostale elemente te gramatičke jedinice.

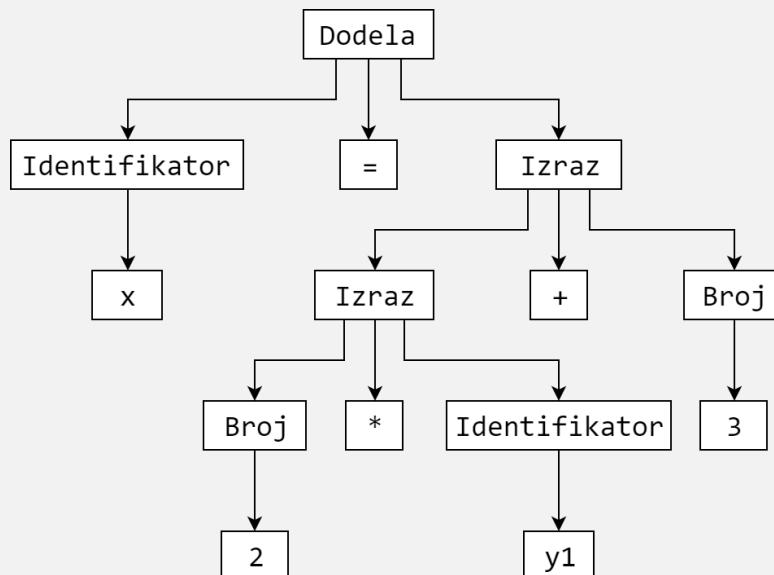
Primer 1.4 Izlaz iz leksičkog analizatora za fragment koda iz primera 1.1 predstavlja sekvenca tokena:

```
Identifikator(x) Operator(=) Broj(2) Operator(*) Identifikator(y1)
Operator(+) Broj(3) Separator(;)
```

U skladu sa definicijom programskog jezika C, ovo predstavlja validnu gramatičku jedinicu. Prikažimo kako bi sintaksički analizator programskog jezika C mogao da konstruiše sintaksičko stablo.

Sintaksički analizator programskog jezika C prepoznaje da ova sekvenca tokena predstavlja naredbu dodele vrednosti promenljivoj, te u koren stabla postavlja rečenični konstrukt koji označava naredbu dodele — **Dodela**. Podstabla korena predstavljaju podelemente ovog rečeničnog konstrukta: (1) izraz sa leve strane kome se dodeljuje vrednost, (2) oznaka operatora dodele i (3) vrednost koja se dodeljuje. Na osnovu tokena, sintaksički analizator za svaki ovaj element prepoznaje: (1) identifikator, (2) karakter = i (3) izraz, i upravo ovo predstavljuju nove rečenične konstrukte koji se dalje analiziraju.

Obratimo pažnju na desnu stranu naredbe dodele koja predstavlja složeni izraz. U skladu sa prioritetima operadora, sintaksički analizator prepoznaje kao operaciju sabiranja, te se u koren desnog podstabla postavlja **Izraz** koji predstavlja sabiranje dva operanda. Zašto je odabrana operacija sabiranja umesto množenja kada množenje ima veći prioritet?



Slika 1.3: Sintaksičko stablo koje formira sintaksički analizator prilikom obrade izraza iz primera 1.1.

Razmislimo kako bismo izvršili izračunavanje izraza u stablu. Mi u korenu nekog stabla vidimo koja se operacija koristi (na primer, sabiranje), ali da bismo izvršili tu operaciju, moramo da imamo izračunate vrednosti u podstablima. Ukoliko su operandi doslovne

vrednosti, ovo je moguće uraditi. Međutim, u opštem slučaju, operandi ne moraju biti doslovne vrednosti, već mogu biti memorijske lokacije ili rezultati izračunavanja funkcija. Zbog toga, prvo je potrebno izračunati vrednosti u podstablima da bismo mogli da izračunamo vrednost u korenu. Rekurzivnim primenom ovog pravila dolazimo do činjenice da se izračunavanje vrši od listova (u kojima su smeštene konkretnе vrednosti) ka korenu. Zbog toga, operacije koje imaju viši prioritet moraju se nalaziti na većoj dubini stabla.

Opisani postupak se dalje primenjuje dok se ne obradi svi tokeni čime se kreira sintakstičko stablo koje je dato na slici 1.3.

Primer 1.5 Neka je dat naredni fragment koda u programskom jeziku C:

```
2*y1 = x + 3;
```

Primer 1.6 Izlaz iz leksičkog analizatora za fragment koda iz primera 1.5 predstavlja sekvenca tokena:

```
Broj(2) Operator(*) Identifikator(y1) Operator(=) Identifikator(x)  
Operator(+) Broj(3) Separator(;)
```

Leksički analizator je prepoznao sve lekseme kao ispravne i proces leksičke analize je prošao bez grešaka.

Nakon što sintakski analizator dobije datu struju tokena kao ulaz, prilikom analiziranja naredbe dodele, sintakstički analizator prepoznaće da izraz sa leve strane izraza dodele sadrži binarni operator, što nije dozvoljeno gramatikom programskog jezika C. U tom slučaju, sintakski analizator bi prijavio grešku, čime bi se ceo proces kompilacije završio neuspešno, što je i očekivano.

Dajemo još jedan, ne značajno komplikovaniji primer fragmenta koda, ovog puta u programskom jeziku C++, pomoću koga su prikazani procesi leksičke i sintakski analize.

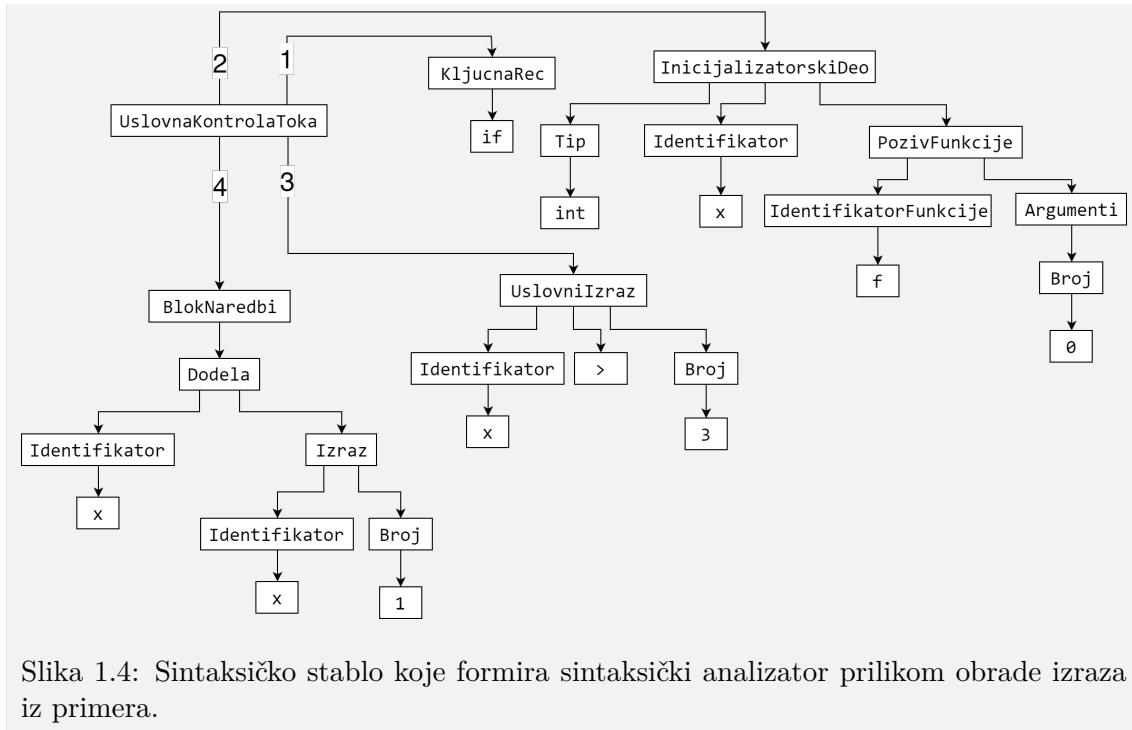
Primer 1.7 Neka je dat naredni fragment koda u programskom jeziku C++ (verzija C++17):

```
if(int x = f(0); x > 3) {  
    x = x + 1;  
}
```

Primer izlaza leksičkog analizatora bi mogla predstavljati naredna sekvenca tokena:

```
KljučnaRec(if) Zagrada() Tip(int) Identifikator(x) Operator(=)  
IdentifikatorFunkcije(f) Zagrada() Broj(0) Zagrada() Separator();  
Identifikator(x) Operator(>) Broj(3) Zagrada() Zagrada("{}")  
Identifikator(x) Operator(=) Identifikator(x) Operator(+)  
Broj(1) Separator(); Zagrada()}
```

Primer izlaza sintakstičkog analizatora bi moglo predstavljati sintakstičko stablo:

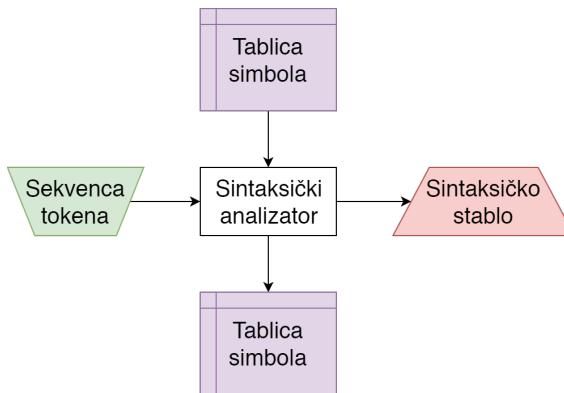


Slika 1.4: Sintakško stablo koje formira sintakški analizator prilikom obrade izraza iz primera.

Prepoznavanje sintakške strukture izvornog koda direktno utiče na kreiranje struktura podataka koje se koriste u kasnijim fazama prevođenja. Često se sintaksa programskog jezika (preciznije, neki njegovi delovi) mogu definisati na više načina, usled čega se može dobiti različito sintakško stablo. Zbog toga se često za proces prevođenja kaže da je *sintaksno-voden*. Konceptom sintaksno-vodenog prevođenja se omogućuje da je konstrukcija izvršnog jezika ekvivalentna izvornom jeziku. Ovo je važna napomena jer, u slučaju da nemamo ovo osiguranje, došlo bi do narušavanja prvog osnovnog principa prevođenja:

Prevodilac mora da očuva značenje programa koji se prevodi.

Kao što se lekser ne bavi sintaksom programskog jezika, tako se i parser ne bavi značenjem rečeničnih formi, odnosno, semantikom programskog jezika. Prikaz procesa sintakške analize dat je na slici 1.5.



Slika 1.5: Prikaz procesa koji izvršava sintakški analizator, sa ulazima i izlazima.

1.3.3 Osnovno o semantičkoj analizi

Provera sintaksičke konzistentnosti izvornog koda nam daje veliki broj koristi. Međutim, sve dok sintaksničkim konstruktima nisu pridružena značenja, te rečenice nemaju realnu važnost. Na primer, rečenica „Danas je bezbojno povrće“ predstavlja sintaksički korektan konstrukt, ali je bez realnog značenja. Nasuprot tome, rečenica „Prevodioci su korisni alati“, koja ima istu sintaksičku formu kao i prethodna rečenica, ovoga puta nosi korisno značenje. Zbog toga je važno da izvršimo proveru značenja koje dati rečenični konstrukti imaju. Za razliku od sintakse jezika koja izučava strukturu konstrukata čime se nešto izražava (tj. rečenica), značenje tih konstrukata izučava semantika jezika. Shodno tome, semantika programskih jezika izučava značenje programskih konstrukcija.

Definicija 1.3.8 — Semantički analizator. Deo kompilatora koji obavlja zadatku semantičke analize naziva se *semantički analizator* (engl. *semantic analyser*).

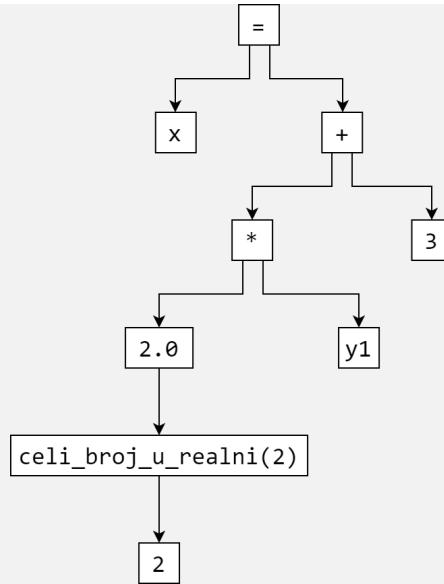
Semantički modeli koji su definisani nekim programskim jezikom predstavljaju značajno jednostavnije modele od onih u prirodnim jezicima. Ovi modeli, izraženi u terminima matematičkih alata, najčešće se ogledaju kroz nekoliko važnih provera, kao što su: provera tipova i deklaracija u sintaksičkom stablu dobijenog iz prethodne faze, provera prava pristupa u objektno-orientisanim jezicima, provera pravila opsega i vidljivosti identifikatora, provera broja i tipova argumenata pri pozivu funkcija, provera upotrebi naredbi `break` i `continue` na mestima gde one nisu dozvoljene i dr. Takođe, semantički modeli izvršavaju i implicitnu konverziju kod primene operatora, poziva funkcija i dr.

Ova faza ima poseban značaj i u slučaju interpretatora. Naime, čvorovima sintaksičkog stabla je moguće pridružiti značenje putem navođenja operacija koje je potrebno izvršiti u obilasku tog stabla. Na primer, prilikom nailaska na čvor u stablu koji sadrži operator `+`, moguće je dodeliti akciju „saberi vrednosti u levom i desnom podstablu i sačuvaj rezultat“, uz eventualne promene informacije u tablici simbola. Drugim rečima, interpretatore je moguće implementirati jednostavno definisanjem ovakvih operacija nad sintaksičkim stablom. Ove operacije se mogu definisati bilo u terminima nekog međukoda za već postojeću virtualnu mašinu koja taj međukod razume ili u terminima nekog programskog jezika, pri čemu će lekser, parser i semantički analizator biti prevedeni kao programi tog programskog jezika.

Sve opisane operacije se u fazi semantičke analize vrše nad strukturonim podatkovima koje se naziva apstraktno sintaksičko stablo.

Definicija 1.3.9 — Apstraktno sintaksičko stablo. Struktura podataka koja nastaje na osnovu sintaksičkog stabla eliminacijom detalja sintaksičke analize i korišćenjem informacija iz tablice simbola naziva se *apstraktno sintaksičko stablo* (engl. *abstract syntax tree*, skr. *AST*).

Primer 1.8 Posmatrajmo sintaksičko stablo sa slike 1.3. Ukoliko smo promenljivu čiji je identifikator `y1` deklarisali tipom `double`, onda, da bi se operandi operacije sabiranja mogli sabratи, prilikom provere tipova, semantički analizator mora da izvrši implicitnu konverziju doslovne celobrojne vrednosti `2` u doslovnu vrednost u pokretnom zarezu `2.0`. Apstraktno sintaksičko stablo koje se formira se može grafički prikazati kao na slici 1.6.



Slika 1.6: Apstraktno sintaksičko stablo koje formira semantički analizator prilikom obrade izraza iz primera 1.1.

Prilikom provere tipova prevodilac analizira tipove koji su dodeljeni svakoj imenovanoj adresabilnoj jedinici (promenljive) i svakom izrazu i vodi računa da su oni iskorišćeni u skladu sa validnim semantičkim kontekstima programskog jezika. Ovde prepoznajemo dve vrste procesa: identifikacija tipova i provera grešaka koji nastaju u nepoklapanju tipova.

Jedan moguć način implementacije provere tipova jeste u dva koraka:

1. Prilikom konstrukcije apstraktnog sintaksičkog stabla se izvode tipovi za svaku promenljivu, izraz, i dr. i ove informacije se čuvaju u tablici simbola. Vremenska složenost ovog koraka je $O(nh \cdot k)$, pri čemu je n broj sintaksičkih konstrukcija, h najveća visina sintaksičkog stabla, što je u prosečnom slučaju $O(\log n)$, a $O(k)$ vremenska složenost upisivanja informacija u tablicu simbola, pri čemu se uz inteligentan izbor strukture podataka, za vremensku složenost $O(k)$ se može postići $O(1)$.
2. Rekurzivnim obilaskom prethodno konstruisanog apstraktnog sintaksičkog stabla vrši se provera u skladu sa semantičkim modelom programskog jezika. Za svako podstablo važi: ako su poznati tipovi naslednika, tip korena se dobija primenom odgovarajućeg pravila koje dolazi iz definicije programskog jezika, na osnovu toga koji izraz se izračunava. Vremenska složenost ovog koraka je $O(m)$, gde je m broj čvorova u apstraktnom sintaksičkom stablu. Ponovo podrazumevamo da je vremenska složenost dobijanja pravila iz definicije programskog jezika $O(1)$, što je izvodljivo korišćenjem odgovarajućih struktura podataka.

Zapitajmo se kolika je ukupna vremenska složenost provere tipova. Videli smo da implicitne konverzije umeću nove čvorove koji nisu postojali u sintaksičkom stablu, te bi trebalo da važi $n \leq m$. Međutim, moguće je izmeniti strukturu čvora stabla tako da sadrži i konvertovanu vrednost, čime se eliminiše potreba za uvođenjem novog čvora u apstraktном stablu sintakse. Dakle, kako važi $n = m$, to je ukupna vremenska složenost operacije provere tipova u dva koraka $O(nh) \sim O(n \log n)$.

Naredni primer ilustruje analizu izvornog koda napisanog u programskom jeziku C pomoću kompilatora GCC. Leksička i sintaksička analiza prolaze bez grešaka, ali u semantičkoj

analizi dolazi do greške tokom provere tipova. Obratiti pažnju na veoma čitljivu poruku koju kompilator GCC prikazuje prilikom prevođenja.

Primer 1.9 Posmatrajmo sledeći izvorni kod napisan u programskom jeziku C:

```
#include <stdio.h>

int main() {
    double x = 3.7;
    char* niska;

    double y = x * niska;

    printf("lf", y);return 0;
}
```

Iako sintakški korektan, fragment sadrži semantičku grešku u kojoj se broj u pokretnom zarezu (`double`) `x` množi niskom (`char*`) `niska`, što dovodi do toga da kompilator prijavi grešku. Na primer, prilikom pokušaja prevođenja datog izvornog koda, GCC kompilator može prikazati grešku sličnu sledećoj:

```
primer.c: In function 'main':
primer.c:7:15: error: invalid operands to binary * (have 'double' and
      'char *')
        double y = x * niska;
               ^
```

1.4 Etapa sinteze programskih prevodilaca

Kao što smo rekli, etapa sinteze se sastoji od tri faze. To su: generisanje međukoda, optimizacija međukoda i generisanje izvršnog koda. U ovoj sekciji dajemo kratak osvrt na ove faze, bez ulazeњa u detalje.

Faza generisanja međukoda služi za generisanje međukoda koji može doprineti prednosti u prenosivosti prevodioca, odnosno, međukodom se mogu odvojiti zavisnosti programskog jezika od konkretnе mašine. Primeri međukodova koji se dobijaju u ovoj fazi su *p-kod* (engl. *p-code*) programskog jezika Pascal i već pomenuti *bajtkod* (engl. *bytecode*) programskog jezika Java. Najčešći oblik međureprezentacije koda je tzv. *troadresni kod* u kome se javljuju dodele promenljivama u kojima se sa desne strane dodele javlja najviše jedan operator. Otuda naziv „troadresni” — u svakoj instrukciji se navode „adrese” najviše dva operanda i rezultata operacije.

Primer 1.10 Neka je dat naredni fragment koda u programskom jeziku C:

```
if (v >= 120) {
    s = s0 + v * t;
}
```

Prilikom generisanja međukoda, naredbe kontrola toka (`if`, `if-else`, `while`, `for` i `do-while`) uklanjuju se i svode na uslovne i bezuslovne skokove (predstavljenih često u obliku naredbi `goto`).

Primer 1.11 Generisani troadresni međukod za fragment koda iz primera 1.10 bi mogao da izgleda:

```
ifFalse v < 120 goto L
```

```
t1 = v * t
s = s0 + t1
L:
```

Ako je, na primer, promenljiva `t` deklarisana celobrojnim tipom, a promenljive `s` i `v` su deklarisane tipom brojeva u pokretnom zarezu, onda bi se tokom semantičke analize u drvo ubacio čvor konverzije tipa i to bi se oslikalo i u međukodu:

```
ifFalse v < 120 goto L
t1 = int2float(t)
t2 = v * t1
s = s0 + t2
```

L:

Da bi se postigla troadresnost, u međukodu se vrednost svakog podizraza smešta u novouvedenu *privremenu promenljivu* (engl. *temporary*). U prethodnom primeru takve su promenljive `t1` i `t2`. Njihov broj je potencijalno neograničen, a tokom faze generisanja koda (tj. alokacije registara) svim promenljivama se dodeljuju fizičke lokacije gde se one skladište (to su ili registri procesora ili lokacije u glavnoj memoriji).

Optimizacija međukoda predstavlja sprovođenje niza različitih, često komplikovanih transformacija međukoda kako bi se dobio kod koji je optimizovaniji od početnog koda. Pod optimizacijom koda podrazumevamo popravljanje kvaliteta koda u skladu sa nekim metrikama. Primeri metrika su: brzina izvršavanja nekih operacija, veličina alocirane memorije koja je potrebna za izvršavanje naredbi i dr. Optimizacija se najčešće odvija na dva nivoa:

1. Optimizacija međukoda se generiše na početku faze sinteze i podrazumeva mašinski nezavisne optimizacije, tj. optimizacije koje ne uzimaju u obzir specifičnosti ciljne arhitekture.
2. Optimizacija ciljnog koda se izvršava na samom kraju sinteze i zasniva se na detaljnem poznavanju ciljne arhitekture i asemblerorskog i mašinskog jezika na kome se izražava ciljni program.

Navedimo nekoliko primera najčešćih optimizacija koje se vrše nad međukodom:

- Konstantni izrazi se mogu izračunati (engl. *constant folding*).

Primer 1.12 Fragment koda:

```
x = 2 + 2;
y = z * x;
može se optimizovati u:
x = 4;
y = z * x;
```

- Izbegava se upotreba promenljivih čija je vrednost konstantna (engl. *constant propagation*).

Primer 1.13 Fragment koda:

```
x = 4;  
y = z * x;  
može se optimizovati u:  
y = z * 4;
```

- Operacije se zamenjuju onim za koje se očekuje da će se izvršiti brže (engl. *strength reduction*).

Primer 1.14 Fragment koda:

```
y = 4 * x;  
može se optimizovati u:  
y = x << 2;
```

- Izbegava se vršenje istog izračunavanja više puta (engl. *common subexpression elimination*).

Primer 1.15 Fragment koda:

```
a = x + y;  
b = x - y;  
c = x + y;  
d = a + c;  
može se optimizovati u:  
a = x + y;  
b = x - y;  
c = a;  
d = a + c;
```

- Izbegava se uvodđenje promenljivih koje samo čuvaju vrednosti nekih postojećih promenljivih (engl. *copy propagation*).

Primer 1.16 Fragment koda:

```
a = x + y;  
b = x - y;  
c = a;  
d = a + c;  
može se optimizovati u:  
a = x + y;  
b = x - y;  
d = a + a;
```

- Izračunavanja vrednosti promenljivih koje se dalje ne koriste, eliminišu se (engl. *dead code elimination*).

Primer 1.17 Pod pretpostavkom da se nakon sledećeg fragmenta koda, promenljive **a** i **b** koriste dalje, ali se promenljiva **d** više ne koristi, tada naredni fragment koda:

```
a = x + y;
b = x - y;
d = a + a;
```

može se optimizovati u:

```
a = x + y;
b = x - y;
```

Tokom faze generisanja izvršnog koda optimizovani međukod se prevodi u izvršni asembleri, tj. mašinski kod. U tom procesu potrebno je napraviti nekoliko značajnih odluka koje utiču na kvalitet generisanog koda. One se donose tokom faze odabira instrukcija (tada se određuje kojim mašinskim instrukcijama se modeluju instrukcije troadresnog koda), faze alokacije registara (tada se određuje lokacija na kojoj se svaka od promenljivih skladišti) i faze raspoređivanja instrukcija (tada se određuje redosled instrukcija koji doprinosi kvalitetnjem iskorišćavanju protočne obrade i paralelizacije na nivou instrukcija). Ove faze su prilično isprepletane. Najčeće se prvo radi odabir instrukcija dok se raspoređivanje instrukcija nekada rade i pre i posle alokacije registara.

1.5 Pitanja i zadaci

Pitanje 1.1 Šta je programski jezik?

Pitanje 1.2 Šta je izvorni kod, a šta izvršni kod?

Pitanje 1.3 Šta je neophodno uraditi da bismo napisani kod mogli pokrenuti u vidu programa na računaru?

Pitanje 1.4 Šta je programski prevodilac, a šta kompiliranje?

Pitanje 1.5 Koje vrste programskega prevodilaca postoje? Koje su sličnosti, a koje su razlike između tih vrsta? Koje su prednosti korišćenja jednih u odnosu na druge?

Pitanje 1.6 Navesti etape kompiliranja, njihove faze, i kratko ih objasniti.

Pitanje 1.7 Da li su faze kompiliranja međusobno izolovane ili ne? Objasniti i dati (kontra)primer.

Pitanje 1.8 Šta je međukod? Navesti primer međukoda.

Pitanje 1.9 Šta je lekser, a šta parser?

Pitanje 1.10 Koja je uloga leksičkog analizatora?

Pitanje 1.11 Šta je leksema, a šta token?

Pitanje 1.12 Šta je tablica simbola? U kojim fazama analize se ona koristi?

Pitanje 1.13 Koja je uloga sintaksičkog analizatora i šta je rezultat njegovog rada?

Pitanje 1.14 Koja je uloga semantičkog analizatora?

Pitanje 1.15 Da li su svi sintaksički ispravni fragmenti ujedno i semantički ispravni? Da li su svi semantički ispravni fragmenti ujedno i sintaksički ispravni? Objasniti.

Pitanje 1.16 Koja od faza analize je najsporija? Objasniti.

Zadatak 1.17 U kojoj fazi analize će biti prijavljena greška prilikom analiziranja narednih fragmenata koda programskog jezika C, i objasniti gde se javila greška:

- int main() {

```
    double pi = 3.14;
    float f = 10; d = 0;

    return 0;
}
• int x = 0,
      y = 1,
      z = x * y - 12x0;
• char zdravo[] = "zdravo",
      ni$ka[] = "ni$ka";
size_t duzina_z = strlen(zdravo),
      duzina_n = strlen(ni$ka);
(pod uslovom da je dostupna funkcija strlen)
• int a[10];
a = 100;
• int f(int n) {
    if(n == 0)
        return 1
    else
        return return n*f(n-1);
}
```

Zadatak 1.18 Izdvojiti lekseme iz narednih fragmenata koda programskog jezika C, i za svaki fragment navesti koliko razlicitih tokena sadrži:

```
• int x = 3;
• double a = 1.4, b = 3;
• for(i = 0; i < n; ++i) {}
• int main() { return 0; }
• int nzd(int a, int b) {
    if(b == 0)
        return a;
    else
        return nzd(b, a % b);
}
```

Zadatak 1.19 Za svaki fragment koda iz zadatka 1.18, prikazati rezultat rada sintaksičkog analizatora.

2. Regularni jezici

Teorija formalnih jezika se bavi teorijskim problemima i rešenjima vezanim za leksičku i sintaksičku analizu. U ovom poglavlju ćemo prvo uvesti elementarne pojmove u teoriji, poput azbuke, reči, jezika, zatim ćemo se upoznati sa dva načina opisivanja jezika (pomoću regularnih izraza i kontekstnoslobodnih gramatika) i operacijama nad rečima i jezicima.

Svi ovi pojmovi će nam služiti da formalno uvedemo pojam regularnog jezika, sagledamo njihova proširenja i njihov značaj u praktičnoj primeni. Regularnim jezicima ćemo posvetiti najviše pažnje u ovom poglavlju.

Da bismo razumeli kako funkcioniše leksička analiza kroz regularne jezike, uvodimo pojam konačnog automata, njegove varijante koje se odnose na determinizam, kao i jezik automata. Značaj ovog poglavlja ogleda se u Klinijevoj teoremi, čiji se dokaz sastoji od sekvensijalne primene nekoliko algoritama koji će biti detaljno opisani i objašnjeni kroz primere.

Poglavlje završavamo objašnjavanjem kako se određuje presek dva jezika i značaj ovog postupka za određivanje ostalih operacija sa jezicima, kao i prikazivanjem granice moći regularnih jezika.

2.1 Azbuka i jezici

Ovo poglavlje započećemo dvama pitanjima:

1. Da li u programskom jeziku C može postojati promenljiva imena **reč**?
2. Da li u programskom jeziku Java može postojati promenljiva imena **reč**?

Čitaocu je verovatno poznato da je odgovor na prvo pitanje „ne”, a na drugo „da”. Kao što vidimo, ne postoji jedinstven skup reči koji možemo da pridružimo domenu imena promenljivih u svim programskim jezicima. Neki programski jezici podržavaju šire tabele karaktera (poput *Unicode*, ili nekih njegovih podskupova, kao što je slučaj sa programskim

jezikom Java), dok neke imaju samo osnovni skup karaktera (najčešće *ASCII*, kao što je slučaj sa programskim jezikom C). Zbog ovog razloga, na početku ovog poglavlja ćemo se prvo upoznati sa osnovnim pojmovima koji će nam biti važni za dalji rad.

Definicija 2.1.1 *Azbuka ili alfabet*, u oznaci Σ , skup je simbola koje nazivamo *karakteri*.

Po konvenciji, elemente azbuke obeležavamo simbolima a, b, c itd. Kako skupovi mogu biti konačni i beskonačni, tako i azbuke mogu biti konačne i beskonačne. Međutim, pošto beskonačne azbuke ne možemo da predstavimo u računaru, koncentrisaćemo se samo na konačne azbuke, te ćemo u daljem tekstu pod pojmom azbuka misliti na konačnu azbuku.

Definicija 2.1.2 Konačan niz $a_1, a_2, \dots, a_n \in \Sigma$ nazivamo *reč*, *niska* ili *string* azbuke Σ . Broj n naziva se *dužina niske* i obeležava se sa $|x|$. *Prazna reč*, u oznaci ϵ , reč je dužine nula.

Po konvenciji, reči obeležavamo simbolima x, y, z , itd.

Primer 2.1 Neka je zadata azbuka $\Sigma = \{a, b\}$. Neke od reči azbuke Σ su $x = ab$, $y = bba$ i $z = bbbb$.

Definicija 2.1.3 *Opšti jezik* azbuke Σ , u oznaci Σ^* , skup je svih reči nad azbukom Σ .

Primetimo da je opšti jezik neke azbuke beskonačan skup, osim u trivijalnom slučaju, kad je jedini element azbuke prazna reč.

Primer 2.2 Neka je zadata azbuka $\Sigma = \{a, b\}$. Opšti jezik azbuke Σ je $\Sigma^* = \{\epsilon, a, b, ab, ba, aaa, aab, aba, abb, baa, bab, bba, bbb, \dots\}$.

Definicija 2.1.4 Skup svih reči bez prazne reči nad Σ se obeležava sa Σ^+ :

$$\Sigma^+ = \Sigma^* - \epsilon$$

Definicija 2.1.5 *Jezik* nad azbukom Σ je bilo koji skup $L \subseteq \Sigma^*$.

Dakle, jezik je proizvoljan podskup reči opštег jezika neke azbuke. Zbog toga je jasno da postoje konačni i beskonačni jezici.

Primer 2.3 Jezik $L_1 = \{b^k a \mid k \geq 0\}$ sastoji se od reči u kojima se slovo b pojavljuje proizvoljan nenegativan broj puta i završavaju se slovom a . Jeziku pripadaju niske a , ba , bba , $bbba$, $bbbba$, itd.

Primer 2.4 Jezik $L_2 = \{a^n b^n \mid n > 0\}$. sadrži sve reči koje počinju pozitivnim brojem slova a i završavaju se istim brojem slova b . Ovom jeziku pripadaju niske ab , $aabb$, $aaabbb$, $aaaabbbb$, ...

Jezik možemo opisati pomoću klase:

1. regularnih jezika (RJ), ili
2. kontekstnoslobodnih gramatika (KSG).

Činjenica je da nijedna klasa nije dovoljno jaka da opiše sve jezike. Ono što je zanimljivo jeste da je primena klase KSG šira od klase RJ. Međutim, klasa RJ može da opiše sve

jezike bitne za leksičku analizu, te čemo zbog toga tu klasu i izučavati nadalje u tekstu. Sa druge strane, klasa KSG pokriva sintakšku analizu, pa čemo se njom baviti u narednom poglavlju.

Još jedna zanimljivost jeste ta da, ako pogledamo jezik L_2 iz primera 2.4, on ne može da se opiše pomoću klase RJ. Naime, ispostavlja se da važi sledeće pravilo. Ukoliko se traži uparivanje dva entiteta (u posmatranom primeru imamo da reči jezika L_2 imaju jednak broj karaktera a i b), takav jezik ne pripada klasi RJ. U narednom poglavlju čemo precizirati ovaj problem lemom o razrastanju. Za sada, razmotrimo naredni primer.

Primer 2.5 Da li je moguće iskoristiti klasu RJ za izraze poput

$$(2 + 5) * (3 - 9/3)?$$

Na prvi pogled se ne čini da postoji uparivanje. Međutim, pažljivijim posmatranjem uvidićećemo da postoji uparivanje zagrada. To nam daje odgovor da klasa RJ nije dovoljno dobra za opisivanje aritmetičkih izraza.

Ovim smo postavili glavni problem kojim čemo se baviti u narednom tekstu, a to je problem pripadnosti reči jeziku.

2.2 Operacije nad rečima i jezicima

U ovom odeljku čemo se upoznati sa operacijama nad rečima i jezicima. Ograničićemo se na jednu operaciju nad rečima i devet operacija nad jezicima. Cilj ovog odeljka jeste da uvedemo neophodne pojmove za definisanje pojmove regularni jezik i regularni izraz.

Definicija 2.2.1 Neka su date dve reči x i y . Operacija **spajanje** (dopisivanje, konkatenacija) **reči y na reč x** , u oznaci $x \cdot y$, predstavlja operaciju:

$$x, y \longmapsto x \cdot y = xy.$$

Uz prethodnu definiciju važi i da je dužina novodobijene reči jednaka zbiru dužina reči koje su spojene, odnosno:

$$|ab| = |a| + |b|$$

Primer 2.6 Neka je $x = auto$, a $y = put$. Tada je $x \cdot y = autoput$ (odnosno, $auto \cdot put$).

Operacija spajanje je primer binarne operacije. Ispostavlja se da je ona asocijativna (ovu osobinu nije teško dokazati) odnosno za proizvoljne reči x , y i z proizvoljnog opštег jezika Σ^* važi:

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$

Kako se ispostavlja da za svaku reč x proizvoljnog opštег jezika Σ^* važi i

$$\epsilon \cdot x = x \cdot \epsilon = x,$$

možemo zaključiti da je struktura $(\Sigma^*, \cdot, \varepsilon)$ jedan monoid. Da li je ta struktura možda i grupa? Lako se vidi da važi

$$(\forall x, x^{-1} \in \Sigma^* \setminus \{\varepsilon\}) : x \cdot x^{-1} = \varepsilon \wedge x^{-1} \cdot x = \varepsilon,$$

te zaključujemo da prethodna struktura ne može biti grupa. Međutim, to što nije grupa, ne znači da nije komutativan monoid. Ipak, ukoliko pogledamo primer 2.6, vidimo da važi

$$x \cdot y = auto \cdot put \neq put \cdot auto = y \cdot x,$$

te smo dali jedan kontraprimer koji dokazuje da pomenuta struktura nije komutativan monoid (u opštem slučaju). Napomenimo da postoje određeni potrebni i dovoljni uslovi za komutativnost ovog monoida koji se mogu pronaći u Vitas 2006.

Lema 2.2.1 — Levi. Neka su a, b, c i d reči nad azbukom Σ . Tada važi jednakost $ab = cd$ ako i samo ako je zadovoljen neki od sledećih uslova:

1. $a = c$ i $b = d$
2. Postoji neprazna reč x nad azbukom Σ , takva da je $a = cx$ i $d = xb$.
3. Postoji neprazna reč y nad azbukom Σ , takva da je $c = ay$ i $b = yd$.

Pređimo sada na operacije nad jezicima. Neka su L_1 i L_2 dva jezika proizvoljne azbuke Σ . S obzirom da je jasno da su jezici skupovi elemenata (koje nazivamo rečima), to za njih važe opšte operacije nad skupovima:

Definicija 2.2.2 *Presek jezika L_1 i L_2 , u označi $L_1 \cap L_2$, skup (jezik) je svih elemenata koji se nalaze u jeziku L_1 i L_2 , tj.*

$$L_1 \cap L_2 \stackrel{df}{=} \{x \mid x \in L_1 \wedge x \in L_2\}.$$

Definicija 2.2.3 *Unija jezika L_1 i L_2 , u označi $L_1 \cup L_2$, skup (jezik) je svih elemenata koji se nalaze u jeziku L_1 ili L_2 , tj.*

$$L_1 \cup L_2 \stackrel{df}{=} \{x \mid x \in L_1 \vee x \in L_2\}.$$

Definicija 2.2.4 *Razlika jezika L_1 i L_2 , u označi $L_1 \setminus L_2$, skup (jezik) je svih elemenata koji se nalaze u jeziku L_1 i ne nalaze se u jeziku L_2 , tj.*

$$L_1 \setminus L_2 \stackrel{df}{=} \{x \mid x \in L_1 \wedge x \notin L_2\}.$$

Definicija 2.2.5 *Komplement jezika L_1 , u označi L_1^c , skup (jezik) je svih elemenata koji se ne nalaze u jeziku L_1 , tj.*

$$L_1^c \stackrel{df}{=} \{x \mid x \in \Sigma^* \setminus L_1\}.$$

Definicija 2.2.6 *Dekartov proizvod jezika L_1 i L_2 , u označi $L_1 \times L_2$, skup (jezik) je svih uređenih parova takvih da se prva koordinata nalazi u jeziku L_1 i druga koordinata nalazi u L_2 , tj.*

$$L_1 \times L_2 \stackrel{df}{=} \{(x, y) \mid x \in L_1 \wedge y \in L_2\}.$$

Primer 2.7 Neka su $L_1 = \{ab, b\}$ i $L_2 = \{ba, a\}$. Dekartov proizvod datih jezika je

$$L_1 \times L_2 = \{(ab, ba), (ab, a), (b, ba), (b, a)\}.$$

Osim standardnih operacija nad skupovima, definišemo i operaciju proizvoda jezika.

Definicija 2.2.7 Proizvod jezika L_1 i L_2 , u oznaci $L_1 \cdot L_2$, skup (jezik) je svih elemenata oblika $x \cdot y$, pri čemu se x nalazi u jeziku L_1 i y u L_2 , tj.

$$L_1 \cdot L_2 \stackrel{df}{=} \{x \cdot y \mid x \in L_1 \wedge y \in L_2\}.$$

Neutral je skup $\{\epsilon\}$ i važi:

$$L \cdot \{\epsilon\} = \{\epsilon\} \cdot L = L$$

Primer 2.8 Neka su $L_1 = \{ab, b\}$ i $L_2 = \{ba, a\}$. Proizvod datih jezika je

$$L_1 \cdot L_2 = \{abba, aba, bba, ba\}.$$

Bitno je obratiti pažnju na to da prazan skup ne može biti neutral jer za svako $L \subseteq \Sigma^*$, gde je Σ proizvoljna azbuka, važi:

$$L \cdot \emptyset = \emptyset \cdot L = \emptyset$$

Definicija 2.2.8 Stepen jezika L , u oznaci L^n , definiše se rekurentnom relacijom:

$$\left\{ \begin{array}{l} L^0 \stackrel{df}{=} \{\epsilon\}, \\ L^n \stackrel{df}{=} L \cdot L^{n-1} = L^{n-1} \cdot L \end{array} \right\}.$$

Primer 2.9 Neka je $L = \{ab, b\}$. Tada je

$$\begin{aligned} L^3 &= \{ab, b\}^3 = \\ &= \{ab, b\} \cdot \{ab, b\}^2 = \\ &= \{ab, b\} \cdot \{ab, b\} \cdot \{ab, b\} = \\ &= \{abab, abb, bab, bb\} \cdot \{ab, b\} = \\ &= \{ababab, ababb, abbab, abbb, babab, babb, bbab, bbb\}. \end{aligned}$$

Nije teško dokazati da važi $L \cdot L^{n-1} = L^{n-1} \cdot L$ u opštem slučaju (dokaz se izvodi indukcijom po n), ali mi ćemo prikazati dokaz za $n = 3$:

$$\begin{aligned} L^3 &= L \cdot L^2 = \\ &= L \cdot (L \cdot L) = \\ &= L \cdot (L \cdot (L \cdot \{\epsilon\})) = \\ &= ((L \cdot L) \cdot L) \cdot \{\epsilon\} = \\ &= (L^2 \cdot L) \cdot \{\epsilon\} = \\ &= L^3 \cdot \{\epsilon\} = \\ &= L^3, \end{aligned}$$

pri čemu četvrta jednakost sledi iz asocijativnosti operacije spajanja.

Primetimo da je u primerima 2.8 i 2.9 ispunjeno da je kardinalnost proizvoda jezika jednaka proizvodu kardinalnosti svakog od jezika. Da li ovo važi za bilo koji odabir jezika?

Primer 2.10 Neka je $L_1 = \{ab, a\}$ i $L_2 = \{b, \epsilon\}$. Tada je $L_1 \cdot L_2 = \{ab, a\} \cdot \{b, \epsilon\} = \{abb, ab, a\}$ (u pitanju je skup, pa je element ab koji je dobijen spajanjem $ab \cdot \epsilon$ jednak elementu ab koji je dobijen spajanjem $a \cdot b$, te se on piše samo jednom u skupu). Dakle, dobili smo tri elemenata umesto četiri, te u opštem slučaju ne važi da je kardinalnost proizvoda jednaka proizvodu kardinalnosti.

Stepen jezika je veoma korisna operacija. Posmatrajmo naredni primer.

Primer 2.11 Posmatrajmo jezik $L_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Primetimo da su $L_1^2 = \{00, 01, ..., 09, 10, 11, ..., 19, ..., 90, 91, ..., 99\}$ i $L_1^3 = \{000, 001, ..., 009, ..., 990, 991, ..., 999\}$. Ukoliko bismo posmatrali prirodne brojeve kao reči (različitih dužina) koje imaju cifre za svoje simbole, onda bismo mogli reći da je jezik skupa prirodnih brojeva, označimo taj jezik \mathbf{N} , zadat sa

$$L_1^1 \cup L_1^2 \cup L_1^3 \cup \dots \cup L_1^i \cup \dots,$$

gde je $i > n$, $\forall n \in \mathbf{N}$. Međutim, \mathbf{N} ne predstavlja jezik koji odgovara jeziku prirodnih brojeva \mathbf{N} .

Definicija 2.2.9 Neka je L jezik proizvoljne azbuke Σ . Skup

$$L^* \stackrel{df}{=} \bigcup_{i=0}^{\infty} L^i$$

nazivamo *Klinijevo zatvorene jezika* L , a skup

$$L^+ \stackrel{df}{=} \bigcup_{i=1}^{\infty} L^i$$

nazivamo *pozitivno Klinijevo zatvorene jezika* L .

Primetimo da je $L^* = L^+ \cup \{\epsilon\}$.

Primer 2.12 Jezik prirodnih brojeva (tj. jezik čije reči odgovaraju brojevima skupa \mathbf{N}) definišemo korišćenjem operacija nad jezicima na sledeći način. Neka su $L_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ i $L_2 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Tada je

$$\mathbf{N} \stackrel{df}{=} L_2 \cdot L_1^* = \{1, 2, \dots, 9, 10, 11, \dots, 19, \dots\}.$$

Dodatno,

$$\mathbf{N}_0 \stackrel{df}{=} L_2 \cdot L_1^* \cup \{0\} = \mathbf{N} \cup \{0\}.$$

2.2.1 Prioritet operacija nad jezicima

Primer 2.13 Da li regularni izrazi ab^* i $(ab)^*$ opisuju isti jezik? Odgovor je ne, jer regularni izraz ab^* opisuje reči jezika $\{a, ab, abb, \dots\}$, dok regularni izraz $(ab)^*$ opisuje reči jezika $\{\epsilon, ab, abab, ababab, \dots\}$.

Ovo nas dovodi do pitanja prioriteta operacija koje možemo primeniti nad jezicima. Važi sledeća raspodela:

- Operator $*$ ima najveći prioritet,
- Operator \cdot ima srednji prioritet,
- Operatori \cup i \cap imaju najmanji prioritet.

Naravno, ukoliko želimo da sami damo veći prioritet nekoj operaciji, onda možemo nju i njene operate da ogradimo zagradama (i), i time ćemo postići željeni prioritet.

Primer 2.14 Jeziku koji je opisan regularnim izrazom $ab|c$ pripadaju reči iz skupa $\{ab, c\}$, dok jeziku koji je opisan regularnim izrazom $a(b|c)$ pripadaju reči iz skupa $\{ab, ac\}$.

Sada ćemo dati formulaciju jedne poznate leme čiji dokaz nećemo prikazati, ali napomenemo da se izvodi indukcijom po dužini reči.

2.3 Regularni jezici

U mnogim primenama dovoljno je posmatrati klasu jezika koja se formira polazeci od jezika $\{a\}$ i $\{\epsilon\}$ primenom operacija unije, dopisivanja i zatvorenja. Ovu klasu jezika nazivamo regularni jezici.

Definicija 2.3.1 Neka je Σ proizvoljna azbuka. Definicija pojma **regularni jezik** glasi:

1. Ako $a \in \Sigma$, onda je jezik $\{a\}$ regularni jezik.
2. Jezik $\{\epsilon\}$ je regularni jezik.
3. Jezik \emptyset je regularni jezik.
4. Ako su $L_1, L_2 \subseteq \Sigma^*$ regularni jezici, onda je i jezik $L_1 \cup L_2$ regularan jezik.
5. Ako su $L_1, L_2 \subseteq \Sigma^*$ regularni jezici, onda je i jezik $L_1 \cdot L_2$ regularan jezik.
6. Ako je $L \subseteq \Sigma^*$ regularan jezik, onda je i jezik L^* regularan jezik.

Primetimo da je ova definicija rekurzivna. Prva tri iskaza predstavljaju bazu rekurzije, a preostala tri iskaza njen korak. Ova definicija se kraće (i neformalno) može opisati sledećom rečenicom: *Jezici su regularni ako su trivijalni ili ako se mogu dobiti od jednostavnijih regularnih jezika primenom operacija \cup , \cdot , i * .*

Primer 2.15 Ovom definicijom smo pokazali da je jezik \mathbf{N}_0 regularan jezik. Možemo ga zapisati na sledeći način:

$$\mathbf{N}_0 \stackrel{df}{=} \left[\underbrace{(\{1\} \cup \{2\} \cup \dots \cup \{9\})}_{L_2} \cdot \underbrace{(\{0\} \cup \{1\} \cup \dots \cup \{9\})^*}_{L_1^*} \right] \cup \{0\}.$$

Sada kada imamo definiciju regularnog jezika, možemo definisati regularni izraz.

Definicija 2.3.2 Neka je Σ proizvoljna azbuka. Definicija pojma **regularni izraz** glasi:

1. Ako $a \in \Sigma$, onda regularni izraz a opisuje regularni jezik $\{a\}$.
2. Regularni izraz ϵ opisuje regularni jezik $\{\epsilon\}$.
3. Regularni izraz \emptyset opisuje regularni jezik \emptyset .
4. Ako su e_1 i e_2 regularni izrazi koji opisuju regularne jezike $L(e_1)$, $L(e_2) \subseteq \Sigma^*$ redom, onda je i izraz $e_1|e_2$ regularni izraz koji opisuje regularni jezik $L(e_1) \cup L(e_2)$.
5. Ako su e_1 i e_2 regularni izrazi koji opisuju regularne jezike $L(e_1)$, $L(e_2) \subseteq \Sigma^*$ redom, onda je i izraz e_1e_2 regularni izraz koji opisuje regularni jezik $L(e_1) \cdot L(e_2)$.
6. Ako je e regularni izraz koji opisuje regularni jezik $L(e) \subseteq \Sigma^*$, onda je i izraz e^* regularni izraz koji opisuje regularni jezik $L(e)^*$.

Primetimo da je i ova definicija rekurzivna.

Primer 2.16 Regularni jezik \mathbb{N}_0 opisuje se regularnim izrazom $((1|2|\dots|9)(0|1|\dots|9)^*)|0$.

Primer 2.17 Regularni izraz ab^*a opisuje regularni jezik $L(ab^*a) = \{aa, aba, abba, abbba, \dots\}$.

Videli smo da svi regularni jezici mogu nastati primenom operacija \cup , \cdot ili * na jednostavnije regularne jezike. Neko bi mogao postaviti pitanje da li i jezik $L_1 \cap L_2$ predstavlja regularni jezik ako su L_1 i L_2 regularni jezici. Odgovor bi bio potvrđan, ali ovu činjenicu nije jednostavno dokazati. Dokaz ćemo prikazati u sekciji 3.3.

2.3.1 Proširenja regularnih izraza i jezika

Veličina zapisa regularnog izraza brzo raste kako raste složenost regularnog jezika, što se možemo uveriti već iz primera 2.16. Radi skraćivanja zapisa i povećanja čitljivosti, uvode se tzv. *prošireni regularni izrazi*, koji predstavljaju kraće zapise nekih osnovnih regularnih izraza. Neki jednostavni prošireni regularni izrazi i ekvivalentni regularni izrazi koje oni zamenjuju su dati u tabeli 2.1.

Primer 2.18 U nastavku dajemo primere konkretnih proširenih regularnih izraza i ekvivalentnih regularnih izraza koje oni zamenjuju:

- RI aa^* možemo kraće zapisati PRI a^+ .
- RI $0|1|2|3|4|5|6|7$ možemo kraće zapisati PRI $[0-7]$.
- RI $a|\epsilon$ možemo kraće zapisati PRI $a?$.
- Jezik svih reči koje se sastoje od tačno 6 malih slova ASCII kodne šeme može se opisati PRI $[a-z]^6$.

Tabela 2.1: Primeri proširenih regularnih izraza i regularnih izraza koje oni zamenjuju.

<i>Prošeni regularni izraz</i>	<i>Regularni izraz</i>
$[a_1 a_2 \dots a_n]$	$a_1 a_2 \dots a_n$
$[^a a_1 a_2 \dots a_n]$	$b_1 b_2 \dots b_m$, pri čemu $b_1, b_2, \dots, b_m \in \Sigma \setminus \{a_1, a_2, \dots, a_n\}$
$[a_1 - a_n]$	$a_1 a_2 \dots a_n$, pri čemu se pretpostavlja postojanje uređenja $a_1 \prec a_2 \prec \dots \prec a_n$
$X?$	$X \epsilon$, gde je X neki regularan izraz
X^+	XX^* , gde je X neki regularan izraz
$X\{n\}$	$\underbrace{XX \dots X}_{n \text{ puta}}$, gde je X neki regularan izraz
$X\{n, m\}$	$\underbrace{XX \dots X}_{\text{od } n \text{ do } m \text{ puta}}$, gde je X neki regularan izraz
$X\{n, \}$	$\underbrace{XX \dots}_{n \text{ ili više puta}}$, gde je X neki regularan izraz

Prošireni regularni izrazi ne utiču na to da li je jezik koji je opisan proširenim regularnim izrazom regularan ili ne. Sve što možemo zapisati proširenim može se zapisati i osnovnim regularnim izrazima, ali po cenu dužine zapisa.

Regularnim definicijama možemo da imenujemo neki regularni izraz i da ga takvog koristimo u drugim izrazima. Ako je Σ azbuka, onda je regularna definicija niz definicija oblika:

$$\begin{aligned} d_1 &\longrightarrow r_1 \\ d_2 &\longrightarrow r_2 \\ &\dots \\ d_n &\longrightarrow r_n \end{aligned}$$

pri čemu važi:

- Svako d_i je simbol koji se ne pojavljuje u azbuci Σ .
- Svako d_i je različito od ostalih.
- Svako r_i je regularni izraz nad azbukom $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$.

Navedimo primere ovakve upotrebe.

Primer 2.19 Pomoću

$$okt \longrightarrow [0 - 7]$$

definisali smo oktalnu cifru kao cifru između cifara 0 i 7, a pomoću

$$oktbr \longrightarrow okt +$$

definisali smo oktalni broj kao jednu ili više oktalskih cifara.

Primer 2.20 Identifikator u programskom jeziku C je reč koja može da sadrži slova, cifre i podvlake. Regularna definicija koja opisuje jedan takav identifikator:

$\text{slово_} \longrightarrow [A - Za - z_]$
 $\text{broj} \longrightarrow [0 - 9]$
 $\text{identifikator} \longrightarrow \text{slово_}(\text{slово_}|\text{broj})^*$

Regularnim definicijama možemo olakšati ispravljanje drugih regularnih izraza u kojima smo koristili definiciju. Naime, potrebno je izmenu primeniti samo na jednom mestu (pri definisanju) da bi se ona oslikala na sve ostale, što značajno olakšava posao.

2.3.2 Pitanja i zadaci

Pitanje 2.1 Šta je azbuka? Šta je reč?

Pitanje 2.2 Šta je opšti jezik neke azbuke? Šta je jezik?

Pitanje 2.3 Na koje načine možemo zadati jezik, a na koje načine ga možemo opisati?

Pitanje 2.4 Na šta se odnose skraćenice RJ i KSG? Koje su njihove sličnosti, a koje su razlike?

Pitanje 2.5 Šta je operacija spajanja reči? Šta je neutral za tu operaciju?

Pitanje 2.6 Operacija spajanja reči je:

- (a) asocijativna, (b) komutativna, (c) unarna, (d) binarna.

Pitanje 2.7 Struktura $(\Sigma^*, \cdot, \epsilon)$ je primer:

- (a) monoida, (b) grupe, (c) prstena, (d) polja.

Pitanje 2.8 Da li je struktura $(\Sigma^*, \cdot, \epsilon)$ komutativna? Obrazložiti.

Pitanje 2.9 Šta je: (a) presek, (b) unija, (c) razlika, (d) komplement, (e) Dekartov proizvod, (f) proizvod, (g) stepen jezika?

Pitanje 2.10 Šta je neutral za operaciju proizvoda jezika?

Pitanje 2.11 Dokazati ili opovrgnuti tvrđenje $(\forall \Sigma)(\forall L \subseteq \Sigma^*) : L \cdot L^{n-1} = L^{n-1} \cdot L$.

Pitanje 2.12 Dokazati ili opovrgnuti tvrđenje $(\forall \Sigma)(\forall L_1, L_2 \subseteq \Sigma^*) : \text{card}(L_1 \cdot L_2) = \text{card}(L_1) \cdot \text{card}(L_2)$, pri čemu operacija $\text{card}(L)$ predstavlja kardinalnost jezika L .

Pitanje 2.13 Šta je Klinijevo zatvorene jezika, a šta je pozitivno Klinijevo zatvorene jezika? Da li postoji veza između njih? Ako postoji, kako ona glasi?

Zadatak 2.14 Konstruisati primer sa dva jezika L_1 i L_2 u kojem ne važi da je kardinalnost proizvoda jezika jednaka proizvodu kardinalnosti jezika (različit od primera 2.10), a zatim konstruisati primer koji ispunjava isti uslov, ali u kojem važi da nijedan skup ne sadrži ϵ . ■

Zadatak 2.15 Koristeći operacije nad jezicima definisati jezik kojim se opisuju standardne registracione tablice. ■

Pitanje 2.16 Dat je regularni izraz $(ab*a)^+$. Koje od sledećih reči pripadaju jeziku koji je opisan datim regularnim izrazom: (a) ϵ , (b) aaa , (c) $aaaa$, (d) $ababa$, (e) $abaaa$?

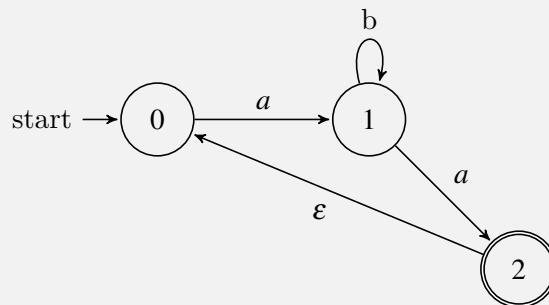
Zadatak 2.17 Napisati regularni izraz za jezik identifikatora u programskom jeziku C. ■

3. Konačni automati

3.1 Konačni automati

Konačni automati opisuju formalizam pomoću kojeg možemo proveriti da li neka reč pripada određenom regularnom jeziku. Ilustrujmo funkcionisanje konačnih automata sledećim primerom.

Primer 3.1



Ovo je primer jednog **nedeterminističkog** konačnog automata ((N)KA). Njemu odgovara regularni izraz $(ab^*a)^+$. Konačni automat se sastoji od **stanja**, pri čemu se **unutrašnja stanja** označavaju krugom, a **završna stanja** duplim krugom. **Početno stanje** je označeno strelicom koja vodi iz oznake *start*.

Primenimo ovaj automat nad niskom "abaaa" kako bismo proverili da li pripada jeziku. Krećemo iz početnog stanja 0, nailazimo na karakter *a* koji čitamo i prelazimo u stanje 1. Nakon toga, nailazimo na karakter *b* koji čitamo i prelazimo u stanje 1. Nakon toga, nailazimo na karakter *a* koji čitamo i prelazimo u stanje 2. Nakon toga, nailazimo na karakter *a* koji ne možemo da pročitamo iz stanja 2, ali umesto toga možemo da

posmatramo kao da smo naišli na praznu reč, pročitali je i prešli u stanje 0^1 . U nastavku čitamo a , prelazimo u stanje 1, čitamo a i prelazimo u stanje 2. Ovo se kraće može zapisati sledećom notacijom

$$_0a_1b_1a_2\varepsilon_0a_1a_2,$$

koju ćemo koristiti nadalje u tekstu. Pošto smo završili u stanju 2, onda možemo da prihvatimo zadatu reč jer je stanje 2 završno.

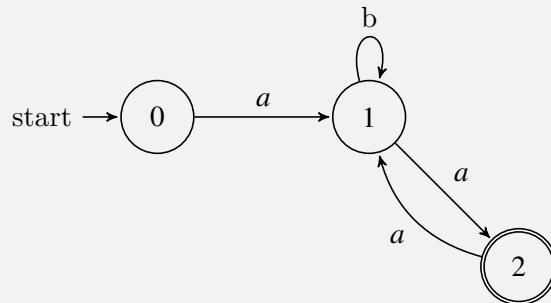
Primenimo isti automat nad niskom "aaa". Vidimo da je rezultat primene automata:

$$_0a_1a_2\varepsilon_0a_1,$$

ali kako smo završili u stanju 1, a to stanje nije završno, to zadata reč ne može biti prihvaćena.

Vidimo da je poprilično nezgodno raditi sa automatima koji podržavaju ε -prelaze. Moguće je napraviti konačne automate koji su ekvivalentni ovakvim konačnim automatima, ali koji ne sadrže pomenute ε -prelaze.

Primer 3.2 Konačni automat iz primera 3.1 ekvivalentan je konačnom automatu:

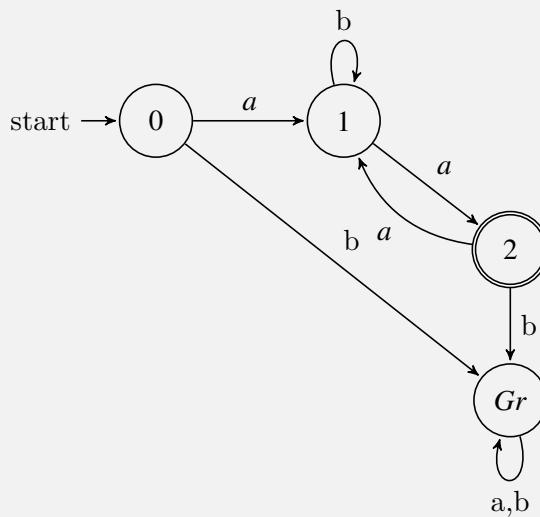


Međutim, ovaj konačni automat naziva se **deterministički** konačni automat (DKA).

Problem koji se ovde može javiti jeste ako automat ne stigne do kraja reči. Ovaj problem nastaje zbog toga što ne postoje prelazi iz svih stanja preko svih karaktera (recimo, iz stanja 0 i 2 ne postoje prelazi preko karaktera b). Pravilo je da i u tom slučaju ne treba da prihvatimo tu reč. Zbog pojave ovog problema, uvodimo stanje greške, kao u sledećem primeru.

Primer 3.3 Konačni automat

¹Reč $abaaa$ ekvivalentna je reči $aba\varepsilon aa$ (podsetite se zašto), te je ovakvo rezonovanje potpuno ispravno.

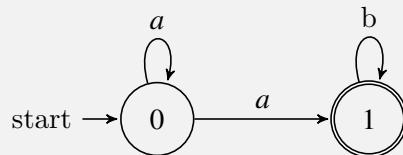


predstavlja upotpunjivanje konačnog automata iz primera 3.2 i naziva se **potpuni konačni** deterministički automat (PDKA). Stanje Gr naziva se **stanje greške**, i ne predstavlja ništa drugo do unutrašnje stanje. Stanje greške ne sme da bude završno jer bi u tom slučaju automat prihvatio pogrešnu reč. Postupak pri čitanju reči koja dospe u stanje greške je takav da čitamo ostatak reči do kraja i sve karaktere vodimo u stanje greške.

Posmatrajmo složenost izvršavanja algoritma čitanja reči u konačnom automatu. Jasno je da je vremenska složenost linearna po dužini reči koju čitamo ($O(n)$, gde je n dužina reči). Prostorna složenost je konstantna ($O(1)$) za proizvoljan ulaz jednom kada je konačni automat konstruisan.

Napomenimo da ϵ uvodi nedeterminizam jer postojanje ϵ -prelaza dovodi do toga da nije u svakom trenutku jasno kojom granom treba nastaviti. Međutim, to nije jedini slučaj nedeterminizma.

Primer 3.4 Regularni izraz a^*ab^* možemo opisati sledećim konačnim automatom:



On je takođe nedeterministički jer postoje dve grane kojima se može nastaviti kad se nađe na karakter a (jedna vodi u stanje 0, a druga u stanje 1).

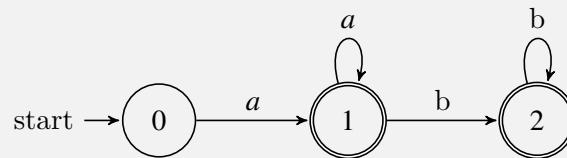
Primer 3.5 Primenimo konačni automat iz primera 3.4 na reč "aabbb". Moguća su tri slučaja čitanja ove reči:

1. $0a_0a_0b_?$ – nakon čitanja drugog karaktera a prelazimo u stanje 0 i tu se proces zaglavljuje,
2. $0a_0a_1b_1b_1b_1$ – nakon čitanja drugog karaktera a prelazimo u stanje 1 i daljim čitanjem preostalih karaktera b prelazimo u stanje 1, u kojem smo čitanje i završili

- što rezultira u prihvatanju reči, i
3. $0\alpha_1\alpha_2$ – nakon čitanja prvog karaktera a prelazimo u stanje 1 i tu se proces zaglavljuje.

Napomenimo da je moguće implementirati simulator ponašanja nedeterminističkog automata predstavljen primerom 3.5 rekurzijom, tj. primenjivanjem algoritamske strategije pretrage (*backtracking*). U tom slučaju je vremenska složenost eksponencijalna. U praksi se radi sledeće: od regularnog izraza se prvo napravi (N)KA, a potom se od (N)KA napravi DKA. Ilustrijmo ovo sledećim primerom.

Primer 3.6 Nedeterministički konačni automat iz primera 3.4 možemo transformisati u sledeći deterministički konačni automat:



Videćemo u daljem tekstu da važi da se svaki nedeterministički konačni automat može transformisati u deterministički konačni automat.

3.1.1 Definicija (N)KA

Predimo sada na formalno definisanje pojma konačni automat. I ovde ćemo koristiti oznaku $(N)KA$, jer se može smatrati da su nedeterministički konačni automati natkласа determinističkih konačnih automata, tj. za druge važe neke posebne osobine. Zato možemo reći da su pojmovi (*opšti*) *konačni automat* i *nedeterministički konačni automat* identični.

Definicija 3.1.1 (Nedeterministički) Konačni automat, u oznaci $(N)KA$, uređena je petorka

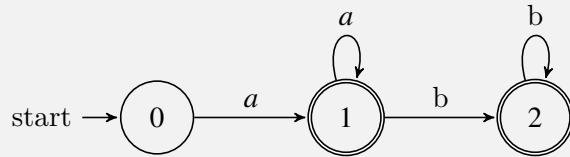
$$(\Sigma, Q, I, F, \Delta),$$

za koju važi:

- Σ je **azbuka** nad čijim rečima primenjujemo automat,
- Q je **konačan skup stanja**,
- $I \subseteq Q$ je **skup početnih stanja** (oznaka potiče od reči *Initial*),
- $F \subseteq Q$ je **skup završnih stanja** (oznaka potiče od reči *Final*),
- Δ je **relacija prelaska automata**, tj. $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$, a često se naziva još i **skup grana automata**.

Pre nego što predemo na primer, razjasnimo zašto je neophodno uključiti skup $\{\epsilon\}$ u definiciju relacije prelaska automata. Videli smo da smo još u primeru 3.1 dozvoljavali da konačni automat pređe iz jednog stanja u drugo stanje preko ϵ -prelaza. Međutim, ϵ je (prazna) reč, a ne karakter azbuke, te nismo mogli da stavimo samo Σ u definiciji. Za označavanje pojedinačnih automata možemo koristiti pisana matematička slova $\mathcal{A}, \mathcal{B}, \dots$

Primer 3.7 (N)KA iz primera 3.6 zadat sledećim grafom:



možemo zapisati kao uređenu petorku iz prethodne definicije na sledeći način:

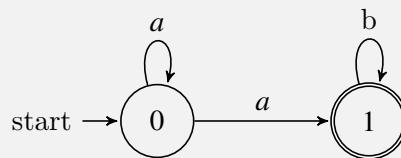
- $\Sigma = \{a, b\}$
- $Q = \{0, 1, 2\}$
- $I = \{0\}$
- $F = \{1, 2\}$
- $\Delta = \{(0, a, 1), (1, a, 1), (1, b, 2), (2, b, 2)\}$.

Na ovom mestu ćemo reći da se (N)KA može predstaviti i tabelarno na sledeći način:

$Q \setminus \Sigma$	a	b
0	1	-
1	1	2
2	-	2

Posmatrajmo primer 3.7 malo detaljnije. Ako pokušamo da utvrdimo neka svojstva skupa Δ , videćemo da za svaki skup uređenih parova prvih i drugih koordinata imamo različite slike trećih koordinata. U tabelarnoj reprezentaciji to se vidi time što je u svakom polju tabele tačno jedno stanje. Ukoliko je ovo ispunjeno za neki (N)KA, onda kažemo da se relacija Δ naziva **i funkcija prelaska automata**, označava se simbolom δ i predstavlja preslikavanje $\delta : (Q \times \Sigma) \rightarrow Q$.

Primer 3.8 (N)KA iz primera 3.4 predstavljen grafom



može se tabelarno predstaviti na sledeći način:

$Q \setminus \Sigma$	a	b
0	0, 1	-
1	-	1

a relacija prelaska datog automata predstavlja sledeći skup:

$$\Delta = \{(0, a, 0), (0, a, 1), (1, b, 1)\}.$$

U primeru 3.8 vidimo da se iz stanja 0 može granom a preći ili u stanje 0 ili u stanje 1. Ovo možemo da posmatramo i na drugi način: ako relaciju prelaska automata Δ posmatramo kao funkciju prelaska automata δ , onda original $(0, a)$ ima dve različite slike 0 i 1. Tada nije teško uočiti da je u opštem slučaju funkcija δ definisana sa $\delta : Q \times \Sigma \rightarrow P(Q)$, gde je $P(Q)$ označen partitivni skup skupa Q .

3.1.2 Uslovi za deterministički konačni automat (DKA)

Napomenuli smo da su u okviru definicije (N)KA opisani i DKA. Na ovom mestu ćemo navesti uslove koji određuju determinizam jednog konačnog automata.

Definicija 3.1.2 Neka je $(\Sigma, Q, I, F, \Delta)$ jedan (N)KA. Tada se taj (N)KA naziva **deterministički konačni automat**, i označava se DKA, ukoliko je svaki od sledeća tri uslova ispunjen:

1. Nisu dopušteni ε -prelazi, tj. za sve $p, q \in Q$ važi:

$$(p, \varepsilon, q) \notin \Delta.$$

2. Svi prelazi su jednoznačni, tj. za sve $p, q_1, q_2 \in Q$ i $a \in \Sigma$ važi:

$$((p, a, q_1) \in \Delta \wedge (p, a, q_2) \in \Delta) \implies q_1 = q_2.$$

3. Postoji tačno jedno početno stanje, tj.

$$|I| = 1,$$

gde je sa $|I|$ označen broj elemenata skupa I .

3.1.3 Jezik automata

Neka je petorkom $(\Sigma, Q, I, F, \Delta)$ zadat jedan (N)KA. Radi lakšeg uvodenja pojma jezika automata, uvešćemo prvo sledeće dve oznake:

1. Oznaka

$$p \xrightarrow{a} q$$

ekvivalentna je

$$(p, a, q) \in \Delta$$

za sve $p, q \in Q$ i $a \in \Sigma$.

2. Neka je $w = a_1 a_2 \dots a_n \in \Sigma^*$. Tada je oznaka

$$p \xrightarrow{w} q$$

ekvivalentna

$$(\exists r_1, r_2, \dots, r_{n+1} \in Q) : p = r_1 \xrightarrow{a_1} r_2 \xrightarrow{a_2} r_3 \xrightarrow{a_3} \dots \xrightarrow{a_n} r_{n+1} = q$$

za sve $p, q \in Q$.

Oznaka $p \xrightarrow{w} q$ može se protumačiti sledećim rečenicama: „Preko reči w se iz stanja p može stići u stanje q “ ili „Postoji put w iz stanja p u stanje q “.

Definicija 3.1.3 Jezik automata \mathcal{A} , u oznaci $L(\mathcal{A})$, predstavlja sledeći skup:

$$L(\mathcal{A}) = \{w \mid (\exists p \in I \wedge \exists q \in F) : p \xrightarrow{w} q\}.$$

Ova definicija se kraće (i neformalno) može opisati sledećom rečenicom: „Da bi neka reč w pripala jeziku automata \mathcal{A} , dovoljno je pronaći bilo koji put w od početnog do završnog stanja automata \mathcal{A} “.

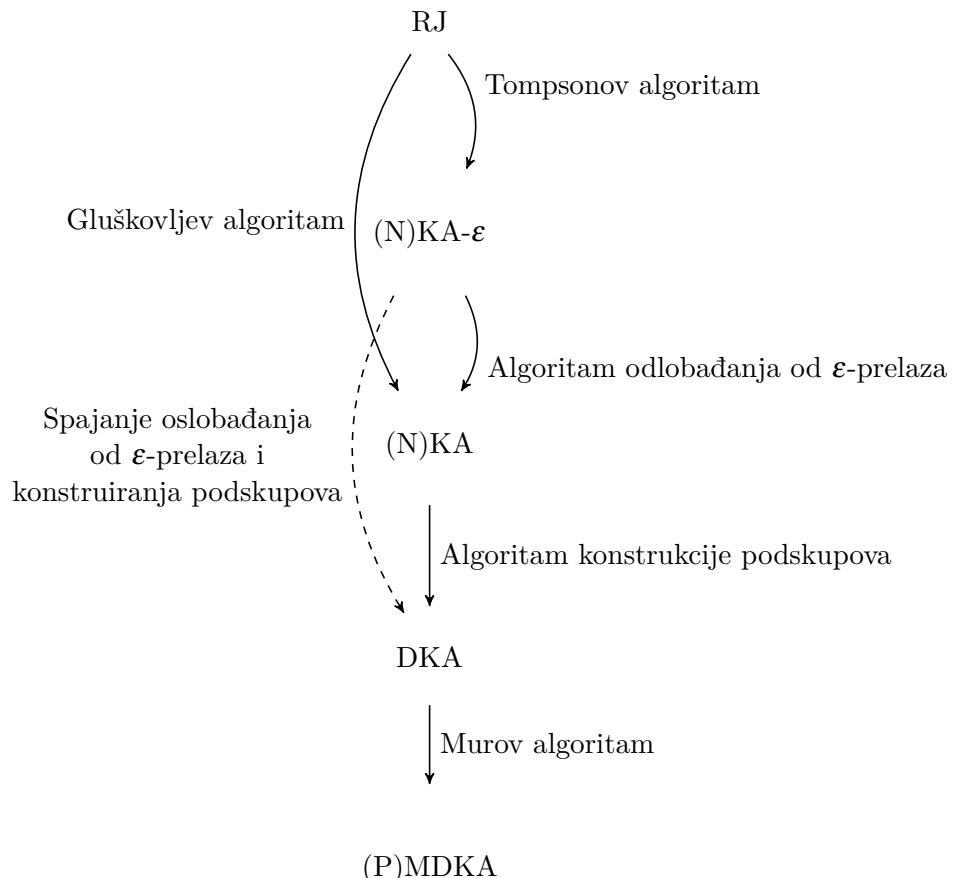
Za DKA važi da postoji najviše jedan put iz početnog stanja u automatu po svakoj reči w , a ako je automat još i potpun, tada za svaku reč w postoji tačno jedan put iz početnog stanja. Ako se taj put završava u završnom stanju, onda reč w pripada jeziku automata.

3.2 Klinijeva teorema

Teorema 3.2.1 — Klini. Klasa regularnih jezika se poklapa sa klasom jezika koji prepoznaju konačni automati.

Dokaz. Teoremu ćemo dokazati konstruktivno – navešćemo algoritme i primenjivati ih na primerima.

(Smer \Rightarrow):



(Smer \Leftarrow): Od proizvoljnog konačnog automata se primenom **Metoda eliminisanja stanja** može dobiti regularni izraz. ■

Teorema 3.2.2 Za dati regularni izraz postoji jedinstven PMDKA (potpuni minimalni deterministički konačni automat) koji prepoznaje odgovarajući regularni jezik. Dokaz

ove teoreme može se pronaći u Vitas 2006.

3.2.1 Tompsonov algoritam

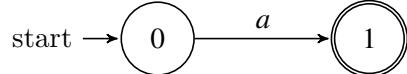
Definicija 3.2.1 Za (N)KA- ϵ kažemo da je **normalizovan** ako ima:

1. jedinstveno početno stanje u koje ne ulazi ni jedna grana,
2. jedinstveno završno stanje iz kojeg ne izlazi ni jedna grana, i
3. iz svakog stanja izlazi ili jedna grana označena simbolom iz azbuke ili najviše dve grane obeležene sa ϵ .

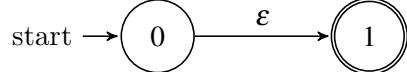
Jedno od najznačajnijih osobina Tompsonovih (N)KA- ϵ jeste ta da su svi normalizovani. Nije loše primetiti da ako regularni izraz ima r operacija, (N)KA- ϵ koji se dobija kao rezultat Tompsonovog algoritma ima najviše $2r$ stanja.

Tompsonov algoritam prati rekurzivnu definiciju regularnih izraza:

1. Ako je $a \in \Sigma$, automat koji odgovara regularnom izrazu a koji opisuje jezik $\{a\}$ je:



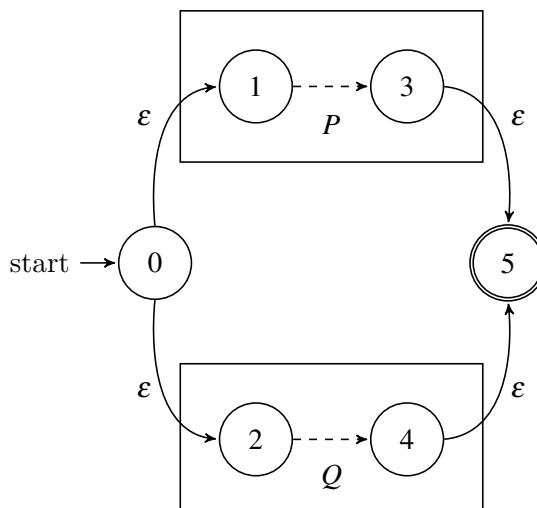
2. Automat koji odgovara regularnom izrazu ϵ koji opisuje jezik $\{\epsilon\}$ je:



3. Automat koji odgovara regularnom izrazu \emptyset koji opisuje jezik \emptyset je:



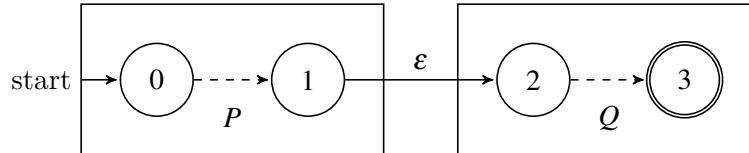
4. Neka su dati regularni izrazi p i q i njihovi odgovarajući automati P i Q , redom. Želimo da konstruišemo (N)KA- ϵ koji odgovara regularnom izrazu $p|q$. Dodajemo novo početno stanje (stanje 0) i završno stanje (stanje 5) i ϵ -prelazima ih povezujemo sa početnim i završnim stanjima automata P (stanja 1 i 3, redom) i automata Q (stanja 2 i 4, redom). Početno stanje automata P prestaje da bude početno stanje i završno stanje automata P prestaje da bude završno stanje. Isto važi i za automat Q .



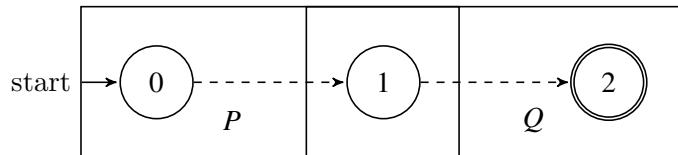
5. Neka su dati regularni izrazi p i q i njihovi odgovarajući automati P i Q , redom. Želimo da konstruišemo (N)KA- ϵ koji odgovara regularnom izrazu $p \cdot q$ (odnosno,

pq). Ovo možemo uraditi na dva načina:

- (a) Prvi način je da ϵ -prelazom povežemo završno stanje automata P (stanje 1) i početno stanje automata Q (stanje 2).

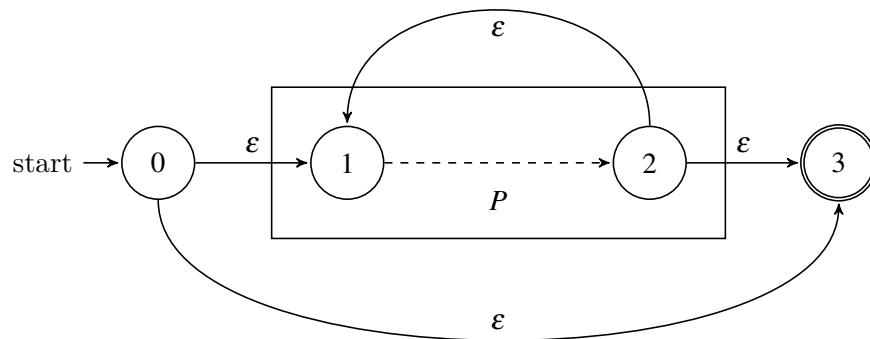


- (b) Drugi način je da spojimo završno stanje automata P i početno stanje automata Q (time dobijamo jedinstveno stanje 1).

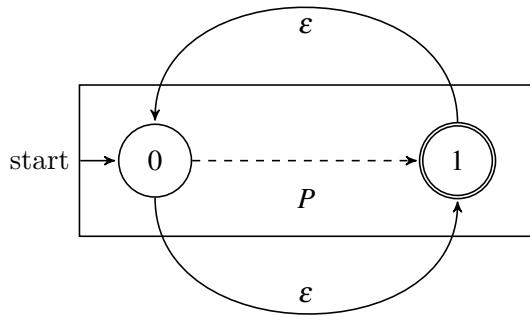


U oba slučaja je početno stanje (N)KA- ϵ koji odgovara regularnom izrazu $p \cdot q$ zapravo početno stanje automata P (stanje 0, u oba slučaja), dok je njegovo završno stanje zapravo završno stanje automata Q (stanje 3 u prvom slučaju, a stanje 2 u drugom slučaju).

6. Neka je dat regularni izraz p i njegov odgovarajući automat P . Želimo da konstruišemo (N)KA- ϵ koji odgovara regularnom izrazu p^* . Dodajemo dva nova stanja, početno (stanje 0) i završno (stanje 3) i ϵ -prelazima spajamo novo početno stanje sa početnim stanjem automata P (stanje 1) i spajamo završno stanje automata P (stanje 2) sa novim završnim stanjem. Takođe, dodajemo ϵ -prelaze od završnog do početnog stanje automata P , kao i ϵ -prelaz od novog početnog stanja do novog završnog stanja. Početno stanje automata P prestaje da bude početno stanje i završno stanje automata P prestaje da bude završno stanje.

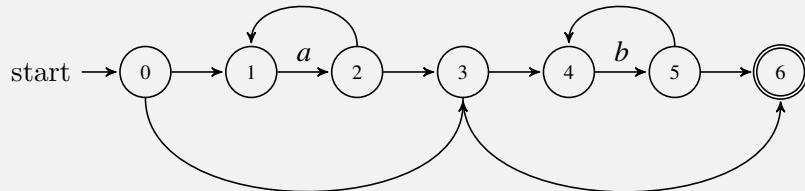


Na prvi pogled deluje da smo mogli da konstruišemo (N)KA- ϵ koji odgovara jeziku p^* i na kraći način: Dodajemo samo dva ϵ -prelaza koji povezuju početno stanje automata P sa završnim stanjem automata P i obrnuto.

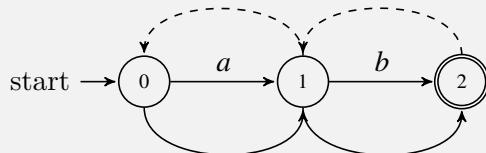


Druga konstrukcija takođe opisuje p^* , ali ne smemo da je koristimo, iako je štedljivije. Ova konstrukcija narušava definiciju normalizovanih automata jer iz završnog stanja postoji izlazna grana. Preporuka je da kod Klinijevog zatvorenja ne treba štedeti sa dodavanjem prelaza. Naredna dva primera nam ilustruju i zašto.

Primer 3.9 Konstruisati (N)KA- ϵ koji odgovara regularnom izrazu a^*b^* koristeći ispravan način konstruisanja p^* .



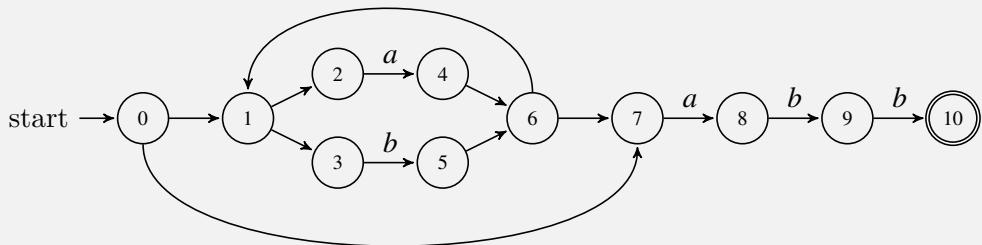
Primer 3.10 Konstruisati (N)KA- ϵ koji odgovara regularnom izrazu a^*b^* koristeći neispravan način konstruisanja p^* .



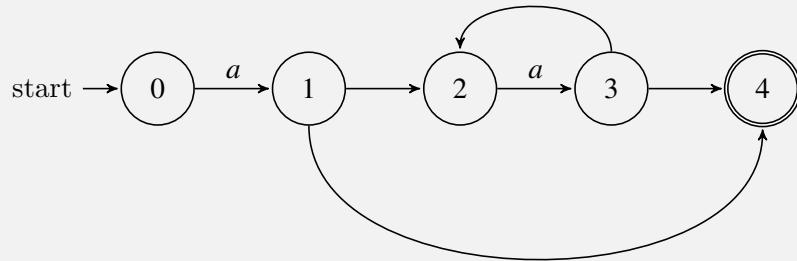
Primetimo da ovaj automat ima put od završnog stanja do početnog (to je put označen isprekidanim prelazima). Zbog toga, on je u stanju da prihvata i reči u kojima se a pojavljuje posle b , a takve reči nisu obuhvaćene jezikom a^*b^* .

Primetimo da u primerima 3.9 i 3.10 nismo eksplisitno obeležavali ϵ -prelaze. Nadalje ćemo se pridržavati ove konvencije i smatrati svaki neoznačen prelaz za ϵ -prelaz. Pogledajmo sada još neke primere koji ilustruju primenu Tompsonovog algoritma.

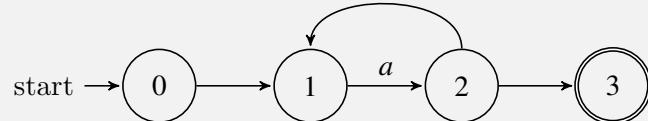
Primer 3.11 Konstruisati automat koji odgovara regularnom jeziku $(a|b)^*abb$.



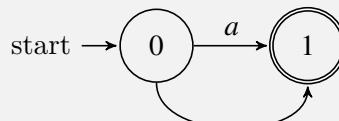
Primer 3.12 Konstruisati automat koji odgovara regularnom jeziku a^+ . Koristićemo činjenicu koju znamo od ranije da je $a^+ \equiv aa^*$.



Mada je potpuno korektan, prethodni (N)KA- ϵ koji odgovara regularnom jeziku a^+ se može konstruisati i efikasnije, jednostavno uklanjanjem ϵ -prelaza od početnog do završnog stanja (N)KA- ϵ koji odgovara regularnom jeziku a^* .



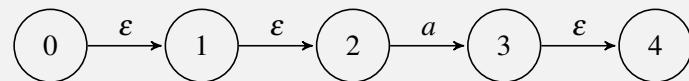
Primer 3.13 Konstruisati automat koji odgovara regularnom jeziku $a^?$.



3.2.2 Algoritam oslobođanja od ϵ -prelaza

Prvo ćemo prikazati postupak u opštem slučaju, a potom ćemo fokus staviti na oslobođanje od ϵ -prelaza Tompsonovih (N)KA- ϵ od ostalih koje je znatno jednostavnije baš zbog njihove osobine da su normalizovani. Ideja je pronaći sva stanja iz kojih se prelazi u drugo stanje po ϵ i eliminisati ih dodavanjem novih grana po odgovarajućim slovima azbuke.

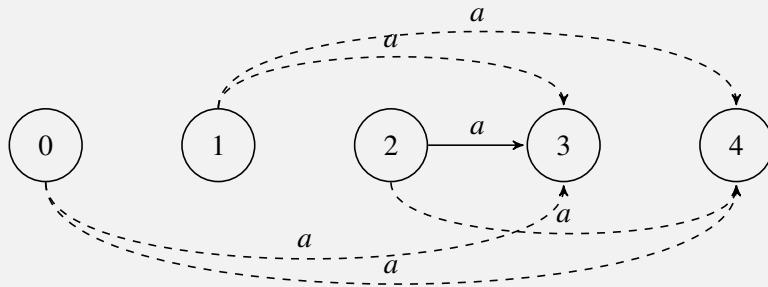
Primer 3.14 Neka je dat sledeći (N)KA- ϵ :



Postupak oslobođanja od ϵ -prelaza je sledeći:

- K1: Dodavanje prelaza a iz stanja 0 u stanje 3
- K2: Dodavanje prelaza a iz stanja 1 u stanje 3
- K3: Dodavanje prelaza a iz stanja 0 u stanje 4
- K4: Dodavanje prelaza a iz stanja 1 u stanje 4
- K5: Dodavanje prelaza a iz stanja 2 u stanje 4

Nakon dodavanja potrebnih prelaza, sledi brisanje svih ϵ -prelaza. Rezultujući automat je sledeći:



Isprekidanom linijom su označeni dodavani prelazi.

Formalno gledano, prvo smo posmatrali sva stanja p, q, q' i r i sve prelaze a za koje važi:

$$p \xrightarrow{\epsilon} q \xrightarrow{a} q' \xrightarrow{\epsilon} r,$$

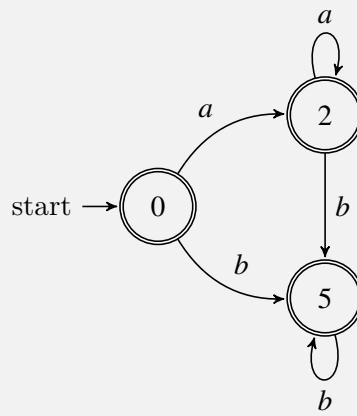
i dodavali smo putanju (p, a, r) u Δ . Potom smo eliminisali sve putanje oblika (p, ϵ, q) iz Δ . Time smo dobili (N)KA bez ϵ -prelaza.

Pri eliminaciji ϵ -prelaza, u opštem slučaju se pronalaze sve putanje koje imaju ϵ -prelaz, pa prelaz po nekom slovu, pa zatim ponovo ϵ -prelaz. Kod (N)KA- ϵ koji se dobijaju kao rezultat Tompsonovog algoritma situacija se menja. Naime, dovoljno je prespojiti putanje sledećeg oblika:

$$p \xrightarrow{\epsilon} q \xrightarrow{a} r.$$

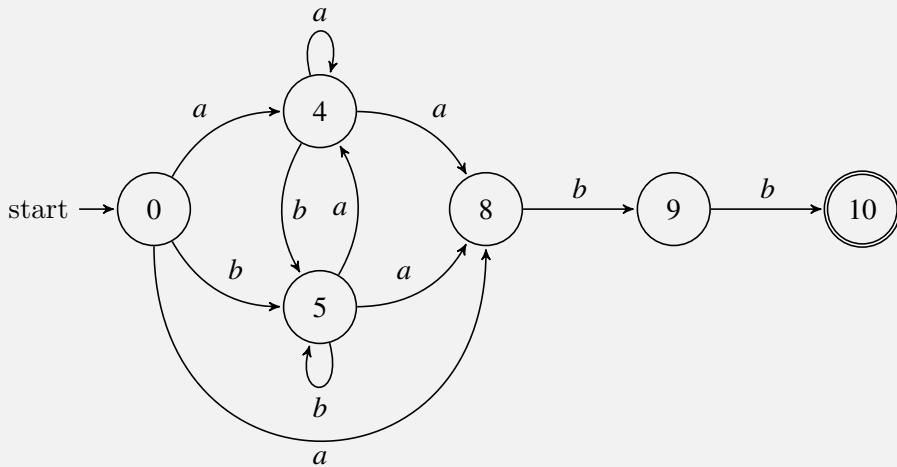
Pokažimo ovo sledećim primerom:

Primer 3.15 Oslobođanjem ϵ -prelaza u (N)KA iz primera 3.9 dobija se sledeći (N)KA:



Možda nije sasvim jasno zašto su sva tri stanja iz primera 3.15 završna. Ispostavlja se da važi sledeće: Za završno stanje $(N)KA$ proglašavamo sva stanja u čijem se epsilon zatvorenju nalazi završno stanje početnog $(N)KA-\epsilon$. Pokažimo da ovo važi na još jednom primeru.

Primer 3.16 Oslobođanjem ϵ -prelaza u $(N)KA$ iz primera 3.11 dobija se sledeći $(N)KA$:

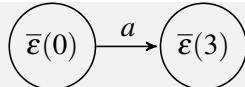


Ovo nije jedini način za eliminaciju ϵ zatvorenja. Kao stanja novog automata mogu da se uzmu ϵ -zatvorenja stanja polaznog automata. U tom slučaju potrebno je gledati za svako zatvorenje $\epsilon(i)$ sve neposredne prelaze po nekom simbolu, a iz svih stanja koja su u $\epsilon(i)$ povući odgovarajuće prelaze ka odgovarajućim ϵ -zatvorenjima. U tom slučaju se razmatraju samo ϵ -prelazi pre konkretnog prelaza po slovu azbuke, a ne i oni posle tog prelaza (oni su apstrahovani time što smo za stanja uzeli ϵ -zatvorenja).

Definicija 3.2.2 Epsilon zatvorenje stanja p , u označi $\bar{\epsilon}(p)$, predstavlja sledeći skup:

$$\bar{\epsilon}(p) = \{q \mid p \xrightarrow{\epsilon} q\}.$$

Primer 3.17 Epsilon zatvorenja stanja iz primera 3.14 su: $\bar{\epsilon}(0) = \{0, 1, 2\}$, $\bar{\epsilon}(1) = \{1, 2\}$, $\bar{\epsilon}(2) = \{2\}$ i $\bar{\epsilon}(3) = \{3, 4\}$. Konstruišemo novi automat čija su stanja upravo ovi prelazi:



Kao što smo videli, ukoliko nam je dat regularni jezik, primenom Tompsonovog algoritma, pa zatim algoritma oslobađanja od ϵ -prelaza, možemo dobiti (N)KA. Postoji još jedan algoritam koji neposredno izvodi (N)KA iz zadatog regularnog izraza koji se naziva Gluškovljev algoritam. Iako nama jednostavniji i brži za rad, ispostavlja se da je do sada opisani metod (primena prethodna dva algoritma) lakši za automatizaciju i implementaciju u računaru. Ipak, prikazaćemo i Gluškovljev algoritam u narednoj sekciji.

3.2.3 Gluškovljev algoritam

Konstrukcija automata Gluškovljevim algoritmom sastoji se iz 3 koraka:

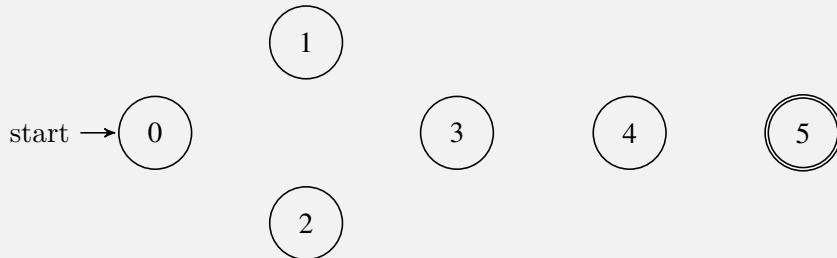
1. Sva slova u regularnom uzrazu numerošu se brojevima od 1 do n , pri čemu je n broj karaktera koji se pojavljuju u regularnom izrazu.
2. Svakom broju dodeljuje se po jedno stanje automata. Dodaje se i stanje 0 koje predstavlja početno stanje. Završna stanja su sva stanja koja odgovaraju indeksima slova kojima se može završiti reč jezika. U slučaju da je prazna reč deojezika, onda i stanje 0 može biti završno stanje.
3. Granu koja vodi os stanja i do stanja j obeležavamo slovom na poziciji j ako se $x_i x_j$ može pojaviti bar u jednoj reči jezika. Takođe, potrebno je povezati početno stanje sa svim stanjima koja predstavljaju indeks početnog slova neke reči jezika.

Primer 3.18 Konstruisati automat koji odgovara regularnom izrazu $(a|b)^*abb$.

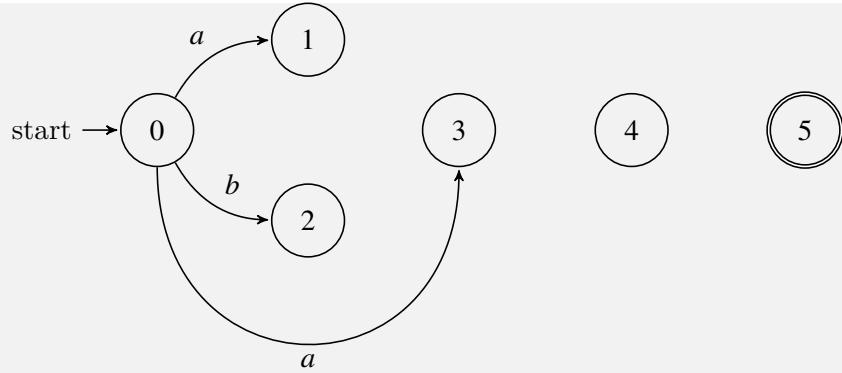
Prvo, obeležimo svako slovo odgovarajućim brojem.

		(a			b)		*		a		b		b
0				1			2						3		4		5

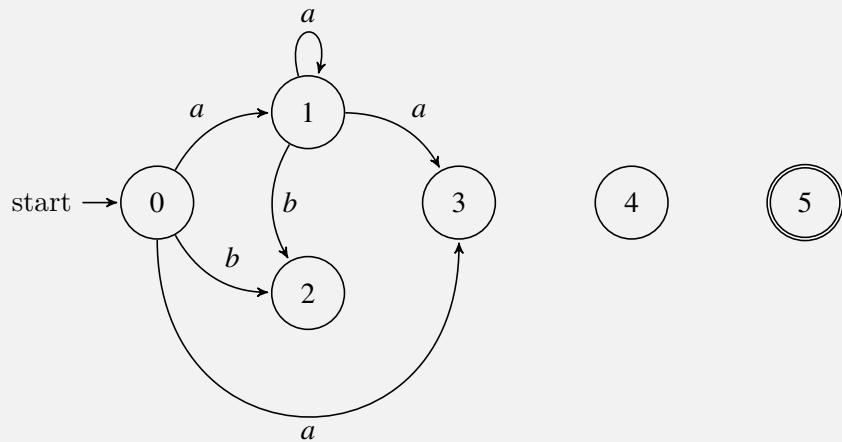
Zatim pravimo stana za svaki broj.



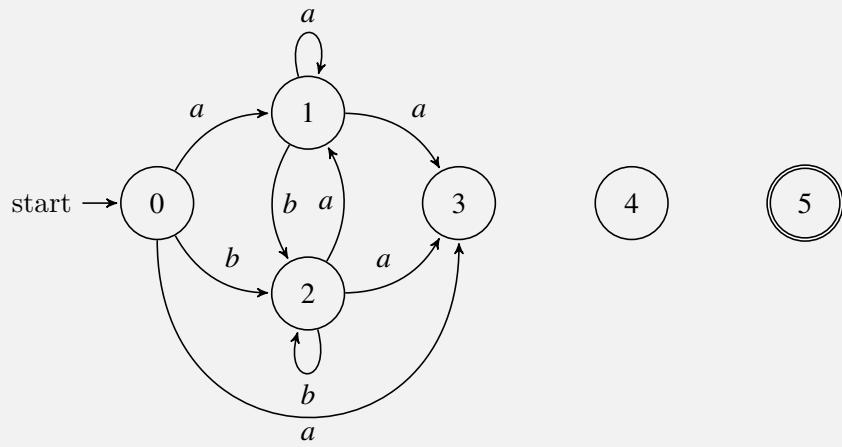
Sada treba dodati odgovarajuće grane. Kao početno slovo mogu se pojaviti slovo a na poziciji 1, slovo b na poziciji 2 ili slovo a na poziciji. Zbog toga dodajemo prelaze iz stanja 0 u stanje 1 i 3 po slovu a , i u stanje 2 po slovu b :



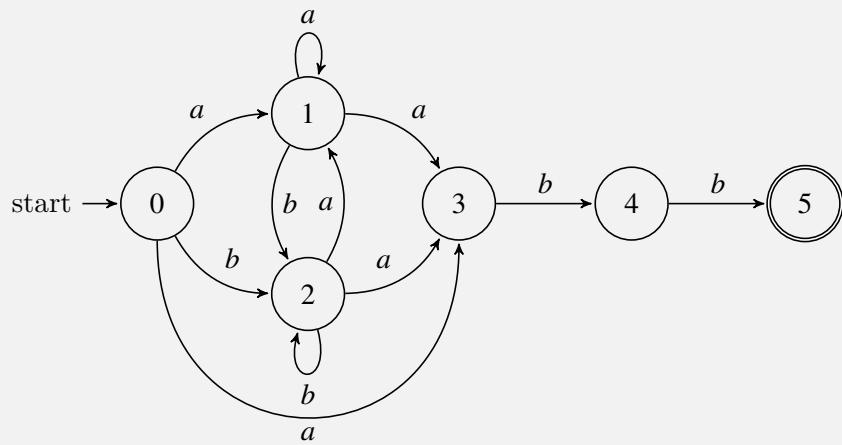
Dalje, iz stanja 1 može se preći u stanje 1 ili u stanje 3, ako je naredno slovo a , u stanje 2, ako je naredno slovo b



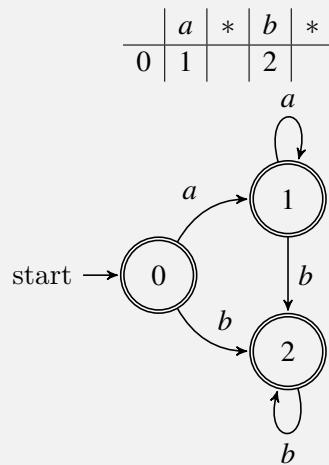
Iz stanja 2 prelazi se u stanje 1 ili 3, ako je naredno slovo a , ili se ostaje u stanju 2, ako je naredno slovo b :



Kada smo došli do stanja 3, put do poslednjeg stanja je pravolinijski jer reč mora da se završi sa abb . Zbog toga, iz stanja 3 u stanje 4 prelazi se po karakteru b , i iz stanja 4 u stanje 5 takođe po karakteru b .



Primer 3.19 Konstruisati automat koji odgovara regularnom izrazu a^*b^* .



3.2.4 Algoritam konstrukcije podskupova

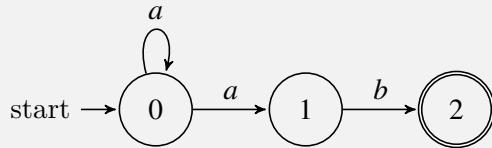
Algoritam konstrukcije podskupova nam omogućava da od (N)KA dobijemo DKA. Postupak razlikuje nekoliko koraka:

1. Konstruisati (N)KA koji odgovara zadatom regularnom izrazu.
2. Napraviti tabelu prelazaka dobijenog (N)KA.
3. Konstruisati DKA prema sledećem uputstvu:
 - Na početku, skupovi Q , I , F i Δ iz DKA su prazni skupovi.
 - Stanja DKA biće podskupovi stanja (N)KA.
 - Stanja koja su bila početna u (N)KA stavimo u jedan podskup i označimo taj podskup kao početno stanje DKA. Prema tablici prelaska određuje se podskup stanja u koje se može preći iz početnog stanja preko odgovarajućeg slova, tako da se po jednom slovu prelazi u jedinstveno stanje.
 - Postupak se nastavlja sa novim podskupovima dok se ne obrade svi.
 - U slučaju da se iz nekog stanja preko nekog karaktera ne može stići ni u jedno drugo stanje, podskup je prazan skup stanja koje predstavlja stanje greške. Jednom kad se stigne u to stanje, ne može se vratiti u neko validno stanje.
 - Stanja DKA koja sadrže završna stanja (N)KA označavamo završnim.

- Broj stanja u najgorem slučaju iznosi 2^n , ali takvi slučajevi su retki.

Primer 3.20 Determinizovati automat koji odgovara izrazu $a+b$.

Korak 1. (N)KA koji odgovara zadatom regularnom izrazu je:



Korak 2. Tabela prelazaka dobijenog (N)KA:

	a	b
$\rightarrow 0$	0, 1	—
1	—	2
(2)	—	—

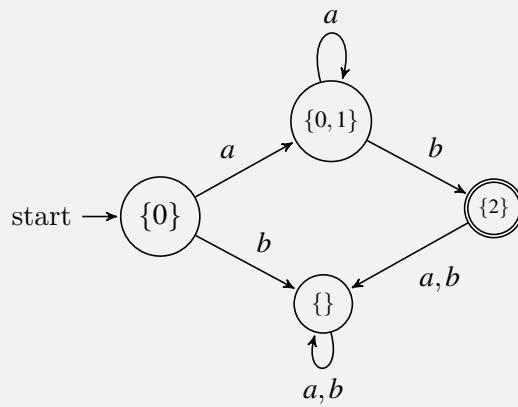
Napomena: Strelicom su obeležena početna stanja, a zaokružena stanja predstavljaju završna stanja.

Korak 3. Kako je u ovom slučaju jedino stanje 0 bilo početno u (N)KA, onda je traženo početno stanje (tj. podskup koji predstavlja početno stanje) DKA podskup $\{0\}$ i označimo ga q_1 , tj. $q_1 \in Q$ i $q_1 \in I$.

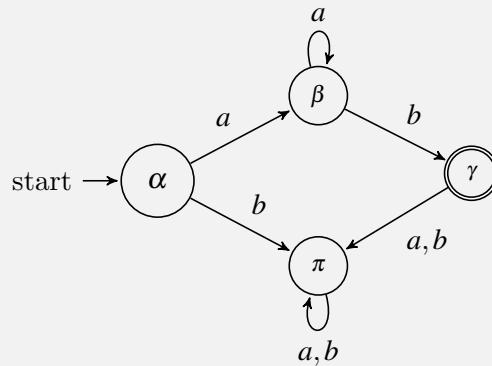
Kada pravimo novi podskup, mi zapravo gledamo svaki element podskupa od kojeg krećemo (ti elementi su stanja u (N)KA) i pravimo novi podskup tako što dodajemo sva stanja do kojih se može stići u (N)KA:

- Iz stanja q_1 :
 - preko karaktera a, može se stići do stanja 0 ili 1, te je $\{0, 1\} = q_2 \in Q$ i $(q_1, a, q_2) \in \Delta$
 - preko karaktera b, nema stanja u koje se može stići te je podskup prazan, tj. $\{\} = q_3 \in Q$ i $(q_1, b, q_3) \in \Delta$
- Iz stanja q_2 može se stići:
 - preko karaktera a, do stanja 0 ili 1, što je stanje q_2
 - preko karaktera b, do stanja 2, te je $\{2\} = q_4 \in Q$ i $(q_2, b, q_4) \in \Delta$
- Iz stanja q_3 je stanje greške i za svaki karakter važi da vodi nazad do ovog stanja, te je $(q_3, a, q_3) \in \Delta$ i $(q_3, b, q_3) \in \Delta$
- Iz stanja q_4 završno stanje i za svaki karakter važi da vodi do stanja greške, tj. do $\{\}$, te je $(q_4, a, q_3) \in \Delta$ i $(q_4, b, q_3) \in \Delta$

Ovo nam daje sledeći DKA:



Stanja DKA koja sadrže završna stanja (N)KA označavamo završnim. U ovom slučaju, to je stanje q_4 jer sadrži stanje 2 koje je bilo završno u (N)KA. Naravno, stanja ne moramo obeležavati skupovima. Umesto toga možemo ih preimenovati, na primer, slovima grčkog alfabetu:



Definicija 3.2.3 Neka je zadat (N)KA koji je predstavljen petorkom $\mathcal{A} = (\Sigma, Q, I, F, \Delta)$. Tada je odgovarajući DKA predstavljen petorkom $\mathcal{D} = (\Sigma, P(Q), i, F', \delta)$, gde je:

- $P(Q)$ partitivni skup skupa Q ,
- $i = I$,
- $F' = \{q \in P(Q) \mid q \cap F \neq \emptyset\}$,
- $\delta(q, a) = \{p' \in Q \mid (\exists p \in q) : (p, a, p') \in \Delta\} = \bigcup_{p \in q} \{p' \in Q \mid (p, a, p') \in \Delta\}$

Pojasnimo δ sledećim primerom.

Primer 3.21 Za automat iz primera 3.22 važi:

	a	b
1	1, 3	2
2	...	
3	-	4
4	...	

Tablica (N)KA:

	a	b
1, 3	...	{2, 4}
2	{1, 3}	
3	{1, 3}	
4	...	

Tablica DKA:

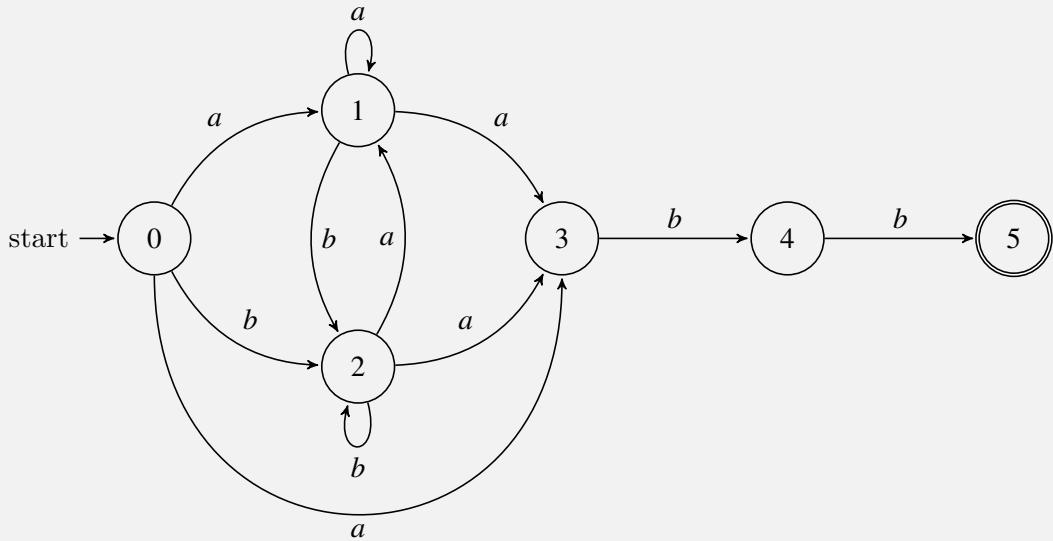
što se može zapisati:

$$\delta(q = \{p = 1, p = 3\}, a) = \{p' = 1, p' = 3\} \wedge \delta(q = \{p = 1, p = 3\}, b) = \{p' = 2, p' = 4\},$$

a to odgovara definiciji skupa δ .

Primer 3.22 Konstruisati DKA koji odgovara regularnom izrazu $(a|b)^*abb$.

Konstrukcija (N)KA:



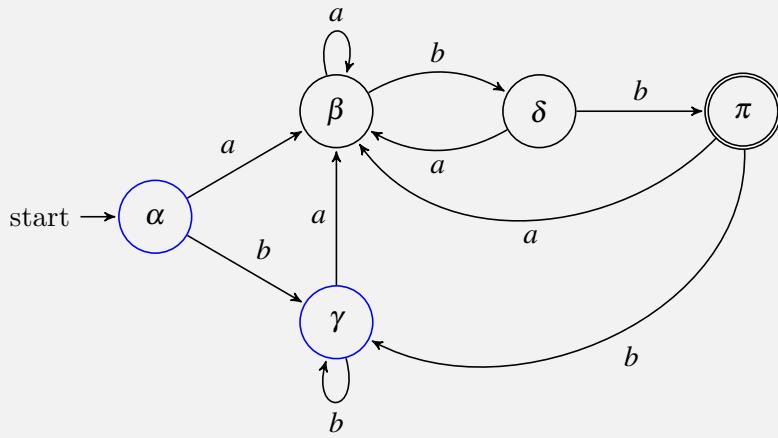
U ovom slučaju ćemo DKA prvo predstaviti tabelarno, pa zatim i grafički. Čitajući tablicu za (N)KA dobijamo tablicu za DKA.

Tablica (N)KA:

	a	b
$\rightarrow 0$	1, 3	2
1	1, 3	2
2	1, 3	2
3	—	4
4	—	5
(5)	—	—

	a	b
$\rightarrow \{0\}$	{1, 3}	{2}
{1, 3}	{1, 3}	{2, 4}
{2}	{1, 3}	{2}
{2, 4}	{1, 3}	{2, 5}
{2, 5}	{1, 3}	{2}

Imenujmo dobijene podskupove $\{0\}, \{1, 3\}, \{2\}, \{2, 4\}, \{2, 5\}$ slovima $\alpha, \beta, \gamma, \delta$ i π , redom. Čitajući tablicu prelaska DKA, dobijeni DKA možemo prikazati grafički:



Primetimo da ovaj automat nije minimizovan, tj. da postoje stanja sa identičnim prelascima, kao što su α i γ u ovom slučaju i ona se mogu spojiti. Minimizacijom automata ćemo se pozabaviti u narednom poglavljju.

Teorema 3.2.3 Za sve skupove stanja $q, q' \in Q_{DKA}$, sva stanja $p, p' \in Q_{(N)KA}$ i reč w važi:

$$\delta^*(q, w) = q' \iff q' = \{p' \in Q \mid (\exists p \in q) : p \xrightarrow{w} p'\}.$$

Ova teorema se neformalno može opisati rečenicom: „U DKA, može se krenuti iz nekog skupa stanja q i čitanjem neke reči w doći u neki skup stanja q' akko je q' skup svih stanja (N)KA do kojih se može doći čitajući reč w “.

Daćemo primer ove teoreme, ali pre toga uvedimo sledeću oznaku:

$$p \xrightarrow{w} q,$$

koja označava da se krenuvši od *skupa* stanja p , čitajući reč w , može stići do *skupa* stanja q .

Primer 3.23 Iz primera 3.22 možemo uočiti da važi sledeće:

$$\{0\} \xrightarrow{abb} \{2, 5\},$$

tj. ako se u DKA krene iz stanja $\{0\}$ i pročita reč abb , stić će se do stanja $\{2, 5\}$. Teorema tvrdi da onda u (N)KA postoji stanje p (u ovom slučaju 0) i p' (2 i 5) tako da čitajući reč abb može se stići od p do p' . U NKA, čitajući reč abb iz stanja 0 može se stići u stanje 2 (to je put $0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 2$) ili čitajući istu reč se iz stanja 0 može stići u stanje 5 (to je put $0 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{b} 5$).

Slično važi:

$$\{1, 3\} \xrightarrow{ab} \{2, 4\}.$$

Dokaz teoreme 3.2.3 se izvodi indukcijom po dužini reči, ali on neće biti prikazan.

3.2.5 Murov algoritam

Pre samog algoritma minimalizacije DKA, uvešćemo nekoliko oznaka i definicija koje ćemo koristiti nadalje. Ključni pojam će biti ekvivalencija stanja.

U (N)KA, **jezik stanja** q , u označi L_q , sledeći je skup:

$$L_q = \{w \in \Sigma^* \mid (\exists q' \in F) : q \xrightarrow{w} q'\}.$$

Ako se podsetimo definicije jezika automata \mathcal{A} , onda možemo primetiti da se jezik automata \mathcal{A} zapravo može predstaviti kao unija jezika njegovih početnih stanja, tj.

$$L(\mathcal{A}) = \bigcup_{q \in I} L_q$$

Lema 3.2.4 Ako jezik stanja sadrži praznu reč, onda je to stanje završno pod uslovom da nema ϵ -prelaza.

Dokaz. Sledi direktno iz definicije jezika stanja za $w = \epsilon$. Pošto trenutno govorimo o DKA, onda je jasno da neće postojati ϵ -prelazi, te će ovo uvek biti ispunjeno. ■

Nakon uvođenja ovih označa, definisaćemo pojam ekvivalencije stanja.

Po definiciji **Nerodove ekvivalencije** stanje p je ekvivalentno stanju q AKKO su im jezici isti tj. ako su skupovi reči koji mogu da se prepozna polazeći iz stanja p i q jednaki. Pišemo:

$$p \sim q \stackrel{df}{\iff} L_p = L_q.$$

Primetimo da važi sledeće: Može se desiti da dva stanja imaju različite prelaze, a da su ekvivalentna. Nije potreban uslov da imaju iste prelaze, ali jeste dovoljan.

Lema 3.2.5 Nerodova ekvivalencija je relacija ekvivalencije.

Jezik stanja q za reči dužine najviše k , u označi $L_q^{(k)}$, sledeći je skup:

$$L_q^{(k)} = \{w \in \Sigma^* \mid (\exists q \in F) : p \xrightarrow{w} q \wedge |w| \leq k\}.$$

Uvodimo relaciju **Nerodove ekvivalencije za reči dužine najviše k** na sledeći način:

$$p \sim_k q \stackrel{df}{\iff} L_p^{(k)} = L_q^{(k)},$$

tj. dva stanja su ekvivalentna za reči dužine k , ali može postojati prelaz po nekom slovu tako da se stigne do različitih stanja (različitih u smislu da je jedno završno, a drugo nezavršno). Ilustrujmo ovo sledećim primerom:

Primer 3.24 1. Bez gubitka opštosti, pretpostavimo da je a takvo slovo i w takva reč za koje važi $|w| \leq k$, tj. $|aw| \leq k+1$ i važi:

$$p \xrightarrow{a} p' \xrightarrow{w} p'' \in F,$$

$$q \xrightarrow{a} q' \xrightarrow{w} q'' \notin F.$$

Vidimo da su stanja p' i q' neekvivalentna za reči dužine k , tj. $p' \not\sim_k q'$. Tada možemo zaključiti da je

$$p \not\sim_{k+1} q.$$

Zaključak: Dovoljno je da nađemo prelaz po jednom slovu da bismo utvrdili da se stanja razlikuju.

2. Međutim, da bismo utvrdili da su stanja ekvivalentna, moramo proveriti prelaze po svim slovima azbuke:

Za svako $a \in \Sigma$:

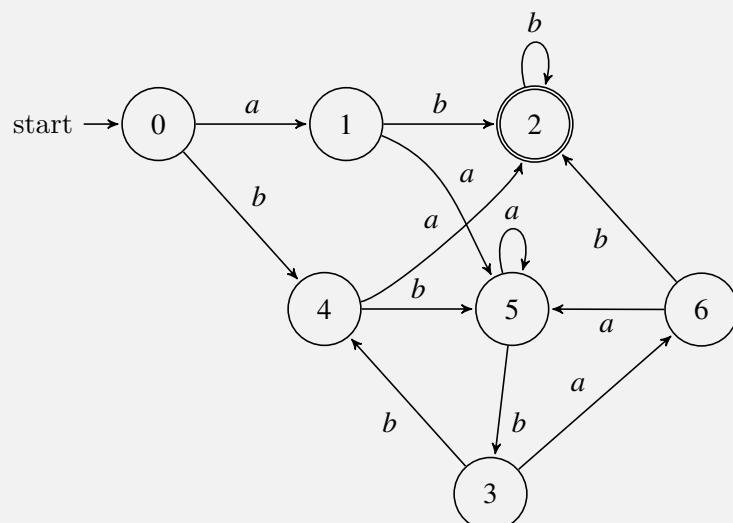
$$\text{ako } p \xrightarrow{a} p' \wedge q \xrightarrow{a} q' \wedge p' \sim_k q', \text{ onda } p \sim_{k+1} q.$$

Neka je \mathcal{A} potpuni deterministički konačan automat koji prepoznaće neki jezik L . Stanja minimalnog automata $\mathcal{B} = \mathcal{A}/\sim$ su klase ekvivalencije po relaciji \sim koje nastaju štapanjem stanja polaznog automata \mathcal{A} . Dva stanja p i q su razlikujuća ako je jedno od stanja završno, a drugo nije, ili ako postoji slovo $a \in \Sigma$ tavo da su stanja $\delta(p, a)$ i $\delta(q, a)$ razlikujuća. U prvom koraku stanja se dele na skup završnih i skup nezavršnih stanja. Zatim se razmatra da li postoji slovo $a \in \Sigma$ koje razlikuje neka dva stanja u konstruisanim skupovima stanja. Skupovi se na kraju razlažu na skupove koji sadrže samo nerazlikujuća stanja.

Sada ćemo dati primer koji detaljno ilustruje Murov algoritam aproksimiranja Nerodove ekvivalencije pomoću Nerodovih ekvivalencija za reči dužine najviše k , za sve $k = \overline{1, n}$, pri čemu je $n \in \mathbb{N}$ takav broj da važi $\sim_{n-1} \equiv \sim_n$. Nakon što nađemo traženo n , dolazimo do zaključka:

$$\sim \equiv \sim_n.$$

Primer 3.25 Minimizovati sledeći DKA:



Murov algoritam:

$k = 0$: Posmatramo relaciju \sim_0 i tražimo njene klase ekvivalencije:

To su stanja koja se razlikuju za reči dužine 0. Kako čitanjem prazne reči ostajemo u istom stanju, u prvom koraku uvek imamo tačno dve klase ekvivalencije, jednu koja sadrži završna i drugu koja sadrži nezavršna stanja. Dakle, klase ekvivalencije posmatrane relacije su: $\{0, 1, 4, 5, 6, 3\}$ i $\{2\}$.

$k = 1$: Posmatramo relaciju \sim_1 i tražimo njene klase ekvivalencije:

Razmatramo stanja unutar klase ekvivalencije prethodno posmatrane relacije. Kako je stanje 2 jedino u svojoj klasi, njega ne razmatramo, ono ostaje tako kako je.

- razmatramo stanja 0 i 1:
 - a) $0 \xrightarrow{a} 1 \wedge 1 \xrightarrow{a} 5$.

Pošto su stanja 1 i 5 u istoj klasi ekvivalencije po dužini 0, ona su ekvivalentna po 0, tj. važi $1 \sim_0 5$. Pošto smo naišli na ekvivalenciju, moramo proveriti i za preostala slova azbuke automata kako bismo utvrdili ekvivalentiju stanja 0 i 1.

- b) $0 \xrightarrow{b} 4 \wedge 1 \xrightarrow{b} 2$.

Stanja 2 i 4 nisu u istoj klasi ekvivalencije po dužini 0, tj. važi $4 \not\sim_0 2$.

Iz ovoga zaključujemo: $0 \not\sim_1 1$.

- razmatramo stanja 0 i 4:

- a) $0 \xrightarrow{a} 1 \wedge 4 \xrightarrow{a} 2 \wedge 1 \not\sim_0 2$.

Iz ovoga zaključujemo: $0 \not\sim_1 4$.

- razmatramo stanja 0 i 5:

- a) $0 \xrightarrow{a} 1 \wedge 5 \xrightarrow{a} 5 \wedge 1 \sim_0 5$.
- b) $0 \xrightarrow{b} 4 \wedge 5 \xrightarrow{b} 3 \wedge 4 \sim_0 3$.

Pošto u azbuci imamo samo slova a i b , pokazali smo da su 0 i 5 ekvivalentna stanja po dužini 1, tj. važi $0 \sim_1 5$. Dalje ne moramo proveravati ekvivalentnost stanja 1 i 5 ili 4 i 5, zbog tranzitivnosti Nerodove ekvivalencije.

- razmatramo stanja 0 i 6:

- a) $0 \xrightarrow{a} 1 \wedge 6 \xrightarrow{a} 5 \wedge 1 \sim_0 5$.
- b) $0 \xrightarrow{b} 4 \wedge 6 \xrightarrow{b} 2 \wedge 4 \not\sim_0 2$.

Iz ovoga zaključujemo: $0 \not\sim_1 6$.

- razmatramo stanja 0 i 3:

- a) $0 \xrightarrow{a} 1 \wedge 3 \xrightarrow{a} 6 \wedge 1 \sim_0 6$.
- b) $0 \xrightarrow{b} 4 \wedge 3 \xrightarrow{b} 4 \wedge 4 \sim_0 4$.

Pošto u azbuci imamo samo slova a i b , pokazali smo da su 0 i 3 ekvivalentna stanja po dužini 1, tj. važi $0 \sim_1 3$.

- razmatramo stanja 1 i 4:

- a) $1 \xrightarrow{a} 5 \wedge 4 \xrightarrow{a} 2 \wedge 5 \sim_0 2$.

Iz ovoga zaključujemo: $1 \not\sim_1 4$.

- razmatramo stanja 1 i 6:

- a) $1 \xrightarrow{a} 5 \wedge 6 \xrightarrow{a} 5 \wedge 5 \sim_0 5$.
- b) $1 \xrightarrow{b} 2 \wedge 6 \xrightarrow{b} 2 \wedge 2 \sim_0 2$.

Pošto u azbuci imamo samo slova a i b , pokazali smo da su 1 i 6 ekvivalentna stanja po dužini 1, tj. važi $1 \sim_1 6$.

Klase ekvivalencije posmatrane relacije su: $\{0, 3, 5\}, \{1, 6\}, \{4\}, \{2\}$.

$k = 2$: Posmatramo relaciju \sim_2 i tražimo njene klase ekvivalencije:

Razmatramo stanja unutar klase ekvivalencije prethodno posmatrane relacije.

- razmatramo stanja 0 i 5:
 - $0 \xrightarrow{a} 1 \wedge 5 \xrightarrow{a} 5 \wedge 1 \not\sim_1 5$.
Iz ovoga zaključujemo: $0 \not\sim_2 5$.
 - $0 \xrightarrow{a} 1 \wedge 3 \xrightarrow{a} 6 \wedge 1 \sim_1 6$.
 - $0 \xrightarrow{b} 4 \wedge 3 \xrightarrow{b} 4 \wedge 4 \sim_1 4$.

Pošto u abecedi imamo samo slova a i b , pokazali smo da su 0 i 3 ekvivalentna stanja po dužini 2, tj. važi $0 \sim_2 3$.

- razmatramo stanja 1 i 6:
 - $1 \xrightarrow{a} 5 \wedge 6 \xrightarrow{a} 5 \wedge 5 \sim_1 5$.
 - $1 \xrightarrow{b} 2 \wedge 6 \xrightarrow{b} 2 \wedge 2 \sim_1 2$.

Pošto u abecedi imamo samo slova a i b , pokazali smo da su 1 i 6 ekvivalentna stanja po dužini 2, tj. važi $1 \sim_2 6$.

Klase ekvivalencije posmatrane relacije su: $\{0, 3\}, \{1, 6\}, \{5\}, \{4\}, \{2\}$.

$k = 3$: Posmatramo relaciju \sim_3 i tražimo njene klase ekvivalencije:

Razmatramo stanja unutar klase ekvivalencije prethodno posmatrane relacije.

- razmatramo stanja 0 i 3:
 - $0 \xrightarrow{a} 1 \wedge 3 \xrightarrow{a} 6 \wedge 1 \sim_2 6$.
 - $0 \xrightarrow{b} 4 \wedge 3 \xrightarrow{b} 4 \wedge 4 \sim_2 4$.

Pošto u abecedi imamo samo slova a i b , pokazali smo da su 0 i 3 ekvivalentna stanja po dužini 3, tj. važi $0 \sim_3 3$.

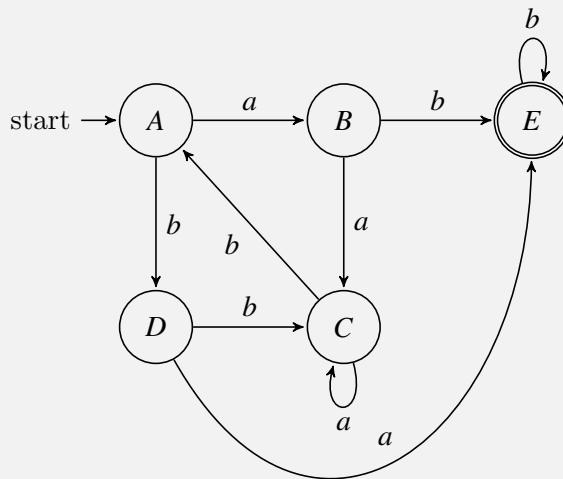
- razmatramo stanja 1 i 6:
 - $1 \xrightarrow{a} 5 \wedge 6 \xrightarrow{a} 5 \wedge 5 \sim_2 5$.
 - $1 \xrightarrow{b} 2 \wedge 6 \xrightarrow{b} 2 \wedge 2 \sim_2 2$.

Pošto u abecedi imamo samo slova a i b , pokazali smo da su 1 i 6 ekvivalentna stanja po dužini 3, tj. važi $1 \sim_3 6$.

Klase ekvivalencije posmatrane relacije su: $\{0, 3\}, \{1, 6\}, \{5\}, \{4\}, \{2\}$.

Vidimo da se klase ekvivalencije \sim_2 i \sim_3 ne razlikuju i tu stajemo sa proverom.
Pronašli smo Nerodovu ekvivalenciju, to je $\sim \equiv \sim_3$.

Svaka klasa ekvivalencije predstavlja jedno stanje. Obeležimo ih, redom, slovima A, B, C, D, E . Početno stanje MDKA je ono koje sadrži početno stanje polaznog DKA (slično važi za završno stanje). Traženi MDKA predstavlja sledeći automat:



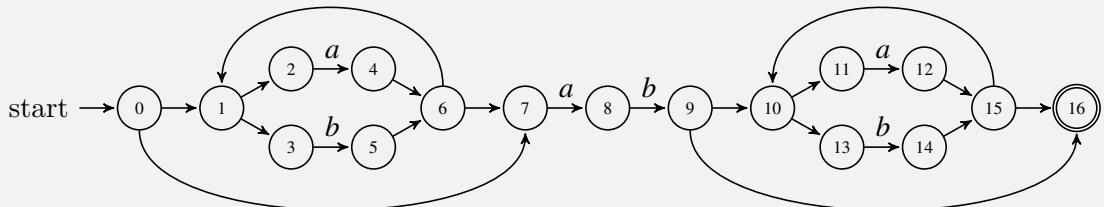
Ispravnost Murovog algoritma nam garantuje naredna teorema koju ćemo navesti bez dokaza.

Teorema 3.2.6 Rezultujući MDKA koji se dobija primenom Murovog algoritma na (pret-hodno upotpunjenoj) DKA ima minimalan broj stanja i jedinstven je do na preimenovanje stanja.

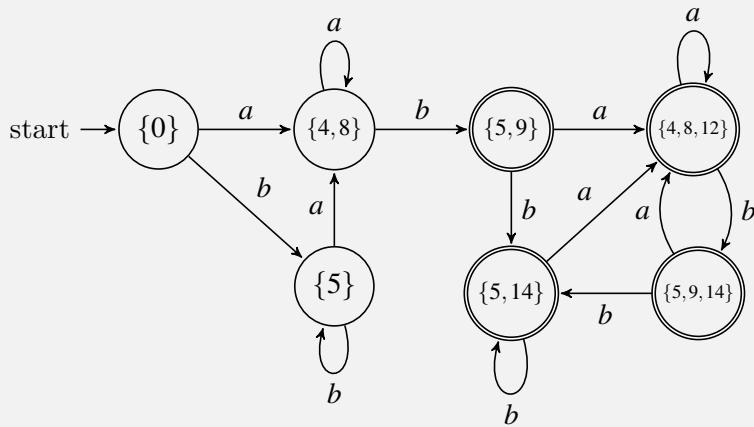
Ovo poglavlje o Klinijevoj teoremi sumiraćemo sledećim dvama primerima koji pokazuju potpun postupak primene Klinijeve teoreme na proizvoljno zadate regularne izraze i to koristeći dva različita puta od regularnog izraza do MDKA.

Primer 3.26 Konstruisati MDKA koji odgovara regularnom izrazu $(a|b)^*ab(a|b)^*$.

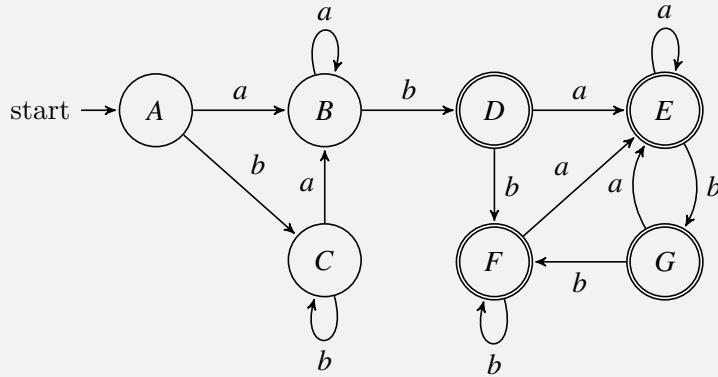
Tompsonov algoritam:



Algoritmi oslobođanja od ϵ -prelaza i konstrukcije podskupova:



Preimenovanje:



Murov algoritam:

Napomena: U nastavku ćemo pisati samo stanja za koja smo zaključili da nisu ekvivalentna, kao i dokaz na osnovu kojeg smo došli do tog zaključka. Ukoliko neka stanja nisu razmatrana, podrazumevaće se da su ekvivalentna.

$k = 0$: Klase ekvivalencije $\sim_0 : \{A, B, C\}$ i $\{D, E, F, G\}$

$k = 1$: Posmatramo \sim_1 :

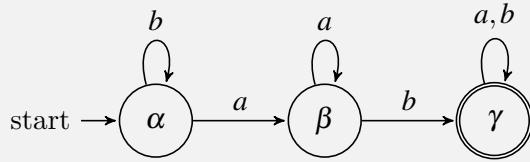
- $A \not\sim_1 B$ ($A \xrightarrow{b} C \wedge B \xrightarrow{b} D \wedge C \not\sim_0 D$).

Klase ekvivalencije posmatrane relacije su: $\{A, C\}$, $\{B\}$ i $\{D, E, F, G\}$.

$k = 2$: Posmatramo \sim_2 : Nema klasa u kojima stanja nisu ekvivalenta, pa su klase ekvivalencije posmatrane relacije: $\{A, C\}$, $\{B\}$ i $\{D, E, F, G\}$.

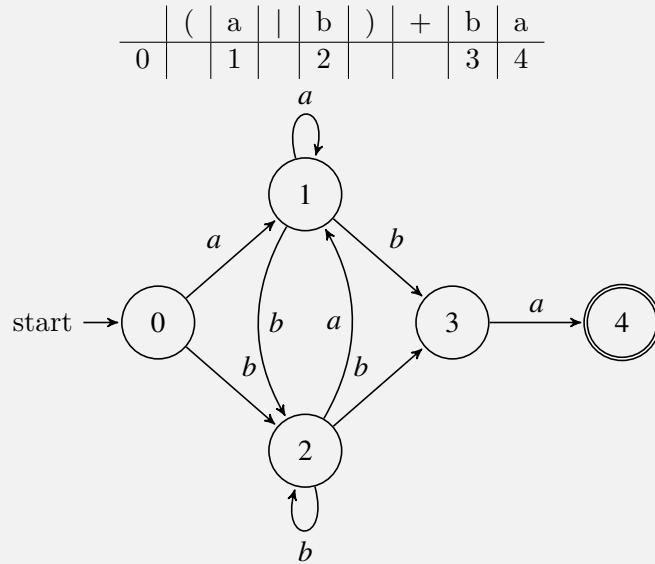
Pošto su klase ekvivalencije \sim_1 i \sim_2 jednake, pronašli smo Nerodovu ekvivalenciju, to je $\sim \equiv \sim_2$.

Obeležimo redom stanja slovima α , β i γ . Traženi MDKA je sledeći automat:



Primer 3.27 Konstruisati MDKA koji odgovara regularnom izrazu $(a|b)+ba$.

Gluškovljev algoritam:



Algoritam konstrukcije podskupova:

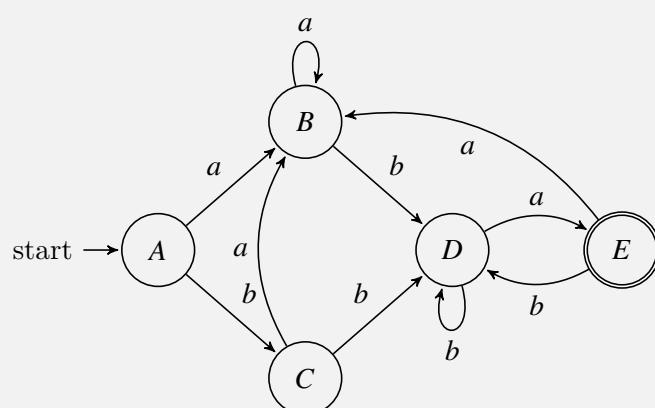
	a	b
→ 0	1	2
1	1	2, 3
2	1	2, 3
3	4	—
(4)	—	—

Tablica (N)KA:

	a	b
→ {0}	{1}	{2}
{1}	{1}	{2, 3}
{2}	{1}	{2, 3}
{2, 3}	{1, 4}	{2, 3}
{1, 4}	{1}	{2, 3}

Tablica DKA:

Imenujmo podskupove $\{0\}, \{1\}, \{2\}, \{2, 3\}$ i $\{1, 4\}$ slovima A, B, C, D i E , redom. Tada dobijeni DKA predstavlja sledeći automat:



Murov algoritam:

$k = 0$: Klase ekvivalencije $\sim_0 : \{A, B, C, D\}$ i $\{E\}$.

$k = 1$: Posmatramo \sim_1 :

- $A \not\sim_1 D$ ($A \xrightarrow{a} B \wedge D \xrightarrow{a} E \wedge B \not\sim_0 E$).

Klase ekvivalencije posmatrane relacije su: $\{A, B, C\}$, $\{D\}$ i $\{E\}$.

$k = 2$: Posmatramo \sim_2 :

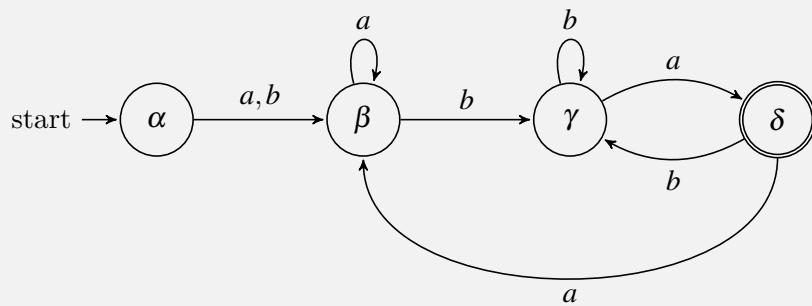
- $A \not\sim_2 B$ ($A \xrightarrow{b} C \wedge B \xrightarrow{b} D \wedge C \not\sim_1 D$).
- $A \not\sim_2 C$ ($A \xrightarrow{b} C \wedge C \xrightarrow{b} D \wedge C \not\sim_1 D$).

Klase ekvivalencije posmatrane relacije su: $\{A\}$, $\{B, C\}$, $\{D\}$ i $\{E\}$.

$k = 3$: Posmatramo \sim_3 : Nema klasa u kojima stanja nisu ekvivalenta, pa su klase ekvivalencije posmatrane relacije: $\{A\}$, $\{B, C\}$, $\{D\}$ i $\{E\}$.

Pošto su klase ekvivalencije \sim_2 i \sim_3 jednake, pronašli smo Nerodovu ekvivalenciju, to je $\sim \equiv \sim_3$.

Obeležimo redom stanja slovima α , β , γ i δ . Traženi MDKA je sledeći automat:



Teorema 3.2.7 Dva regularna jezika su ekvivalentna akko su odgovarajući PMDKA ekvivalentni.

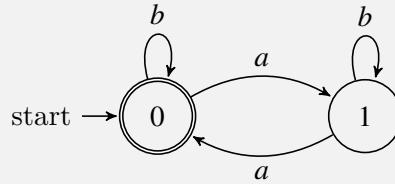
3.2.6 Metod eliminacije stanja

Videli smo kako od regularnog izraza dobijamo PMDKA. Sada nam je zadatak da na osnovu zadatog automata odredimo koji je regularni jezik njime opisan. Meotd se sastoji u transformaciji polaznog automata u ekvivalentni automat sa samo dva stanja (početnim i završnim) i jednim prelazom čija je oznaka jezik prepoznat automatom (može biti i prazan). L_{pq} označava jezik opisan oznakom (neposrednog) prelaza iz stanja p u stanje q . Na samom početku dodajemo dva nova stanja. Prvo stanje će biti novo početno stanje i povezujemo ga sa početnim stanjem ulaznog automata, a drugo predstavlja novo završno stanje i povezujemo ga sa završnim stanjem polaznog automata. Zatim eliminišemo jedno po jedno stanje na sledeći način: neka su p, q, r tri različita stanja automata, a L_{pqr} jezik svih oznaka svih izračunavanja koja počinju u p , završavaju u r , a prolaze kroz stanje q . Metod se sastoji u uklanjanju stanja q i obeležavanju grane između p i r oznakom L_{pqr} . Time se eliminiše jedno stanje, a jezik koji automat prepoznaće ostaje isti. Ponavljanjem ovog postupka, polazni automat se svodi na automat sa dva stanja - početnim i završnim,

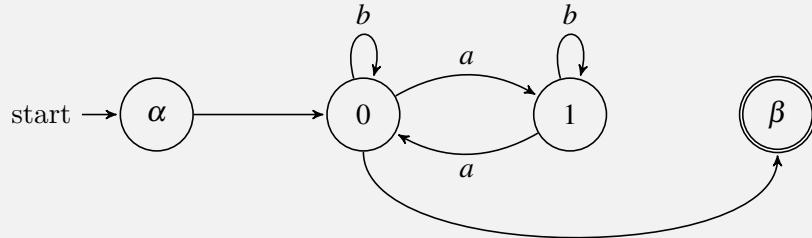
a grana koja ih povezuje označena je jezikom koji ovaj automat prepoznaće.

Primenu metoda ilustrovaćemo narednim primerima.

Primer 3.28 Odrediti koji je regularni jezik opisan sledećim automatom:

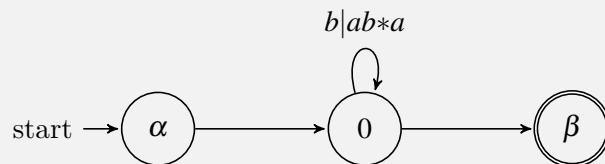


Pripremni korak je dodavanje novih stanja. Dodajemo novo jedinstveno početno stanje α i novo jedinstveno završno stanje β . Cilj je da se oslobođimo ostalih stanja, tj. da nam ostanu samo stanja α i β . Prvo što radimo jeste povezivanje novih stanja sa starim. Iz stanja α dodajemo grane u sva stara početna stanja, a u stanje β dovodimo grane iz svih starih završnih stanja.

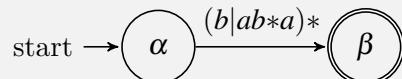


Da bismo se oslobođili starih stanja dopuštamo da se na granama javljaju regularni izrazi. Rezultat primene ove metode je automat sa jednim početnim i jednim završnim stanjem koja su povezana jednom granom koja prepoznaće regularni izraz koji je opisan početnim automatom. Ne postoji neko pravilo po kom treba birati koje stanje prvo eliminisati.

Eliminišimo, na primer, stanje 1:



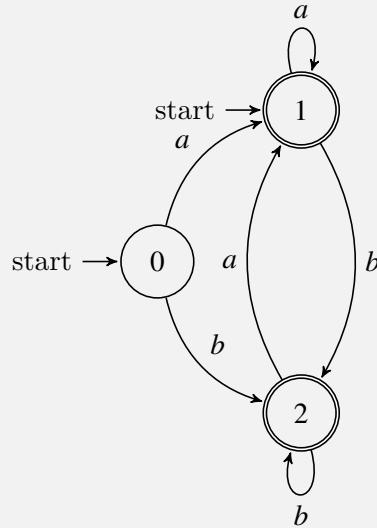
Zatim, eliminisemo stanje 0:



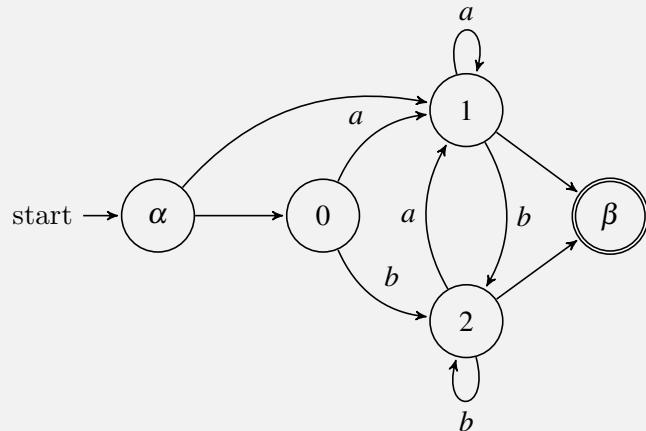
Dakle, zadati automat prepoznaće regularni jezik $(b|ab^*a)^*$.

Napomenimo već sada da dužina dobijenog regularnog izraza veoma brzo raste što je automat složeniji. Ponekad je moguće vršiti skraćivanja izraza u hodu, što ćemo ilustrovati sledećim primerom.

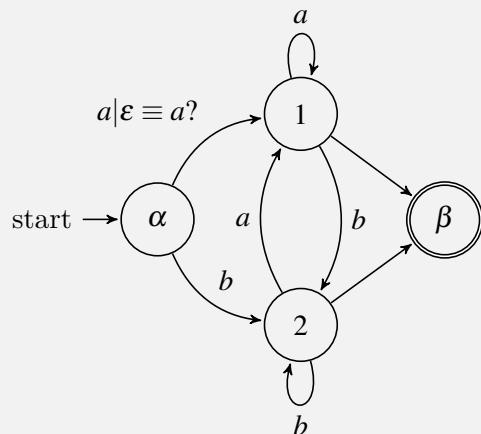
Primer 3.29 Odrediti koji je regularni jezik opisan sledećim automatom:



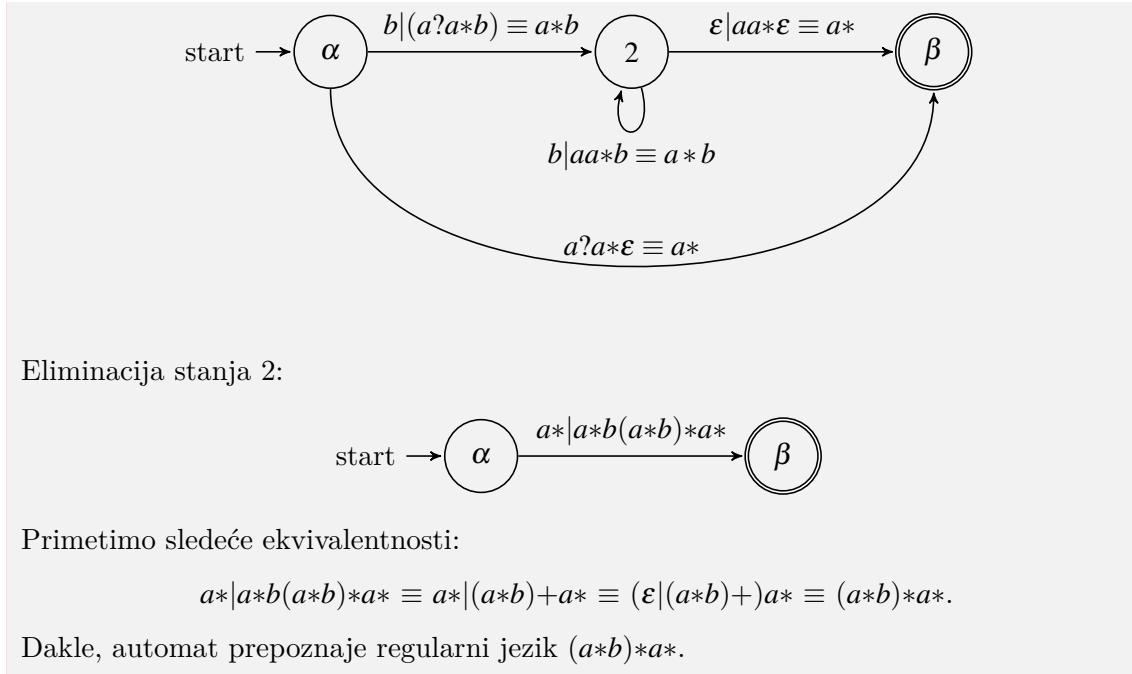
Pripremni korak:



Eliminacija stanja 0:



Eliminacija stanja 1:



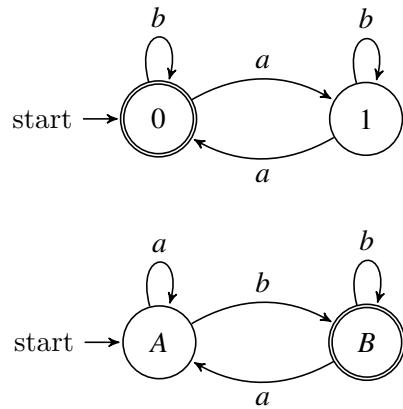
3.3 Skupovne operacije i konačni automati

Neka je $\Sigma = \{a_1, a_2, \dots, a_n\}$. Pošto je Σ^* regularan jezik (i znamo da ga možemo predstaviti regularnim izrazom $(a_1|a_2|\dots|a_n)^*$), ako su regularni jezici koji mogu biti konstruisani podskupovima regularnog jezika Σ^* zatvoreni za presek i uniju, onda je jasno da će biti zatvoreni i za komplement, a samim tim i za sve ostale Bulovske operacije.

Lema 3.3.1 Za dva konačna automata moguće je napraviti treći konačni automat koji prepozna njihov presek, uniju i razliku.

Dokaz. Automati kojima računamo preseke, uniju i razlike su PDKA. Teorijski nije potrebno da budu i minimalani, ali u praksi je lakše raditi sa manjim brojem stanja. Takođe, dokaz ćemo prikazati na jednostavna dva automata, ali postupak se u potpunosti skalira za proizvoljne automate.

Neka su zadata sledeća dva automata, \mathcal{A} i \mathcal{B} , redom:

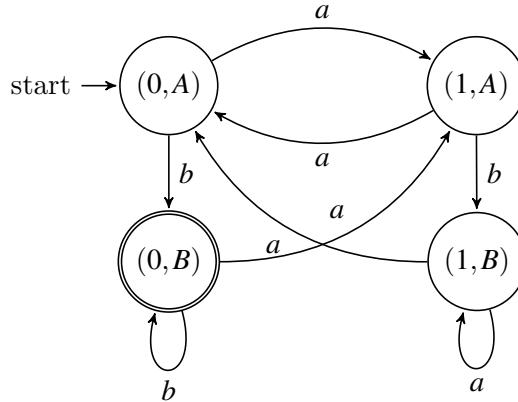


Jednostavnosti radi, drugačije obeležavamo stanja automata \mathcal{A} od stanja automata \mathcal{B} .

Ključan deo dokaza za sve tri operacije je konstrukcija Dekartovog proizvoda stanja ova dva automata. Znači, konstruišemo automat $\mathcal{A} \times \mathcal{B}$ čiji je skup stanja Dekartov proizvod stanja datih automata, tj. $Q_{\mathcal{A} \times \mathcal{B}} = Q_{\mathcal{A}} \times Q_{\mathcal{B}}$. Grane ovog automata konstruišemo tako što posmatramo svaki par $(x, y) \in Q_{\mathcal{A} \times \mathcal{B}}$ i gledamo pojedinačno u koje se stanje dolazi iz stanja x u prvom automatu, odnosno iz stanja y u drugom. Početna stanja tog automata moraju da sadrže početna stanja oba automata; u ovom slučaju to je stanje $(0, A)$. Proizvod istovremeno simulira oba automata. Ono zbog čega je proizvod automata koristan jeste što pravim odabirom završnih stanja možemo dobiti različite automate, a tabela 3.3 određuje završna stanja za svaku operaciju.

Operacija	Završna stanja
$\mathcal{A} \cap \mathcal{B}$	$(0, B)$
$\mathcal{A} \cup \mathcal{B}$	$(0, A), (0, B), (1, B)$
$\mathcal{A} \setminus \mathcal{B}$	$(0, A)$
$\mathcal{B} \setminus \mathcal{A}$	$(1, B)$

Dekartov proizvod automata \mathcal{A} i \mathcal{B} ilustrovan je narednim automatom gde je kao završno stanje obeleženo stanje $(0, B)$, dakle, prikazan je presek ova dva automata.

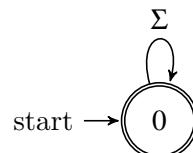


■

Uvedimo još i operaciju komplementa automata. Neka je zadat automat \mathcal{A} nad azbukom Σ . **Komplement automata \mathcal{A}** , u oznaci \mathcal{A}^c , predstavlja sledeći automat:

$$\mathcal{A}^c = \Sigma^* \setminus \mathcal{A},$$

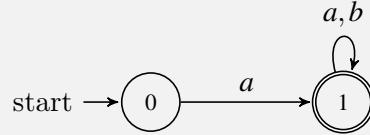
pri čemu je Σ^* zapis za automat, koji opisuje regularni jezik Σ^* , a koji je predstavljen sledećim automatom:



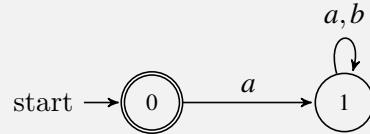
Kada znamo da konstruišemo Dekartov proizvod automata, onda znamo i razliku, pa znamo i komplement (na osnovu prethodne definicije). Ipak, operaciju komplementa automata možemo sprovesti na sledeći način: Neka je zadat PDKA $\mathcal{A} = (\Sigma, Q, I, F, \Delta)$. Komplementarni automat \mathcal{A}^c je automat $(\Sigma, Q, I, Q \setminus F, \Delta)$. Dakle, samo smo zamenili skup završnih stanja njegovim komplementom, tj. stanja koja su bila završna sada označimo

kao nezavršna i obrnuto. Ipak, treba biti oprezan sa ovakvim sprovođenjem komplementa automata. Ilustrijmo ovo sledećim primerom:

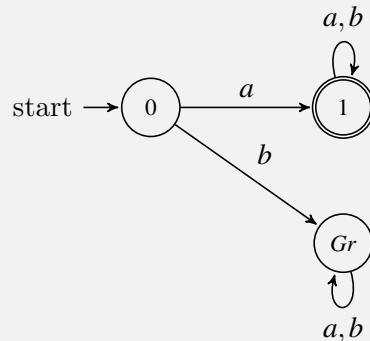
Primer 3.30 Automat koji prepoznaje regularni jezik $a(a|b)^*$ je sledeći automat:



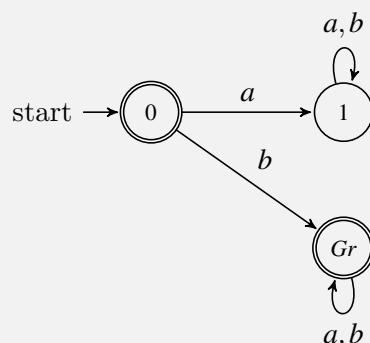
Prateći prethodno uputstvo, automat koji prepoznaje regularni jezik $(a(a|b)^*)^c$ trebalo bi da bude sledeći automat:



Međutim, reč $w = b$ ne može biti prepoznata ni jednim automatom, što je kontraprimer hipotezi da je drugi automat komplement prvom. Ovaj rezultat se dobija zato što početni (prvi) automat nije prethodno upotpunjjen. Ispravno bi bilo:



Sada se ispravan komplement dobija primenom prethodnog pravila:



3.3.1 Lema o razrastanju

Lema 3.3.2 Svaka reč jezika L može razbiti na tri dela, tj. $w = xyz$, tako da važi $xy^i z \in L$, za sve $i \geq 0$.

Dokaz. Neka automat \mathcal{A} ima k stanja (tj. $|Q| = k$). Međutim, koliko god da automat

ima stanja, uvek možemo naći reč čija je dužina veća od k . Za svaku reč $w \in L$ koja je duža od k važi sledeće: prilikom njenog prihvatanja će se kroz barem jedno stanje (označimo ga q) proći dva puta (na osnovu Diriheleovog principa), tj. za neke $q_0 \in I$ i $q_r \in F$:

$$q_0 \xrightarrow{x} q \xrightarrow{y} q \xrightarrow{z} q_r,$$

gde je $w = xyz$. Rečima, postoji neko stanje q tako da ukoliko se krene iz stanja q_0 , pročita neka reč x , zatim stigne do stanja q koje se može ponavljati (preko reči y se opet stigne do stanja q) i nadalje se čita reč z i stigne do stanja q_r .

Kako $w \in L$, to i $xz \in L$ (jer postoji put $q_0 \xrightarrow{x} q \xrightarrow{z} q_r$ u automatu). Već znamo da $xyz \in L$. Takođe, reč $xyyz \in L$ (jer postoji put $q_0 \xrightarrow{x} q \xrightarrow{y} q \xrightarrow{y} q \xrightarrow{z} q_r$ u automatu). Lako se vidi da sve reči oblika xy^iz pripadaju jeziku L , za $i \geq 0$. ■

Na početku ovog poglavlja rekli smo da nikoji jezik koji zahteva uparivanje (barem) dva entiteta ne može biti opisan regularnim jezikom. Kao primer dali smo jezik $L = \{a^n b^n \mid n \geq 0\}$. Na ovom mestu ćemo pokazati dokaz za ovu činjenicu. Naravno, iako sledeća teorema važi za proizvoljan jezik koji ispunjava navedeni zahtev uparivanja, mi ćemo pokazati dokaz za konkretan jezik, i to će biti baš jezik L . Ovo ćemo pokazati korišćenjem leme o razrastanju.

Posledica 3.3.3 Jezik $L = \{a^n b^n \mid n \geq 0\}$ nije regularan.

Dokaz. Prepostavimo suprotno, tj. da je jezik L regularan. Na osnovu Klinijeve teoreme, postoji (N)KA koji prepozna jezik L , i neka je to automat $\mathcal{A} = (\Sigma, Q, I, F, \Delta)$. Na osnovu prethodne leme, postoji reč $w \in L$, takva da je $|w| > |Q|$ i koja je oblika:

$$\underbrace{aa\dots}_{n} \underbrace{abb\dots}_{n} b.$$

Reč je dovoljno dugačka da možemo da nađemo njeno razbijanje na tri reči x , y i z . Postavlja se pitanje na koje sve načine možemo da podelimo datu reč?

Posmatrajmo razbijanje:

$$\underbrace{aa\dots}_{x} \underbrace{aaa\dots}_{y} \underbrace{aaa\dots}_{z} \underbrace{abbb\dots}_{bbb} bbb.$$

ili simetrično razbijanje:

$$\underbrace{aa\dots}_{x} \underbrace{abb\dots}_{y} \underbrace{bbb\dots}_{z} \underbrace{bbb\dots}_{bbb} b.$$

Primetimo da u ovim slučajevima važi $xz \notin L$, te ovakva razbijanja nisu moguća.

Posmatrajmo razbijanje:

$$\underbrace{aa\dots}_{x} \underbrace{aaa\dots}_{y} \underbrace{abb\dots}_{z} \underbrace{bbb\dots}_{bb} b.$$

Jasno je da ako y obuhvata različite brojeve karaktera a i b , onda će ponovo važiti $xz \notin L$. Dakle, jedino razbijanje reči w koje je preostalo je takvo da y sadrži jednak broj karaktera a i b . Na primer, neka $y = aabb$. Međutim, sada iako $xy \in L$ i $xyz \in L$, dogodilo se da $xy^2z \notin L$ (jer $xy^2z = \underbrace{aa\dots}_{x} \underbrace{aabbaabb}_{y} \underbrace{bb\dots}_{z} b \notin L$). Dakle, ni ovo razbijanje nije moguće.

Zaključujemo da ne možemo da rastavimo reč w na tri reči. Kontradikcija. ■

3.3.2 Pitanja i zadaci**Zadatak 3.1** Upotpuniti deterministički konačni automat iz primera 3.6. ▀**Zadatak 3.2** Konstruisati nedeterministički konačni automat koji odgovara regularnim izrazima:

1. $((a|ab)+b?)^*$,
2. $(ab^*)b$,
3. $(a?b|a)^*$, i
4. $(a|b)^*c(ca^*)^+$,

prvo primenom Tompsonovog algoritma i algoritma oslobođanja od ϵ -prelaza, a potom i primenom Gluškovljevog algoritma. ▀

Zadatak 3.3 Dokazati lemu 3.2.5. ▀

4. Kontekstno-slobodne gramatike

Kontekstno-slobodne gramatike (češće ćemo koristiti oznaku KS gramatike) jesu, po pravilu, dovoljne da se opiše sintaksička struktura jednog programskog jezika, kao što su regularni izrazi dovoljni da se opiše leksička struktura programskih jezika. Neformalno, jedna gramatika sadrži simbole i pravila izvođenja u toj gramatici.

Simboli mogu biti:

- **završni simboli** (koriste se i sinonimi terminalni simboli, terminali, i tokeni), i
- **nezavršni simboli** (koriste se i sinonimi neterminalni simboli, neterminali, i pomoći simboli).

Posmatrajmo sledeći primer:

Primer 4.1 Neka je data gramatika G sa sledećim skupom pravila:

1. $A \rightarrow aAb$, i
2. $A \rightarrow \epsilon$.

Ovo je jedan jednostavan primer KS gramatike. Završni simboli su a i b , a pomoćni simbol je A . Izvođenje počinje od pomoćnog simbola kojeg menjamo na osnovu nekog pravila. Možemo primeniti pravila prezapisivanja, na primer, na sledeći način:

$$A \xrightarrow{1} aAb \xrightarrow{1} aaAbb \xrightarrow{2} aabb.$$

Kada dobijemo reč u kojoj nema neterminalnih simbola, onda za tu reč možemo reći da je **reč jezika opisane gramatike**. Vidimo da reč $w = aabb$ pripada jeziku date gramatike jer smo pronašli izvođenje do te reči u datoj gramatici. Nije teško primetiti još i da je jezik opisane gramatike zapravo jezik $L = \{a^n b^n \mid n \geq 0\}$ (lako se pokazuje indukcijom po dužini reči). Za $n = 0$, jasno je da $\epsilon = a^0 b^0 \in L$. Kako dobijamo ϵ koristeći prethodno navedena pravila? Primenom pravila 2 (samo jednom), dobija se tražena pražna reč:

$$A \xrightarrow{2} \epsilon.$$

Navedimo još dva primera KS gramatike.

Primer 4.2 Jezik $L_2 = \{a^n b^n \mid n > 0\}$ opisuje se KS gramatikom G_2 koja sadrži sledeća dva pravila:

1. $A \rightarrow aAb$, i
2. $A \rightarrow ab$,

što se često kraće zapisuje:

$$\begin{array}{lcl} A & \longrightarrow & aAb \\ & | & ab. \end{array}$$

Primer 4.3 Jezik $L_3(a^*b^*)$ se opisuje KS gramatikom G_3 koja sadrži sledeća pravila:

$$\begin{array}{lcl} S & \longrightarrow & AB \\ A & \longrightarrow & aA \\ & | & \varepsilon \\ B & \longrightarrow & bB \\ & | & \varepsilon. \end{array}$$

Iz primera 4.3 vidimo da u nekim slučajevima može biti mnogo jednostavnije zapisati jezik pomoću regularnog izraza (a^*b^*) nego pomoću gramatike (G_3). Stoga ćemo i koristiti zapis pomoću regularnih izraza kada god nam to odgovara. Naravno, ovo nije uvek moguće uraditi, kao što smo videli u primeru 4.1.

4.1 Definicija KS gramatike

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 4.4 Gramatika iz primera 4.2, koju smo označili G_2 , obuhvata sledeće komponente:

$$\Sigma = \{a, b\}, N = \{A\}, S = A \text{ i } P = \{(A, aAb), (A, ab)\}.$$

Napomenimo da pravila nećemo pisati kao uređene parove, već onako kako smo pisali u primerima 4.1 do 4.3.

Po konvenciji, velika slova predstavljaju nezavršne simbole, a mala slova završne simbole. Aksioma je prvo slovo koje se navodi u prvom pravilu sa leve strane. Takođe, na osnovu pravila gramatike možemo zaključiti koji simboli su neterminalni. Neterminalni simboli su oni simboli koji se nalaze sa leve strane strelice svakog pravila.

Već smo napomenuli da KS gramatike predstavljaju samo jedan od oblika gramatika koje su dovoljne za opis veštačkih jezika. Nazivamo ih kontekstno slobodnim zbog činjenice da se sa leve strane pravila nalazi tačno jedan neterminalni simbol. Pojasnimo ovu činjenicu na primeru.

Primer 4.5 Neka je zadato pravilo gramatike $A \rightarrow aAbb$ i neka smo nekim izvođenjem nezavršnog simbola S došli do $aAbAb$.

Dalje možemo zameniti prvi simbol A na sledeći način:

$$S \Rightarrow \dots \Rightarrow a\underline{A}bAb \Rightarrow aa\underline{A}bbAb.$$

Ali mogli smo da zamenimo i drugi simbol A , na isti način:

$$S \Rightarrow \dots \Rightarrow aAb\underline{A}b \Rightarrow aAba\underline{A}bb.$$

Dakle, kontekst u kom se javlja simbol nam nije bitan. Mi ga menjamo na potpuno isti način, nezavisno od simbola između kojih se nalazi.

Navedimo sada primer kontekstno zavisne gramatike:

Primer 4.6 Neka je zadato pravilo gramatike $bAb \rightarrow aABb$ i neka smo nekim izvođenjem nezavršnog simbola S došli do $aAbAb$.

Ako bismo sad želeli da zamenimo nezavršni simbol A , mogli bismo da izaberemo samo drugi simbol jer se prvi ne nalazi u zadatom kontekstu, tj. ne nalazi se između dva mala slova b :

$$S \Rightarrow \dots \Rightarrow aAb\underline{A}b \Rightarrow aAa\underline{A}bb.$$

Usvojimo dodatna pravila zapisivanja:

- \rightarrow koristimo u zapisu pravila umesto uređenih parova. Sa leve strane ove strelice može da se nađe samo neterminalni simbol, dok sa desne strane može da se nađe kombinacija završnih i nezavršnih simbola, i
- \Rightarrow koristimo u zapisu izvođenja.

Uvedimo sada definiciju relacije izvođenja.

Relacija domena $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ naziva se **relacija izvođenja** i obeležava se oznakom \Rightarrow . Kažemo da **niska $\alpha X \beta$ izvodi nisku $\alpha \gamma \beta$** , u oznaci $\alpha X \beta \Rightarrow \alpha \gamma \beta$, akko postoji pravilo $X \rightarrow \gamma \in P$, pri čemu su $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$ i $X \in N$.

Relacija označena simbolom \Rightarrow^* je refleksivno tranzitivno zatvoreno zatvorenje relacije izvođenja. To je najmanja tranzitivno-refleksivna relacija koja sadrži datu relaciju, tj. ima 0 ili više koraka izvođenja. Relacija označena simbolom \Rightarrow^+ je tranzitivno zatvoreno zatvorenje relacije izvođenja. To je najmanja tranzitivna relacija koja sadrži datu relaciju, tj. može se izvesti u 1 ili više koraka.

Koristeći ove definicije, možemo definisati jezik gramatike na sledeći način.

Jezik gramatike G , u oznaci $L(G)$, predstavlja sledeći skup:

$$L(G) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}.$$

Za nadalje će nam biti važno da posmatramo neke određene klase pravila u gramatikama. Posmatrajmo sledeća pravila:

$$\begin{aligned} A &\longrightarrow A\alpha \\ B &\longrightarrow \beta B \\ C &\longrightarrow C\gamma C \\ D &\longrightarrow \delta D\varepsilon \end{aligned}$$

gde su $A, B, C, D \in N$, a niske $\alpha, \beta, \gamma, \delta, \varepsilon \in (\Sigma \cup N)^+$. Kažemo da su ova pravila *direktno rekurzivna* jer se simbol sa leve strane pravila pojavljuje i na njegovoj desnoj strani. Za prvo pravilo kažemo da je *levo rekurzivno* jer se pomoćni simbol s leve strane pravila javlja sasvim levo na desnoj strani pravila. Slično tome, kažemo da je drugo pravilo *desno rekurzivno*, treće pravilo *dvostruko rekurzivno* i četvrto pravilo *rekurzivno po sredini*.

Pored direktne rekurzije, u pravilima se može javiti i *indirektna rekurzija*:

$$\begin{aligned} A &\longrightarrow B\alpha \\ B &\longrightarrow A\beta \end{aligned}$$

Indirektna rekurziva se može uvek svesti na direktnu rekurziju. Na primer, prethodna dva pravila su ekvivalentna narednom pravilu:

$$A \longrightarrow A\beta\alpha$$

4.2 Stablo izvođenja

Pogledajmo primer konstrukcije jedne jednostavne gramatike koja opisuje aritmetičke izraze sabiranja celih brojeva i primera izvođenja u toj gramatici.

Primer 4.7 Dati su nam aritmetički izrazi u kojima učestvuju celi brojevi i operacija sabiranja. Primeri izraza koje želimo da prepoznamo gramatikom su: $32 + 7 + 15$, $13 + 5$, i 431 . Odrediti gramatiku ovakvih (i njima sličnih) aritmetičkih izraza.

Kako smo napomenuli da KS gramatike koristimo za sintaksičku analizu, često ćemo apstrahovati date niske u tokene, tako da umesto navedenih primera izraza, posmatraćemo izraze tipa: $a + a + a$, $a + a$, i a , tj. obeležićemo oznakom a token koji predstavlja ceo broj, a oznakom E (*expression*) aritmetički izraz. Na ovom mestu, tokene i terminale možemo unifikovati.

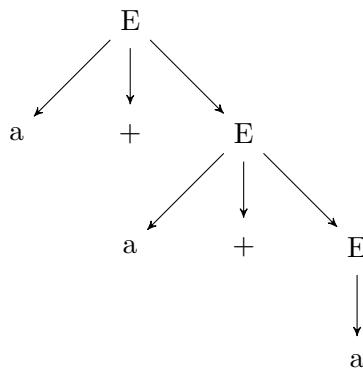
Definišemo pravila gramatike koju ćemo označiti G_{+D} :

1. $E \longrightarrow a$, i
2. $E \longrightarrow a + E$.

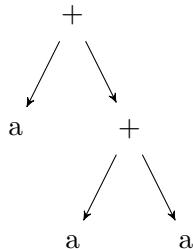
Primer izvođenja izraza $a + a + a$:

$$E \xrightarrow{2} a + E \xrightarrow{2} a + a + E \xrightarrow{1} a + a + a.$$

Primer izvođenja izraza iz prethodnog primera se takođe može predstaviti i **drvetom izvođenja**:



Kao što vidimo, stablo izvođenja čuva detalje koje se vide u izvođenju, a koja ne moraju nužno biti važne. Zbog toga, izvođenje se može predstaviti i **apstraktnim sintaksičkim stablom**, u kojem se ne navode neterminali. Apstraktno sintaksičko stablo izvođenja iz prethodnog primera je dano na narednoj slici:



U narednom primeru ćemo za aritmetičke izraze iz prethodnog primera pokazati da postoji još jedna gramatika koja im odgovara.

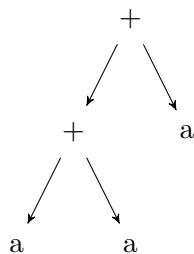
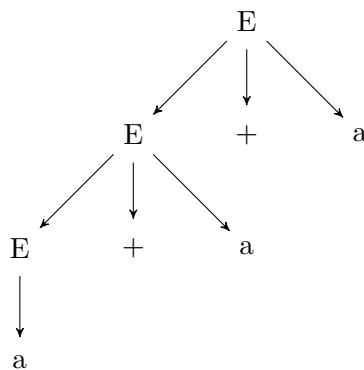
Primer 4.8 Definišemo pravila gramatike koju ćemo označiti G_{+L} :

1. $E \rightarrow a$, i
2. $E \rightarrow E + a$.

Primer izvođenja izraza $a + a + a$:

$$E \xrightarrow{2} E + a \xrightarrow{2} E + a + a \xrightarrow{1} a + a + a.$$

Stablo izvođenja iz prethodnog primera je:



Izvođenja koja su data u primerima 4.7 i 4.8 imaju važnu osobinu koja utiče na asocijativnost izraza koje oni izvode. Zbog toga ćemo formalizovati klasu ovakvih izvođenja narednim definicijama.

Definicija 4.2.1 — Najdešnje izvođenje. *Najdešnje izvođenje* (engl. *rightmost derivation*) predstavlja izvođenje u kojem se uvek menja najdešnji simbol u tekućoj rečeničnoj formi.

Definicija 4.2.2 — Najlevlje izvođenje. *Najlevlje izvođenje* (engl. *leftmost derivation*) predstavlja izvođenje u kojem se uvek menja najlevlji simbol u tekućoj rečeničnoj formi.

Izvođenje iz primera 4.7 predstavlja primer najdešnjeg izvođenja. Izvođenje iz primera 4.8 predstavlja primer najlevljeg izvođenja.

N Postoje izvođenja koja nisu ni najlevlja ni najdešnja.

Primer 4.9 Definišemo pravila gramatike koju ćemo označiti G_{+LD} :

1. $E \rightarrow a$, i
2. $E \rightarrow E + E$.

U ovoj gramatici, za izraz $a+a+a$ postoji više izvođenja. Primeri nekih od tih izvođenja su:

1. Izvođenje \mathcal{I}_1 je primer najlevljeg izvođenja::

$$\mathcal{I}_1 : E \xrightarrow{2} E + E \xrightarrow{1} a + E \xrightarrow{2} a + E + E \xrightarrow{1} a + a + E \xrightarrow{1} a + a + a.$$

Stablo izvođenja i apstraktno sintaksičko stablo koja se dobijaju na osnovu \mathcal{I}_1 su

Stablo izvođenja	Apstraktno sintaksičko stablo
<pre> graph TD E1[E] --> E2[E] E1 --> P[+] E1 --> E3[E] E2 --> a1[a] E2 --> P E2 --> E4[E] E3 --> a2[a] P --> P E4 --> a3[a] a1 --> a1 P --> P a2 --> a2 E4 --> a4[a] a3 --> a3 a4 --> a4 </pre>	<pre> graph TD P1[+] --> a1[a] P1 --> P2[+] P1 --> a3[a] P2 --> a2[a] P2 --> P3[+] P2 --> a4[a] P3 --> a1 P3 --> a2 P3 --> a4 </pre>

2. Izvođenje \mathcal{I}_2 je primer najdešnjeg izvođenja:

$$\mathcal{I}_2 : E \xrightarrow{2} E + E \xrightarrow{1} E + a \xrightarrow{2} E + E + a \xrightarrow{1} E + a + a \xrightarrow{1} a + a + a,$$

Stablo izvođenja i apstraktno sintaksičko stablo koja se dobijaju na osnovu \mathcal{I}_2 su

Stablo izvođenja	Apstraktno sintaksičko stablo
<pre> graph TD E1[E] --> E2[E] E1 --> P[+] E1 --> E3[E] E2 --> a1[a] E2 --> P E2 --> E4[E] E3 --> a2[a] P --> P E4 --> a3[a] a1 --> a1 P --> P a2 --> a2 E4 --> a4[a] a3 --> a3 a4 --> a4 </pre>	<pre> graph TD P1[+] --> a1[a] P1 --> P2[+] P1 --> a3[a] P2 --> a2[a] P2 --> P3[+] P2 --> a4[a] P3 --> a1 P3 --> a2 P3 --> a4 </pre>

3. Izvođenja \mathcal{I}_3 i \mathcal{I}_4 su primjeri izvođenja koja nisu ni najlevlja ni najdešnja:

$$\mathcal{I}_3 : E \xrightarrow{2} E + E \xrightarrow{1} E + a \xrightarrow{2} E + E + a \xrightarrow{1} a + E + a \xrightarrow{1} a + a + a$$

$$\mathcal{I}_4 : E \xrightarrow{2} E + E \xrightarrow{2} E + E + E \xrightarrow{1} a + E + E \xrightarrow{1} a + a + E \xrightarrow{1} a + a + a.$$

Stablo izvođenja i apstraktno sintaksičko stablo koja se dobijaju na osnovu izvođenja \mathcal{I}_3 i \mathcal{I}_4 su

Stablo izvođenja	Apstraktno sintaksičko stablo
<pre> graph TD E1[E] --> E2[E] E1 --> P[+] E1 --> E3[E] E2 --> a1[a] E2 --> P E2 --> E4[E] E3 --> a2[a] P --> P E4 --> a3[a] a1 --> a1 P --> P a2 --> a2 E4 --> a4[a] a3 --> a3 a4 --> a4 </pre>	<pre> graph TD P1[+] --> a1[a] P1 --> P2[+] P1 --> a3[a] P2 --> a2[a] P2 --> P3[+] P2 --> a4[a] P3 --> a1 P3 --> a2 P3 --> a4 </pre>

Kao što znamo, svakom izvođenju u nekoj gramatici G se dodeljuje jedno stablo izvođenja. Nasuprot ovome, u opštem slučaju u dатој gramatici G , jedna niska x može imati više izvođenja, kojima odgovara isto stablo izvođenja. Ova izvođenja se razlikuju prema redosledu u kome se primenjuju pravila gramatike tokom izvođenja. Otuda kažemo da su dva izvođenja u gramatici G *ekvivalentna* ako imaju isto stablo izvođenja. Primetimo u primeru 4.9 da smo za izvođenja $\mathcal{I}_2, \mathcal{I}_3$ i \mathcal{I}_4 dobili ista stabla izvođenja.

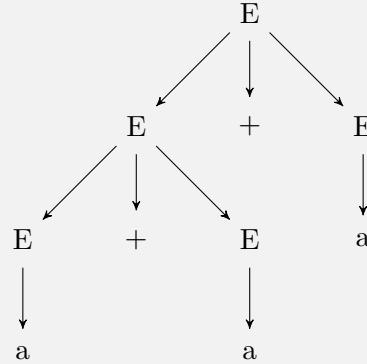
4.3 Višezačne gramatike

Za gramatiku iz primera 4.9 važi jedno zanimljivo svojstvo.

Primer 4.10 Posmatrajmo ponovo izraz $a+a+a$. Za ovaj izraz u gramatici G_{+LD} postoji najlevlje izvođenje \mathcal{I}_{L1} koje je dato u nastavku:

$$\mathcal{I}_{L1} : E \xrightarrow{2} E+E \xrightarrow{2} E+E+E \xrightarrow{1} a+E+E \xrightarrow{1} a+a+E \xrightarrow{1} a+a+a.$$

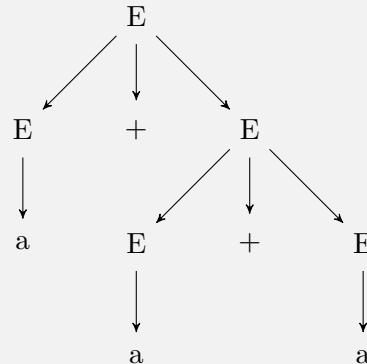
Stablo izvođenja koje se dobija na osnovu \mathcal{I}_{L1} je



Međutim, za isti izraz u gramatici G_{+LD} postoji još jedno najlevlje izvođenje \mathcal{I}_{L2} koje je dato u nastavku:

$$\mathcal{I}_{L2} : E \xrightarrow{2} E+E \xrightarrow{1} a+E \xrightarrow{2} a+E+E \xrightarrow{1} a+a+E \xrightarrow{1} a+a+a.$$

Stablo izvođenja koje se dobija na osnovu \mathcal{I}_{L2} je



Ono što je važnije primetiti jeste da se dogodilo da smo za istu nisku generisanu gramatičkom G_{+LD} dobili dva različita stabla izvođenja. Ovaj rezultat nam je bitan zbog sledeće leme i prateće definicije.

Lema 4.3.1 Ako je za datu gramatiku G tačan jedan od sledećih iskaza:

- Svaka reč u gramatici G ima jedinstveno najlevlje izvođenje.
- Svaka reč u gramatici G ima jedinstveno najdešnje izvođenje.
- Svaka reč u gramatici G ima jedinstveno drvo izvođenja.

onda su tačna i druga dva iskaza.

Na osnovu ove leme uvodi se sledeća definicija.

Definicija 4.3.1 — Jednoznačna i višezačna gramatika. KS-gramatika u kojoj za barem jednu nisku postoje barem dva različita stabla izvođenja nazivamo *višezačnom*. U suprotnom slučaju, tu gramatiku nazivamo *jednoznačnom*.

N Višezačnost je osobina gramatike, a ne jezika. Neke gramatike jednog jezika mogu biti jednoznačne, a druge višezačne.

N Problem određivanja da li je neka gramatika višezačna je u opštem slučaju neodlučiv, odnosno, ne postoji opšti algoritam koji određuje da li je data gramatika višezačna ili ne. Ipak, postoji klasa gramatika za koju se zna da su njeni elementi više značne gramatike. To su gramatike koje sadrže bar jedno pravilo koje je dvostruko rekurzivno. Otuda je gramatika iz primera 4.9 višezačna.

4.4 O asocijativnosti, prioritetima i razrešavanju višezačnosti

Kao što je čitaocu verovatno već poznato, redosled izračunavanja aritmetičkih izraza se određuje na osnovu dogovora o *asocijativnosti operatora*. Tako, na primer, aritmetički izraz $a + a + a$ se može izračunati na dva načina:

1. Ako se prvo saberi prva dva sabirka, pa se onda na taj zbir doda treći sabirak: $(a + a) + a$.
2. Ako se prvo saberi poslednja dva sabirka, pa se onda na taj zbir doda prvi sabirak: $a + (a + a)$.

U prvom slučaju kažemo da se operator sabiranja *asocira nalevo*, a u drugom slučaju da se *asocira nadesno*.

Zanimljivo je primetiti da je asocijativnost operatora $+$ u navedenim gramatikama G_{+L} i G_{+D} leva, odnosno desna, redom. Ispostavlja se da je ovo posledica tipa direktnе rekurzije koje pravila u ovim gramatikama nameću. Tako, levo rekurzivno pravilo $E \longrightarrow E + a$ u gramatici G_{+L} čini da operator sabiranja bude levo asocijativan, dok desno rekurzivno pravilo $E \longrightarrow a + E$ u gramatici G_{+D} čini da operator sabiranja bude desno asocijativan.

Pored ovog pravila, možemo primetiti da postoji i vizualna naznaka o asocijativnosti ovog operatora. U primeru 4.7, stablo izvođenja (kao i apstraktno sintaksičko stablo) izraza $a + a + a$ deluje kao da se „agnulu” na desnu stranu. Slično tome, u primeru 4.8, stablo

izvođenja izraza $a + a + a$ deluje kao da se „nagnulo” na levu stranu. Ova „nagnutost” stabala upravo govori o tipu asocijativnosti operatora u svakoj od gramatika.

Posledica ovog zapažanja jeste da je potrebno da vodimo računa prilikom konstrukcije gramatike ukoliko bismo želeli da nametnemo određenu asocijativnost. Na primer, s obzirom da su u programkom jeziku C aritmetički operatori u definisani kao levo asocijativni, gramatika G_{+L} bi bila pogodnija za korišćenje u odnosu na gramatiku G_{+D} .

Takođe, primetimo da se u primeru 4.9 u izvođenju \mathcal{I}_1 nameće desna asocijativnost, dok se u izvođenjima \mathcal{I}_2 , \mathcal{I}_3 i \mathcal{I}_4 nameće leva asocijativnost. Sa druge strane, pravila kao što je $E \rightarrow E + E$ predstavljaju primer pravila sa dvostrukom rekurzijom i takva pravila ostavljaju slobodan izbor asocijativnosti. Zbog toga što je asocijativnost najčešće strogo nametnuta, u praksi treba izbegavati dvostrukе rekurzije, a samim tim i više značne gramatike.

Pogledajmo još jedan primer gramatike aritmetičkih izraza.

Primer 4.11 Dati su nam aritmetički izrazi u kojima učestvuju celi brojevi i operacije sabiranja i množenja. Primeri izraza koje želimo da prepoznamo gramatikom su: $3 + 5 * 6 * 3 + 2 * 5$, $2 * 4$, $4 * 2 + 3 * 7 + 1$ i 5 . Definišemo pravila gramatike koju ćemo označiti G_{+*LD} :

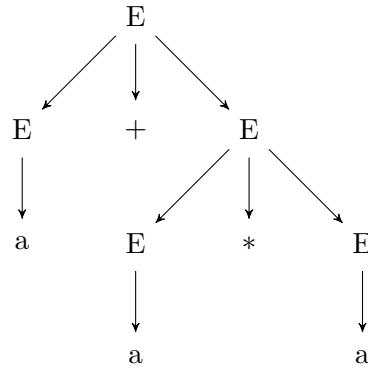
1. $E \rightarrow a$,
2. $E \rightarrow E + E$, i
3. $E \rightarrow E * E$.

I u prethodnom primeru vidimo primer dvostrukе rekurzije, i već smo videli da ona daje više značnost što se tiče asocijativnosti. Dodatno se pojavio još jedan problem. Na primer, za izraz $a + a * a$ postoje dva najlevlja izvođenja:

1. Korisćenjem prvo drugog pravila:

$$\mathcal{I}_1 : E \xrightarrow{2} E + E \xrightarrow{1} a + E \xrightarrow{3} a + E * E \xrightarrow{1} a + a * E \xrightarrow{1} a + a * a$$

Stablo izvođenja koje se dobija na osnovu \mathcal{I}_1 je

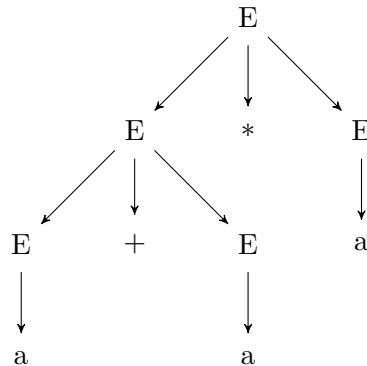


Izračunavanje izraza $a + a * a$ se interpretira kao $a + (a * a)$.

2. Korisćenjem prvo trećeg pravila:

$$\mathcal{I}_2 : E \xrightarrow{3} E * E \xrightarrow{2} E + E * E \xrightarrow{1} a + E * E \xrightarrow{1} a + a * E \xrightarrow{1} a + a * a$$

Stablo izvođenja koje se dobija na osnovu \mathcal{I}_2 je



Izračunavanje izraza $a + a * a$ se interpretira kao $(a + a) * a$.

Pravila o asocijativnosti operatora ne mogu razrešiti ovu dvosmislicu. Zbog toga se uvodi pojam *prioriteta operatora*. Kažemo da je operator $*$ višeg prioriteta u odnosu na operator $+$ jer se primenjuje na svoje operande pre nego $+$.

Poštovanjem matematičkog dogovora da operacija množenja ima viši prioritet nad operacijom sabiranja, dolazimo do zaključka da je stablo izvođenja dobijeno na osnovu izvođenja \mathcal{I}_1 jedino ispravno.

Sada smo uvereni da je višeznačnost gramatike nepogodna za sintaksičku analizu. Slično kao što smo nametanje asocijativnosti postigli odgovarajućom rekurzijom, nametanje prioriteta je moguće izvesti „razdvajanjem pravila na nivoe“. Ilustrujmo ovo neformalno pravilo sledećim primerom.

Primer 4.12 Aritmetičke izraze iz primera 4.11 možemo posmatrati kao niz sabiraka (kojih može biti jedan ili više) razdvojenih operacijom sabiranja. Svaki sabirak možemo posmatrati kao broj ili kao niz brojeva razdvojenih operacijom množenja. Prateći ovakvo posmatranje, definišemo pravila gramatike koju ćemo označiti G_{+*L} :

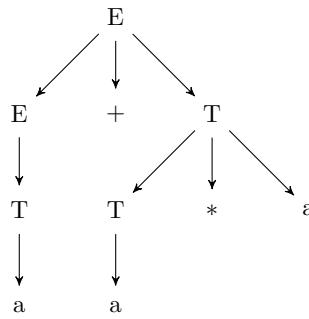
1. $E \rightarrow E + T$,
2. $E \rightarrow T$,
3. $T \rightarrow T * a$, i
4. $T \rightarrow a$.

Primetimo da gramatika G_{+*L} nije višeznačna. Njena pravila „razdvajaju operatore sabiranje i množenje na dva različita nivoa“ upravo da bi se postigao različiti prioritet tih operatora.

Izvođenje izraza $a + a * a$ u gramatici G_{+*L} je:

$$E \xrightarrow{1} E + T \xrightarrow{2} T + T \xrightarrow{4} a + T \xrightarrow{3} a + T * a \xrightarrow{4} a + a * a$$

Stablo izvođenja koje se dobija na osnovu ovog izvođenja je



Dopunimo gramatiku iz prethodnog primera tako da obuhvata sve četiri osnovne operacije nad njima.

Primer 4.13 Definišemo pravila gramatike koju ćemo označiti G_{op} :

$$\begin{array}{lcl} E & \longrightarrow & E + T \\ & | & E - T \\ & | & T \\ T & \longrightarrow & T * a \\ & | & T/a \\ & | & a \end{array}$$

Primetimo da, kako su operacije sabiranja i oduzimanja istog prioriteta, njihovi nivoi su jednaki. Isto važi za operacije množenja i deljenja. Dodatno, kako se koristi leva rekurzija u pravilima, svi operatori su leve asocijativnosti. Konačno, dopunjujemo gramatiku iz prethodnog primera tako da obuhvata i operatore zagrada.

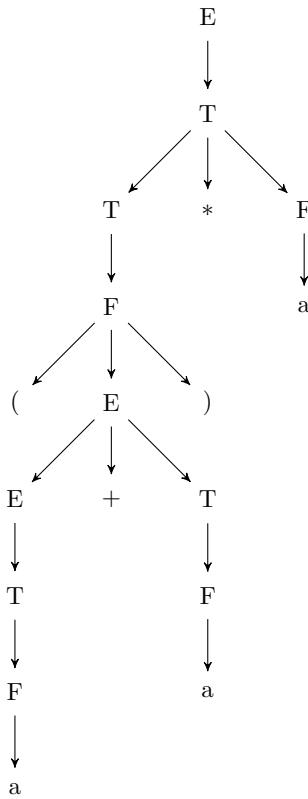
Primer 4.14 Definišemo pravila gramatika koju ćemo označiti G_{opz} :

$$\begin{array}{lcl} E & \longrightarrow & E + T \\ & | & E - T \\ & | & T \\ T & \longrightarrow & T * F \\ & | & T/F \\ & | & F \\ F & \longrightarrow & a \\ & | & (E) \end{array}$$

Izvođenje izraza $(a+a)*a$ u gramatici G_{opz} je:

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

Stablo izvođenja koje se dobija na osnovu ovog izvođenja je



Navedimo i dva primera koja ilustruju gramatike koje sadrže unarne operatore.

Primer 4.15 Gramatika iskaznih formula G_{if} je data pravilima:

$$\begin{array}{lcl}
 F & \longrightarrow & F \vee K \\
 & | & \\
 & K & \\
 K & \longrightarrow & K \wedge L \\
 & | & \\
 & L & \\
 L & \longrightarrow & \sim L \\
 & | & \\
 & A & \\
 A & \longrightarrow & p \\
 & | & \\
 & (F) &
 \end{array}$$

Izraz $p \wedge (\sim p \vee p)$ u gramatici G_{if} se izvodi na sledeći način:

$$\begin{array}{cccc}
 F \Rightarrow K & \Rightarrow K \wedge L & \Rightarrow L \wedge L & \Rightarrow A \wedge L \\
 \Rightarrow p \wedge L & \Rightarrow p \wedge A & \Rightarrow p \wedge (F) & \Rightarrow p \wedge (F \vee K) \\
 \Rightarrow p \wedge (K \vee K) & \Rightarrow p \wedge (L \vee K) & \Rightarrow p \wedge (\sim L \vee K) & \Rightarrow p \wedge (\sim A \vee K) \\
 \Rightarrow p \wedge (\sim p \vee K) & \Rightarrow p \wedge (\sim p \vee L) & \Rightarrow p \wedge (\sim p \vee A) & \Rightarrow p \wedge (\sim p \vee p)
 \end{array}$$

Primer 4.16 Gramatika koja predstavlja fragment gramatike programskog jezika C u kojoj učestvuju neki unarni operatori G_{Cun} je data pravilima:

$$\begin{array}{lcl} E & \longrightarrow & *E \\ & | & P \\ P & \longrightarrow & P++ \\ & | & A \\ A & \longrightarrow & id \\ & | & (E) \end{array}$$

Izraz $*(id)++$ se izvodi u gramatici G_{Cun} na sledeći način:

$$\begin{aligned} E &\Longrightarrow *E && \Longrightarrow *P && \Longrightarrow *P++ && \Longrightarrow *A++ \\ &&& \Longrightarrow *(E)++ && \Longrightarrow *(P)++ && \Longrightarrow *(A)++ \Longrightarrow *(id)++ \end{aligned}$$

Naredni primer ilustruje primer razrešavanja višezačnosti u gramatici.

Primer 4.17 Gramatika koja prepoznaje uslovnu naredbu **if** može biti predstavljena narednim pravilima:

$$\begin{array}{lcl} S & \longrightarrow & e; \\ & | & \{ L \} \\ & | & \text{if}(e) S \\ & | & \text{if}(e) S \text{ else } S \\ L & \longrightarrow & LS \\ & | & \epsilon \end{array}$$

Ova gramatika je višezačna, jer se rečenica **if**(e) **if**(e) $e;$ **else** $e;$ može izvesti na dva načina¹. Ovo se rešava uvodenjem novih pomoćnih simbola U i N koji određuju „uparenost“ uslovne naredbe **if** (U predstavlja „uparenu“, dok N predstavlja „neuparenu“ naredbu), na sledeći način:

$$\begin{array}{lcl} S & \longrightarrow & U \\ & | & N \\ N & \longrightarrow & \text{if}(e) S \\ & | & \text{if}(e) U \text{ else } N \\ U & \longrightarrow & e; \\ & | & \{ L \} \\ & | & \text{if}(e) U \text{ else } U \end{array}$$

U ovoj gramatici rečenica **if**(e) **if**(e) $e;$ **else** $e;$ se može izvesti na samo jedan način:

$$\begin{aligned} S &\Longrightarrow N \Longrightarrow \text{if}(e) S \Longrightarrow \text{if}(e) U \Longrightarrow \text{if}(e) \text{ if}(e) U \text{ else } U \\ &\qquad\qquad\qquad \Longrightarrow^* \text{if}(e) \text{ if}(e) e; \text{ else } e; \end{aligned}$$

4.5 Transformacije KS gramatika

Ovaj deo započinjemo navođenjem nekoliko osnovnih definicija.

Definicija 4.5.1 — Neproduktivan simbol. Simbol $X \in \Sigma \cup N$ je *neproduktivan* u KS gra-

¹Pokušajte da pronadete ova dva izvođenja za vežbu.

matici $G = (\Sigma, N, P, S)$ ako ne postoji izvođenje oblika:

$$S \longrightarrow^* uXv \longrightarrow^* uxv$$

gde su $u, v, x \in \Sigma^*$. Simbol koji nije neproduktivan je *produktivan*.

Definicija 4.5.2 — Nedostižan simbol. Simbol $X \in \Sigma \cup N$ je *nedostižan* u KS gramatici $G = (\Sigma, N, P, S)$ ako ne učestvuje ni u jednoj rečeničnoj formi gramatike G . Simbol koji nije nedostižan je *dostižan*.

Prema definiciji, neproduktivni simboli su ili nedostižni simboli iz $\Sigma \cup N$, ili pomoćni simboli iz kojih se ne izvodi nijedna završna niska iz Σ^* .

Definicija 4.5.3 — Čista gramatika. Za gramatiku $G = (\Sigma, N, P, S)$ kažemo da je *čista po simbolu S* ako su ispunjeni naredni uslovi:

1. Svaki pomoćni simbol učestvuje u izvođenju neke niske završnih simbola.
2. Svaki pomoćni simbol je dostižan iz početnog simbola S .

Svaka KS gramatika se može efektivno svesti na ekvivalentnu čistu gramatiku.

Definicija 4.5.4 — ϵ -slobodna gramatika. Gramatika $G = (\Sigma, N, P, S)$ je *ϵ -slobodna* ako je ispunjen jedan od narednih uslova:

1. U skupu pravila P ne postoji ϵ -pravilo.
2. Postoji pravilo $S \longrightarrow \epsilon$ u skupu pravila P , pri čemu S ne učestvuje na desnoj strani nijednog pravila u P .

Definicija 4.5.5 — Jednostruko pravilo. Za pravilo $X \longrightarrow Y$, gde su $X, Y \in N$ se kaže da je *jednostruko* pravilo.

Definicija 4.5.6 — Svojstvena gramatika. Gramatika $G = (\Sigma, N, P, S)$ je *svojstvena* ako je ϵ -slobodna i bez jednostrukih pravila.

Za datu KS gramatiku može se efektivno konstruisati ekvivalentna svojstvena KS gramatika. U daljem tekstu razmatramo procedure za konstruisanje čistih i svojstvenih gramatika.

4.5.1 Eliminacija neproduktivnih simbola

Procedura za eliminaciju neproduktivnih simbola se vrši u dva koraka:

1. U prvom koraku se eliminisu oni pomoćni simboli X iz kojih se ne može izvesti nijedna reč koja se sastoji od završnih simbola. Eliminacija se vrši na osnovu sledećih rekurzivnih definicija:
 - (a) Simbol $X \in N$ je produktivan ako u gramatici postoji bar jedno pravilo oblika $X \longrightarrow x$ i $x \in \Sigma^*$.
 - (b) Simbol $X \in N$ je produktivan ako postoji bar jedno pravilo takvo da je $X \Longrightarrow \alpha$, a α se sastoji samo od produktivnih simbola iz N i simbola iz Σ .

Simboli koji nisu produktivni se uklanaju iz skupa N zajedno sa pravilima iz skupa P u kojima oni učestvuju i tako se dobija novi skup pomoćnih simbola N' i pravila P' . Ovako modifikovana gramatika je ekvivalentna polaznoj gramatici.

2. U drugom koraku se eliminišu nedostižni simboli iz $\Sigma \cup N'$. Eliminacija se vrši na osnovu sledećih rekurzivnih definicija:
- Simbol $S \in N'$ je dostižan simbol.
 - Simboli koji se pojavljuju na desnoj strani pravila dostižnih simbola su dostižni simboli.

Simboli koji se ne identifikuju u ovom koraku su nedostižni, pa se uklanjuju iz skupa N' zajedno sa pravilima iz skupa P' u kojima oni učestvuju i tako se dobija novi skup pomoćnih simbola N'' i pravila P'' . Ovako modifikovana gramatika je ekvivalentna polaznoj gramatici.

Primer 4.18 Neka je zadata gramatika G nad $\Sigma = \{a, b, c, d\}$, gde je $N = \{S, A, B, C, D, E, F\}$, S je aksioma, a skup pravila P :

$$\begin{array}{ll} S & \longrightarrow A \\ A & \longrightarrow aB \quad | \quad bA \\ B & \longrightarrow cD \quad | \quad E \\ C & \longrightarrow CA \quad | \quad Bd \\ D & \longrightarrow a \quad | \quad b \\ E & \longrightarrow dE \quad | \quad Ed \\ F & \longrightarrow Ca \quad | \quad Fb \end{array}$$

Potrebno je da ispratimo korake opisane procedure kako bismo eliminisale neproduktivne simbole. Na osnovu koraka (1a) dolazimo do zaključka da je samo simbol D produktivan (zbog postojanja pravila $D \rightarrow a$ ili $D \rightarrow b$). Na osnovu koraka (1b) se za produktivne simbole identificuje prvo B (na osnovu pravila $B \rightarrow cD$), zatim A, C (na osnovu pravila $A \rightarrow aB$ i $C \rightarrow Ba$) i konačno S, F (na osnovu pravila $S \rightarrow A$ i $F \rightarrow Ca$). Nakon koraka 1 dobijamo $N' = \{S, A, B, C, D, F\}$, a skup pravila P' postaje:

$$\begin{array}{ll} S & \longrightarrow A \\ A & \longrightarrow aB \quad | \quad bA \\ B & \longrightarrow cD \\ C & \longrightarrow CA \quad | \quad Bd \\ D & \longrightarrow a \quad | \quad b \\ F & \longrightarrow Ca \quad | \quad Fb \end{array}$$

Na osnovu koraka (2a), simbol S je dostižan. Na osnovu koraka (2b) se za dostižne simbole prvo identificuje simbol A (na osnovu pravila $S \rightarrow A$), zatim B (na osnovu pravila $A \rightarrow aB$) i konačno simbol D (na osnovu pravila $B \rightarrow cD$). Nakon koraka 2 dobijamo $N'' = \{S, A, B, D\}$, a skup pravila P'' postaje:

$$\begin{array}{ll} S & \longrightarrow A \\ A & \longrightarrow aB \quad | \quad bA \\ B & \longrightarrow cD \\ D & \longrightarrow a \quad | \quad b \end{array}$$

4.5.2 Eliminacija ϵ -pravila

Za potrebe opisa ove transformacije, uvedimo narednu definiciju.

Definicija 4.5.7 — Anulirajući simbol. Za simbol $X \in N$ gramatike G kažemo da je *anulirajući* ako $X \Rightarrow^* \epsilon$.

Za zadatu gramatiku G , procedura eliminacije ϵ -pravila se sastoji iz dva koraka: (1)

konstrukcije skupa $A(G) \subseteq N$ anulirajućih simbola i (2) transformacije pravila koja sadrže anulirajuće simbole. Procedura glasi:

1. Za datu KS gramatiku G bez neproduktivnih simbola, skup anulirajućih simbola $A(G)$, inicijalno prazan, dobija se primenom narednih rekurzivnih pravila:
 - (a) Simbol $X \in N$ je anulirajući ako je $X \rightarrow \epsilon \in P$. Svaki takav simbol X se dodaje u skup $A(G)$.
 - (b) Simbol $X \in N$ je anulirajući ako postoji bar jedno pravilo $X \rightarrow \alpha \in P$, gde su svi pomoćni simboli u niski α anulirajući ($\alpha \in N^*$).
2. Kada je određen skup $A(G)$, modifikuju se pravila gramatike G koja sadrže anulirajuće simbole tako što se u svakom pravilu $X \rightarrow \alpha \in P$ u niski α zamene anulirajući simboli praznom reči ϵ na sve moguće načine, a zatim se eliminisu ϵ -pravila.

KS gramatika koja se dobija kao rezultat ove transformacije je ekvivalentna polaznoj gramatici do na praznu reč.

Primer 4.19 Neka je zadata gramatika G nad $\Sigma = \{a, b\}$, gde je $N = \{S, A, B, C, D, E\}$, S je aksioma, a skup pravila P :

$S \rightarrow Aa$	$ $	Bba
$A \rightarrow aBB$	$ $	$CC \quad \quad aD$
$B \rightarrow aB$	$ $	b
$C \rightarrow CD$	$ $	$DE \quad \quad a$
$D \rightarrow aB$	$ $	$bBa \quad \quad \epsilon$
$E \rightarrow aD$	$ $	DD

Na osnovu koraka (1a), dobijamo da je $A(G) = \{D\}$, na osnovu pravila $D \rightarrow \epsilon$. Na osnovu koraka (1b), skup $A(G)$ se proširuje simbolom E (na osnovu pravila $E \rightarrow DD$), zatim C (na osnovu pravila $C \rightarrow DE$) i konačno A (na osnovu pravila $A \rightarrow CC$). Time dobijamo $A(G) = \{A, C, D, E\}$.

Na osnovu koraka (2) pronalazimo sva pravila u kojima sa desne strane učestvuju anulirajući simboli. Ta pravila zamenjujemo novim pravilima u svaki anulirajući simbol zamenjujemo praznom reči ϵ na sve moguće načine. U narednoj tabeli su data pronađena pravila, kao i nova pravila koja su na ovaj način kreirana:

Originalno pravilo	Nova pravila
$S \rightarrow Aa$	$S \rightarrow Aa, S \rightarrow a$
$A \rightarrow CC$	$A \rightarrow CC, A \rightarrow C$
$C \rightarrow CD$	$C \rightarrow CD, C \rightarrow C, C \rightarrow D$
$C \rightarrow DE$	$C \rightarrow DE, C \rightarrow D, C \rightarrow E$
$E \rightarrow aD$	$E \rightarrow aD, E \rightarrow a$
$E \rightarrow DD$	$E \rightarrow DD, E \rightarrow D$

Nakon dodavanja novih pravila u skup P i eliminacije svih ϵ -pravila, dobijamo ϵ -slobodnu gramatiku zadatu pravilima:

$S \rightarrow Aa$	$ $	$a \quad \quad Bba$
$A \rightarrow aBB$	$ $	$CC \quad \quad C \quad \quad aD \quad \quad a$
$B \rightarrow aB$	$ $	b
$C \rightarrow CD$	$ $	$C \quad \quad D \quad \quad DE \quad \quad E \quad \quad a$
$D \rightarrow aB$	$ $	bBa
$E \rightarrow aD$	$ $	$a \quad \quad DD \quad \quad D$

koja opisuje isti jezik, ali bez prazne reči. U skladu sa definicijom 4.5.4, gramatika se može modifikovati uvođenjem nove aksiome S' umesto simbola S i dodavanjem novog pravila:

$$S' \rightarrow S \mid \epsilon$$

Novonastala gramatika opisuje isti jezik kao i polazna gramatika.

4.5.3 Eliminacija jednostrukih pravila

Neka je data KS gramatika $G = (\Sigma, N, P, S)$. Prepostavimo da u gramatici G nema neproduktivnih simbola, kao ni ϵ -pravila. Postupak eliminacije jednostrukih pravila se svodi na postupak traženja svih izvođenja oblika

$$X \xrightarrow{*} Y, \text{ gde } X, Y \in N.$$

Procedura za pronalaženje ovih izvođenja se vrši rekurzivno polazeći od jednostrukih pravila:

1. Eliminisati sva pravila oblika $X \rightarrow X$.
2. Pronaći skup svih jednostrukih pravila J u tekućem skupu P . Za svako pravilo $X \rightarrow Y \in J$:
 - (a) Ukloniti pravilo $X \rightarrow Y$ iz skupa P .
 - (b) Pravilima koja na levoj strani imaju X dodaju se desne strane svih pravila koja nisu jednostruka, a koja na levoj strani imaju simbol Y .
3. Ponavlјati korak 2 sve dok $J \neq \emptyset$.

Ovako transformisana gramatika može imati neproduktivnih simbola, koje treba eliminisati. Dobijena gramatika je ekvivalentna polaznoj gramatici, a nema neproduktivnih simbola, ϵ -pravila niti jednostrukih pravila.

Primer 4.20 Uzmimo ϵ -slobodnu gramatiku iz primera 4.19. Nakon trivijalnog koraka (1), gramatika koja se dobija ima skup pravila P :

$S' \rightarrow S$	$ $	ϵ
$S \rightarrow Aa$	$ $	$a \quad \quad Bba$
$A \rightarrow aBB$	$ $	$CC \quad \quad C \quad \quad aD \quad \quad a$
$B \rightarrow aB$	$ $	b
$C \rightarrow CD$	$ $	$D \quad \quad DE \quad \quad E \quad \quad a$
$D \rightarrow aB$	$ $	bBa
$E \rightarrow aD$	$ $	$a \quad \quad DD \quad \quad D$

Na osnovu ovog skupa pravila P formiramo skup $J = \{S' \rightarrow S, A \rightarrow C, C \rightarrow D, C \rightarrow E, E \rightarrow D\}$. Nakon koraka (2a) svako pravilo iz ovog skupa eliminisemo iz skupa P . U koraku (2b) za svako pravilo iz skupa J dodajemo nova pravila na osnovu opisane procedure (pravila koja se ponavlјaju, naravno, nećemo duplirati u rezultujućem skupu P). Pravila koja se dodaju su data narednom tabelom:

Eliminisano pravilo iz J	Nova pravila koja se dodaju u skup P
$S' \rightarrow S$	$S' \rightarrow Aa \mid a \mid Bba$
$A \rightarrow C$	$A \rightarrow CD \mid D \mid DE \mid E \mid a$
$C \rightarrow D$	$C \rightarrow CD \mid D \mid DE \mid E \mid a \mid aB \mid bBa$
$C \rightarrow E$	$C \rightarrow aD \mid a \mid DD \mid D$
$E \rightarrow D$	$E \rightarrow aB \mid bBa$

Gramatika koja je nastala nakon koraka (2) ima skup pravila P :

$S' \rightarrow \epsilon$	$ $	Aa	$ $	a	$ $	Bba
$S \rightarrow Aa$	$ $	a	$ $	Bba		
$A \rightarrow aBB$	$ $	CC	$ $	aD	$ $	a
$B \rightarrow aB$	$ $	b				
$C \rightarrow CD$	$ $	DE	$ $	a	$ $	CD
$D \rightarrow aB$	$ $	bBa			$ $	DE
$E \rightarrow aD$	$ $	a	$ $	DD	$ $	aB
					$ $	bBa

Na osnovu ovog skupa pravila P formiramo skup $J = \{A \rightarrow D, A \rightarrow E, C \rightarrow D, C \rightarrow E\}$. Ponovo, primenom koraka (2a) svako pravilo iz ovog skupa eliminisemo iz skupa P . U koraku (2b) za svako pravilo iz skupa J dodajemo nova pravila koja su data narednom tabelom:

Eliminisano pravilo iz J	Nova pravila koja se dodaju u skup P
$A \rightarrow D$	$A \rightarrow aB \mid bBa$
$A \rightarrow E$	$A \rightarrow aD \mid a \mid DD \mid aB \mid bBa$
$C \rightarrow D$	$C \rightarrow aB \mid bBa$
$C \rightarrow E$	$E \rightarrow aD \mid a \mid DD \mid aB \mid bBa$

Gramatika koja je nastala nakon koraka (2) ima skup pravila P :

$S' \rightarrow \epsilon$	$ $	Aa	$ $	a	$ $	Bba
$S \rightarrow Aa$	$ $	a	$ $	Bba		
$A \rightarrow aBB$	$ $	CC	$ $	aD	$ $	a
$B \rightarrow aB$	$ $	b				
$C \rightarrow CD$	$ $	DE	$ $	a	$ $	CD
$D \rightarrow aB$	$ $	bBa			$ $	DE
$E \rightarrow aD$	$ $	a	$ $	DD	$ $	aB
					$ $	bBa

Na osnovu ovog skupa pravila P formiramo skup $J = \emptyset$. Procedura je završena i dobijena je gramatika bez jednostrukih pravila koja je ekvivalentna polaznoj gramatici.

4.5.4 Eliminacija leve rekurzije

Ranije smo uveli pojam levo rekurzivnog pravila. Ispostavlja se da ovakva pravila mogu biti nepodesna za neke metode sintaksičke analize, te bi bilo dobro oslobođiti se takvih pravila. Za ovu proceduru će nam biti neophodna naredna definicija.

Definicija 4.5.8 — Imenovano pravilo. Pravilo neke KS gramatike G oblika $A \rightarrow \alpha$ ($A \in N$, $\alpha \in (\Sigma \cup N)^*$) naziva se *A-pravilo*.

Naredna teorema i prateći dokaz daju postupak za eliminaciju neposredne leve rekurzije, kao i garanciju da taj postupak neće uticati na jezik koji je opisan novonastalom gramatikom.

Teorema 4.5.1 Za svaki KS jezik bez prazne reči postoji čista gramatika G bez jednostrukih pravila i bez levo rekurzivnih pravila koja generiše taj jezik.

Dokaz. Pretpostavimo da je KS gramatika $G = (\Sigma, N, P, S)$ čista gramatika bez jednostrukih pravila. Neka je $A \in N$ pomoći simbol koji je levo rekurzivan. Tada je A -pravila mogu podeliti na pravila koja sadrže levu rekurziju i na pravila koja je ne sadrže:

$$\begin{aligned} A &\longrightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \\ A &\longrightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

gde niske oblike β_i ne počinju simbolom A . U skup pomoćnih simbola N dodajemo novi simbol B , a gornja A -pravila zamenjujemo pravilima:

$$\begin{aligned} A &\longrightarrow \beta_1B \mid \beta_2B \mid \dots \mid \beta_nB \\ B &\longrightarrow \alpha_1B \mid \alpha_2B \mid \dots \mid \alpha_mB \mid \epsilon \end{aligned}$$

Ovako transformisana gramatika G' generiše isti jezik kao i gramatika G . Kako se B -pravila koriste jedino kao rezultat primene nekog A -pravila, za dokaz je dovoljno uočiti da A -pravila i u polaznoj i u transformisanoj gramatici opisuju na kraju izvođenja isti skup reči. Svakoj primeni A -pravila u izvođenju nalevo u gramatici G odgovara u gramatici G' primena jednog A -pravila i nekog broja B -pravila u izvođenju nadesno. ■



Teorema 4.5.1 se može dokazati i pod nešto tvrdim uslovima. Moguće je dokazati da će teorema važiti pod uslovima da je gramatika G svojstvena. Dokaz je sličan onom koji smo prikazali, sa određenim izmenama u proceduri eliminacije pravila i dodavanja novih pravila. Međutim, ukoliko bismo koristili tu proceduru, dobili bismo veći broj pravila. Činjenica da se uvode ϵ -pravila procedurom koja je opisana iznad, skup novih pravila koji se dodaju je dvostruko manji od onog koji se dobija procedurom za dobijanje svojstvene gramatike G . Za više informacija, pogledati Vitas 2006.

Primer 4.21 Gramatika G_{+*z} aritmetičkih izraza sa operacijama sabiranja i množenja i zagradama je data pravilima:

$$\begin{array}{lcl} E & \longrightarrow & E + T \quad | \quad T \\ T & \longrightarrow & T * F \quad | \quad F \\ F & \longrightarrow & (E) \quad | \quad b \end{array}$$

Iz ovog skupa pravila P primećujemo da su E -pravila i T -pravila levo rekurzivna. Uvedimo nove pomoćne simbole E' i T' . U narednoj tabeli su data levo rekurzivna pravila koja se eliminisu, zajedno sa pravilima koja se dodaju u skup pravila P , na osnovu procedure opisane u dokazu teoreme 4.5.1.

Eliminisana pravila	Novouvedena pravila
$E \longrightarrow E + T \mid T$	$E \longrightarrow TE', E' \longrightarrow +TE' \mid \epsilon$
$T \longrightarrow T * F \mid F$	$T \longrightarrow FT', T' \longrightarrow *FT' \mid \epsilon$

Gramatika koja se dobija kao rezultat nema leva rekurzivna pravila:

$$\begin{array}{l}
 E \rightarrow TE' \\
 E' \rightarrow +TE' \mid \epsilon \\
 T \rightarrow FT' \\
 T' \rightarrow *FT' \mid \epsilon \\
 F \rightarrow (E) \mid b
 \end{array}$$

Eliminacija posredne leve rekurzije

Procedura opisana dokazom teoreme 4.5.1 ne eliminiše levu rekurziju koja povlači za sobom izvođenje u dva ili više koraka. Na primer, razmotrimo narednu gramatiku:

$$\begin{array}{l}
 S \rightarrow Aa \mid b \\
 A \rightarrow Ac \mid Sd
 \end{array}$$

Za pomoći simbol S kažemo da je *posredno levo rekurzivan* zato što postoji izvođenje

$$S \Rightarrow Aa \Rightarrow Sda$$

koje u dva ili više koraka izvodi simbol S u nisku čiji je najlevlji simbol takođe simbol S . Ipak, postoji procedura za eliminisanje i ovakvih rekurzija u gramatikama.

Procedura za eliminaciju posredne leve rekurzije je data narednim koracima:

1. Urediti sve pomoći simbole u neki poredak A_1, A_2, \dots, A_n .
2. Za svaki pomoći simbol $A_i, i \in \{1, \dots, n\}$ uraditi sledeće:
 - (a) Za svaki pomoći simbol $A_j, j \in \{1, \dots, i-1\}$ pronaći sva pravila oblika

$$A_i \rightarrow A_j \gamma$$

- (b) Zameniti pronađena pravila iz prethodnog koraka pravilima oblika

$$A_i \rightarrow \delta_1 \gamma \mid \dots \mid \delta_k \gamma$$

pri čemu su $A_j \rightarrow \delta_1 \mid \dots \mid \delta_k$ sva tekuća A_j -pravila.

- (c) Eliminisati neposrednu levu rekurziju među A_i -pravilima procedurom iz dokaza teoreme 4.5.1.

Teorema koju navodimo govori o uslovima za garanciju primene ove procedure.

Teorema 4.5.2 Primena opisane procedure za eliminaciju posredne leve rekurzije nad KS gramatikom G proizvodi gramatiku G' bez pravila koja proizvode bilo posrednu ili neposrednu rekurziju, a koja opisuje isti jezik kao i početna gramatika G ako gramatika G nema ciklusa (tj. izvođenja oblika $A \Rightarrow^+ A$) ili ϵ -pravila.

Dokaz. Primeničemo tehniku potpune matematičke indukcije da bismo dokazali teoremu. Za bazu indukcije biramo slučaj $i = 1$. U toj iteraciji procedure, korakom (2c) biće eliminisane sve neposredne leve rekurzije među svim A_1 -pravilima. Sva preostala pravila oblika $A_1 \rightarrow A_l \alpha$ moraju da zadovoljavaju $l > 1$. U induktivnoj hipotezi, nakon $i - 1$ iteracija procedure, svi pomoći simboli A_k , gde je $k < i$ su „očišćeni”, tj. za sva pravila oblika $A_k \rightarrow A_l \alpha$ mora da važi $l > k$. U i -toj iteraciji procedure, koracima (2a) i (2b) progresivno se spušta donja granica proizvoljnog pravila oblika $A_i \rightarrow A_m \alpha$ sve dok ne postane $m \geq i$. Tada, eliminacijom direktnе leve rekurzije za A_i pravila u koraku (2c) „forsiraju” m da bude veće od i . ■

Primer 4.22 Razmotrimo narednu gramatiku koja sadrži i neposrednu i posrednu levu rekurziju:

$$\begin{aligned} S &\longrightarrow Sa \mid Ab \mid c \mid d \\ A &\longrightarrow Aa \mid Sbd \mid Sc \end{aligned}$$

Želimo da primenimo proceduru za eliminiranje posredne leve rekurzije. Uvidećemo da će procedura takođe eliminisati i neposrednu levu rekurziju na osnovu teoreme 4.5.2.

U koraku (1) je potrebno da ustanovimo poređenje pomoćnih simbola. Neka to bude poređenje $[S, A]$, odnosno, $A_1 = S, A_2 = A$.

U iteraciji $i = 1$ koraka (2), indeks j neće uzimati nijednu vrednost, tako da se koraci (2a) i (2b) preskaču. U koraku (2c) eliminišemo pravila

$$\begin{aligned} S &\longrightarrow Sa \\ S &\longrightarrow Ab \mid c \mid d \end{aligned}$$

i uvodimo nova pravila

$$\begin{aligned} S &\longrightarrow AbS' \mid cS' \mid dS' \\ S' &\longrightarrow aS' \mid \epsilon \end{aligned}$$

Trenutni skup pravila u gramatici je:

$$\begin{aligned} S &\longrightarrow AbS' \mid cS' \mid dS' \\ S' &\longrightarrow aS' \mid \epsilon \\ A &\longrightarrow Aa \mid Sbd \mid Sc \end{aligned}$$

U iteraciji $i = 2$ koraka (2), indeks j uzima redom vrednosti iz skupa $\{1\}$. Za svaku vrednost iz skupa (dakle, za samo jednu vrednost u ovom slučaju), ponavljamo korake (2a) i (2b). Za $j = 1$, u koraku (2a) pronalazimo skup pravila

$$A \longrightarrow Sbd \mid Sc$$

Pravila iz ovog skupa eliminišemo i umesto njih, u koraku (2b) uvodimo pravila

$$A \longrightarrow cS'bd \mid cS'c \mid dS'bd \mid dS'c \mid AbS'bd \mid AbS'Sc$$

U koraku (2c) eliminišemo pravila

$$\begin{aligned} A &\longrightarrow Aa \mid AbS'bd \mid AbS'Sc \\ A &\longrightarrow cS'bd \mid cS'c \mid dS'bd \mid dS'c \end{aligned}$$

i umesto njih uvodimo pravila

$$\begin{aligned} A &\longrightarrow cS'bdA' \mid cS'cA' \mid dS'bdA' \mid dS'cA' \\ A' &\longrightarrow aA' \mid bS'bdA' \mid bS'ScA' \mid \epsilon \end{aligned}$$

Kako smo icrpeli sve iteracije za indeks i , procedura je završena. Rezultujuća gramatika se sastoji od pravila:

$$\begin{aligned} S &\longrightarrow AbS' \mid cS' \mid dS' \\ S' &\longrightarrow aS' \mid \epsilon \\ A &\longrightarrow cS'bdA' \mid cS'cA' \mid dS'bdA' \mid dS'cA' \\ A' &\longrightarrow aA' \mid bS'bdA' \mid bS'ScA' \mid \epsilon \end{aligned}$$

4.5.5 Leva faktorizacija

Korisna transformacija gramatike je *leva faktorizacija*. Ako u gramatici postoje pravila nekog pomoćnog simbola čije desne strane počinju istim prefiksima, onda je takva pravila moguće formulisati na takav način da se ovaj nedostatak otkloni.

Procedura leve faktorizacije se sastoji u sledećem: neka je $\alpha \neq \epsilon$ pravi levi faktor za neke desne strane A -pravila. Tada A -pravila imaju oblik:

$$A \longrightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_n \mid \gamma,$$

gde su γ sve desne strane koje nemaju prefiks α . Uvedimo novi pomoćni simbol A' i gornja A -pravila zamenimo pravilima:

$$\begin{aligned} A &\longrightarrow \alpha A' \mid \gamma \\ A' &\longrightarrow \beta_1 \mid \dots \mid \beta_n \end{aligned}$$

Ovaj postupak se ponavlja sve dok nikoje dve desne strane nekog pravila nemaju zajednički prefiks.

Primer 4.23 Posmatrajmo gramatiku sa pravilima:

$$\begin{aligned} S &\longrightarrow cAd \\ A &\longrightarrow ab \mid a \end{aligned}$$

Prateći proceduru za levu faktorizaciju, gramatika koja se dobija na osnovu prethodnih pravila je data novim pravilima:

$$\begin{aligned} S &\longrightarrow cAd \\ A &\longrightarrow aA' \\ A' &\longrightarrow b \mid \epsilon \end{aligned}$$

5. Potisni automati

5.1 Osnovni pojmovi

Posmatrajmo konačan automat \mathcal{M} koji prihvata jezik

$$\mathcal{L}_1(\mathcal{M}) = \{a^m b^n \mid m, n \geq 1\}.$$

Konačni automat \mathcal{M} se može konstruisati tako da se pomera iz početnog stanja q_0 u stanje q_1 kada se u ulaznoj struji primeti a . Nakon što se primeti b , \mathcal{M} se pomera iz stanja q_1 u stanje q_2 i nastavlja da bude u tom stanju sve dok dobija b iz ulazne struje. U slučaju da je data niska $a^m b^n$, rezultujuće stanje konačnog automata \mathcal{M} je završno stanje, tako da automat prihvata nisku $a^m b^n$.

Posmatrajmo sada jezik $\mathcal{L}_2(\mathcal{M}) = \{a^n b^n \mid n \geq 1\}$ u kojem niske imaju jednak broj simbola a i b . KA koji smo konstruisali za jezik \mathcal{L}_1 razlikuje se od onog koji treba da se konstruiše za jezik \mathcal{L}_2 . Zašto?

Za jezik $\mathcal{L}_1(\mathcal{M})$ ne postoji potreba za pamćenjem broja pročitanih simbola a . Sve što treba biti zapamćeno je:

- Da li je prvi simbol b ? (Da bi se odbacila niska.)
- Da li simbolu b sledi simbol a ? (Da bi se odbacila niska.)
- Da li je simbolu a sledi simbol a i simbolu b sledi simbol b ? (Da bi se prihvatile niske.)

Dodatno, poznato nam je da KA ima samo konačan broj stanja. Automat \mathcal{M} ne može da pamti broj simbola a u niski $a^n b^n$ gde je n veće od broja stanja u automatu \mathcal{M} . Da bismo ovo uspeli da uradimo, potrebno je da uvedemo novi model automata.

Definicija 5.1.1 — NPA. Nedeterministički potisni automat, skr. NPA (engl. *nondeterministic pushdown automata*, skr. *NPDA*) formalno se definiše kao uređena 7-orka:

$$\mathcal{A} = (\Sigma, Q, \Gamma, I, Z_0, K, \Delta)$$

gde je:

- Σ ulazna azbuka,
- Q azbuka stanja automata,
- Γ azbuka potisne liste,
- $I \subseteq Q$ skup početnih stanja,
- $Z_0 \in \Gamma$ početni simbol potisne liste,
- $K \subseteq Q \times \Gamma^*$ skup završnih konfiguracija, a
- $\Delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times \Gamma \times Q \times \Gamma^*$ skup pravila prelaza.

Potisna lista (engl. *stack*) predstavlja dodatnu komponentu koja je dostupna kao deo NPA koja ima ulogu povećanja njegove memorije. Ako se ponovo vratimo na jezik $\mathcal{L}_2(\mathcal{M}) = \{a^n b^n \mid n \geq 1\}$, problem koji smo imali kod konstrukcije KA bismo mogli da prevaziđemo skladištenjem simbola a u potisnoj listi. Kadase primeti simbol b na ulaznoj struji, onda se sa vrha potisne liste uklanja simbol a . Ukoliko je potisna lista prazna na kraju procesiranja niske, onda NPA prihvata datu nisku.

Obratimo nešto detaljniju pažnju na skup pravila prelaza NPA.

Definicija 5.1.2 — Pravilo, ϵ -pravilo. Za dati NPA \mathcal{A} petorka $(q, a, Z, r, \gamma) \in \Delta$ se naziva pravilo. Ako je $a = \epsilon$, onda se naziva ϵ -pravilo.

Prve tri komponente pravila predstavljaju preduslov, a poslednje dve postuslov ponašanja automata \mathcal{A} . Otuda se Δ obično posmatra kao preslikavanje

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \longrightarrow \text{konačni podskupovi od } Q \times \Gamma^*$$

takvo da

$$(r, \gamma) \in \delta(q, a, Z) \iff (q, a, Z, r, \gamma) \in \Delta.$$

Argumenti ovog preslikavanja su (1) stanje, (2) bilo ϵ bilo simbol iz ulazne azbuke i (3) simbol na vrhu potisne liste. Baš kao što se ulazni simbol „konzumira“ pri izvršavanju preslikavanja, tako se simbol iz potisne liste „konzumira“ (uklanja iz potisne liste).

Primetimo da iako drugi argument može biti ϵ umesto nekog simbola iz ulazne azbuke (tako da se nijedan ulazni simbol ne konzumira), ne postoji takva opcija za treći argument. Drugim rečima, preslikavanje δ uvek konzumira simbol iz potisne liste. Posledica toga je da ne postoji korak u automatu ako je potisna lista prazna.

Primer 5.1 Posmatrajmo naredni skup pravila prelaza NPA zadat preslikavanjem:

$$\delta(q_1, a, b) = \{(q_2, cd), (q_3, \epsilon)\}.$$

Ako je u bilo kom trenutku automat u stanju q_1 , na ulaznoj struji je pročitan simbol a i simbol na vrhu potisne liste je b , tada je moguć jedan od naredna dva slučaja:

1. Automat prelazi u stanje q_2 i niska cd zamenjuje b na vrhu potisne liste.

2. Automat prelazi u stanje q_3 i simbol ϵ zamenjuje b na vrhu steka (drugim rečima, simbol b je uklonjen sa vrha steka).

Ovaj pojam prelaska potisnog automata se može formalizovati pojmom konfiguracije i relacijom prelaska.

Definicija 5.1.3 — Konfiguracija. *Konfiguracija* potisnog automata \mathcal{A} je uređena trojka

$$(q, w, \gamma) \in Q \times \Sigma^* \times \Gamma^*,$$

gde je w reč koju će automat pročitati, a (a, γ) *unutrašnja konfiguracija* automata (prvi karakter niske γ je na vrhu potisne liste).

Definicija 5.1.4 — Relacija prelaska. *Relacija prelaza*, u oznaci \vdash , predstavlja relaciju među konfiguracijama definisanu na sledeći način: ako su $k = (q, aw, Z\alpha)$ i $k' = (r, w, \beta\alpha)$ dve konfiguracije, gde je $a \in \Sigma \cup \{\epsilon\}$, tada

$$k \vdash k' \text{ ako je } (r, \beta) \in \delta(q, a, Z).$$

Ako je $a = \epsilon$, prelaz se naziva ϵ -prelaz, a ako $a \in \Sigma$, kaže se da prelaz *uključuje čitanje* simbola a . Prelaz automata iz jedne konfiguracije u drugu sastoji se iz četiri dela: (1) skidanja simbola sa vrha potisne liste Z , (2) prenošenja na potisnu listu niske β , (3) čitanja simbola a sa ulazne struje i (4) prelaska iz stanja r u stanje q . Svakom upisivanju na potisnu listu, dakle, prethodni skidanje sa potisne liste. Prema ovoj konvenciji, automat se zaustavlja (blokira) kada je potisna lista prazna jer nema ničega što bi sa nje moglo biti skinuto.

Refleksivno i tranzitno zatvorene relacije prelaska se obeležava oznakom \vdash^* . S obzirom da relacija prelaza definiše način promene stanja automata, a da za dato stanje i slovo ulazne azbuke postoji više mogućih prelaza, PA je nedeterministički.

Za bilo koje $w \in \Sigma^*$, pojam konfiguracije se može svesti na relaciju među unutrašnjim konfiguracijama automata, u oznaci \models^w , definisanoj sa:

$$(q, \gamma) \models^w (q', \gamma') \iff (q, w, \gamma) \vdash^* (q', \epsilon, \gamma').$$

Među svim konfiguracijama, posebno govorimo o *početnim unutrašnjim konfiguracijama* automata koja se sastoje od početnog stanja i početnog simbola potisne liste, tj. (i, Z_0) , gde je $i \in I$. Po konvenciji, postoji tačno jedna ovakva konfiguracija. Za završne konfiguracije K postoji više mogućih izbora, koje se nazivaju *način prihvatanja* ili *prepoznavanja*. Postoje tri uobičajena načina da se odredi skup K :

1. $K = F \times \Gamma^*$, gde je $F \subseteq Q$, koji se naziva *skup završnih stanja*. Ovaj način prihvatanja se naziva *prihvatanje završnim stanjem*. Jezik prihvaćen ovim načinom prihvatanja obeležimo sa \mathcal{T} .
2. $K = Q \times \epsilon$. Ovaj način se naziva *prihvatanje praznom potisnom listom*, a odgovarajući jezik obeležimo sa \mathcal{N} .
3. $K = F \times \{\epsilon\}$, gde je $F \subseteq Q$. Ovaj način prihvatanja se naziva *prihvatanje završnim stanjem i praznom potisnom listom*. Jezik prihvaćen na ovaj način označimo sa \mathcal{J} .

Kažemo da je reč $w \in \Sigma^*$ *prihvaćena* PA $\mathcal{A} = (\Sigma, Q, \Gamma, I, Z_0, K, \Delta)$ ako za neko $k \in K$ (pri određenom načinu prihvatanja) važi $(i, Z_0) \models^w k$.

Definicija 5.1.5 — Jezik prihvaćen automatom. Uspešno izračunavanje. Blokiran automat.
Neka je $\mathcal{A} = (\Sigma, Q, \Gamma, I, Z_0, K, \Delta)$ PA, a K neki od načina prihvatanja. Tada je *jezik prihvaćen automatom* \mathcal{A} skup:

$$\mathcal{L}(\mathcal{A}, K) = \{w \in \Sigma^* \mid (i, Z_0) \models^w k, k \in K\}.$$

Kažemo tada da je automat \mathcal{A} obavio *uspešno izračunavanje*. Ako izračunavanje nije uspešno, kažemo da je automat *blokiran*.

N Prepozнати jezik zavisi od načina prihvatanja, te zbog toga jezik \mathcal{L} u definiciji 5.1.5 zavisi i od skupa K . Ako je jezik \mathcal{L} prihvaćen završnim stanjem, onda

$$\mathcal{T}(\mathcal{A}) = \mathcal{L}(\mathcal{A}, F \times \Gamma^*) = \{w \in \Sigma^* \mid (i, w, Z_0) \vdash^* (q, \varepsilon, \gamma), q \in F, \gamma \in \Gamma^*\}.$$

Ako je jezik \mathcal{L} prihvaćen praznom potisnom listom, onda

$$\mathcal{N}(\mathcal{A}) = \mathcal{L}(\mathcal{A}, Q \times \{\varepsilon\}) = \{w \in \Sigma^* \mid (i, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}.$$

Ako je jezik \mathcal{L} prihvaćen završnim stanjem i praznom potisnom listom, onda

$$\mathcal{J}(\mathcal{A}) = \mathcal{L}(\mathcal{A}, F \times \{\varepsilon\}) = \{w \in \Sigma^* \mid (i, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon), q \in F\}.$$

Može se pokazati da za svaki NPA \mathcal{A} koji prihvata jezik na jedan način prihvatanja postoji ekvivalentan NPA \mathcal{B} koji prihvata jezik na drugi način. Na osnovu te činjenice, moguće je govoriti o familiji jezika koje prepoznaju NPA bez obzira na način prihvatanja.

5.2 Primeri potisnih automata

PA se obično ne reprezentaciji vizualno, ali uz nekoliko dodatnih proširenja u odnosu na KA, možemo ih predstaviti grafom na sledeći način:

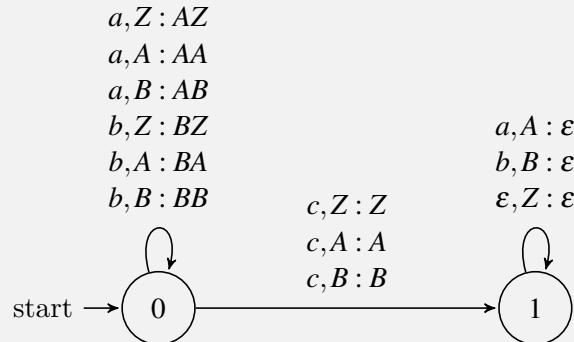
- Svakom stanju $q \in Q$ odgovara jedan čvor grafa.
- Ako $(r, \gamma) \in \delta(q, a, Z)$, tada postoji grana od čvora q do čvora r obeležena oznakom (a, Z, γ) , što se može zapisati i u obliku $a, Z : \gamma$.
- Ako automat prihvata završnim stanjem, onda se $f \in F$ obeležava kao kod konačnog automata.

Primer 5.2 Neka je dat PA $\mathcal{A} = (\Sigma, Q, \Gamma, I, Z_0, K, \Delta)$, gde je $\Sigma = \{a, b, c\}$, $Q = \{0, 1\}$, $I = \{0\}$, $\Gamma = \{Z, A, B\}$, $Z_0 = Z$. Dalje, neka automat prepoznae praznom potisnom listom, tj. $K = Q \times \{\varepsilon\}$ i neka je skup pravila dat preslikavanjima:

$$\begin{aligned}\delta(0, a, Z) &= \{(0, AZ)\}, \\ \delta(0, a, A) &= \{(0, AA)\}, \\ \delta(0, a, B) &= \{(0, AB)\}, \\ \delta(0, b, Z) &= \{(0, BZ)\}, \\ \delta(0, b, A) &= \{(0, BA)\}, \\ \delta(0, b, B) &= \{(0, BB)\},\end{aligned}$$

$$\begin{aligned}\delta(0,c,Z) &= \{(1,Z)\}, \\ \delta(0,c,A) &= \{(1,A)\}, \\ \delta(0,c,B) &= \{(1,B)\}, \\ \delta(1,a,A) &= \{(1,\varepsilon)\}, \\ \delta(1,b,B) &= \{(1,\varepsilon)\}, \\ \delta(1,\varepsilon,Z) &= \{(1,\varepsilon)\}\end{aligned}$$

Dati PA se može vizualno predstaviti na sledeći način:



Automat na slici prepoznaje praznom potisnom listom jezik $\mathcal{L} = \{wc\bar{w} \mid w \in \{a,b\}^*\}$, gde je oznakom \bar{w} označena slika u ogledalu niske w . Na primer, niz konfiguracija koji prepoznaje nisku $abcba \in \mathcal{L}$ je:

$$\begin{aligned}k_0 &= (0, abcba, Z) \vdash (0, bcba, AZ) \vdash (0, cba, BAZ) \\ &\quad \vdash (1, ba, BAZ) \vdash (1, a, AZ) \vdash (1, \varepsilon, Z) \\ &\quad \vdash (1, \varepsilon, \varepsilon).\end{aligned}$$

Primetimo da je ovaj automat deterministički: za svako stanje i simbol na vrhu potisne liste, dozvoljen je ili ε -prelaz ili, za svako slovo ulazne azbuke, najviše jedan prelaz.

5.3 Odnos PA sa KS jezicima

Značaj PA se ogleda u tome što PA predstavlja uređaj koji je sposoban da prepozna da li jedna niska pripada nekom KS jeziku ili ne. Formalizujmo ovo narednim teoremmama koje navodimo bez dokaza.

Teorema 5.3.1 Ako je \mathcal{L} jezik generisan nekom KS gramatikom, tada postoji PA \mathcal{A} koji prepozna jezik \mathcal{L} praznom potisnom listom.

Teorema 5.3.2 Ako PA \mathcal{A} prepozna jezik \mathcal{L} praznom potisnom listom, tada postoji KS gramatika G koja generiše jezik \mathcal{L} .

Dokazi ovih teorema se mogu pronaći u Vitas 2006.

5.3.1 Lema o razrastanju za KS jezike

Lema o razrastanju za regularne jezike nam je pokazala da niske dovoljno velike dužine u regularnom jeziku imaju podnisku koja se može ponoviti proizvoljan broj puta tako da rezultujuća niska i dalje bude sadržana u jeziku. Za KS jezike, efekat „razrastanja“ se odnosi na simultano ponavljanje sve podnische. Formalizaciju ovog utiska ostvarujemo navođenjem naredne teoreme.

Teorema 5.3.3 — Lema o razrastanju za KS jezike. Neka je \mathcal{L} KS jezik. Postoji broj k , zavisan od jezika \mathcal{L} , takav da proizvoljna niska $z \in \mathcal{L}$ za koju važi $|z| \geq k$ može biti zapisana u obliku $z = uvwxy$ tako da važi:

1. $|vwx| \leq k$.
2. $|v| + |x| > 0$
3. $uv^iwx^i y \in \mathcal{L}$ za sve $i \geq 0$.

Dokaz se može pronaći u Sudkamp 1988.

5.4 Deterministički KS jezici

Za razliku od konačnih automata, kod kojih se familije nedeterminističkih i determinističkih automata poklapaju, uvođenje determinizma u rad potisnog automata sužava klasu prepoznatih jezika na tzv. *determinističke (kontekstno slobodne) jezike*. Ovi jezici su pravi podskupa klase KS jezika, a pravi nadskup klase regularnih jezika. Ova klasa jezika je osetljivija od KS jezika jer ne postoji njene karakterizacije preko gramatika, kao ni leme analogne lemama o razrastanju za KS gramatike. Iz ugla sintakstičke analize, ova klasa predstavlja osnovnu relevantnu klasu jezika jer se za nju sintaksička analiza vrši u linearном vremenu. Štaviše, generatori sintaksičkih analizatora generišu determinističke potisne automate.

Definicija 5.4.1 — DPA. PA $\mathcal{A} = (\Sigma, Q, \Gamma, I, Z_0, K, \Delta)$ je *deterministički potisni automat*, skr. *DPA* ako za svako $(q, a, Z) \in Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ važi:

1. $\text{card}(\Delta(q, a, Z)) \leq 1$ i
2. $\Delta(q, \epsilon, Z) \neq \emptyset \implies \Delta(q, a, Z) = \emptyset$ za $a \in \Sigma$.

Suština definicije 5.4.1 se ogleda u tome da kod DPA, u bilo kojoj konfiguraciji moguće je napraviti najviše jedan prelaz za simbol iz ulazne struke (uključujući i ϵ -prelaz) i simbol sa vrha potisne liste (uslov 1 iz definicije). Takođe, DPA pravi prelaz ili po ϵ -prelazu ili po simbolu ulazne azbuke (uslov 2 iz definicije).

Moguće je pokazati da je familija jezika koju prepoznaće DPA praznom potisnom listom sadržana u familiji jezika prepoznatih završnim stanjem. S druge strane, jezik DPA prepoznat praznom potisnom listom je *prefiksni jezik*, tj. jezik kod koga nijedan pravi prefiks reči jezika ne pripada jeziku. Dakle, na osnovu ovoga, razlikujemo:

1. *Determinističke jezike*, koji se prepoznaju završnim stanjem i
2. *Determinističke prefiksne jezike*, koji se prepoznaju praznog potisnom listom (ili, ekvivalentno, praznom potisnom listom i završnim stanjem).

Zbog ove diskusije naredno tvrđenje izdvajamo kao teoremu da bismo sumirali ovu važnu razliku u odnosu na NPA.

Teorema 5.4.1 Svaki deterministički prefiksni KS jezik je i deterministički KS jezik, ali obrnuto ne važi.

Primer 5.3 Jezik $\mathcal{L}_1 = \{a^n b^p \mid p > n > 0\}$ je deterministički, ali nije prefiksni, za razliku od jezika $\mathcal{L}_2 = \{a^n b^n \mid n > 0\}$ koji je deterministički i prefiksni.

Kao što smo rekli, ne postoji karakterizacija determinističkih KS jezika preko gramatika. Jedini model za ove jezike su DPA. Otuda teorema 5.4.2 koja govori o determinizmu KS jezika i teorema 5.4.3 koja govori o odnosu determinističkih KS jezika i regularnih jezika. Teoreme dajemo bez dokaza.

Teorema 5.4.2 Neka je G KS gramatika. Problem da li je jezik $\mathcal{L} = \mathcal{L}(G)$ deterministički KS jezik je neodlučiv.

Teorema 5.4.3 Ako je $\mathcal{L} = \mathcal{L}(\mathcal{A})$ jezik koji prepoznaje neki DPA \mathcal{A} , a R regularni jezik, tada su sledeći problemi odlučivi:

- $\mathcal{L}(\mathcal{A}) = R$.
- $R \subseteq \mathcal{L}(\mathcal{A})$.
- $\mathcal{L}(\mathcal{A}) = \Sigma^*$.
- $\mathcal{L}(\mathcal{A})$ je regularan.

5.5 Podfamilije determinističkih KS jezika

Neke od podfamilija determinističkih KS jezika o kojima ćemo diskutovati u ovoj sekciji su:

- LL jezici
- LR jezici

5.5.1 LL jezici

Neka je $w \in \Sigma^*$ i neka je $Prvi_k(w)$ prefiks dužine k reči w . Ako je $|w| < k$, neka je $Prvi_k(w) = w$.

Definicija 5.5.1 — $LL(k)$ -gramatika. KS gramatika $G = (\Sigma, N, P, S)$ je $LL(k)$ -gramatika ako za $u, v, v' \in \Sigma^*$ i $X \in N$ važi da iz (oznaka \implies^* u donjim formulama predstavlja najlevlje izvođenje)

$$\begin{aligned} S \implies^* uX\beta &\implies u\alpha\beta \implies^* uv \\ S \implies^* uX\beta &\implies u\alpha'\beta \implies^* uv' \end{aligned}$$

i $Prvi_k(v) = Prvi_k(v')$ sledi $\alpha = \alpha'$.

Smisao definicije 5.5.1 je sledeći: ako je data reč $uv \in \Sigma^*$ i najlevlje izvođenje iz S u $uX\beta$, tada prefiks dužine k reči v dopušta da se odredi koje će sledeće pravilo biti primenjeno. Otuda i akronim $LL(k)$ koji ukazuje da se čitanje ulazne reči vrši sleva nadesno (prvo L , od $Left$) i konstruiše najlevlje izvođenje (drugo L , od $Left$) uvidom u prvim k simbola iz ulazne struje.

Definicija 5.5.2 — $LL(k)$ -jezik. **LL -jezik.** Ako je neki jezik generisan $LL(k)$ -gramatikom, onda je on $LL(k)$ -jezik. Jezik je LL -jezik ako je $LL(k)$ -jezik za neko k .

Primer 5.4 Neka je jezik \mathcal{L} definisan skupom pravila:

- (1) $S \rightarrow aSbS$
- (2) $S \rightarrow cT$
- (3) $T \rightarrow aTbc$
- (4) $T \rightarrow b$

Posmatrajmo najlevlje izvođenje niske $w = acbbcabbc$. Na početku je $Prvi_1(w) = a$, pa biramo pravilo (1):

$$S \Rightarrow aSbS.$$

Sada je $X = S, \beta = bS, u = a, v = cbcabbc$. Na osnovu $Prvi_1(v) = c$ možemo da zaključimo da treba birati pravilo (2):

$$aSbS \Rightarrow acTbs.$$

Dalje, $X = T, \beta = bS, u = ac, v = bbcabbc$. Na osnovu $Prvi_1(v) = b$ možemo da zaključimo da treba birati pravilo (4):

$$acTbs \Rightarrow acbbS.$$

Na isti način je dalje primenom pravila 2, 3 i 4:

$$acbbS \Rightarrow acbbcT \Rightarrow acbbcaTbc \Rightarrow acbbcabbc = w.$$

Kako smo svako naredno pravilo određivali na osnovu krajnjeg levog pomoćnog simbola u rečeničnoj formi i sledećeg jednog preduvidnog simbola iz niske w , to je ovaj jezik $LL(1)$ -jezik.

5.5.2 LR jezici

Definicija 5.5.3 — $LR(k)$ -gramatika. KS gramatika $G = (\Sigma, N, P, S)$ je $LR(k)$ -gramatika ako za $u, u', v, v' \in \Sigma^*$, $X \in N$ i $p \in (\Sigma \cup N)^*N$ važi da iz (oznaka \Rightarrow^* u donjim formulama predstavlja najdešnje izvođenje)

$$\begin{aligned} S &\Rightarrow^* \beta X u \Rightarrow \beta \alpha u = p v \\ S &\Rightarrow^* \beta' X' u' \Rightarrow \beta' \alpha' u' = p v' \end{aligned}$$

i $Prvi_k(v) = Prvi_k(v')$ sledi $X = X'$ i $\alpha = \alpha'$.

Definicija 5.5.4 — $LR(k)$ -jezik. **LR -jezik.** Ako je neki jezik generisan $LL(k)$ -gramatikom, onda je on $LL(k)$ -jezik. Jezik je LL -jezik ako je $LL(k)$ -jezik za neko k .

Akronim $LR(k)$ koji ukazuje da se čitanje ulazne reči vrši sleva nadesno (*L*, od *Left*) i konstruiše najdešnje izvođenje (*R*, od *Right*) uvidom u prvim k simbola iz ulazne struje.

Princip rada LR -gramatike se sastoji u sledećem: za zadatu rečeničnu formu pv gde je v

najduži sufiks sastavljen samo od završnih simbola, prvih k slova u niski v je dovoljno da se odredi ono pravilo koje je bilo primenjeno da bi se dobila rečenična forma pv .

Postupak analize u LR -gramatici se svodi na konstrukciju najdešnjeg izvođenja niske w iz aksioma S . Postupak je sledeći: polazi se od niske w koju svodimo na leve strane pravila posmatrajući prefikse nepročitanog dela reči. Reč pripada jeziku ako se može svesti na simbol S .

Primer 5.5 Posmatrajmo gramatiku G_{+*z} aritmetičkih izraza u kojem učestvuju operacije sabiranja i množenja i zagrade zadatu pravilima:

$$\begin{aligned} (1-2) \quad E &\longrightarrow E + T \mid T \\ (3-4) \quad T &\longrightarrow T * F \mid F \\ (5-6) \quad F &\longrightarrow (E) \mid a \end{aligned}$$

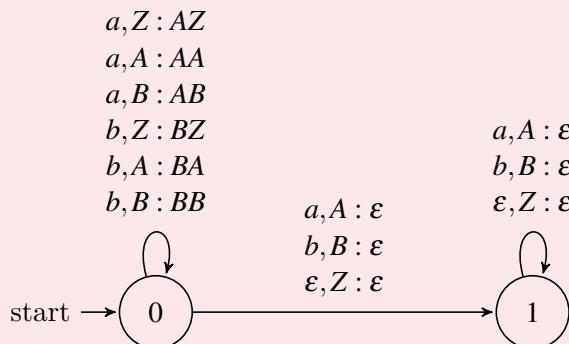
i izvođenje nadesno niske $w = a + a * a$:

$$\begin{aligned} E &\Longrightarrow E + T \Longrightarrow E + T * F \Longrightarrow E + T * a \Longrightarrow E + F * a \\ &\Longrightarrow E + a * a \Longrightarrow T + a * a \Longrightarrow F + a * a \Longrightarrow a + a * a. \end{aligned}$$

Lako se proverava da koraci u izvođenju zadovoljavaju uslove iz definicije 5.5.3 za $k = 1$, te je otuda ova gramatika $LR(1)$ -gramatika.

5.6 Pitanja i zadaci

Zadatak 5.1 Neka je dat PA $\mathcal{A} = (\Sigma, Q, \Gamma, I, Z_0, K, \Delta)$, gde je $\Sigma = \{a, b\}$, $Q = \{0, 1\}$, $I = \{0\}$, $\Gamma = \{Z, A, B\}$, $Z_0 = Z$. Dalje, neka automat prepoznae praznom potisnom listom, tj. $K = Q \times \{\epsilon\}$ i neka je skup pravila prelaza dat narednom slikom.



1. Napisati skup pravila prelaza preko preslikavanja δ .
2. Da li je ovaj automat deterministički ili ne?
3. Odrediti jezik koji prepoznae ovaj automat.
4. Za proizvoljnu reč dužine 4 koju prihvata ovaj automat napisati niz konfiguracija.

Zadatak 5.2 Konstruisati PA koji prepoznae jezik $\mathcal{L} = \{a^n b^m a^n \mid m, n \geq 1\}$ praznom potisnom listom. ■

Zadatak 5.3 Konstruisati PA koji prepoznaje jezik $\mathcal{L} = \{a^n b^{2n} \mid n \geq 1\}$ praznom potisnom listom. ■

Zadatak 5.4 Konstruisati PA koji prepoznaje jezik $\mathcal{L} = \{a^m b^m c^n \mid m, n \geq 1\}$ praznom potisnom listom. ■

6. Sintaksička analiza

Ovo poglavlje je posvećeno nekim od popularnih metoda sintaksičke analize koji se koriste u kompilatorima. Prema dizajnu, svaki programski jezik sadrži precizna pravila koja definišu sintaksičku strukturu programa koji su *dobro oformljeni*. Na primer, programski jezik C se (ugrubo posmatrano) sastoji od funkcija, pri čemu se funkcije sastoje od deklaracija i naredbi; dalje se naredbe sastoje od izraza, itd. Neke od pogodnih svojstava gramatika koje su značajne i dizajnerima jezika i razvijačima kompilatora su:

- Gramatika daje preciznu, a u isto vreme jednostavnu za razumevanje sintaksičku specifikaciju programskog jezika.
- Za određene klase gramatika, možemo da automatizujemo proces konstrukcije efikasnog sintaksičkog analizatora koji određuje sintaksičku strukturu izvornog programa.
- Sam proces konstrukcije parsera može da ukaže na sintaksičke višeiznačnosti i problematičnih mesta u gramatici koje možda nisu bile vidljive u fazi dizajna jezika.
- Gramatika omogućava iterativan razvoj jezika, dodavanjem novih konstrukata koji će izvršavati nove operacije.

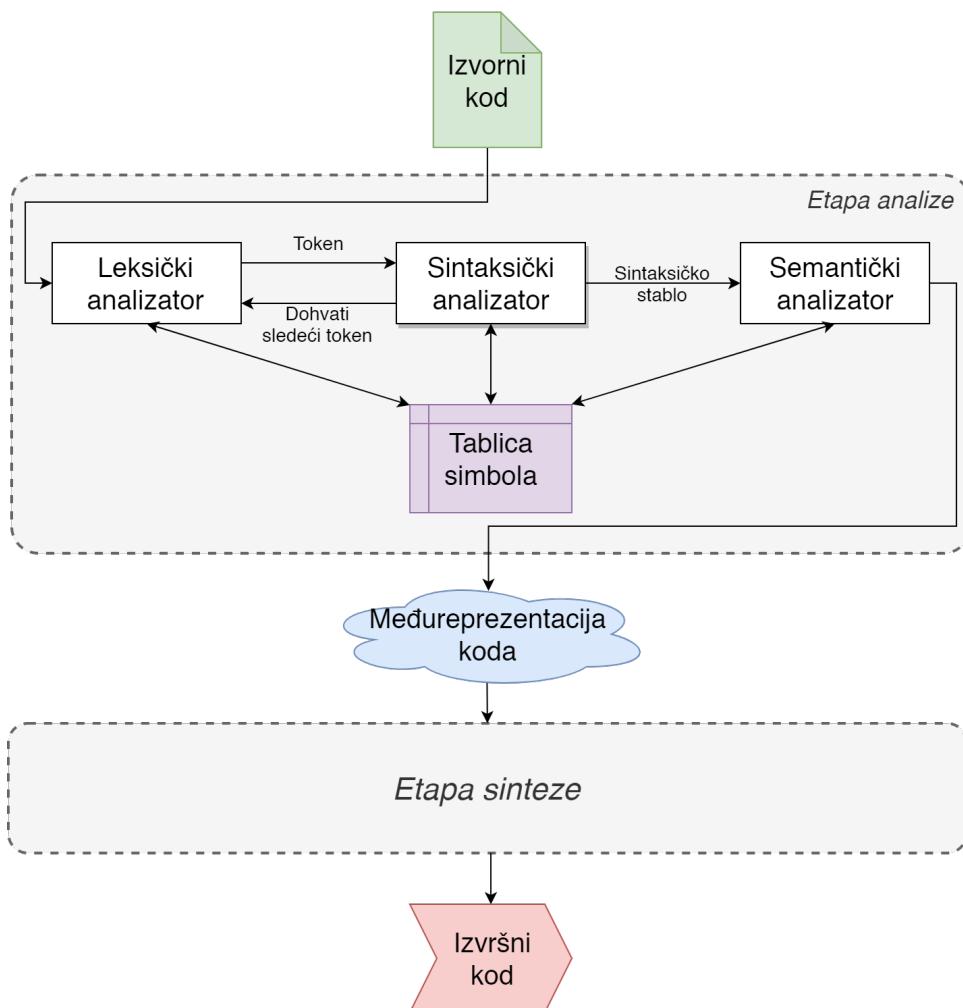
Podsetimo se da u modelu kompilatora, sintaksički analizator dobija nisku tokena od leksičkog analizatora, kao što je prikazano na slici 6.1, i verifikuje da niska tokena može biti generisana gramatikom programskog jezika. Takođe, za dobro oformljene programe, sintaksički analizator konstruiše apstraktno sintaksičko stablo i priprema fazu prevođenja polaznog jezika na međujezik. Pored ovih funkcija, sintaksički analizator obrađuje i eventualne (sintaksičke) greške koje su se pojavile na ulazu.

Postoje tri opšte vrste sintaksičkih analizatora za gramatike:

1. *Univerzalni*, koji dopuštaju da se analizira bilo koja KS gramatika. Ovi opšti metodi se sastoje u tome da se ispitaju, polazeći od početnog simbola gramatike, sva moguća izvođenja dok se ne utvrdi da li analizirana reč pripada jeziku ili ne. Blokiranje procesa analize se sprečava posebnim pravilima o dužini izvođenja. Međutim, složenost ovih metoda je reda $O(n^3)$, gde je n dužina ulaza, što se smatra za izuzetno sporim

u praktičnim implementacijama.

2. *Analizatori naniže*, koji konstruišu apstraktno sintaksičko stablo polazeći od korena do listova. Ovakva konstrukcija odgovara situaciji u kojoj se, polazeći od korena stabla, dodaju novi čvorovi u poretku koji bi odgovarao infiksnom obilasku stabla. Ovi analizatori su dovoljno jednostavnvi da se mogu ručno implementirati za datu gramatiku. Najznačajniji predstavnik ove klase su *LL*-gramatike.
3. *Analizatori naviše*, koji konstruišu apstraktno sintaksičko stablo polazeći od listova do korena. Ovim metodama se može analizirati šira klasa jezika nego metodama analize naniže. Međutim, zbog složenosti njihove konstrukcije, često se pribegava korišćenju nekih od automatskih metoda za njihovu konstrukciju. Najznačajniji predstavnik ove klase su *LR*-gramatike.



Slika 6.1: Pozicija sintaksičkog analizatora u modelu kompilatora.

Za određene klase determinističkih KS jezika postoje analizatori naniže i naviše koji vrše sintaksičku analizu u linearном vremenu, tj. čija je složenost reda $O(n)$. U oba slučaja se ulazni niz tokena čita sleva nadesno, a najčešće je dovoljan jedan token *preduviđa* da se donese odluka o sledećem koraku analize.

6.1 Sintaksna analiza naniže

Analiza naniže počinje od aksioma gramatike S , a tokom analize se postupno generiše najlevlje izvođenje niske koja se poredi sleva nadesno sa ulaznim nizom tokena. Poželjno je da ovo izvođenje bude takvo da je u svakom koraku moguće odrediti na osnovu tekучeg preduvidnog simbola koje će sledeće pravilo biti primenjeno u najlevljem izvođenju. Kao što smo videli, najlevlje izvođenje „otkriva“ postupno sve duži prediks ulazne reči. Otuda, uvid u tekući preduvidni simbol bi trebalo da omogući da se jednoznačno odredi sledeće pravilo izvođenja. Odatle sledi da je poželjno svojstvo gramatike da se odluka o sledećem pravilu može doneti na osnovu *tačno jednog* preduvidnog simbola. U suprotnom slučaju, potrebno je izvršiti *vraćanje unazad* (engl. *backtracking*) što podrazumeva ponovno skeniranje dela ulaza.

Napomenimo da gramatike koje sadrže levo rekurzivna pravila i koja nisu levo faktorisana mogu dovesti do problema u analizi naniže. Takođe, pojava ϵ slova može predstavljati problem. Za više detalja i primera ovih problema, pogledati Vitas 2006.

6.1.1 Marker kraja ulaza

Tokom analize izvornog programa, poseban indikator treba da ukaže na kraj izvorne datoteke. On se naziva *marker kraja ulaza* i obeležava se simbolom \dashv . Marker kraja ulaza se može posmatrati kao poseban token koji se pojavljuje kao krajnji desni simbol desne strane pravila gramatike koja odgovaraju aksiomu S gramatike. Prilikom zapisivanja gramatika, obično se izostavlja navođenje ovog markera (premda se njegovo prisustvo podrazumeva), a to najčešće ne stvara poseban problem. U suprotnom, moguće je uvesti novi pomoći simbol S' kao aksiom gramatike G i dodati pravilo

$$S' \longrightarrow S \dashv$$

gde je S aksiom gramatike pre proširenja. Gramatika sa ovakvim pravilom se naziva *proširena gramatika*.

Predimo sada na konkretnе tehnike sintaksnih analiza naniže.

6.1.2 Rekurzivni spust

Tehnika *rekurzivnog spusta* (engl. *recursive-descent parsing*) predstavlja generalnu formu sintaksne analize naniže, koja može da zahteva vraćanje unazad da bi pronašla korektno A-pravilo koje treba da se iskoristi u izvođenju.

Procedura za konstrukciju sintaksičkog analizatora koji koristi tehniku rekurzivnog spusta je:

1. Za svaki pomoći simbol $A \in N$ je potrebno konstruisati proceduru čiji je potpis `void A()` i čija je uloga da prepozna pojavljivanje simbola A . Telo ove procedure se izvodi iz A -pravila na način koji je opisan u narednim koracima.
2. Ako više pravila imaju isti simbol sa leve strane, telo procedure sadrži uslovnu kontrolu toka koja, prema vrednosti preduvidnog simbola, omogućava izbor desne strane odgovarajućeg pravila. Preduvidni simbol se, pri tome, ne modifikuje.
3. Ako je $A \longrightarrow A_1A_2\dots A_n$ pravilo gramatike, onda za svaki $i \in \{1, \dots, n\}$:
 - (a) Ako $A_i \in N$, onda pozovi proceduru `A_i()`.

- (b) Inače, ako $A_i \in \Sigma$, onda poredi simbol A_i i tekući preduvidni simbol. Ako su ovi karakteri jednaki, čita se sledeći ulazni simbol.
- (c) Inače, došlo je do greške.

Izvršavanje analize počinje pozivanjem procedure za aksiom gramatike, pri čemu se za preduvidni karakter smatra prvi karakter ulazne niske. Po završetku ove procedure, potrebno je signalizirati uspeh ako je telo procedure skeniralo čitavu ulaznu nisku, odnosno, ako je poslednji preduvidni karakter marker kraja ulaza. U suprotnom, ulazna niska sadrži grešku.

Procedura koja je opisana iznad generiše sintaksički analizator koji prepoznae $LL(1)$ -jezike. Nešto uopšteni slučaj bi podrazumevalo postojanje prethodno pomenuto vraćanje unazad, što dalje povlači ponovljena čitanja delova ulazne niske. Na primer, da bismo uveli vraćanje unazad, potrebno je da izmenimo prethodnu proceduru tako da se u koraku (2), umesto biranja jedinstvenog A -pravila, pokuša sa biranjem više pravila u nekom poretku. Zatim, greška u koraku (3c) ne predstavlja finalnu grešku, već sugestiju da je potrebno probati neko drugo pravilo u koraku (2). Samo u situaciji kada smo iscrpeli sva A -pravila prijavljujemo da je došlo do greške. Dodatno, da bismo uspeli da pokušamo sa biranjem nekog drugog A -pravila, potrebno je da postavimo „pokazivač” u ulaznoj niski na poziciju gde je bio kada se prvi put došlo do koraka (2).

Da bismo primenili tehniku rekurzivnog spusta, gramatika ne sme sadržati levu rekurziju, čak i kada procedura sadrži vraćanje unazad. U suprotnom, može doći do beskonačne rekurzije prilikom analize. Ovu važnu napomenu formalizujemo narednom teoremom.

Teorema 6.1.1 Proces sintaksičke analize pomoću sintaksičkog analizatora generisanog tehnikom rekurzivnog spusta za gramatiku G se zaustavlja ako gramatika G ne sadrži levu rekurziju.

Dokaz. Jedina situacija kada nastaje beskonačna rekurzija jeste u koraku (3a) kada na steku poziva procedura postoji poziv procedure `void A()`, a u tekućoj proceduri je odabранo pravilo kojem se ponovo poziva procedura `void A()`. Ova situacija odgovara najlevljem izvođenju oblika

$$A \implies^* A\alpha,$$

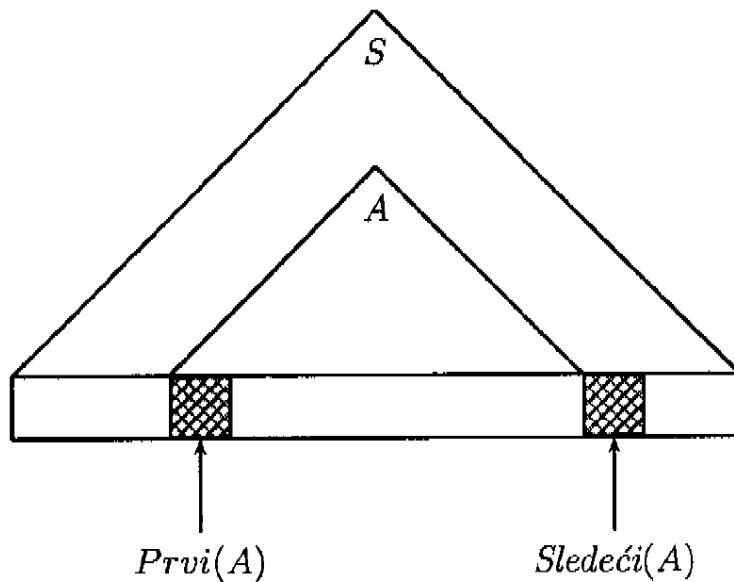
gde je $A \in N$ i $\alpha \in (\Sigma \cup N)^*$. Uklanjanjem leve rekurzije iz gramatike G se dobija gramatika G' u kojoj ne postoji najlevlje izvođenje gornjeg oblika, pa samim tim ni ne dolazi do beskonačne rekurzije, odn. sintaksička analiza se zaustavlja. ■

6.1.3 $LL(1)$ gramatike

Tokom najlevljeg izvođenja nastaje situacija koja je predstavljena slikom 6.2. Kada je simbol A krajnji levi simbol u nekoj rečeničnoj formi izvedenoj iz S , neophodno je znati koji se završni simboli mogu očekivati u daljem izvođenju kao posledica zamene simbola A desnom stranom A -pravila. Slično tome, potrebno je znati koji se završni simboli mogu očekivati nakon niske koja se dobija zamenom simbola A desnom stranom A -pravila.

Definicija 6.1.1 — Skup Prvi. Za datu gramatiku $G = (\Sigma, N, P, S)$ skup $Prvi(\alpha)$, gde je $\alpha \in \Sigma \cup N$, predstavlja skup svih završnih simbola koji se mogu pojaviti kao prefiks u nekom izvođenju iz simbola α . Dodatno, ako je α anulirajući simbol, tada u $Prvi(\alpha)$

treba dodati i ϵ .



Slika 6.2: Funkcije $Prvi$ i $Sledeći$.

Procedura za izračunavanje skupa $Prvi(X)$ za sve $X \in \Sigma \cup N$ data je narednim koracima:

1. Ponavljati naredne korake sve dok se nijedan završni simbol $a \in \Sigma$ ili ϵ ne mogu dodati nijednom $Prvi$ skupu:
 - (a) Ako je $X \in \Sigma$, onda je $Prvi(X) = \{X\}$.
 - (b) Ako je $X \in N$ i $X \rightarrow Y_1Y_2 \dots Y_k \in P$ za neko $k \geq 1$, onda dodati a u skup $Prvi(X)$ ako za neko i važi $a \in Y_i$ i ϵ se nalazi u svim $Prvi(Y_1), \dots, Prvi(Y_{i-1})$ skupovima, odnosno, $Y_1 \dots Y_{i-1} \implies^* \epsilon$. Ako je $\epsilon \in Y_j$ za sve $j \in \{1, 2, \dots, k\}$, onda dodati ϵ u $Prvi(X)$.
 - (c) Ako je $X \rightarrow \epsilon \in P$, tada dodati ϵ u $Prvi(X)$.

Primer 6.1 Posmatrajmo gramatiku datu narednim pravilima:

$$\begin{array}{lcl} E & \longrightarrow & TE' \\ E' & \longrightarrow & +TE' \quad | \quad \epsilon \\ T & \longrightarrow & FT' \\ T' & \longrightarrow & *FT' \quad | \quad \epsilon \\ F & \longrightarrow & (E) \quad | \quad b \end{array}$$

Prateći opisanu proceduru, dobijamo naredne skupove $Prvi$:

$$\begin{aligned} Prvi(E) &= \{(, b\} \\ Prvi(E') &= \{+, \epsilon\} \\ Prvi(T) &= \{(, b\} \\ Prvi(T') &= \{*, \epsilon\} \\ Prvi(F) &= \{(, b\} \end{aligned}$$

Definicija 6.1.2 — Skup Sledeći. Za datu gramatiku $G = (\Sigma, N, P, S)$ skup $\text{Sledeći}(A)$, gde je $A \in N$, predstavlja skup svih završnih simbola $a \in \Sigma$ koji se mogu pojaviti neposredno desno od A u nekoj rečeničnoj formi. Preciznije, skup svih $a \in \Sigma$ za koje postoji izvođenje oblika

$$S \implies^* \alpha A a \beta$$

za neke $\alpha, \beta \in (\Sigma \cup N)^*$.

Procedura za izračunavanje skupa $\text{Sledeći}(A)$ za sve $A \in N$ data je narednim koracima:

1. Ponavljati naredne korake sve dok se ništa ne može dodati nijednom Sledeći skupu:
 - (a) Ako je $A = S$, onda dodati \dashv (marker kraja ulaza) u $\text{Sledeći}(A)$.
 - (b) Ako je $A \rightarrow \alpha B \beta \in P$, tada dodati sve što se nalazi u $\text{Prvi}(\beta)$ osim ϵ u $\text{Sledeći}(B)$.
 - (c) Ako je $A \rightarrow \alpha B \in P$, tada dodati sve što se nalazi u $\text{Sledeći}(A)$ u $\text{Sledeći}(B)$.
 - (d) Ako je $A \rightarrow \alpha B \beta \in P$ i ako $\epsilon \in \text{Prvi}(\beta)$, tada dodati sve što se nalazi u $\text{Sledeći}(A)$ u $\text{Sledeći}(B)$.

Primer 6.2 Posmatrajmo gramatiku datu narednim pravilima:

$$\begin{array}{lcl} S & \longrightarrow & E \dashv \\ E & \longrightarrow & TE' \\ E' & \longrightarrow & +TE' \quad | \quad \epsilon \\ T & \longrightarrow & FT' \\ T' & \longrightarrow & *FT' \quad | \quad \epsilon \\ F & \longrightarrow & (E) \quad | \quad b \end{array}$$

Prateći opisanu proceduru, dobijamo naredne skupove Sledeći :

$$\begin{aligned} \text{Sledeći}(E) &= \{\dashv,)\} \\ \text{Sledeći}(E') &= \{\dashv,)\} \\ \text{Sledeći}(T) &= \{\dashv,), +\} \\ \text{Sledeći}(T') &= \{\dashv,), +\} \\ \text{Sledeći}(F) &= \{\dashv,), +, *\} \end{aligned}$$

Skupovi Prvi i Sledeći omogućavaju da se precizno opišu gramatike koje se mogu analizirati naniže sa tačno jednim preduvidnim simbolom, što se formalizuje narednom teoremom koju dajemo bez dokaza.

Teorema 6.1.2 Gramatika $G = (\Sigma, N, P, S)$ je $LL(1)$ ako u svakom koraku najlevljeg izvođenja tačno jedno pravilo $A \rightarrow \alpha \in P$ zadovoljava uslov da je primenljivo za preduvidni simbol a ako važi

$$a \in \text{Prvi}(\text{Sledeći}(A)).$$

Uloga ϵ -pravila

Načelno, simbol ϵ ima ulogu završnog simbola, ali on nije token jer ne poredi nijednu leksemu iz ulazne struje. U tom smislu, ϵ ukazuje na odsustvo tokena u ulaznoj struji. Ovo dalje znači da će ϵ pravilo biti primenjeno kada se nijedno drugo pravilo ne može primeniti za dati pomoćni simbol.

Analizator odlučuje da li neki simbol poredi praznu nisku ili ne, gledajući u sledeći token, ali i u gramatičke simbole koji slede iza onog pomoćnog simbola koji se svodi na ϵ .

Primer 6.3 Neka je data gramatika sa pravilima:

$$\begin{aligned} S &\longrightarrow aAb \\ A &\longrightarrow BC \mid \epsilon \end{aligned}$$

Kada se odlučuje kako treba razviti A , onda ako se preduvidni simbol ne nalazi u skupu $Prvi(BC)$, biće upotrebljeno ϵ -pravilo ako je preduvidni simbol b . U svakom drugom slučaju aktivira se poruka o grešci.

6.1.4 Tablice LL-analize

Za datu gramatiku $G = (\Sigma, N, P, S)$ proširenu pravilom o markeru kraja ulaza, analiza naniže podrazumeva poznavanje skupova $Prvi$ i $Sledeći$, koji usmeravaju tok sintaksičke analize. Ako je $A \longrightarrow \alpha$ pravilo gramatike G , a $a \in Prvi(\alpha)$, tada će sintaksički analizator razviti simbol A u α ako mu je preduvidni simbol na ulazu a . U slučaju da je α anulirajuća niska, tada se A može razviti u nisku α pod uslovom da je na ulazu neki simbol koji pripada skupu $Sledeći(A)$.

Rezultat ovakvog postupka za sve pomoćne simbole gramatike vodi u konstrukciju matrice M koja za dati pomoćni simbol A i preduvidni karakter a daju pravilo gramatike koja zadovoljava uslov iz teoreme 6.1.2. Konstrukcija tablica se odvija prema sledećoj proceduri:

1. Inicijalizovati matricu M tako da sve vrednosti u njoj odgovaraju stanju *greške* u analizi.
2. Za svako pravilo $A \longrightarrow \alpha$:
 - (a) Za svako $a \in \Sigma$ takvo da $a \in Prvi(\alpha)$, izvršiti dodelu $M[A, a] = A \longrightarrow \alpha$.
 - (b) Ako je $\epsilon \in Prvi(\alpha)$, tada:
 - i. Za svako $b \in \Sigma$ i $b \in Sledeći(A)$, izvršiti dodelu $T[A, b] = A \longrightarrow \alpha$.
 - ii. Ako $\dashv \in Sledeći(A)$, izvršiti dodelu $T[A, \dashv] = A \longrightarrow \alpha$.

Primer 6.4 Numerišimo pravila uprošćene gramatike aritmetičkih izraza koja je proširena pravilom $S \longrightarrow E \dashv$:

0. $S \longrightarrow E \dashv$	1. $E \longrightarrow TE'$	2. $E' \longrightarrow +TE'$
3. $E' \longrightarrow \epsilon$	4. $T \longrightarrow FT'$	5. $T' \longrightarrow *FT'$
6. $T' \longrightarrow \epsilon$	7. $F \longrightarrow (E)$	8. $F \longrightarrow b$

Tada se tabela za *LL*-analizu konstruiše na sledeći način, prateći opisanu proceduru:

- Za pravilo 1. važi $Prvi(TE') = Prvi(E) = \{(, b\}$, pa je $M[E, ()] = M[E, b] = E \longrightarrow TE'$.
- Za pravilo 2. važi $M[E', +] = E' \longrightarrow +TE'$.
- Za pravilo 3. važi $Sledeći(E') = \{), \dashv\}$, pa je $M[E', ()] = M[E', \dashv] = E' \longrightarrow \epsilon$.
- ...

Nastavljujući ovaj postupak dobija se naredna tabela. Primetimo da informacije iz ove tabele ukazuju na grananja u procedurama analize rekurzivnim spustom.

	+	*	()	b	\dashv
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'	$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$		$E' \rightarrow \epsilon$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F			$F \rightarrow (E)$		$F \rightarrow b$	

Prethodno opisana procedura se može koristiti za proizvoljnu gramatiku G radi konstruisanja tablice sintaksičke analize M . Za svaku $LL(1)$ gramatiku, svaki element ove tablice jedinstveno određuje pravilo ili signalizira grešku. Ipak, za neke gramatike, matrica M može imati elemente koji su višestruko definisani. Na primer, ako je G levo rekurzivna ili višeiznačna, onda će M imati makar jedan višestruko definisan element. Ovo je važna napomena s obzirom da, iako su eliminacija leve rekurzije i leva faktorizacija transformacije koje se jednostavno izvršavaju, postoje gramatike za koje nijedan broj transformacija ne može da proizvede $LL(1)$ gramatiku.

6.1.5 Nerekurzivna LL -analiza

Metoda rekurzivnog spusta rekurzivnim pozivima procedura prikriva upotrebu potisne liste koja je neophodna u analizi KS jezika. Prirodno je otuda da je analizu naniže moguće organizovati i koristeći eksplicitno potisnu listu (odnosno, DPA). Ovaj postupak je poznat kao *nerekurzivna analiza naniže*.

Neka je za gramatiku $G = (\Sigma, N, P, S)$ data LL -tablica M , konstruisana na način opisan u podsekciji 6.1.4 i neka je data niska $w \in \Sigma^*$. Ako $w \in \mathcal{L}(G)$, algoritam daje najlevlje izvođenje $S \implies^* w$, a inače, izveštaj o grešci. Postupak se sastoji od sledećih koraka:

1. *Inicijalizacija.* Na početku rada analizatora, potisna lista sadrži jedino marker dna potisne liste \dashv i početni simbol S na vrhu liste. U ulaznom baferu se nalazi niska $w \dashv$ (sa markerom kraja ulaza). Neka promenljiva p pokazuje na prvi karakter niske w .
2. *Iteracija.* Neka je X simbol na vrhu potisne liste, a a simbol niske w na koji pokazuje promenljiva p . Onda, sve dok se potisna lista ne isprazni (tj. dok ne bude $X = \dashv$), ponavljati korake:
 - (a) Ako je $X \in \Sigma$, onda:
 - i. Ako je $X = a$, skinuti X sa potisne liste, a p povećati tako da pokazuje na sledeći simbol niske w .
 - ii. Ako je $X \neq a$, uputiti izveštaj o grešci (na primer, „Na ulazu se očekuje X ”).
 - (b) Ako je $X \in N$, onda:
 - i. Ako je $M[X, a] = X \rightarrow Y_1 \dots Y_n$, onda sa potisne liste skinuti X , a na nju potisnuti Y_n, \dots, Y_1 tako da Y_1 bude na vrhu potisne liste. Na izlazu ispisati pravilo $X \rightarrow Y_1 \dots Y_n$.
 - ii. Inače, ako $M[X, a]$ nije definisano (stanje greške), uputiti izveštaj o grešci.
3. *Završetak.* Ako je potisna lista prazna, tada:
 - (a) Ako je na ulazu samo marker kraja ulaza (\dashv), analiza je uspešno obavljena, niska $w \in \mathcal{L}(G)$, a na izlazu su ispisana pravila koja učestvuju u najlevljem izvođenju niske w .
 - (b) Inače, niska nije prihvaćena i treba uputiti izveštaj o grešci.

Primer 6.5 Koristeći tablicu konstruisanu u primeru 6.4, simulirajmo rad nerekurzivnog analizatora aritmetičkih izraza za slučaj kada se na ulazu pojavi niska $w = b * b + b$ prema opisanoj proceduri.

Rezultat simulacije je prikazan narednom tabelom koja prikazuje konfiguracije analizatora.

Potisna lista	Ulagana niska	Broj pravila
$\dashv E$	$b * b + b \dashv$	
$\dashv E'T$	$b * b + b \dashv$	1
$\dashv E'T'F$	$b * b + b \dashv$	4
$\dashv E'T'b$	$b * b + b \dashv$	8
$\dashv E'T'$	$*b + b \dashv$	
$\dashv E'T'F*$	$*b + b \dashv$	5
$\dashv E'T'F$	$b + b \dashv$	
$\dashv E'T'b$	$b + b \dashv$	8
$\dashv E'T'$	$+b \dashv$	
$\dashv E'$	$+b \dashv$	6
$\dashv E'T+$	$+b \dashv$	2
$\dashv E'T$	$b \dashv$	
$\dashv E'T'F$	$b \dashv$	4
$\dashv E'T'b$	$b \dashv$	8
$\dashv E'T'$	\dashv	
$\dashv E'$	\dashv	6
\dashv	\dashv	3

Kada je potisna lista ostala prazna, ulazna reč je pročitana do kraja, pa niska w pripada analiziranom jeziku. Najlevlje izvođenje niske w je numerisano upotrebljenim pravilima:

$$\begin{aligned}
 E &\Rightarrow TE' \Rightarrow FT'E' \Rightarrow bT'E' \Rightarrow b*FT'E' \Rightarrow b*bT'E' \Rightarrow b*bE' \\
 &\Rightarrow b*b + TE' \Rightarrow b*b + FT'E' \Rightarrow b*b + bT'E' \Rightarrow b*b + bE' \\
 &\Rightarrow b*b + b = w
 \end{aligned}$$

6.2 Sintaksna analiza naviše

Analiza naviše predstavlja pokušaj da se za ulaznu nisku tokena konstruiše sintakško stablo polazeći od njegovih listova, postupnim svodenjem ka korenu, tj. na aksiom gramatike. Postupak svodenja se sastoji u tome da se u pojedinim koracima analize, desna strana nekog pravila gramatike zameni njegovom levom stranom. Ako su podniske izabrane na odgovarajući način, onda ovakav postupak opisuje tok najdešnjeg izvođenja, ali u obrnutom redosledu (od listova ka korenu).

Primer 6.6 Za nisku $w = id * id$ i skup pravila

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

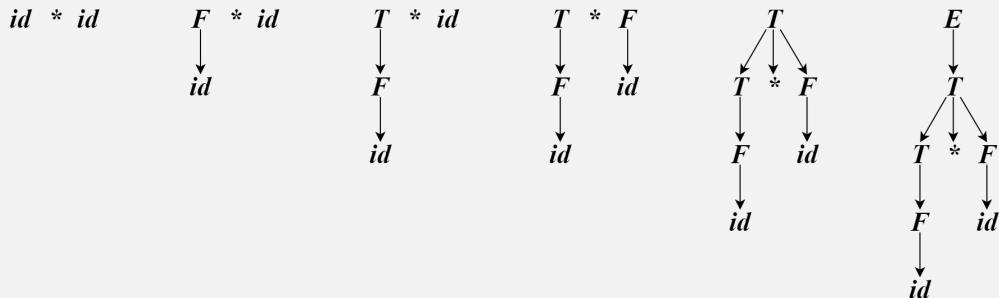
postoji najdešnje izvođenje:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id = w$$

U analizi naviše, proces analize se odvija u suprotnom smeru od smera izvođenja, tj. od niske w ka početkom simbolu E . Ovo se može predstaviti sledećim nizom koraka:

$$w = id * id \Leftarrow F * id \Leftarrow T * id \Leftarrow T * F \Leftarrow T \Leftarrow E$$

koji se grafički mogu predstaviti na sledeći način:



Kao što smo rekli, možemo razumeti proces analize naviše kao proces svodenja niske w ka aksiomu gramatike. U svakom *koraku svodenja*, odgovarajuća podniska koju prepoznaće *desna strana* nekog pravila zamenjuje se *levom stranom* tog pravila.

Primer 6.7 U primeru 6.6 ilustrovana su stanja sintaksičkog stabla koje se postepeno konstruiše. Ova stanja zapravo oslikavaju korake svodenja. Koraci izvođenja dakle formiraju sekvencu niski:

$$id * id, F * id, T * id, T * F, T, E$$

Niske u ovoj sekvenci su formirane od korena svih podstabla u koracima konstrukcije. Sekvenca počinje ulaznom niskom $id * id$. Prvo svodenje formira nisku $F * id$ tako što se bira pravilo $F \rightarrow id$ koje svodi prvo id u simbol F . Drugo svodenje formira nisku $T * id$ tako što se bira pravilo $T \rightarrow F$ koje svodi F u simbol T . Za treće svodenje imamo izbor između: (1) svodenja simbola T u simbol E pomoću pravila $E \rightarrow T$ i (2) svodenja tokena id u simbol F prema pravilu $F \rightarrow id$. Umesto da napravimo izbor (1), biramo izbor (2), što rezultuje niskom $T * F$. Ova niska se dalje svodi na T . Analiza se završava svodenjem simbola T na aksiom E .

Kako se niska w čita sleva nadesno, nije neposredno jasno kako se pravila određuju. U primeru 6.7, analiza započinje tako što se prvi token id zamenjuje simbolom F umesto drugog, a moglo je biti i obrnuto. Zašto je ipak baš odabранo to pravilo za tu podnisku ulazne niske w ?

6.2.1 Ručka

Analiza naviše tokom skeniranja ulaza sleva nadesno konstruiše najdešnje izvođenje u suprotnom smeru. Neformalno, pojam „ručke” se odnosi na podnisku koju prepoznaće *desna strana* nekog pravila i čije svodenje predstavlja jedan korak u suprotnom smeru najdešnjeg izvođenja. Naredni primer nešto detaljnije objašnjava ovaj pojam.

Primer 6.8 Radi jasnoće, označićemo tokene id u ulaznoj niski $w = id * id$ čime dobijamo nisku $w = id_1 * id_2$. U narednoj tabeli su prikazane rečenične forme koje se dobijaju u najdešnjem izvođenju prilikom analize naviše date niske, zatim ručke i pravila koja se biraju na osnovu tih ručki. Primećujemo da iako se simbol T prepoznaže desnom stranom pravila $E \rightarrow T$, simbol T ne predstavlja ručku u rečeničnoj formi $T * id_2$. Da se T zaista zamenjuje simbolom E , dobili bismo nisku $E * id_2$, koja ne može biti izvedena iz aksioma E . Ovim vidimo da nije nužno da najlevlja podniska koju prepoznaže desna strana nekog pravila predstavlja ručku.

Rečenična forma	Ručka	Pravilo
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$E \rightarrow T * F$
...

Formalizujmo ovaj očigledno važan pojam narednom definicijom.

Definicija 6.2.1 — Ručka. *Ručka* (engl. *handle*) rečenične forme γ u najdešnjem izvođenju predstavlja pravilo $A \rightarrow \beta$ takvo da se faktor β pojavljuje i niski γ na onoj poziciji na kojoj se, kada se β zameni sa A , dobija rečenična forma koja neposredno prethodi γ u najdešnjem izvođenju. Ako je $\gamma = \alpha\beta w$, gde je $w \in \Sigma^*$, tada je $A \rightarrow \beta$ ručka u niski γ ako važi:

$$S \xrightarrow{*} \alpha Aw \xrightarrow{} \alpha\beta w = \gamma$$

Oznaka $\xrightarrow{}$ u formuli iznad predstavlja najdešnje izvođenje.

N Radi udobnosti, pod ručkom ćemo smatrati desnu stranu β umesto celo pravilo $A \rightarrow \beta$.

Najdešnje izvođenje u suprotnom smeru se može dobiti procesom *potkresivanje ručke* (engl. *handle pruning*). Započinjemo niskom terminala w koja treba biti analizirana. Ako je w rečenica u dатој gramatici G , onda neka je $w = \gamma_n$, где je γ_n n -ta rečenična forma u nekom, za sada nepoznatom, najdešnjem izvođenju

$$S = \gamma_0 \xrightarrow{} \gamma_1 \xrightarrow{} \dots \xrightarrow{} \gamma_{n-1} \xrightarrow{} \gamma_n = w$$

Da bismo rekonstruisali ovo izvođenje u suprotnom smeru, pronalazimo ručku β_n u γ_n i zamenjujemo β_n pomoćnim simbolom relevantnog pravila $A_b \rightarrow \beta_n$ da bismo dobili pretvodnu rečeničnu formu γ_{n-1} u najdešnjem izvođenju. Videćemo metode za pronalaženje ručke uskoro.

Ovaj proces se zatim ponavlja: pronalazimo ručku β_{n-1} u γ_{n-1} i svodimo tu ručku da bismo dobili rečeničnu formu γ_{n-2} . Ako nastavljanjem ovog procesa proizvedemo rečeničnu formu koja se sastoji samo od S , onda se proces zaustavlja i zaključujemo $w \in \mathcal{L}(G)$.

N Ako je gramatika više značna, za datu rečeničnu formu u najdešnjem izvođenju može postojati više ručki, dok kod jednoznačnih gramatika svaka rečenična forma ima tačno jednu ručku.

6.2.2 Analiza naviše prebacivanjem i svodenjem

Pojam ručke će nam biti važan zbog načina na koji analizatori koji vrše analizu naviše rade. Ona će nam obezbediti način za prepoznavanje kada je potrebno izvršiti određeno svodenje i kojim pravilom.

Objasnimo sada kako analizatori koji vrše analizu naviše prepoznaaju (ili odbacuju) ulaznu nisku. Govorimo smo o tome kako se u analizi naviše vrši zamenjivanje podniske, koju prepoznaće desna strana nekog pravila, levom stranom tog pravila. Naime, desna strana pravila se ne zamenjuje sve dok nije potpuno pročitana. Zbog toga je potrebno sačuvati delimično pročitani deo desne strane pravila sve dok se ne steknu uslovi da se on zameni svojom levom stranom. Podesna struktura za čuvanje ovakvih delimičnih rezultata analize je potisna lista. U zavisnosti od sadržaja potisne liste, moguća su dva tipa akcija koje analizator može da izvrši:

1. *Akcija prebacivanja* (engl. *shift*) predstavlja prebacivanje simbola pročitanog sa ulaza na vrh potisne liste.
2. *Akcija svodenja* (engl. *reduce*) predstavlja zamenu niske sa vrha potisne liste, koja odgovara desnoj strani nekog pravila, njegovom levom stranom. Dakle, desni kraj niske koja se svodi mora biti na vrhu potisne liste. Analizator pretražuje levi kraj niske na potisnoj listi i zamenjuje pronađene vrednosti pomoćnim simbolom iz odabranog pravila.
3. *Prihvatanje* predstavlja signaliziranje da je analiza prošla uspešno.
4. *Greška* predstavlja prepoznavanje sintaksičke greške.

Koristimo oznaku markera kraja ulaza \dashv da označimo dno potisne liste i takođe desni kraj ulazne niske. Neka je data gramatika $G = (\Sigma, N, P, S)$. Za ulaznu nisku $w \in \Sigma^*$, stanje na početku analize izgleda:

$$\begin{array}{ccc} \text{Potisna lista} & & \text{Ulazna struјa} \\ \dashv & & w \dashv \end{array}$$

Proces analize je sledeći: tokom skeniranja ulazne niske sleva nadesno, analizator *prebacuje* nula ili više ulaznih simbola na potisnu listu, sve dok nije spreman da *svede* nisku $\beta \in \Sigma^*$ sa vrha potisne liste. U tom trenutku, analizator svodi β na pomoćni simbol $A \in N$ odgovarajućim pravilom $A \rightarrow \beta \in P$. Analizator ponavlja ovaj ciklus sve dok nije prepoznao *grešku* ili dok potisna lista ne sadrži aksiom, a na ulazu je marker kraja ulaza:

$$\begin{array}{ccc} \text{Potisna lista} & & \text{Ulazna struјa} \\ \dashv S & & \dashv \end{array}$$

Ukoliko se analizator nađe u ovoj konfiguraciji, proces se zaustavlja i analizator *prihvata* reč w .

Primer 6.9 Koraci analizatora prebacivanja-svodenja za gramatiku i (obeleženu) ulaznu reč iz primera 6.6 dati su narednom tabelom. Primerimo da se pored stanja potisne liste i ulazne struke, u svakom koraku navodi i operacija koju je analizator prebacivanja-svodenja izvršio u tom koraku.

Potisna lista	Ulazna struja	Operacija
\vdash	$id_1 * id_2 \vdash$	Prebacivanje
$\vdash id_1$	$*id_2 \vdash$	Svođenje pravilom $F \rightarrow id$
$\vdash F$	$*id_2 \vdash$	Svođenje pravilom $T \rightarrow F$
$\vdash T$	$*id_2 \vdash$	Prebacivanje
$\vdash T *$	$id_2 \vdash$	Prebacivanje
$\vdash T * id_2$	\vdash	Svođenje pravilom $F \rightarrow id$
$\vdash T * F$	\vdash	Svođenje pravilom $T \rightarrow T * F$
$\vdash T$	\vdash	Svođenje pravilom $E \rightarrow T$
$\vdash E$	\vdash	Prihvatanje

Korišćenje potisne liste u ovom pristupu je opravdano narednom činjenicom: ručka rečenične forme će se uvek pojaviti na vrhu potisne liste, a nikad unutar nje. Za više informacija, pogledati u Vitas 2006 ili Aho et al. 2007.

6.2.3 Konflikti tokom analize naviše prebacivanjem i svođenjem

Postoje KS gramatike za koje prethodno opisani proces analize ne može biti iskorišćen. Svaki analizator koji koristi opisani proces analize za takve gramatike može da dostigne konfiguraciju u kojoj, znajući sadržaj čitave potisne liste i sledećeg ulaznog simbola, dolazi do narednih nedoumica:

1. Ne može da odluči da li da izvrši prebacivanje ili svođenje. Ovakav slučaj se naziva *konflikt prebacivanje/svođenje* u analizi (engl. *sift/reduce conflict*).
2. Ne može da odluči, od više pravila, kojim od njih da izvrši svođenje. Ovakav slučaj se naziva *konflikt svođenje/svođenje* u analizi (engl. *reduce/reduce conflict*).

Posledica konflikata je da se proces analize ne odvija na deterministički način. Gramatike koje sadrže sintaksičke konstrukte takve da dovode do nekih od opisanih konflikata ne pripadaju klasi $LR(k)$ gramatika. Za razliku od njih, gramatike koje se koriste u kompilatorima su najčešće $LR(1)$ -gramatike.

6.3 Osnovni *LR*-analizatori

Najpreovlađujući tip analizatora naviše se zasniva na konceptu $LR(k)$ -analize. Već smo videli da to podrazumeva sleva nadesno čitanje sa ulaza (L) i konstruisanje najdešnjeg izvođenja u suprotnom smeru (R), pri čemu je k broj ulaznih karaktera *preduvida* koji se koriste za odlučivanje pravila na osnovu kojeg se najdešnje izvođenje konstruiše. U ovoj sekciji ćemo prikazati osnovne koncepcije LR -analize i najjednostavniji koncept za konstrukciju analizatora naviše prebacivanjem i svođenjem koji se naziva *jednostavan LR* (engl. *simple LR*, skr. *SLR*) analizator. Postoje i neki složeniji mehanizmi koji se zasnivaju na ovom konceptu, kao što su *kanonski LR* i *LALR*, o kojima neće biti reči u ovom tekstu. Detaljne informacije o njima se mogu pronaći u Vitas 2006 ili Aho et al. 2007.

LR -analizatori se zasnivaju na tablicama, slično kao i nerekurzivni LL -analizatori. Gramatika za koju možemo konstruisati te tablice korišćenjem nekih od metoda koje ćemo opisati se naziva *LR-gramatika*. Intuitivno govoreći, dovoljan uslov da je gramatika G LR -gramatika, jeste da sleva nadesno analizator naviše prebacivanjem i svođenjem može da prepozna ručke rečeničnih formi kada se pojave na vrhu potisne liste.

Neka svojstva *LR*-analizatora koja imaju teorijsko-praktični značaj su:

- *LR*-analizatori se mogu konstruisati tako da prepoznaju gotovo sve rečenične konstrukte programskih jezika za koje se mogu napisati KS gramatike. Naravno, već smo diskutovali o tome da postoje KS gramatike koje nisu *LR*-gramatike, ali najčešće se rečenični konstrukti programskih jezika mogu prezapisati na takav način da gramatike u kojima oni učestvuju zaista budu *LR*-gramatike.
- *LR*-analiza predstavlja najuopšteniji metod analize naviše prebacivanjem i svodenjem koji ne koristi vraćanje unazad, a ipak se može implementirati efikasno kao i neki drugi metodi.
- Klasa KS gramatika koje prepoznaju *LR*-analizatori je pravi nadskup klase KS gramatika koje prepoznaju metode analize naniže.

Najveća mana *LR*-analizatora u odnosu na metode analize naniže jeste u tome što njihova ručna konstrukcija može biti veoma naporna. Umesto toga, oni se konstruišu automatski, korišćenjem specijalnih softverskih paketa koji se nazivaju *generatori LR-analizatora*. Takođe generator kao ulaz dobija KS gramatiku zapisanu u određenom formatu i na osnovu nje proizvodi analizator za tu gramatiku. Ako gramatika sadrži više znacnosti ili druge konstrukte koji su teški za analizu skeniranjem ulaza sleva nadesno, tada generator analizatora prepoznaće takve konstrukte i proizvodi detaljne dijagnostičke poruke.

Veliki broj takvih alata je dostupan, a neki od primera uključuju¹: **Bison** (C, C++, Java), **Dragon** (C++, Java), **GOLD** (x86 assembly language, ANSI C, C#, D, Java, Pascal, Object Pascal, Python, Visual Basic 6, Visual Basic .NET, Visual C++), **GPPG** (C#), **jay** (C#, Java), **JS/CC** (JavaScript, JScript, ECMAScript), **Lark** (Python), **Lime** (PHP), i dr.

6.3.1 Automat $LR(0)$ -analize

Jedno od najvažnijih pitanja u prethodnoj sekciji koje smo postavili jeste: kako analizator naviše prebacivanjem i svodenjem ume da prepozna ručku u rečeničnoj formi? Na primer, pri sadržaju $\vdash T$ u potisnoj listi i narednom ulaznom karakteru $*$ u primeru 6.9, kako da analizator zaključi da simbol T , koji se nalazi na vrhu steka nije ručka, pa da na osnovu toga izvrši akciju *prebacivanja* umesto *svodenja* simbola T na simbola E ?

LR-analizatori rešavaju konflikte prebacivanja/svodenja tako što čuvaju informaciju o stanju dokle se stiglo tokom procesa analize. Stanja predstavljaju skupove elemenata koje nazivamo *ajtemi*.

Definicija 6.3.1 — Ajtem. *LR(0) ajtem* (u daljem tekstu samo *ajtem*) KS gramatike $G = (\Sigma, N, P, S)$ predstavlja pravilo iz skupa P koje sadrži *metasimbol* negde u desnoj strani tog pravila. Taj metasimbol se označava oznakom \cdot . Specijalno, pravilo $A \rightarrow \epsilon \in P$ proizvodi tačno jedan ajtem $A \rightarrow \cdot$.

Primer 6.10 Pravilo $A \rightarrow XYZ$ KS gramatike $G = (\Sigma, N, P, S)$, gde je $A \in N$, proizvodi naredne ajteme gramatike G :

$$A \rightarrow \cdot XYZ$$

$$A \rightarrow X \cdot YZ$$

$$A \rightarrow XY \cdot Z$$

$$A \rightarrow XYZ \cdot$$

Intuitivno, ajtem indikuje koliki deo pravila smo videli u nekom trenutku u procesu analize.

¹ U zagradama pored naziva softverskog paketa su navedeni „izlazni jezici”, odnosno programski jezici u kojima *LR*-analizator može biti generisan.

Na primer, ajtem $A \rightarrow \cdot XYZ$ indikuje da očekujemo da na ulazu vidimo nisku koja se izvodi iz XYZ ; ajtem $A \rightarrow X \cdot YZ$ indikuje da smo na ulazu videli nisku koja se izvodi iz X i da očekujemo dalje da vidimo nisku koja se izvodi iz YZ ; ...; ajtem $A \rightarrow XYZ\cdot$ indikuje da smo videli desnu stranu ovog pravila i da je možda vreme za svođenje XYZ na pomoćni simbol A .

Kolekcija skupova $LR(0)$ ajtema, koja se naziva *kanonska $LR(0)$ kolekcija*, snabdeva osnovu za konstrukciju determinističkog KA koji se koristi za izvođenje odluka prilikom analize. Takav automat se naziva *automat $LR(0)$ -analize*. Svako stanje ovog automata predstavlja skup ajtema u kanonskoj $LR(0)$ kolekciji.

Da bismo konstruisali $LR(0)$ kolekciju za datu gramatiku, definišemo *augmentiranu gramatiku* i dve procedure, *Zatvorene* i *GOTO*. Ako je $G = (\Sigma, N, P, S)$ gramatika $G' = (\Sigma, N', P', S')$ koja se dobija na sledeći način: $N' = N \cup \{S'\}$, $P' = P \cup \{S' \rightarrow S\}$ predstavlja *augmentiranu gramatiku* za G . Svrha uvodenja novog početnog pravila jeste da bi se indikovalo analizatoru kada da zaustavi proces analize i *prihvati* ulaznu nisku – prihvatanje se događa samo onda kada analizator treba da svede pravilom $S' \rightarrow S$.

Procedura *Zatvorene*

Neka je I neki skup ajtema gramatike $G = (\Sigma, N, P, S)$. $Zatvorene(I)$ predstavlja skup ajtema konstruisanih iz I na sledeći način:

1. Inicijalno, dodaj svaki ajtem iz I u $Zatvorene(I)$.
2. Ako $A \Rightarrow \alpha \cdot B\beta \in Zatvorene(I)$ i $B \rightarrow \gamma \in P$, onda dodaj pravilo $B \rightarrow \cdot \gamma$ u skup $Zatvorene(I)$ ako se to pravilo već ne nalazi u njemu. Ovaj korak se ponavlja sve dok ne postoji nijedan ajtem koji može biti dodat u $Zatvorene(I)$.

Intuitivno, $A \Rightarrow \alpha \cdot B\beta \in Zatvorene(I)$ indikuje da, u nekom trenutku tokom procesa analize, mislimo da na ulazu možemo da vidimo podnisku koja se izvodi iz $B\beta$. Ta podniska će imati prefiks koji se izvodi iz B primenom nekog B -pravila. Zbog toga, dodajemo sve ajteme B -pravila koji imaju metasimbol ajtema najlevlje, odnosno, ako je $B \Rightarrow \gamma$ pravilo, onda uključujemo $B \rightarrow \cdot \gamma$ u $Zatvorene(I)$.

Primer 6.11 Razmotrimo narednu augmentiranu gramatiku izraza:

$$\begin{aligned} E' &\longrightarrow E \\ E &\longrightarrow E + T \mid T \\ T &\longrightarrow T * F \mid F \\ F &\longrightarrow (E) \mid id \end{aligned}$$

Ako je $I = \{E' \longrightarrow \cdot E\}$, tada je, prateći proceduru opisanu iznad,

$$\begin{aligned} Zatvorene(I) = \{ &E' \longrightarrow \cdot E, \\ &E \longrightarrow \cdot E + T, E \longrightarrow \cdot T \\ &T \longrightarrow \cdot T * F, T \longrightarrow \cdot F, \\ &F \longrightarrow \cdot (E), F \longrightarrow \cdot id \} \end{aligned}$$

Prvo se dodaje pravilo $E' \longrightarrow \cdot E$ na osnovu koraka (1). Zatim, posto se neterminal E nalazi odmah nakon metasimbola ajtema, dodajemo u skup sve ajteme E -pravila koji imaju metasimbol ajtema najlevlje: $E \longrightarrow \cdot E + T$ i $E \longrightarrow \cdot T$. Sada se u skupu nalaze

ajtemi koji imaju T nakon metasimbola ajtema, pa dodajemo $T \rightarrow \cdot T * F$ i $T \rightarrow \cdot F$. Konačno, postojanje ajtema sa F nakon metasimbola ajtema vodi nas ka dodavanju $F \rightarrow (E)$ i $F \rightarrow id$. Više ne možemo dodati nijedan ajtem, pa se procedura zaustavlja.

Procedura $GOTO$

Procedura $GOTO(I, X)$ ima dva argumenta: I je neki skup ajtema gramatike, a X je simbol te gramatike. $GOTO(I, X)$ se definiše kao zatvorenje skupa svih ajtema oblika $A \rightarrow \alpha X \beta$ takvih da $A \rightarrow \alpha \cdot X \beta \in I$. Intuitivno, procedura $GOTO$ se koristi za definisanje prelaza u $LR(0)$ automatu za datu gramatiku. Stanja automata odgovaraju skupovima ajtema, a $GOTO(I, X)$ specifikuje prelaz iz stanja I prema ulazu X .

Primer 6.12 Ako je $I = \{E' \rightarrow E\cdot, E \rightarrow E\cdot + T\}$, tada je

$$\begin{aligned} GOTO(I, +) = & \{E \rightarrow E + \cdot T, T \rightarrow \cdot T * F, T \rightarrow \cdot F, \\ & F \rightarrow \cdot(E), F \rightarrow \cdot id\} \end{aligned}$$

$GOTO(I, +)$ smo izračunali tako što smo pronašli podskup svih ajtema iz I koji imaju $+$ odmah nakon metasimbola ajtema, a to je skup $\{E \rightarrow E\cdot + T\}$. Zatim se za svaki ajtem iz tog skupa (u ovom primeru, za samo jedan) metasimbol ajtema pomeri za jednu poziciju desno i dobija je novi skup $\{E \rightarrow E\cdot + \cdot T\}$. Rezultat predstavlja zatvorenje ovog skupa.

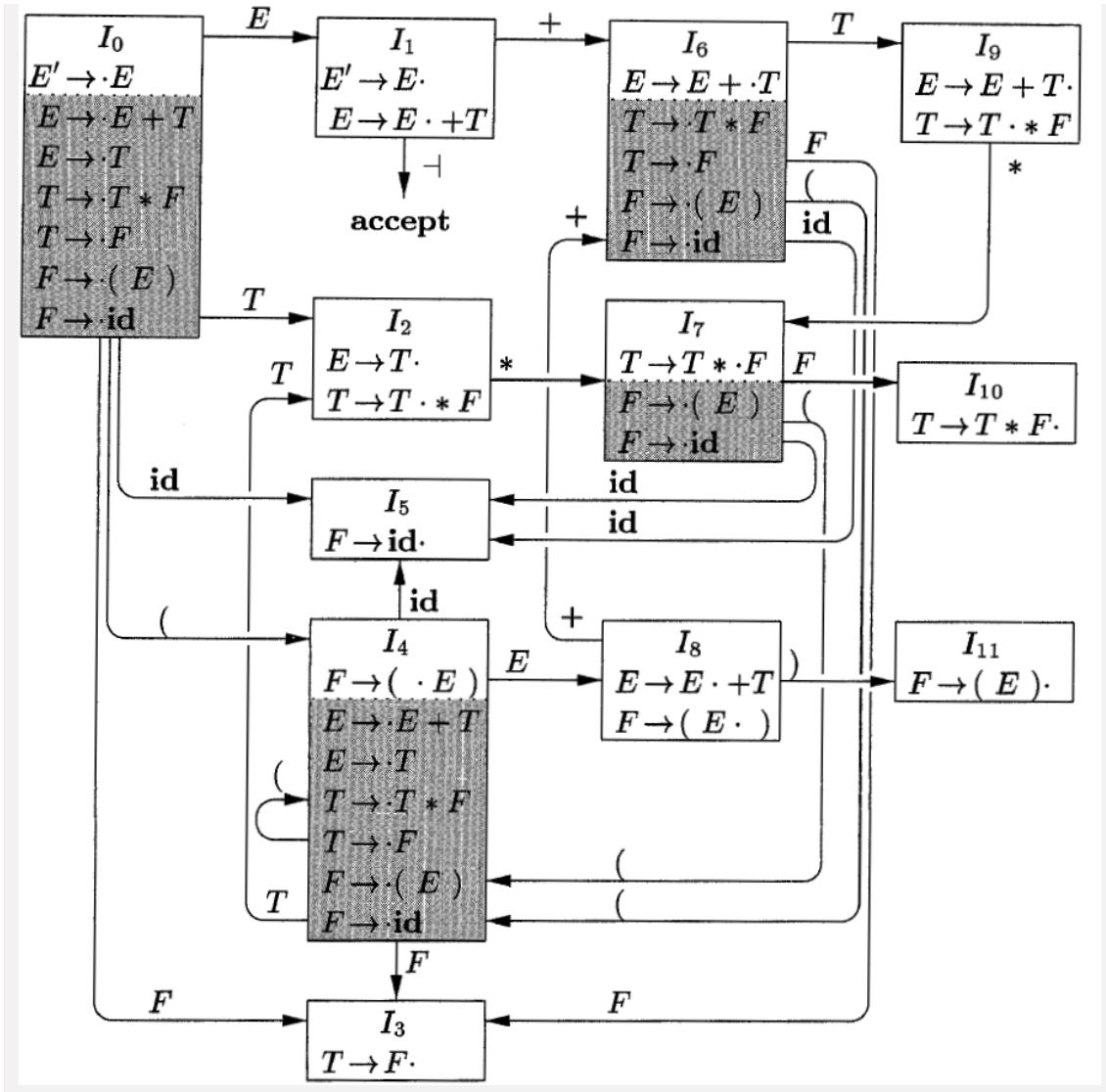
Sada kada imamo definisane ove procedure, vreme je da prikažemo proceduru za konstrukciju kanonske kolekcije C skupa $LR(0)$ ajtema. Procedura za gramatiku $G = (\Sigma, N, P, S)$ je sledeća:

1. Konstruisati augmentiranu gramatiku G' na osnovu G .
2. $C = \text{Zatvorenje}(\{S' \rightarrow \cdot S\})$
3. Sve dok se ne može dodati novi skup ajtema u C , ponavljati naredni korak:
 - (a) Za svaki skup ajtema I u C i za svaki $X \in \Sigma \cup N$:
ako $GOTO(I, X) \neq \emptyset$ i $GOTO(I, X) \notin C$, onda dodaj $GOTO(I, X)$ u C .

Primer 6.13 Razmotrimo narednu augmentiranu gramatiku izraza:

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

Prateći proceduru opisanu iznad, na osnovu kanonske kolekcije C skupa $LR(0)$ ajtema možemo konstruisati automat kao na narednoj slici. Svaki put kada u koraku (3a), za tekuće $I \in C$ i $X \in \Sigma \cup N$ (ako su uslovi iz koraka ispunjeni) dodamo novi skup $GOTO(I, X)$ u C , tada dodajemo u automat čvor koji predstavlja skup $GOTO(I, X)$ (ako on već ne postoji) i napravimo prelaz iz čvora koji odgovara skupu I prema čvoru koji odgovara skupu $GOTO(I, X)$ preko simbola X . Time se $GOTO$ kodira prelazima u automatu. Rezultat je prikazan na slici ispod.



LR(0)-analiza

Centralna ideja $LR(0)$ -analize jeste u konstrukciji $LR(0)$ automata za datu gramatiku $G = (\Sigma, N, P, S)$. Stanja ovog automata su skupovi ajtema iz kanonske $LR(0)$ kolekcije, a prelazi su dati funkcijom $GOTO$.

Početno stanje $LR(0)$ automata je $Zatvorenje(\{S' \longrightarrow \cdot S\})$, gde je S' aksiom augmentirane gramatike G' . Sva stanja su prihvatajuća stanja. Nadalje ćemo pomoći „stanja j “ u automatu referisati na ono stanje koje odgovara skupu ajtema $I_j \in C$.

Kako $LR(0)$ automat može da pomogne u razrešavanju konflikata prebacivanja/svođenja? Ova odluka se može postići na sledeći način. Prepostavimo da niska $\gamma \in \Sigma \cup N$ vodi $LR(0)$ automat od početnog stanja 0 do stanja j . Tada, primeniti tačno jednu od naredne dve operacije:

- Ako je naredni karakter na ulazu a i stanje j ima prelaz po karakteru a , onda izvrši operaciju *prebacivanja*.
- U suprotnom, izvrši operaciju svođenja; ajtemi u stanju j će nam reći koje pravilo je potrebno koristiti za to svođenje.

Algoritam *LR*-analize koji ćemo uvesti u podsekciji 6.3.2 koristi potisnu listu da čuva informaciju o stanjima kao i o gramatičkim simbolima; zapravo, gramatički simbol se može rekonstruisati iz stanja, tako da potisna lista čuva stanja.

Pogledajmo sada primer analize *LR(0)* automata i kako potisna lista stanja može da se koristi za odlučivanje u slučaju prebacivanje/svođenje konflikata.

Primer 6.14 Neka je potrebno analizirati ulaznu nisku $id * id$ korišćenjem *LR(0)* automata konstruisanom u primeru 6.13. Koristićemo potisnu listu za čuvanje stanja. Takođe, prikazaćemo informaciju o gramatičkom simbolu koji odgovara stanju na potisnoj listi u narednoj tabeli u koloni *Simboli*.

Korak	Potisna lista	Simboli	Ulaz	Akcija
1	0	\dashv	$id * id \dashv$	Prebacivanje na 5
2	0 5	$\dashv id$	$*id \dashv$	Svođenje $F \rightarrow id$
3	0 3	$\dashv F$	$*id \dashv$	Svođenje $T \rightarrow F$
4	0 2	$\dashv T$	$*id \dashv$	Prebacivanje na 7
5	0 2 7	$\dashv T *$	$id \dashv$	Prebacivanje na 5
6	0 2 7 5	$\dashv T * id$	\dashv	Svođenje $F \rightarrow id$
7	0 2 7 10	$\dashv T * F$	\dashv	Svođenje $T \rightarrow T * F$
8	0 2	$\dashv T$	\dashv	Svođenje $E \rightarrow T$
9	0 1	$\dashv E$	\dashv	Prihvatanje

U koraku 1, potisna lista sadrži početno stanje 0 automata; odgovarajući simbol je marker kraja ulaza \dashv . Naredni ulazni token je *id* i stanje 0 ima prelaz po *id* u stanje 5. Dakle, vršimo akciju prebacivanja. U koraku 2, stanje 5 (token *id*) se dodaje na vrh potisne liste. Naredni ulazni token je $*$. Kako nemamo prelaz iz stanja 5 po $*$, vršimo svođenje. Na osnovu ajtema $F \rightarrow id$ u stanju 5, pravilo na osnovu kojeg se vrši svođenje je $F \rightarrow id$.

Svođenje se implementira tako što:

- Kada su u pitanju simboli, sa vrha potisne liste se briše desna strana prepoznatog pravila (u koraku 2, desnu stranu čiji niska sa jednim tokenom *id*), a zatim se na vrh potisne liste dodaje leva strana prepoznatog pravila (u ovom slučaju, *F*).
- Kada su u pitanju stanja, sa vrha potisne liste se briše stanje 5 za simbol *id*, što nas ponovo dovodi u stanje 0 koje se sada nalazi na vrhu potisne liste, a zatim se iz stanja 0 traži prelaz po simbolu *F* (leva strana pravila). Kako stanje 0 ima prelaz po *F* u stanje 3, tako se stanje 3 dodaje na vrh potisne liste, zajedno sa simbolom *F*; videti korak 3.

Proces se dalje ponavlja sve dok automat ne prijavi uspeh ili grešku.

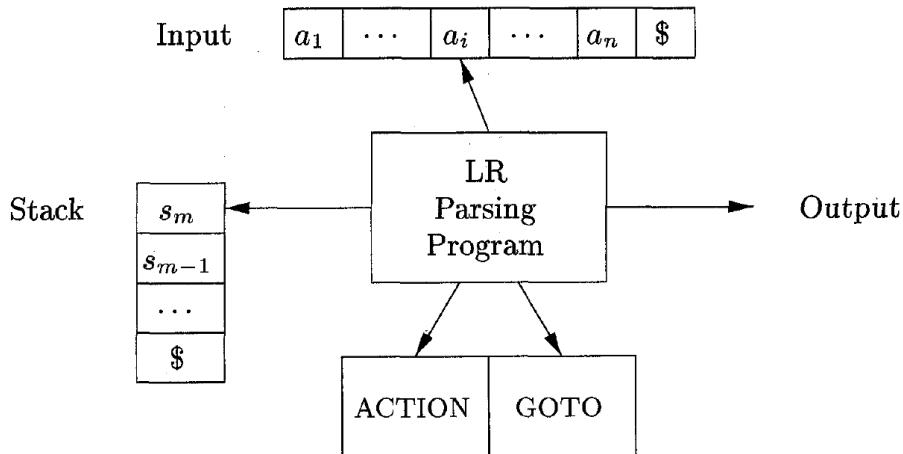
6.3.2 Opšti algoritam *LR*-analize

U ovoj sekciji ćemo detaljno prikazati algoritam *LR*-analize. Shematski prikaz *LR*-analizatora je dat na slici 6.3. Vidimo da se on sastoji od nekoliko elemenata: ulazna struja (*input*), izlazna struja (*output*), potisne liste (*stack*), pogonskog programa (*LR Parsing Program*) i tablice analize (*parsing table*) koja se sastoji od dva dela (*ACTION* i *GOTO*). Svaki *LR*-analizator ima isti pogonski program; samo se tablice analize razlikuju među njima.

Pogonski program čita karaktere iz ulazne struje sleva nadesno. Kao što bi analizator naviše prebacivanjem i svođenjem prebacivao simbol, *LR*-analizator prebacuje *stanje*. Po-

tisna lista sadrži niz stanja $s_0 s_1 \dots s_m$, pri čemu je s_m na vrhu potisne liste. Pri tome, samo simbol stanja koji je na vrhu potisne liste je od značaja za dalji tok analize. Ova stanja predstavljaju stanja determinističkog KA koji prepoznaće radne prefikse rečeničnih formi u najdešnjem izvođenju. Svako stanje sumira informaciju koja se nalazi u potisnoj listi ispod njega. Početni sadržaj potisne liste je početno stanje ovog automata.

Prema konstrukciji, svako stanje ima odgovarajući simbol gramatike. Prisetimo se da stanje odgovara skupu ajtema i da postoji prelaz od stanja i ka stanju j ako $GOTO(I_i, X) = I_j$. Svi prelazi ka stanju j moraju biti za isti gramatički simbol X . Svako stanje, osim stanja 0, ima jedinstveni gramatički simbol koji se asocira sa njim.



Slika 6.3: Model *LR*-analizatora.

Struktura tablice *LR*-analize

Tablica analize se sastoji od dva dela; procedure *ACTION* i *GOTO*:

1. Procedura *ACTION* za argumente ima stanje i i terminal a (ili marker kraja ulaza \vdash). Vrednost $ACTION[i, a]$ može imati jednu od naredne četiri forme:
 - (a) *Prebacivanje na j*, gde je j stanje automata. Ova akcija analizatora prebacuje ulazni simbol a na vrh potisne liste, ali samo što se koristi stanje j da predstavi simbol a .
 - (b) *Svodenje A → β*. Ova akcija analizatora vrši svodenje desne strane pravila β sa vrha potisne liste na levu stranu pravila A .
 - (c) *Prihvatanje*. Analizator prihvata ulaznu nisku i analiza se zaustavlja.
 - (d) *Greška*. Analizator prepoznaće da je došlo do greške u ulaznoj struci i prijavljuje grešku.
2. Proširujemo proceduru *GOTO*, koju smo prethodno definisali na skup ajtema, na stanja: ako $GOTO[I_i, A] = I_j$, onda *GOTO* takođe preslikava stanje i i neterminal A u stanje j .

Ponašanje *LR*-analizatora

Da bismo opisali ponašanje *LR*-analizatora, pomoći će nam da imamo notaciju kojom prestavljamo kompletno stanje analizatora: njegovu potisnu listu i preostali ulaz.

Definicija 6.3.2 — Konfiguracija LR-analizatora. Konfiguracija LR-analizatora predstavlja uređeni par

$$(s_0s_1 \cdots s_m, a_i a_{i+1} \cdots a_n \dashv)$$

gde je prva komponenta sadržaj potisne liste (vrh potisne liste čini najdešnje stanje), a druga komponenta predstavlja preostali ulaz.

Kada se analizator nađe u konfiguraciji $(s_0s_1 \cdots s_m, a_i a_{i+1} \cdots a_n \dashv)$, njegov naredni korak zavisi od:

- karaktera a_i koji će biti pročitan,
- tekućeg ulaznog simbola,
- s_m , odnosno, stanja na vrhu potisne liste

Analizator, na osnovu ovih informacija, konsultuje vrednost $ACTION[s_m, a_i]$ u tablici analize. Konfiguracija koja predstavlja rezultat narednog koraka zavisi od dobijene vrednosti na jedan od naredna četiri načina:

1. Ako je $ACTION[s_m, a_i] = \text{Prebacivanje na } s$, tada analizator izvršava operaciju prebacivanja; on prebacuje stanje s na potisnu listu i ulazi u konfiguraciju

$$(s_0s_1 \cdots s_m s, a_{i+1} \cdots a_n \dashv)$$

Simbol a_i ne mora da se čuva na potisnoj listi, zato što se može rekonstruisati uvidom u s sa vrha potisne liste². Trenutni ulazni simbol je a_{i+1} .

2. Ako je $ACTION[s_m, a_i] = \text{Svođenje } A \longrightarrow \beta$, tada analizator izvršava operaciju svođenja; on ulazi u konfiguraciju

$$(s_0s_1 \cdots s_{m-r}s, a_i a_{i+1} \cdots a_n \dashv)$$

gde je $r = |\beta|$ i $s = GOTO[s_{m-r}, A]$. U ovom slučaju, analizator je prvo obrisao sa vrha potisne liste r stanja, tako da se na vrhu potisne liste nalazi stanje s_{m-r} . Zatim, analizator dodaje na vrh potisne liste stanje $s = GOTO[s_{m-r}, A]$. Tekući ulazni simbol se ne menja u slučaju operacije svođenja.

3. Ako je $ACTION[s_m, a_i] = \text{Prihvatanje}$, analiza je uspešno završena.
4. Ako je $ACTION[s_m, a_i] = \text{Greška}$, analiza je neuspšeno završena i prijavljuje se greška.

Opšti algoritam LR-analize je dat procedurom ispod. Kao što smo već napomenuli, svi LR-analizatori ovu analizu vrše na isti način; jedina razlika je u načinu konstruisanja $ACTION$ i $GOTO$ procedura u tablici analize.

Neka je za gramatiku $G = (\Sigma, N, P, S)$ konstruisana tablica analize i neka je data niska $w \in \Sigma^*$. Opšti algoritam LR-analize se sastoji od narednih koraka:

1. *Inicijalizacija.* Na početku rada analizatora, potisna lista sadrži početno stanje s_0 , a ulazna struka sadrži nisku $w \dashv$ (gde je \dashv marker kraja ulaza). Neka promenljiva p pokazuje na prvi simbol niske w .
2. *Iteracija.* Neka je s simbol stanja na vrhu potisne liste, a a simbol niske w na koji pokazuje promenljiva p . Tada, sve dok je to moguće, ponavljati korake:
 - (a) Ako je $ACTION[s, a] = \text{Prebacivanje na } t$, onda:
 - i. Dodati t na vrh potisne liste.

²U praksi se često ni ne zahteva rekonstrukcija pročitanog simbola, tako da ovo svakako ne predstavlja nedostatak.

- ii. Povećati promenljivu p da pokazuje na naredni ulazni simbol niske w .
- (b) Ako je $ACTION[s, a] = Svođenje A \rightarrow \beta$, onda:
 - i. Sa vrha potisne liste obrisati $|\beta|$ simbola.
 - ii. Neka je stanje t ono koje je sada na vrhu potisne liste.
 - iii. Dodati $GOTO(t, A)$ na vrh potisne liste.
- (c) Ako je $ACTION[s, a] = Prihvatanje$, završi analizu i prijavi $w \in \mathcal{L}(G)$.
- (d) Ako je $ACTION[s, a] = Greška$, prijavi grešku.

Opisani postupak ne precizira kakvu akciju treba preduzeti u slučaju pojave nekog od konfliktata u *LR*-analizi. Ovo potiče otuda što se konflikti razrešavaju na nivou tablica analize. Sama pojava konfliktata zavisi od pravila *LR*-gramatike. Kod različitih potklasa *LR*-gramatika, na različite načine se obrađuje preduvidni simbol (kao sredstvo za otklanjanje konfliktata). Kako se tablice analize formiraju na osnovu pravila gramatike i, eventualno, preduvidnog simbola, njihov sadržaj može biti različit. Na osnovu toga, razlikuju se sledeće potklase *LR*-gramatika:

1. *LR(0)*, ako su tablice analize formirane bez obzira na preduvidni simbol, a u procesu analize se ne javljaju konflikti.
2. *SLR(1)*, ako je konflikte koji se javljaju u *LR(0)*-analizi moguće otkloniti uzimajući u obzir sve skupove *Sledeći* simbola gramatike.
3. *LALR(1)* (skraćeno od engl. *lookahead LR(1)*), ako je konflikte koji su se pojavili u *SLR(1)*-analizi moguće otkloniti ograničavanjem skupova *Sledeći* samo na one simbole koji su neophodni da bi se izvršila akcija *svođenja*.
4. *Kanonska LR(1)*, ako je za razrešavanje konfliktata u *LALR(1)*-analizi potrebno jednoj rečeničnoj formi dodeliti više stanja automata.

6.3.3 Tablice *SLR(1)*-analize

Metod *SLR(1)*-analize započinje kreiranjem *LR(0)* ajtema i *LR(0)* automatom koje smo uveli u podsekciji 6.3.1. Drugim rečima, za datu gramatiku G , konstruišemo augmentiranu gramatiku G' sa novim aksiomom S' . Na osnovu G' , konstruišemo C , kanonsku kolekciju skupa ajtema iz G' zajedno sa funkcijom $GOTO$. Za konstruisanje tablice, neophodno nam je da znamo skupove *Sledeći*(A) za svaki pomoćni simbol A iz gramatike.

Procedura konstrukcije tablica *SLR(1)*-analize je data narednim koracima:

1. Konstruisati $C = \{I_0, I_1, \dots, I_n\}$ kanonsku kolekciju skupova *LR(0)*-ajtema na osnovu gramatike G' .
2. Konstruisati stanje i na osnovu skupa ajtema I_i . Akcija automata za stanje i se određuje na osnovu sledećih uslova:
 - (a) Ako je ajtem $A \rightarrow \alpha \cdot a\beta \in I_i$ i $GOTO(I_i, a) = I_j$, onda postavi vrednost u tablici $ACTION[i, a]$ na *Prebacivanje na j*. Simbol a mora biti terminal.
 - (b) Ako je ajtem $A \rightarrow \alpha \cdot \in I_i$, onda postavi vrednost u tablici $ACTION[i, a]$ na *Svođenje A → α* za svako $a \in Sledeci(A)$. Simbol A ne sme biti S' .
 - (c) Ako je ajtem $S' \rightarrow S \cdot \in I_i$, onda postavi vrednost u tablici $ACTION[i, -]$ na *Prihvatanje*.

Ako dođe do bilo kakvog konflikta na osnovu pravila (2a), (2b) ili (2c), kažemo da gramatika nije *SLR(1)*. U tom slučaju, zaustaviti proceduru i prijaviti da nije moguće konstruisati *SLR(1)*-analizator.

3. Za svaki pomoćni simbol A , konstruiši tablicu $GOTO$ za stanje i na osnovu narednog pravila:

Ako je $GOTO(I_i, A) = I_j$, onda postavi vrednost u tablici $GOTO[i, A] = j$.

4. Sve vrednosti u tablicama $ACTION$ i $GOTO$ koje nisu nastale kao proizvod primene koraka (2) i (3) postaviti na *Greška*.
 5. Postaviti inicijalno stanje analizatora na ono stanje koje se konstruiše na osnovu skupa ajtema koje sadrži ajtem $S \rightarrow \cdot S$.

Primer 6.15 Potrebno je konstruisati $SLR(1)$ -tablice analize za augmentiranu gramatiku izraza čija pravila numerišemo na sledeći način:

$$\begin{array}{lll} 0. S \rightarrow E \dashv & 1. E \rightarrow E + T & 2. E \rightarrow T \\ 3. T' \rightarrow T + F & 4. T \rightarrow F & 5. F \rightarrow (E) \\ 6. F \rightarrow id & & \end{array}$$

Izračuvajmo prvo skupove *Prvi* i *Sledeći* za datu gramatiku:

$$\begin{aligned} Prvi(S) &= Prvi(E) = Prvi(T) = Prvi(F) = \{(, id\}, \\ Sledеći(S) &= \{\dashv\}, \\ Sledеći(E) &= \{\dashv, +,)\}, \\ Sledеći(T) &= Sledеći(F) = \{\dashv, +,), *\}. \end{aligned}$$

Kanonska kolekcija skupa $LR(0)$ -ajtema i $GOTO$ za datu gramatiku su prikazani na slici u primeru 6.13.

Razmotrimo prvo skup ajtema I_0 :

$$\begin{aligned} E' &\longrightarrow \cdot E \\ E &\longrightarrow \cdot E + T \\ E &\longrightarrow \cdot T \\ T &\longrightarrow \cdot T * F \\ T &\longrightarrow \cdot F \\ F &\longrightarrow \cdot (E) \\ F &\longrightarrow \cdot id \end{aligned}$$

Na osnovu ajtema $F \longrightarrow \cdot (E)$, upisujemo $ACTION[0, ()] = Prebacivanje na 4$, dok na osnovu ajtema $F \longrightarrow \cdot id$ upisujemo $ACTION[0, id] = Prebacivanje na 5$. Drugi ajtemi u I_0 ne proizvode nikakve akcije.

Razmotrimo sada skup ajtema I_1 :

$$\begin{aligned} E' &\longrightarrow E \cdot \\ E &\longrightarrow E \cdot + T \end{aligned}$$

Na osnovu prvog ajtema upisujemo $ACTION[1, \dashv] = Prihvatanje$, dok na osnovu drugog ajtema upisujemo $ACTION[1, +] = Prebacivanje na 6$.

Razmotrimo sada skup ajtema I_2 :

$$E \longrightarrow T \cdot$$

$$T \longrightarrow T \cdot *F$$

Kako važi da je $Sledeći(E) = \{\dashv, +,)\}$, to na osnovu prvog ajtema upisujemo

$$ACTION[2, \dashv] = ACTION[2, +] = ACTION[2,)] = Svođenje \quad E \longrightarrow T$$

Na osnovu drugog ajtema upisujemo $ACTION[2, *] = Prebacivanje \text{ na } 7$.

Nastavljanjem ovog procesa, dobijamo $ACTION$ i $GOTO$ tablice prikazane u narednoj tabeli. Napomenimo da ćemo pisati skraćene oznake za vrednosti ovih tablica na sledeći način:

- Oznaka sj označava akciju *Prebacivanje na j* (skraćeno od engl. *shift j*).
- Oznaka rp označava akciju *Svođenje r*, gde je r numeracija nekog pravila $A \longrightarrow \beta \in P$.
- Oznaka acc označava akciju *Prihvatanje*.
- Prazno polje u tablici označava akciju *Greška*.

STATE	ACTION						GOTO		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11	r5	r5			r5	r5			

Neka je sada potrebno analizirati nisku $w = id * id + id$ korišćenjem $SLR(1)$ -analizatora. Rezultat analize je dat narednom tabelom.

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		id * id + id \$	shift
(2)	0 5	id	* id + id \$	reduce by $F \rightarrow \text{id}$
(3)	0 3	F	* id + id \$	reduce by $T \rightarrow F$
(4)	0 2	T	* id + id \$	shift
(5)	0 2 7	T *	id + id \$	shift
(6)	0 2 7 5	T * id	+ id \$	reduce by $F \rightarrow \text{id}$
(7)	0 2 7 10	T * F	+ id \$	reduce by $T \rightarrow T * F$
(8)	0 2	T	+ id \$	reduce by $E \rightarrow T$
(9)	0 1	E	+ id \$	shift
(10)	0 1 6	E +	id \$	shift
(11)	0 1 6 5	E + id	\$	reduce by $F \rightarrow \text{id}$
(12)	0 1 6 3	E + F	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	E + T	\$	reduce by $E \rightarrow E + T$
(14)	0 1	E	\$	accept

6.4 Pitanja i zadaci

Pitanje 6.1 Koje vrste sintaksičkih analizatora postoje?

Pitanje 6.2 Šta su univerzalni sintaksički analizatori? Koje su njihove prednosti, a koje su mane u odnosu na ostale vrste analizatora?

Pitanje 6.3 Šta su sintaksički analizatori naniže? Koje su njihove prednosti, a koje su mane u odnosu na ostale vrste analizatora?

Pitanje 6.4 Šta su sintaksički analizatori naviše? Koje su njihove prednosti, a koje su mane u odnosu na ostale vrste analizatora?

Pitanje 6.5 Koja je osnovna ideja sintaksne analize naniže?

Pitanje 6.6 Šta predstavlja tehnika vraćanja unazad? Čemu ona služi?

Pitanje 6.7 Šta predstavlja marker kraja ulaza? Čemu on služi? Šta su proširene gramatike i kakve veze one imaju sa markerima kraja ulaza?

Pitanje 6.8 Ukratko opisati tehniku rekurzivnog spusta.

Pitanje 6.9 Detaljno napisati proceduru za konstrukciju sintaksičkog analizatora koji koristi tehniku rekurzivnog spusta bez vraćanja unazad.

Pitanje 6.10 Koje jezike mogu da prepoznaju sintaksički analizatori tehnikom rekurzivnog spusta bez vraćanja unazad? A sa vraćanjem unazad? Ukratko opisati kako se implementira vraćanje unazad u rekurzivnom spustu.

Pitanje 6.11 Da li se procedura rekurzivnog spusta uvek zaustavlja ako je ona implementirana:

1. bez vraćanja unazad?
2. sa vraćanjem unazad?

Obrazložiti detaljno.

Pitanje 6.12 Koji je potreban uslov za zaustavljanje procedure rekurzivnog spusta?

Pitanje 6.13 Koji su skupovi važni u *LL*-analizi? Kako se oni dobijaju?

Pitanje 6.14 Koja je uloga ϵ -pravila u *LL*-analizi? Navesti primer upotrebe.

Pitanje 6.15 Šta je tablica *LL*-analize? Kako se ona konstruiše?

Pitanje 6.16 Ako bi vam bila dostupna tablica *LL*-analize M za neku gramatiku G , kako biste implementirali sintaksički analizator tehnikom rekurzivnog spusta koji prepoznaće gramatiku G na osnovu date tablice M ?

Pitanje 6.17 Kako je moguće implementirati nerekurzivnu analizu naviše? Po čemu se ona razlikuje od rekurzivne analize naviše?

Pitanje 6.18 Kako se započinje proces nerekurzivne *LL*-analize?

Pitanje 6.19 Ukratko opisati šta se radi u svakoj iteraciji nerekurzivne *LL*-analize.

Pitanje 6.20 Šta radi sintaksički analizator na kraju nerekurzivne *LL*-analize?

Pitanje 6.21 Kratko opisati ideju sintaksne analize naviše. Za datu gramatiku G i ulazna nisku, šta analiza naviše konstruiše ako $w \in \mathcal{L}(G)$?

Pitanje 6.22 Šta je svođenje u analizi naviše? Dati primer svođenja na proizvoljnoj gramatici i ulaznoj niski.

Pitanje 6.23 Opisati ukratko pojam ručke u analizi naviše. Da li za datu rečeničnu formu u najdešnjem izvođenju može postojati samo jedna ručka ili više njih? Od čega to zavisi?

Pitanje 6.24 Formalno definisati pojam ručke u analizi naviše.

Pitanje 6.25 Opisati proces pokresivanja ručke.

Pitanje 6.26 Opisati analizu naviše prebacivanjem i svedenjem. Koje su operacije koje analizatori ovog tipa izvode.

Pitanje 6.27 Opisati početno i završno stanje analizatora naviše prebacivanjem i svedenjem.

Pitanje 6.28 Šta su konflikti? Koje vrste konflikata postoje?

Pitanje 6.29 Opisati ukratko osnovnu ideju *LR*-analizatora. Koje vrste njih postoje? Nавести bar 5 softverskih paketa koji ih generišu.

Pitanje 6.30 Koje su prednosti, a koje su manje *LR*-analizatora u odnosu na *LL*-analizatore?

Pitanje 6.31 Šta je ajtem? Nавести primer pravila neke KS gramatike i ajtemi koji se proizvode iz tog pravila.

Pitanje 6.32 Šta je sve neophodno da uradimo da bismo mogli da konstruišemo automat *LR(0)*-analize?

Pitanje 6.33 Šta je zatvorene? Kako se ono konstruiše?

Pitanje 6.34 Šta je GOTO? Kako se ono konstruiše?

Pitanje 6.35 Šta je kanonska kolekcija skupa ajtema? Kako se ona konstruiše?

Pitanje 6.36 Koja je osnovna ideja *LR(0)*-analize? Kako *LR(0)* automat može da pomogne u razrešavanju konflikata prebacivanja/svedenje?

Pitanje 6.37 Od kojih elemenata se sastoji svaki *LR*-analizator? Prikazati te elemente shematski i dati kratak opis za svaki od njih.

Pitanje 6.38 Od kojih delova se sastoji tablica analize u *LR*-analizatorima? Koje su moguće vrednosti tih delova?

Pitanje 6.39 Definisati konfiguraciju *LR*-analizatora.

Pitanje 6.40 Kako *LR*-analizator vrši prebacivanje automata iz jedne konfiguracije u drugu? Detaljno opisati ovaj proces.

Pitanje 6.41 Detaljno opisati opšti algoritam *LR*-analize.

Pitanje 6.42 Na osnovu čega se razlikuju potklase *LR*-gramatika? Koje vrste ovih potklasa postoje?

Pitanje 6.43 Detaljno opisati proceduru konstrukcije tablica *SLR(1)*-analize.

Zadatak 6.44 Neka je data naredna gramatika:

$$\begin{aligned} S &\longrightarrow AB \\ A &\longrightarrow aAb \mid \epsilon \\ B &\longrightarrow cB \mid d \end{aligned}$$

- Uvidom u data pravila ispitati da li je gramatika *LL(1)*. Obrazložiti odgovor.

2. Odrediti skupove *Prvi* i *Sledeći* za datu gramatiku.
3. Konstruisati tablicu *LL*-analize.
4. Analizom dobijenih skupova i tablice ispitati da li je gramatika *LL(1)*. Obrazložiti odgovor.

■

Zadatak 6.45 Koristeći tablicu konstruisanu u primeru 6.4, simulirati rad nerekurzivnog analizatora aritmetičkih izraza za slučaj kada se na ulazu pojavi niska

$$w = (b + (b + b)) * b.$$

Napisati najlevlje izvođenje koje se dobija radom analizatora. ■

Zadatak 6.46 Posmatrajmo naredni fragment koda programskog jezika C:

```
int a;
float *b, a, c[10];
char **c, b;
```

Gramatika $G = (\Sigma, N, P, S)$ koja je u stanju da prepozna dati fragment koda data je pravilima:

```

fragment → niz_dekl ⊢
niz_dekl → niz_dekl dekl | ε
dekl → tip niz_prom TZ
tip → INT | FLOAT | CHAR
niz_prom → niz_prom ZAP prom | prom
prom → niz_zv ID dim
niz_zv → niz_zv ZV | ε
dim → LZ BROJ DZ | ε
```

gde je:

- $\Sigma = \{INT, FLOAT, CHAR, TZ, ZAP, ID, ZV, LZ, BROJ, DZ\}$,
- $N = \{fragment, niz_dekl, dekl, tip, niz_prom, prom, niz_zv, dim\}$ i
- $S = fragment$.

1. Da li je data gramatika *LL(1)*? Obrazložiti odgovor.
2. Ukoliko nije *LL(1)*, transformisati je tako da se dobije *LL(1)* gramatika.
3. Odrediti skupove *Prvi* i *Sledeći*.
4. Kreirati tablicu *LL*-analize.
5. Metodom rekurzivne *LL*-analize napisati najlevlje izvođenje koje izvodi gornji fragment koda.
6. Metodom nerekurzivne *LL*-analize napisati najlevlje izvođenje koje izvodi gornji fragment koda.

■

Zadatak 6.47 Neka je data gramatika G :

1. $A \rightarrow Aa$
2. $A \rightarrow B$
3. $B \rightarrow bB$
4. $B \rightarrow b$

$SLR(1)$ -analizom proveriti da li reč $w = bbbaaa$ pripada jeziku $\mathcal{L}(G)$. ■

Zadatak 6.48 Neka je data gramatika G :

1. $S \rightarrow AB$
2. $A \rightarrow aAb$
3. $A \rightarrow \epsilon$
4. $B \rightarrow cB$
5. $B \rightarrow d$

$SLR(1)$ -analizom proveriti da li reč $w = aabbcccd$ pripada jeziku $\mathcal{L}(G)$. ■

Zadatak 6.49 Neka je data gramatika G :

1. $A \rightarrow XY$
2. $A \rightarrow YX$
3. $X \rightarrow xX$
4. $X \rightarrow \epsilon$
5. $Y \rightarrow Yy$
6. $Y \rightarrow x$

$SLR(1)$ -analizom proveriti da li reč $w = xxyy$ pripada jeziku $\mathcal{L}(G)$. ■

7. Semantička analiza

Kontekstno slobodne gramatike su ograničene kada su upitnaju greške koje mogu da otkriju. Na primer, pravila domaćaja promenljivih, nesaglasne tipove u iskazu dodele, nedekvatne konverzije i slično. Takve greške se ne mogu otkriti samo na osnovu pravila gramatike. Kontekstno slobodne gramatike, kao što je opisano u prethodnim sekcijama, koriste za sintaksičku analizu.

Jedan način kako da se gramatika proširi ovakvim ograničenjima pružaju *atributske gramatike*. Ovakve gramatike su veoma složene za specifikaciju realnih pravila o tipovima u jezicima. Sve dok atributska gramatika specifikuje pravila saglasnosti tipova, ta se pravila mogu izraziti i kao akcije kojima se ispituje da li su pravila saglasnosti zadovoljena.

7.1 Atributske gramatike

Atributske gramatike su KSG koje podrazumevaju da je svakom gramatičkom simbolu dodeljen skup atributa. Atributi dobijaju svoju vrednost u nekom čvoru stabla sintaksičke analize na osnovu semantičkih pravila pridruženih pravilu izvođenja koje je dodeljeno tom čvoru.

Definicija 7.1.1 Atribut je svaka informacija pridružena simbolima gramatike. Token ima bar jedan atribut - leksemu kojoj je pridružen, a može imati i druge attribute.

Primer 7.1 Neka je zadata sledeća gramatika:

$$\begin{array}{lcl} E & \longrightarrow & E + E \\ & | & E * E \\ & | & (E) \\ & | & br \end{array}$$

Njom želimo da opišemo izraz $(br+br*br)*br$ i izračunamo njegovu vrednost. Primetimo da ovo nije ni LR ni LL gramatika. Imamo i shift-reduce i reduce-reduce konflikte.

Leksički analizator nam vraća tokene, ali i vrednosti koje su njima pridruženi. Pridružuje vrednosti koje čita sa ulaza. Npr, neka je na ulazu izraz:

$$(3 + 5 * 2) * 10.$$

Prepostavljamo da token *br* ima atribut *v*, što se obeležava *br.v* i prepostavljamo da je leksički analizator taj koji pridružuje vrednosti. Vrednost izraza računamo:

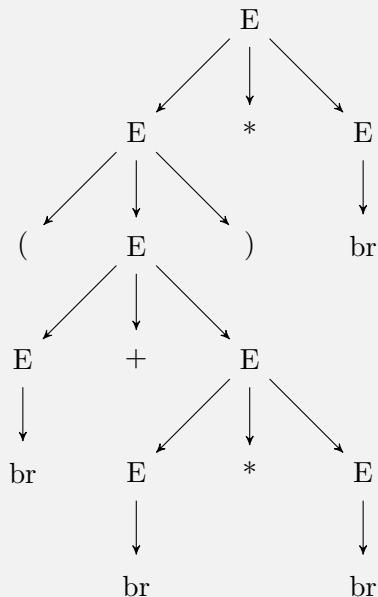
$$\left(\underbrace{br}_3 + \underbrace{br}_5 * \underbrace{br}_2 \right) * \underbrace{br}_{10}$$

 13

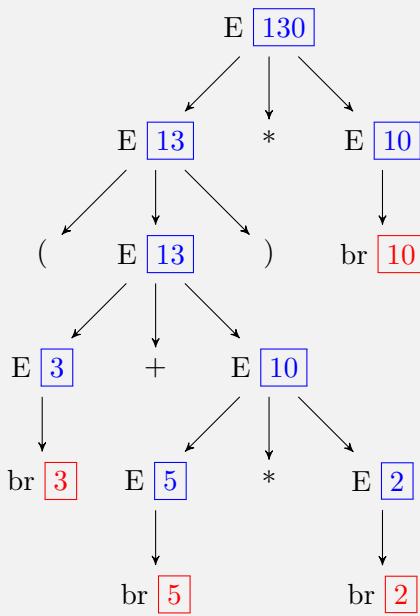
 13

 130

što se češće obeležava stablom izvođenja:



Listovi stabla su tokeni. Krećemo od pretpostavke da je leksički analizator obavio svoj posao i da svaki list stabla ima svoju vrednost. Izračunavanje se vrši odozdo-naviše, a opisujemo ga programskim kodom koji pridružujemo pravilima. Prikažimo kako se izračunavanje vrši na stablu izvođenja. Svakom listu su u leksičkoj analizi dodeljene vrednosti (obeleženo crvenom bojom), a ostalim čvorovima dodeljujemo vrednosti prema pravilima izračunavanja (obeleženo plavom bojom):



Pridružimo pravila izračunavanja našoj gramatici. Obeležićemo svako slovo E u pravilu različitim brojem, kako bismo razlikovali koju vrednost kada koristimo. Izračunavanjaćemo pisati u vitičastim zagradama pored pravila:

$$\begin{array}{lcl}
 E^1 & \longrightarrow & E^2 + E^3 \quad \{E^1.v = E^2.v + E^3.v\} \\
 | & & E^2 * E^3 \quad \{E^1.v = E^2.v * E^3.v\} \\
 | & & (E^2) \quad \{E^1.v = E^2.v\} \\
 | & & br \quad \{E^1.v = br.v\}
 \end{array}$$

U YACC-u to zapisujemo na sledeći način:

$$\begin{array}{lcl}
 E & : & E '+' E \quad \{\$\$ = \$1 + \$3; \} \\
 | & & E '*' E \quad \{\$\$ = \$1 * \$3; \} \\
 | & & '(' E ')' \quad \{\$\$ = \$2; \} \\
 | & & br \quad \{\$\$ = \$1; \} \\
 ;
 \end{array}$$

Primetimo da u YACC-u svaki token ima vrednost, a mi biramo da li ćemo ih koristiti ili ne. Zato je, na primer, za vrednost izraza u prvom pravilu (što se u YACC-u obeležava sa $\$\$$) koristimo $\$1$ i $\$3$, jer je $\$2$ vrednost tokena $+$, koju ne koristimo. Slično i u ostalim pravilima.

Svaki simbol gramatike ima svoje attribute. Pravilima gramatike se dodaju akcije koje omogućavaju izračunavanje atributa. U opštem slučaju, svakom pravilu gramatike $A \rightarrow a$ je dodeljen skup semantičkih pravila oblika $b := f(c_1, c_2, \dots, c_n)$, gde je f neko preslikavanje. Kaže se da atribut b zavisi od atributa c_1, c_2, \dots, c_n . Razlikujemo 2 vrste atributa:

1. Sintetizovani atributi – vrednost atributa simbola sa leve strane računamo na osnovu vrednosti atributa simbola sa desne strane. U stablu izvođenja to predstavlja kretanje odozdo-naviše.
2. Nasleđeni atributi – vrednost atributa simbola sa desne strane izračunavamo na osnovu vrednosti atributa simbola koji mu prethode (u stablu, „otac“ i „starija

braća”). U stablu izvođenja to predstavlja kretanje odozgo-naniže i sleva nadesno. Dakle, moguća su tri smera kretanja kroz stablo. Zdesna nalevo nije moguće jer bi to značilo da koristimo vrednost atributa nekog simbola koji još nismo pročitali.

Gramatika je S -atributska ako su u njoj svi atributi sintetizovani, a L -atributska ako su svi atributi u njoj ili sintetizovani ili nasledeni, pri čemu svaki nasledeni atribut zavisi samo od nasleđenih atributa oca, i/ili nasleđenih ili sintetizovanih atributa starije braće. Ova svojstva omogućavaju da se atributi računaju obilaskom stabla izvođenja u dubinu, sa leva na desno, pri čemu se nasleđeni atributi računaju u dolasku, a sintetizovani u odlasku iz odgovarajućeg čvora.

Primer 7.2 Primer S -gramatike: kreiranje apstraktnog sintaksnog stabla (atributi su pokazivači na čvorove u stablu):

$$\begin{array}{lcl} E & \longrightarrow & E + T \quad \{ \$\$.pok = mk_node('+', \$1.pok, \$3.pok); \} \\ & | & T \quad \{ \$\$.pok = \$1.pok; \} \\ & | & F \quad \{ \$\$.pok = \$1.pok; \} \\ F & \longrightarrow & (E) \quad \{ \$\$.pok = \$2.pok; \} \\ & | & id \quad \{ \$\$.pok = mk_leaf(id.lex); \} \end{array}$$

pri čemu je $id.lex$ leksema koja je pridružena tokenu id .

Primer 7.3 S -gramatika koja određuje tip izraza, uz proveru ispravnosti tipova. U ovom primeru prepostavčemo striktnu varijantu gde nema implicitnih konverzija tipova.

$$\begin{array}{lcl} E & \longrightarrow & E + T \quad \{ if(\$1.tip! = \$3.tip) \\ & & \quad \quad \quad error("type error"); \\ & & \quad \quad \quad else \\ & & \quad \quad \quad \quad \$\$.tip = \$1.tip; \} \\ & | & T \quad \{ \$\$.tip = \$1.tip; \} \\ & | & F \quad \{ \$\$.tip = \$1.tip; \} \\ F & \longrightarrow & (E) \quad \{ \$\$.tip = \$2.tip; \} \\ & | & id \quad \{ \$\$.tip = id.tip; \} \end{array}$$

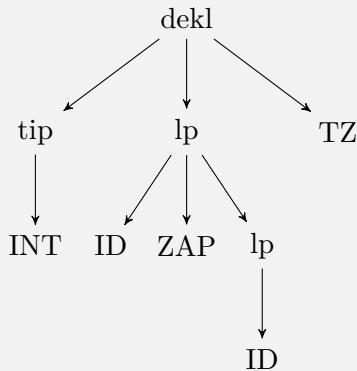
pri čemu je $id.tip$ određen prilikom deklaracije promenljive i informacija o tome je upisana u tabelu simbola.

Pogledajmo konkretnu situaciju gde se koriste nasleđeni atributi:

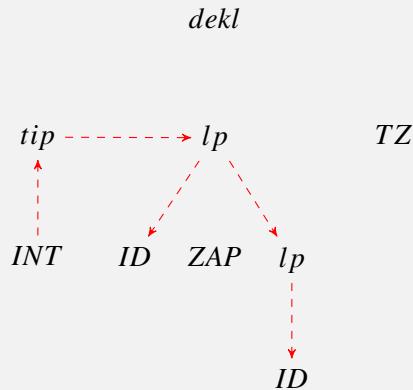
Primer 7.4 Napisati atributsku gramatiku za sledeći fragment koda:

```
int a, b;
```

stablu izvođenja u ovom slučaju izgleda:



U ovom slučaju **lp** nasleđuje svoj tip od svog „starijeg brata” **tip**, a **tip** sintetiše konkretnu vrednost od „sina” **INT**. Prikažimo tok podataka na stablu izvođenja radi detaljnije ilustracije:



Gramatika koja opisuje ovaj kod (pravila sada umećemo u sredinu kada su nasleđena) izgleda:

$$\begin{array}{lcl}
 dekl & \longrightarrow & tip \quad \{lp.t + tip.t\} \\
 lp^1 & \longrightarrow & ID \quad \quad \quad ZAP \quad \quad \quad \{ID.t = lp^2.t = lp^1.t\} \quad lp^2 \\
 & | & ID \quad \{ID.t = lp^2.t\} \\
 tip & \longrightarrow & INT \\
 & | & FLOAT
 \end{array}$$

Primer 7.5 Neka je zadata gramatika

$$\begin{array}{lcl}
 A & \longrightarrow & BC \quad \{\$\$ = \$1 + 2 * \$2;\} \\
 B & \longrightarrow & BbB \quad \{\$\$ = \$1 - \$3;\} \\
 & | & C \quad \{\$\$ = 2 * \$1;\} \\
 C & \longrightarrow & CaC \quad \{\$\$ = 3 * \$1 + \$2;\} \\
 & | & x \quad \{\$\$ = \$1;\}
 \end{array}$$

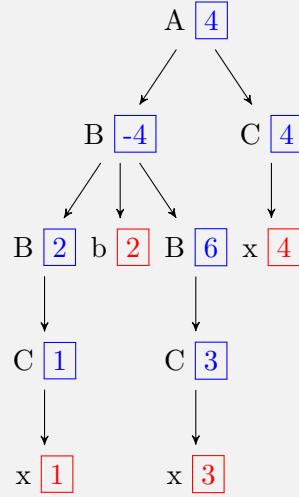
i niska „xbxx”, pri čemu tokeni niske imaju redom attribute 1, 2, 3 i 4. Izračunati vrednost celog izraza.

Najpre treba pronaći izvođenje zadate niske u zadatoj gramatici. Za nisku xbxx nije

teško naći izvođenje:

$$A \implies BC \implies BbBC \implies CbBC \implies CbCC \implies^* xbxx.$$

Dalje nije problem. Treba nacrtati stablo izvođenja i propagirati vrednosti atributa odozdo-nagore prema pravilima gramatike. Iz stabla izvođenja:



dobijamo da je vrednost izraza jednaka 4.

Literatura

Knjige

- [Aho+07] A.V. Aho et al. *Compilers: Principles, Techniques and Tools, Second Edition*. Pearson., 2007. ISBN: 9788131762349 (cited on page 119).
- [Sud88] T. A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. USA: Addison-Wesley Longman Publishing Co., Inc., 1988. ISBN: 0201157683 (cited on page 102).
- [Vit06] D. Vitas. *Prevodioci i interpretatori: Uvod u teoriju i metode kompilacije programskih jezika*. Matematički fakultet, 2006. ISBN: 9788675890560 (cited on pages 30, 44, 92, 101, 109, 119).

