PREVOĐENJE PROGRAMSKIH JEZIKA

Nikola Ajzenhamer
Anja Bukurov

Prevođenje programskih jezika

Beleške sa predavanja

Nikola Ajzenhamer
Anja Bukurov

Matematički fakultet, Univerzitet u Beogradu
17. novembar 2016.

Sadržaj

Poglavlje 1

Uvod

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

Na početku poglavlja prikazujemo proces kompiliranja kroz njegove faze i potfaze, a zatim se nešto detaljnije upoznajemo sa svakom od potfaza.

Na sadržaj u uvodnom poglavlju može se posmatrati kao na sadržaj celokupnog kursa „Prevođenje programskih jezika" koji se izvodi kao obavezan kurs na trećoj godini osnovnih studija smera Informatika na Matematičkom fakultetu.

1.1 Proces kompiliranja

Ovaj odeljak započećemo jednom od mnogobrojnih definicija pojma programskog jezika.

Definicija 1.1

Programski jezik je veštački jezik za opis konstrukcija (pisanje instrukcija) koje mogu biti prevedene u mašinski jezik i izvršene od strane računara.[1]

Za svaki program koji želimo da konstruišemo na nekom unapred odabranom programskom jeziku, moramo imati sačuvan izvorni kôd na disku u vidu datoteke. Izvorni kôd čini uređeni multiskup instrukcija koje su predviđene zadatim programskim jezikom. Međutim, napisani izvorni kôd se ne može izvršiti na računaru, već ga je neop-

[1] Ova definicija je predviđena od strane American Heritage Dictionary (AHD). Originalnu definiciju možete pogledati na zvaničnoj internet prezentaciji AHD.

hodno prevesti u tzv. izvršni kôd[2]. Izvršni kôd čini uređeni multiskup instrukcija na mašinskom jeziku, koji omogućava izvršavanje programa na konkretnom računaru.

Definicija 1.2

Programski prevodilac ili *jezički procesor* je program koji čita izvorni kôd, obrađuje ga i prevodi u izvršni kôd, čime se omogućava izvršavanje programa. Proces prevođenja izvornog kôda u izvršni kôd poznat je pod nazivom *kompiliranje*[3].

Svi programski prevodioci mogu se podeliti u dve grupe:

1. Kompilatori su programski prevodioci kod kojih je jasna razlika između faze prevođenja i faze izvršavanja. Izvorni kôd se prvo prevede u izvršni kôd, a zatim se dobijeni kôd izvršava učitavanjem u memoriju i izvršavanjem instrukcija. Kompilatori se najčešće koriste u programskim jezicima kao što su C, Java[4] i C++.

2. Interpretatori su programski prevodioci kod kojih su faza prevođenja i faza izvršavanja ispripletane, i ne postoji materijalizacija prevođenja, tj. ne postoji izvršni kôd koji se učitava u memoriju, već se uvek čita ceo izvorni kôd (ili međukôd, zavisno od jezika). Jasno je da je ovakav način izvršavanja programa sporiji, ali doprinosi boljoj dijagnostici problema, upravo zato što izvršava program naredbu-po-naredbu. Interpretatori se najčešće koriste u programskim jezicima kao što su JavaScript, Haskell, Prolog.

Sve faze kompiliranja mogu se podeliti u dve etape:

1. Etapa analize, koju vrši prednji deo prevodioca, i

2. Etapa sinteze, koju vrši zadnji deo prevodioca.

Iako su sve faze kompiliranja odvojene na navedene dve etape, one ipak nisu u potpunosti izolovane. Napomenimo da nije nepoznato korišćenje prednjeg dela sa nekim delom zadnjeg dela prevodioca (na primer, analiziranje C kôda se vrši na isti način bez obzira na kojoj se platformi kôd kompilira, ali se koriste različiti zadnji delovi za Linux, Windows, i ostale operativne sisteme) ili zadnjeg dela sa nekim delom prednjeg dela prevodioca (na primer, .NET dozvoljava pisanje kôda u C#, Visual Basic i Visual C++ programskim jezicima, i za njih ima različite prednje delove, ali svi oni se

[2] Pri ovom procesu moguć je nastanak tzv. *međukôda*, odnosno, jezika između izvornog i izvršnog (ciljnog) jezika.

[3] Termini 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.

[4] Preciznije, Java programski prevodilac kombinuje kompiliranje i interpretiranje. Java izvorni kôd se prvo kompilira u međukôd koji se naziva bajtkôd, a zatim se bajtkôd interpretira od strane virtualne mašine. Prednost ovoga se ogleda u tome da bajtkôd kompiliran na jednoj mašini može biti interpretiran na nekoj drugoj (koja, naravno, podržava izvršavanje Java izvršnih programa).

apstrahuju na zajednički međukôd koji se dalje sintetiše na isti način).

Takođe, jedno pitanje koje neko može postaviti jeste u kom jeziku programiramo kompilator. Treba znati da efikasnost kompilatora ne utiče na kompiliranje. Ne mora da znači da će jedan kompilator kompilirati brži program ukoliko je utrošio manje vremena za proces kompiliranja od nekog drugog kompilatora. Što je kompilator bolje konstruisan, to on mora više posla da uradi (tj. više vremena da potroši) prilikom kompiliranja, ali su zato programi koje dobijamo brži. Dakle, to ne znači da je neophodno da kompilatore pišemo u brzim programskim jezicima, već da ih pažljivo konstruišemo.

U ovoj skripti, fokus je stavljen na etapu analize, dok je etapa sinteze nešto čime se bavi proces „konstrukcije kompilatora". Zbog toga ćemo obratiti više pažnje na etapu analize, a zatim ćemo ukratko navešti osnovne delove etape sinteze.

1.1.1 Etapa analize

Etapa analize je, kao što smo napomenuli, prva etapa koju vrši tzv. prednji deo prevodioca. On se bavi analizom, odnosno, velike delove kôda razbija na manje fragmente. Prednji deo prevodioca je vezan za ulazni jezik, tj. viši programski jezik.

Ono što karakteriše etapu analize jeste što je ista i za kompilatore i za interpretatore. Ona je veoma dobro opisana u teoriji. Deli se na tri faze:

1. Leksička analiza, koju vrši leksički analizator,
2. Sintaksička analiza, koju vrši sintaksički analizator, i
3. Semantička analiza, koju vrši semantički analizator.

1.1.2 Etapa sinteze

Etapa sinteze je druga etapa koju vrši tzv. zadnji deo prevodioca. On se bavi sintezom, odnosno, od malih fragmenata pravi jedinstveni izlaz. Kod zadnjeg dela je bitan izlazni jezik, koji je najčešće mašinski jezik ili asembler, ali je bitna i arhitektura, okruženje pod kojim se vrši sinteza zbog optimizacije, itd.

Ova etapa se temelji na specifičnim, često ad hoc rešenjima koja zavise od konkretne mašine. Deli se na tri faze:

1. Generisanje međukôda[5],
2. Optimizacija međukôda, u kojoj se sprovode različite, često komplikovane transformacije, i

[5] Primeri međukôdova koji se dobijaju u ovoj fazi su p-kôd (engl. *p-code*) programskog jezika Pascal i već pomenuti bajtkôd (engl. *bytecode*) programskog jezika Java.

3. Generisanje izvršnog kôda.

Nekada se čitava ova etapa naziva *Generisanje kôda*.

1.1.3 Pitanja i zadaci

Pitanje 1.1.3.1. Šta je programski jezik?

Pitanje 1.1.3.2. Šta je izvorni kôd, a šta izvršni kôd?

Pitanje 1.1.3.3. Šta je neophodno uraditi da bismo napisani kôd mogli pokrenuti u vidu programa na računaru?

Pitanje 1.1.3.4. Šta je programski prevodilac, a šta kompiliranje?

Pitanje 1.1.3.5. Koje vrste programskih prevodilaca postoje? Koje su sličnosti, a koje su razlike između tih vrsta? Koje su prednosti korišćenja jednih u odnosu na druge?

Pitanje 1.1.3.6. Navesti etape kompiliranja, njihove faze, i kratko ih objasniti.

Pitanje 1.1.3.7. Da li su faze kompiliranja međusobno izolovane ili ne? Objasniti i dati (kontra)primer.

Pitanje 1.1.3.8. Šta je međukôd? Navesti primer međukôda.

1.2 Osnovno o fazama analize

Započnimo odeljak o etapi analize sledećim primerom.

Primer 1.2.1

Pretpostavimo da imamo naredni deo programa (fragment kôda) napisanog u programskom jeziku C:

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

Zadatak nam je da predstavimo šta se dešava u svakoj od faza analize pri analiziranju datog fragmenta kôda.

1.2.1 Osnovno o leksičkoj analizi

Definicija 1.3

Deo kompilatora koji obavlja zadatak leksičke analize naziva se *leksički analizator* ili *lekser*.

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 ulazne struje karaktera i identifikuje celine koje nazivamo lekseme.

Definicija 1.4

Leksema je najmanja samostalna jedinica leksičkog sistema.

Primer 1.2.2

Lekseme koje možemo izdvojiti iz primera 1.2.1 su sledeće:

x	=	2	*	y1	+	3	;

Kada se identifikuju, svakoj od leksema se pridružuje odgovarajuća leksička kategorija, kao i jedinstvena odgovarajuća oznaka kategorije koja je pridružena leksemi. Ta oznaka naziva se token. Primeri tokena su identifikator, operator, separator, razni literali (na primer, brojevni literali) itd, što se može zapisati:

```
typedef enum {
    ID, OP, SEP, NUM, ...
} token;
```

Blanko karakteri (razmaci, tabulatori i novi red) se najčešće odbacuju od strane leksičkog analizatora.

Primer 1.2.3

Iz primera 1.2.1 dobijamo da su:

- lekseme `x` i `y1` identifikatori (`ID`),
- lekseme `=`, `*`, i `+` operatori (`OP`),
- lekseme `2` i `3` brojevni literali (`NUM`), i da je
- leksema `;` separator (`SEP`).

Time od početnog (ulaznog) izraza

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

dobijamo (izlazni) izraz

```
ID OP NUM OP ID OP NUM SEP
```

Napomenimo da iako leksički analizator barata sa tokenima, on ne vodi računa o tome da li je, na primer, neki identifikator promenljiva, funkcija, struktura, polje, i slično. U narednom tekstu videćemo da je za to zaslužan semantički analizator.

Osim navedenih operacija, leksički analizator održava i generiše tablicu simbola u kojoj se nalazi spisak svih identifikatora na koje se nailazilo prilikom identifikacije.

Definicija 1.5

Tablica simbola je struktura podataka u kojoj se, tokom etape analize, prikupljaju informacije o tipu, opsegu i memorijskoj lokaciji identifikatora.

Ova tabela, koja se inicijalizuje tokom leksičke analize, dopunjava se i koristi i u ostalim fazama.

Još jedan posao leksičkog analizatora jeste da pamti brojeve linija izvornog kôda.

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).

1.2.2 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 simbola za koje se smatra da daju ispravno struktuiran fragment u traženom programskom jeziku.

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

Definicija 1.6

Deo kompilatora koji obavlja zadatak sintaksičke analize naziva se *sintaksički analizator* ili *parser*.

Primer 1.2.4

Ukoliko bismo samo malo permutovali naredbu iz primera 1.2.1, tj. ako posmatramo fragment

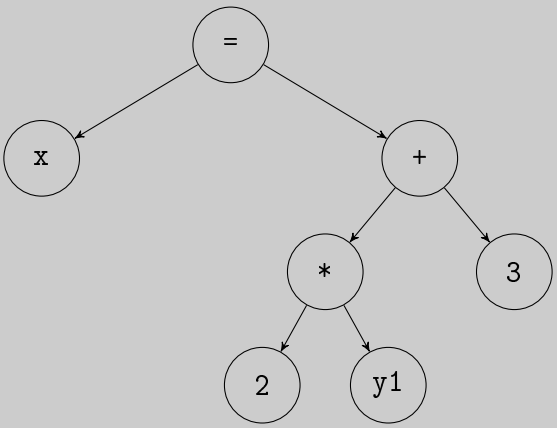
```
2*y1 = x + 3;
```

dobili bismo sintaksički neispravnu naredbu jer vrednosni izraz `2*y1` nije *l-value* u programskom jeziku C, tj. ne možemo joj dodeliti vrednost nekog drugog izraza[6]. U ovom slučaju, sintaksička analiza ne bi uspela, i greška bi bila prijavljena.

Dakle, ulaz za sintaksički analizator predstavljaju tokeni, i on ih proverava. Rezultat rada sintaksičkog analizatora je sintaksičko drvo (engl. *parse tree*).

Primer 1.2.5

Sintaksičko drvo koje se dobija kao rezultat sintaksičke analize fragmenta kôda iz primera 1.2.1 je:



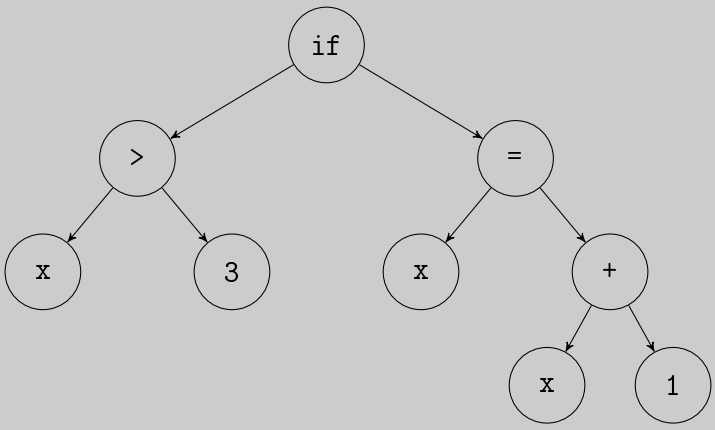
Pogledajmo još jedan, ne značajno komplikovaniji primer sintaksičkog drveta.

Primer 1.2.6

Posmatrajmo još jedan fragment kôda napisan u programskom jeziku C:

```
if(x > 3)
    x = x + 1;
```

Sintaksičko drvo dobijeno sintaksičkom analizom ovog fragmenta izgleda:



[6]Podsetimo se da su jedino promenljive, elementi niza, i memorijske lokacije *l-value* u programskom jeziku C.

1.2.3 Osnovno o semantičkoj analizi

Za razliku od sintakse koja izučava strukture onoga čime se nešto izražava, semantika prirodnih jezika izučava značenje reči.

Definicija 1.7

Deo kompilatora koji obavlja zadatak semantičke analize naziva se *semantički analizator*.

Semantički analizator proverava tipove i deklaracije u sintaksičkom drvetu dobijenog iz prethodne faze, a takođe vrši i implicitnu konverziju kod operatora. U slučaju interpretatora vrši se i izračunavanje vrednosti.

Primer 1.2.7

Posmatrajmo sintaksičko drvo iz primera 1.2.5. Ukoliko smo promenljivu identifikatora `y1` deklarisali tipom `double`, onda će semantički analizator implicitno konvertovati celi broj `2` u broj u pokretnom zarezu `2.0`, da bi se mogle sabrati odgovarajuće vrednosti istog tipa.

Napomenimo i da postoje sintaksički ispravni, a semantički neispravni fragmenti. Ilustrujmo ovaj fenomen sledećim primerom.

Primer 1.2.8

Posmatrajmo sledeći izvorni kôd `primer.c` 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 sintaksički 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, kompiliranjem `gcc primer.c` kompilatorom `gcc` možemo očekivati 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;
               ^
```

1.2.4 Pitanja i zadaci

Pitanje 1.2.4.1. Šta je lekser, a šta parser?

Pitanje 1.2.4.2. Koja je uloga leksičkog analizatora?

Pitanje 1.2.4.3. Šta je leksema, a šta token?

Pitanje 1.2.4.4. Šta je tablica simbola? U kojim fazama analize se ona koristi?

Pitanje 1.2.4.5. Koja je uloga sintaksičkog analizatora i šta je rezultat njegovog rada?

Pitanje 1.2.4.6. Koja je uloga semantičkog analizatora?

Pitanje 1.2.4.7. Da li su svi sintaksički ispravni fragmenti ujedno i semantički ispravni? Da li su svi semantički ispravni fragmenti ujedno i sintaksički ispravni?

Pitanje 1.2.4.8. Koja od faza analize je najsporija? Objasniti.

Pitanje 1.2.4.9. U kojoj fazi analize će biti prijavljena greška prilikom analiziranja narednih fragmenata kôda 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 1.2.4.1. Izdvojiti lekseme iz narednih fragmenata kôda programskog jezika C, i za svaki fragment navesti koliko različitih 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 1.2.4.2. Za svaki fragment kôda iz zadatka 1.2.4.1, prikazati rezultat rada sintaksičkog analizatora.
  ```

Poglavlje 2

Elementi teorije formalnih jezika

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 sekvencijalne 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 promenljih 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

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 2.2

Konačan niz a_1, a_2, ..., $a_n \in \Sigma$ nazivamo *reč*, *niska* ili *string* azbuke Σ. *Prazna reč*, u oznaci ε, reč je dužine nula.

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

Primer 2.1.1

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

Definicija 2.3

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 2.1.2

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

Definicija 2.4

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

Dakle, jezik je proizvoljan podskup reči opšteg jezika neke azbuke. Zbog toga je jasno da postoje konačni i beskonačni jezici. Jezik možemo zadati na dva načina:

1. nabrajanjem njegovih elemenata, i
2. pomoću skupovnih matematičkih definicija.

Primer 2.1.3

Neka je data azbuka $\Sigma = \{a, b\}$. Neka su $L_1, L_2 \subseteq \Sigma^*$ jezici za koje važe da L_1 sadrži sve reči koje počinju nenegativnim brojem slova b i završavaju se slovom a, a da L_2 sadrži sve reči koje počinju pozitivnim brojem slova a i završavaju se istim tim brojem slova b. Jezike L_1 i L_2 možemo zadati nabrajanjem na sledeći način:

$$L_1 = \{a, ba, bba, bbba, bbbba, ...\},$$
$$L_2 = \{ab, aabb, aaabbb, ...\},$$

ili pomoću skupovnih matematičkih definicija na sledeći način:

$$L_1 = \{b^k a \mid k \geq 0\},$$
$$L_2 = \{a^n b^n \mid n > 0\}.$$

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 ćemo zbog toga tu klasu i izučavati nadalje u tekstu. Sa druge strane, klasa KSG pokriva sintaksičku analizu, pa ćemo se njom baviti u narednom poglavlju.

Još jedna zanimljivost jeste ta da, ako pogledamo jezike iz primera 2.1.3, jezik L_2 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), klasa RJ biva preslaba da zadovolji takav uslov. Razmotrimo naredni primer.

Primer 2.1.4

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 uvidićemo da postoji uparivanje zagrada. To nam daje odgovor da klasa RJ nije dovoljno dobra za opisivanje aritmetičkih izraza.

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

2.1.1 Pitanja i zadaci

Pitanje 2.1.1.1. Šta je azbuka? Šta je reč?

Pitanje 2.1.1.2. Šta je opšti jezik neke azbuke? Šta je jezik?

Pitanje 2.1.1.3. Na koje načine možemo zadati jezik, a na koje načine ga možemo opisati?

Pitanje 2.1.1.4. Na šta se odnose skraćenice RJ i KSG? Koje su njihove sličnosti, a koje su razlike?

2.2 Operacije nad rečima i jezicima

U ovom odeljku ćemo 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 pojmova regularni jezik i regularni izraz.

Definicija 2.5

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

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

Jednostavnije rečeno, spajanjem dve reči dobijamo jednu reč tako što „nalepimo" početak druge na kraj prve reči (i, eventualno, pišemo znak $\cdot$ između njih).

Primer 2.2.1

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). Kako se ispostavlja da za svaku reč x proizvoljnog opšteg jezika Σ^* važi i

$$\varepsilon \cdot x = x \cdot \varepsilon = x,$$

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

$$(\forall \mathtt{x},\ \mathtt{x}^{-1} \in \Sigma^* \backslash \{\varepsilon\}) : \ \mathtt{x} \cdot \mathtt{x}^{-1} \neq \varepsilon \wedge \mathtt{x}^{-1} \cdot \mathtt{x} \neq \varepsilon,$$

te zaključujemo da prethodna struktura ne može biti grupa. Međutim, to što nije grupa, ne znači da nije komutativan monoid. Ipak, ukoliko pogledamo primer 2.2.1, vidimo da važi

$$\mathtt{x} \cdot \mathtt{y} = auto \cdot put \neq put \cdot auto = \mathtt{y} \cdot \mathtt{x},$$

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 [1].

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 2.6

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

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

Definicija 2.7

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

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

Definicija 2.8

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

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

Definicija 2.9

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

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

Definicija 2.10

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

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

Primer 2.2.2

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 2.11

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

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

Primer 2.2.3

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 šta je neutral u odnosu na operaciju proizvoda jezika. Prazan skup ne može biti neutral jer za svako $L \subseteq \Sigma^*$, gde je Σ proizvoljna azbuka, važi:

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

dok skup $\{\varepsilon\}$ jeste traženi neutral jer važi:

$$L \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L = L.$$

Definišimo operaciju stepena jezika.

Definicija 2.12

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

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

Primer 2.2.4

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

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

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 $n = 3$:

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

pri čemu četvrta jednakost sledi iz asocijativnosti operacije spajanja.

Primetimo da je u primerima 2.2.3 i 2.2.4 ispunjeno da je kardinalnost proizvoda jezika jednaka proizvodu kardinalnosti svakog od jezika. Da li ovo važi za bilo koji odabir jezika?

Primer 2.2.5

Neka je $L_1 = \{ab, a\}$ i $L_2 = \{b, \varepsilon\}$. Tada je $L_1 \cdot L_2 = \{ab, a\} \cdot \{b, \varepsilon\} = \{abb, ab, a\}$ (u pitanju je skup, pa je element ab koji je dobijen spajanjem $ab \cdot \varepsilon$ 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 2.2.6

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 $L_{\mathbf{N}}$, zadat sa

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

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

Definicija 2.13

Neka je L jezik proizvoljne azbuke Σ. Skup

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

nazivamo *Klinijevo zatvorenje jezika* L, a skup

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

nazivamo *pozitivno Klinijevo zatvorenje jezika* L.

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

Primer 2.2.7

Jezik prirodnih brojeva (tj. jezik čije reči odgovaraju brojevima skupa **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

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

Dodatno,

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

2.2.1 Pitanja i zadaci

Pitanje 2.2.1.1. Šta je operacija spajanja reči? Šta je neutral za tu operaciju?

Pitanje 2.2.1.2. Operacija spajanja reči je:
(a) asocijativna, (b) komutativna, (c) unarna, (d) binarna.

Pitanje 2.2.1.3. Struktura $(\Sigma^*, \cdot, \varepsilon)$ je primer:
(a) monoida, (b) grupe, (c) prstena, (d) polja.

Pitanje 2.2.1.4. Da li je struktura $(\Sigma^*, \cdot, \varepsilon)$ komutativna? Obrazložiti.

Pitanje 2.2.1.5. Šta je: (a) presek, (b) unija, (c) razlika, (d) komplement, (e) Dekartov proizvod, (f) proizvod, (g) stepen jezika?

Pitanje 2.2.1.6. Šta je neutral za operaciju proizvoda jezika?

Pitanje 2.2.1.7. Dokazati ili opovrgnuti tvrđenje $(\forall\Sigma)(\forall L \subseteq \Sigma^*) : L \cdot L^{n-1} = L^{n-1} \cdot L$.

Pitanje 2.2.1.8. 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 2.2.1.9. Šta je Klinijevo zatvorenje jezika, a šta je pozitivno Klinijevo zatvorenje jezika? Da li postoji veza između njih? Ako postoji, kako ona glasi?

Zadatak 2.2.1.1. 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 2.2.5), a zatim konstruisati primer koji ispunjava isti uslov, ali u kojem važi da nijedan skup ne sadrži ε.

Zadatak 2.2.1.2. Koristeći operacije nad jezicima definisati jezik kojim se opisuju standardne registracione tablice.

2.3 Regularni jezici

Iako postoji veliki broj operacija nad jezicima, pokazuje se da se sve operacije mogu svesti na tačno tri operacije, a to su: unija, proizvod i Klinijevo zatvorenje (tj. operacije $\cup$, $\cdot$, i *). Na primer, $L^+ = L \cdot L^*$.

Sada se postavlja sledeće pitanje: Ukoliko su nam ove tri operacije dovoljne, od kojih jezika treba krenuti da bi se dobili ostali jezici? Jasno nam je da možemo jezik L_2 raščlaniti na sledeći način: $L_2 = \{1\} \cup \{2\} \cup \ldots \cup \{9\}$, tj. na skupove čiji je element samo jedna cifra. Slično, kako važi $\{ab\} = \{a\} \cdot \{b\}$, to sve jezike koje imaju reči čiji su karakteri slova, možemo raščlaniti na skupove čiji je element samo jedno slovo. Kako prazna reč nije slovo, to je jasno da će nam biti potreban i skup $\{\varepsilon\}$. Ovom idejom dolazimo do definicije regularnih jezika.

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

1. Ako $a \in \Sigma$, onda je jezik $\{a\}$ regularni jezik.
2. Jezik $\{\varepsilon\}$ 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 2.3.1

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

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

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

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 $\{\varepsilon\}$.
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 ova definicija rekurzivna. Ova definicija se kraće (i neformalno) može opisati sledećom rečenicom: *„Regularan izraz je zapis takav da svakom regularnom izrazu možemo bijektivno pridružiti regularni jezik"*.

Primer 2.3.2

Regularni jezik $\mathbf{N}_0$ opisuje se regularnim izrazom $((1|2|...|9)(0|1|...|9)*)|0$.

Primer 2.3.3

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

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 potvrdan, ali ovu činjenicu nije jednostavno dokazati. Dokaz ćemo prikazati u poglavlju [2.6].

2.3.1 Prioritet operacija nad jezicima

Primer 2.3.4

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

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 operande da ogradimo zagradama (i), i time ćemo postići željeni prioritet.

Primer 2.3.5

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

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

LEMA 2.1. (Levi) Neka su $\mathtt{xy}$ i $\mathtt{zw}$ reči proizvoljnog jezika nad opštom azbukom Σ^*. Tada važi:

$$\mathtt{xy} = \mathtt{zw} \iff (\mathtt{x} = \mathtt{z} \wedge \mathtt{y} = \mathtt{w}) \vee ((\exists \mathtt{t} \in \Sigma^*) : \mathtt{z} = \mathtt{xt} \wedge \mathtt{y} = \mathtt{tw}) \vee ((\exists \mathtt{t} \in \Sigma^*) : \mathtt{x} = \mathtt{zt} \wedge \mathtt{w} = \mathtt{ty}).$$

Napomenimo još i da prvi disjunkt sa desne strane ekvivalencije u Lemi 2.1 ne mora da se eksplicitno napiše jer potpada i u drugi i treći disjunkt sa iste strane ekvivalencije (za $\mathtt{t} = \varepsilon$).

2.3.2 Proširenja regularnih izraza i jezika

Prošireni regularni izrazi

Vremenom kad su ljudi počeli da regularne izraze koriste u računarstvu zaključili da se pisanjem regularnih izraza pomoću samo osnovnih operacija dobijaju veliki zapisi. Zato su uvedeni tvz. **prošireni regularni izrazi**, koji predstavljaju kraće zapise regularnih izraza. Navedimo neke jednostavne proširene regularne izraze:

Primer 2.3.6

- $aa*$ možemo kraće zapisati kao $a+$
- $0|1|2|3|4|5|6|7$ možemo kraće zapisati kao $[0-7]$
- $a|\varepsilon$ možemo kraće zapisati kao $a?$
- ...

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.

Regularne definicije

Regularnim definicijama možemo da imenujemo neki regularni izraz i da ga takvog koristimo u drugim izrazima. Navedimo jedan primer ovakve upotrebe.

Primer 2.3.7

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 oktalnih cifara.

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. Program `lex` poseduje mogućnost upotrebe regularnih definicija.

2.3.3 Pitanja i zadaci

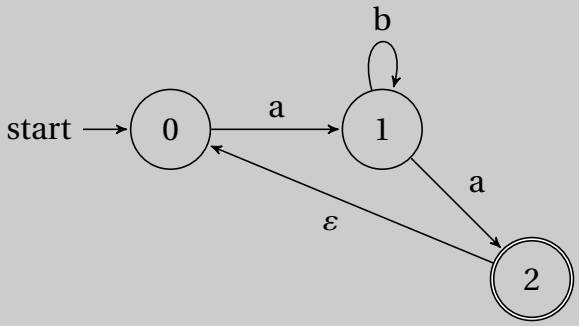
Pitanje 2.3.3.1. Dat je regularni izraz $(ab*a)+$. Koje od sledećih reči pripadaju jeziku koji je opisan datim regularnim izrazom: (a) ε, (b) aaa, (c) $aaaa$, (d) $ababa$, (e) $abaaa$?

Zadatak 2.3.3.1. Napisati regularni izraz za jezik identifikatora u programskom jeziku C.

2.4 Konačni automati

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

Primer 2.4.1



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:

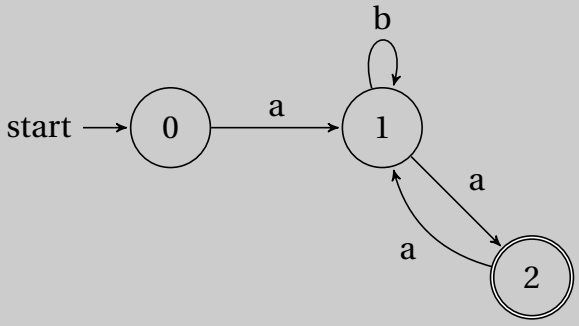
$$_0a_1a_2\varepsilon_0a_1,$$

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 2.4.2

Konačni automat iz primera 2.4.1 ekvivalentan je konačnom automatu:



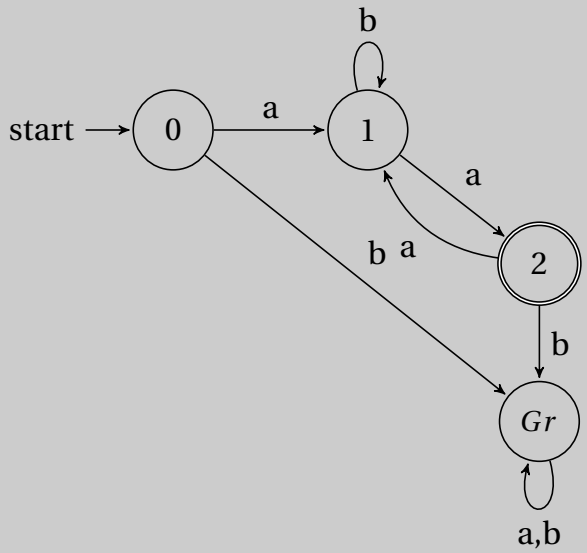
Međutim, ovaj konačni automat naziva se **deterministički** konačni automat (DKA).

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 2.4.3

Konačni automat

[1] Reč *abaaa* ekvivalentna je reči *abaεaa* (podsetite se zašto), te je ovakvo rezonovanje potpuno ispravno.



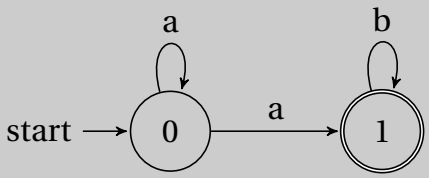
predstavlja upotpunjenje konačnog automata iz primera 2.4.2 i naziva se **potpuni konačni** deterministički automat (PDKA). Stanje Gr naziva se **stanje greške**, i ne prestavlja ništa drugo do unutrašnje stanje. Stanje greške ne sme da bude završno jer bi u tom slučaju automat prihvatio pogrešnu reč. 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 2.4.4

Regularni izraz $a*ab*$ možemo opisati sledećim konačnim automatom:



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

Primer 2.4.5

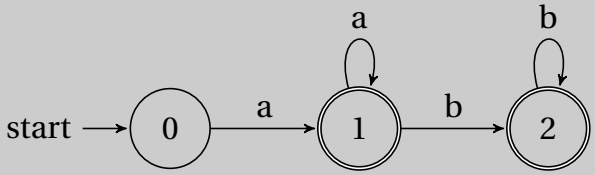
Primenimo konačni automat iz primera 2.4.4 na reč "$aabbb$". Moguća su tri slučaja čitanja ove reči:

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. ${}_0a_1a_?$ – 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 2.4.5 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. Ilustrujmo ovo sledećim primerom.

Primer 2.4.6

Nedeterministički konačni automat iz primera 2.4.4 možemo transformisati u sledeći deterministički konačni automat:



Videćemo u daljem tekstu da važe sledeće dve činjenice:

1. Svaki nedeterministički konačni automat se uspešno može transformisati u deterministički konačni automat, i
2. Rezultat transformacije iz prve činjenice je jedinstven, odnosno, funkcija transformacije iz nedeterminističkog konačnog automata u deterministički konačni automat je injektivna.

2.4.1 Definicija (N)KA

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

automat identični.

(Nedeterministički) Konačni automat, u oznaci *(N)KA*, uređena je petorka

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

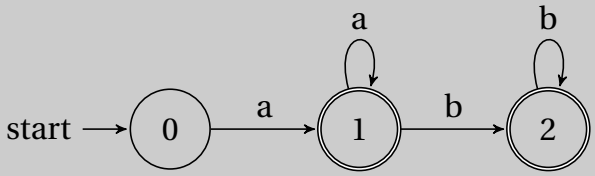
za koju važi:

- Σ je **azbuka** nad čijim rečima primenjujemo automat,
- Q je **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. $\Delta \subseteq Q \times \{\Sigma \cup \{\varepsilon\}\} \times Q$, a često se naziva još i **skup grana automata**.

Pre nego što pređemo na primer, razjasnimo zašto je neophodno uključiti skup $\{\varepsilon\}$ u definiciju relacije prelaska automata. Videli smo da smo još u primeru 2.4.1 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 $\mathcal{A}$, $\mathcal{B}$, ...

Primer 2.4.7

(N)KA iz primera 2.4.6 zadat sledećim grafom:



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

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

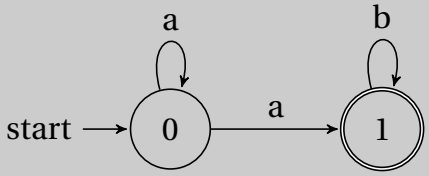
Na ovom mestu ćemo reći da se (N)KA može predstaviti i tabelarno na sledeći način:

$Q\backslash\Sigma$	a	b
0	1	–
1	1	2
2	–	2

Posmatrajmo primer 2.4.7 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 $\delta : (Q \times \Sigma) \longrightarrow Q$.

Primer 2.4.8

(N)KA iz primera 2.4.4 predstavljen grafom



može se tabelarno predstaviti na sledeći način:

$Q\backslash\Sigma$	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 2.4.8 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 $\delta : Q \times \Sigma \longrightarrow P(Q)$, gde je sa $P(Q)$ označen partitivni skup skupa Q.

2.4.2 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.

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:

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

3. Postoji tačno jedno početno stanje, tj.

$$|I| = 1,$$

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

2.4.3 Jezik automata

Neka je petorkom $(\Sigma, Q, I, F, \Delta)$ zadat jedan (N)KA. Radi lakšeg uvođenja 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 $\mathtt{w} = a_1 a_2 \ldots a_n \in \Sigma^*$. Tada je oznaka

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

ekvivalentna

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

za sve $p, q \in Q$.

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

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

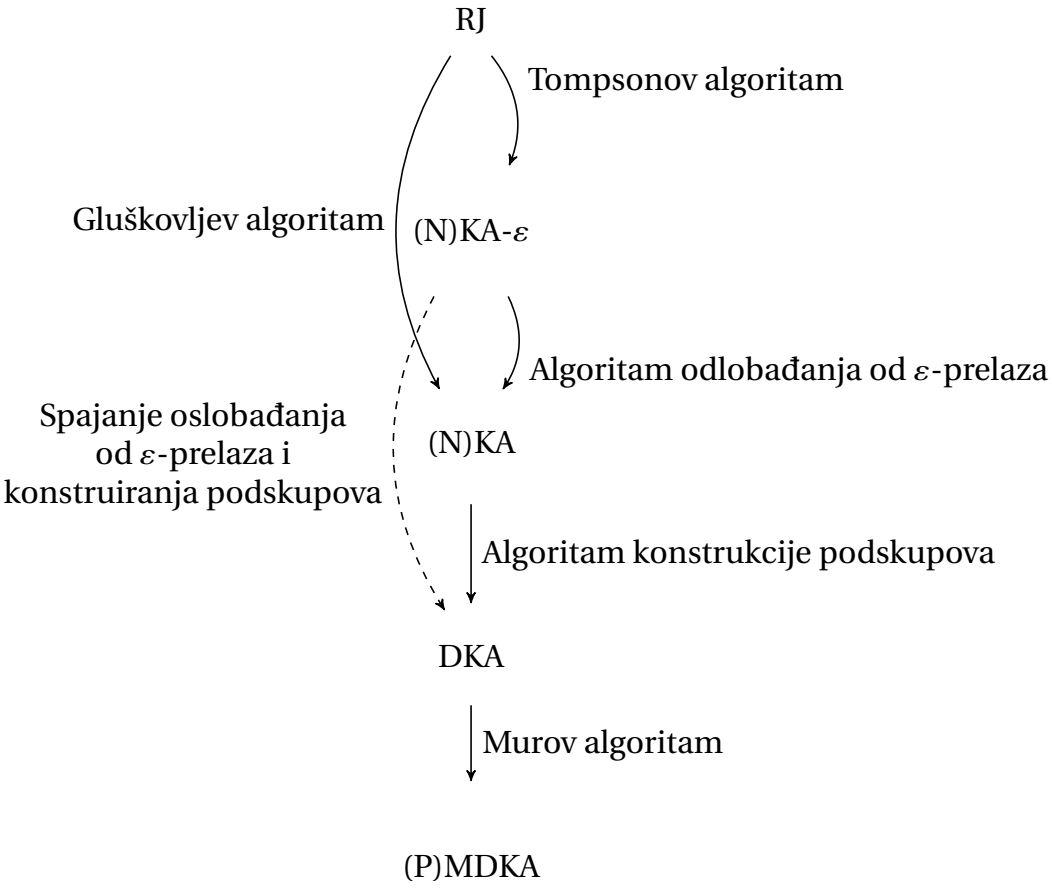
$$L(\mathcal{A}) = \{\mathtt{w} \mid (\exists p \in I \wedge \exists q \in F) : p \xrightarrow{\mathtt{w}} q\}.$$

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

2.5 Klinijeva teorema

TEOREMA 2.2. 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 $\Longrightarrow$):



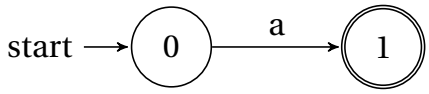
(Smer $\Longleftarrow$): Od proizvoljnog konačnog automata se primenom **Metoda eliminisanja stanja** može dobiti regularni izraz. ■

TEOREMA 2.3. Za dati regularni izraz postoji jedinstven PMDKA (potpuni minimalni deterministički konačni automat).

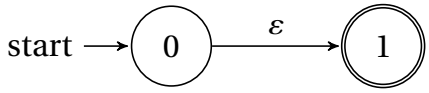
2.5.1 Tompsonov algoritam

Tompsonov algoritam prati rekurzivnu definiciju regularnih izraza:

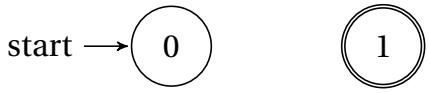
1. Ako je $a \in \Sigma$, automat koji odgovara regularnom izrazu a koji opisuje jezik $\{a\}$ je:



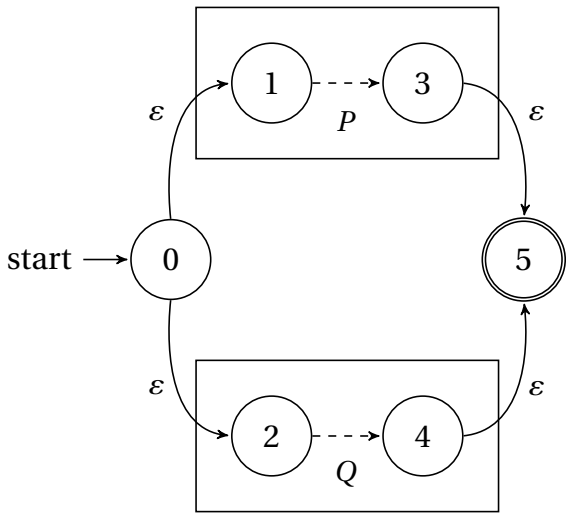
2. Automat koji odgovara regularnom izrazu ε koji opisuje jezik $\{\varepsilon\}$ je:



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

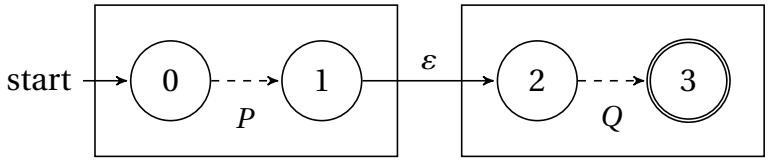


4. Neka su dati regularni izrazi p i q i njihovi odgovarajući automati P i Q, redom. Želimo da konstruišemo (N)KA-ε koji odgovara regularnom izrazu $p|q$. Dodajemo novo početno stanje (stanje 0) i završno stanje (stanje 5) i ε-prelazima ih povezujemo sa početnim i završnim stanjima automata P (stanja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

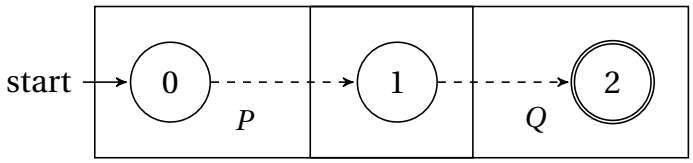


5. Neka su dati regularni izrazi p i q i njihovi odgovarajući automati P i Q, redom. Želimo da konstruišemo (N)KA-ε koji odgovara regularnom izrazu $p \cdot q$ (odnosno, pq). Ovo možemo uraditi na dva načina:

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

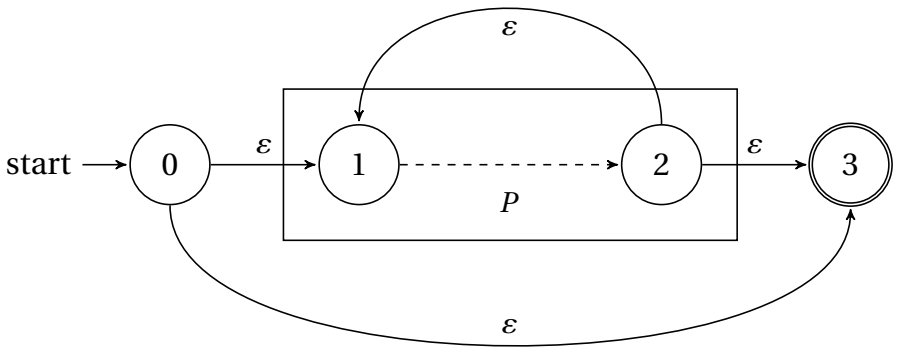


(b) Drugi način je da spojimo završno stanje automata P i početno stanje automata Q (time dobijamo jedinstveno stanje 1).

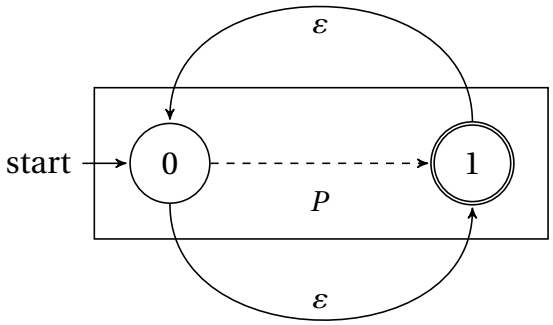


U oba slučaja je početno stanje (N)KA-ε koji odgovara regularnom izrazu $p \cdot q$ zapravo početno stanje automata P (stanje 0, u oba slučaja), dok je njegovo završno stanje zapravo završno stanje automata Q (stanje 3 u prvom slučaju, a stanje 2 u drugom slučaju).

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.



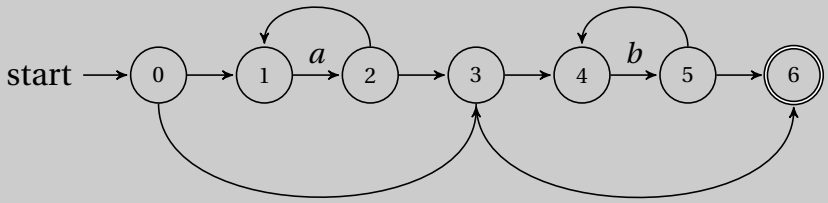
Napomenimo da je konstrukcija (N)KA-ε koji odgovara jeziku $p*$ mogla da se izvede i na sledeći način: Dodajemo samo dva ε-prelaza koji povezuju početno stanje automata P sa završnim stanjem automata P i obrnuto.



Druga konstrukcija takođe opisuje $p*$, ali ne smemo da je koristimo, iako je štedljivije. Preporuka je da kod Klinijevog zatvorenja ne treba štedeti sa dodavanjem prelaza. Naredna dva primera nam ilustruju i zašto.

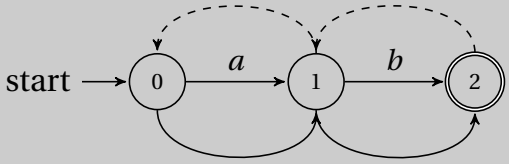
Primer 2.5.1

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



Primer 2.5.2

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



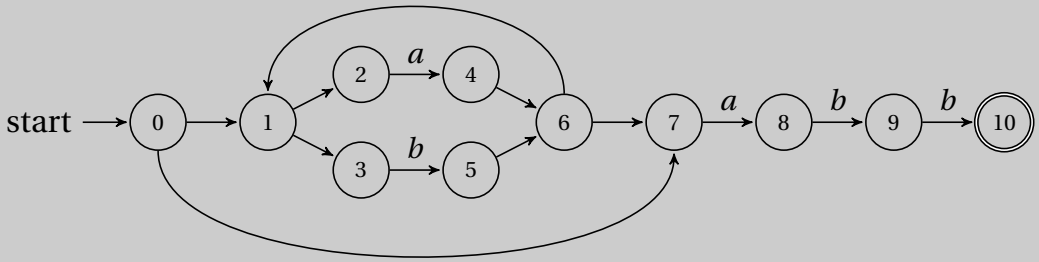
Primetimo da ovaj automat ima put od završnog stanja do početnog (to je put označen isprekidanim prelazima). Zbog toga, on je u stanju da prihvata i reči u kojima se a pojavljuje posle b, a takve reči nisu obuhvaćene jezikom $a*b*$.

Primetimo da u primerima 2.5.1 i 2.5.2 nismo eksplicitno 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.

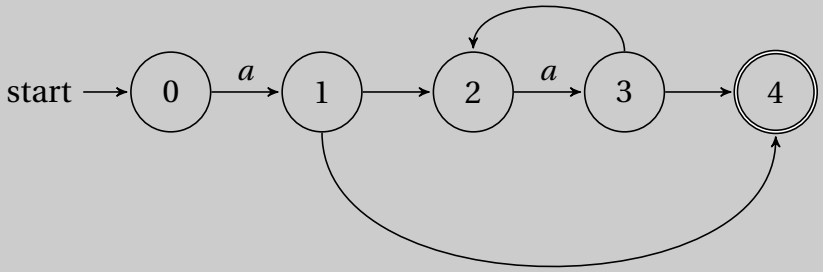
Primer 2.5.3

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

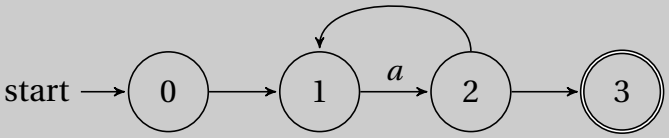


Primer 2.5.4

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

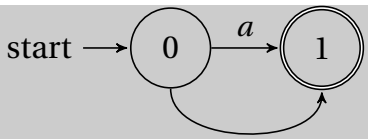


Mada je potpuno korektan, prethodni (N)KA-ε koji odgovara regularnom jeziku $a+$ se može konstruisati i efikasnije, jednostavno uklanjanjem ε-prelaza od početnog do završnog stanja (N)KA-ε koji odgovara regularnom jeziku $a*$.



Primer 2.5.5

Konstruisati automat koji odgovara regularnom jeziku $a?$.



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

1. jedinstveno 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 najviše 2 grana i ulaze najviše 2 grane.

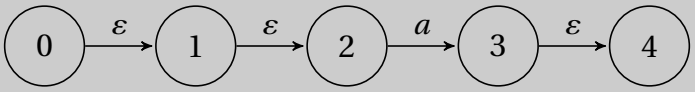
Jedno od najznačajnijih osobina Tompsonovih (N)KA-ε jeste ta da su svi normalizovani. Nije loše primetiti da ako regularni izraz ima r operacija, (N)KA-ε koji se dobija kao rezultat Tompsonovog algoritma ima najviše $2r$ stanja.

2.5.2 Algoritam oslobađanja od ε-prelaza

Prvo ćemo prikazati postupak u opštem slučaju, a potom ćemo fokus staviti na oslobađanje od ε-prelaza Tompsonovih (N)KA-ε od ostalih koje je znatno jednostavnije baš zbog njihove osobine da su normalizovani.

Primer 2.5.6

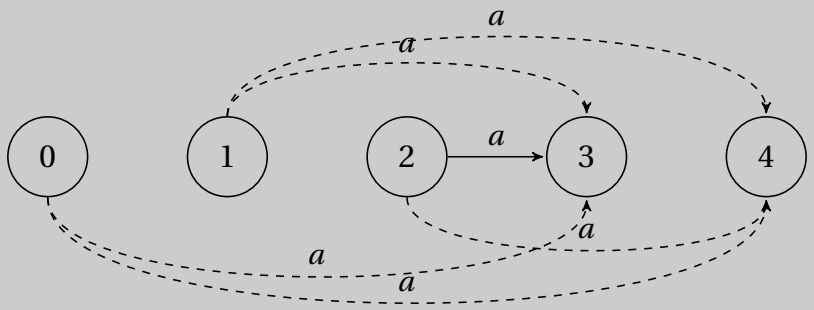
Neka je dat sledeći (N)KA-ε:



Postupak oslobađanja 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:

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

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.

Epsilon zatvorenje stanja p, u oznaci $\overline{\varepsilon}(p)$, predstavlja sledeći skup:

$$\overline{\varepsilon}(p) = \{q \mid p \xrightarrow{\varepsilon} q\}.$$

Primer 2.5.7

Epsilon zatvorenja stanja 0 i 2 iz primera 2.5.6 su: $\overline{\varepsilon}(0) = \{0, 1, 2\}$ i $\overline{\varepsilon}(2) = \{2\}$.

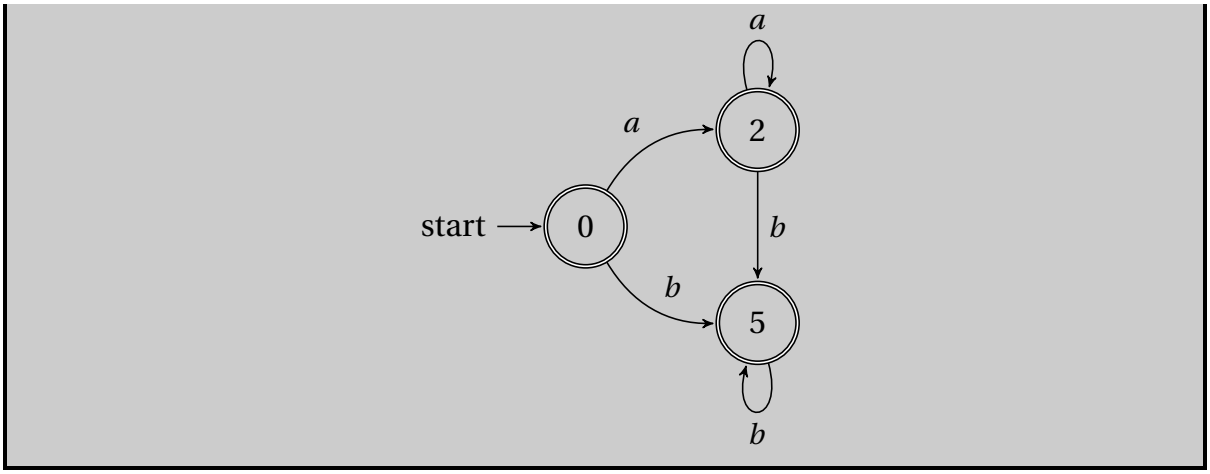
Pri eliminaciji ε-prelaza, u opštem slučaju se pronalaze sve putanje koje imaju ε-prelaz, pa prelaz po nekom slovu, pa zatim ponovo ε-prelaz. Kod (N)KA-ε koji se dobijaju kao rezultat Tompsonovog algoritma situacija se menja. Naime, dovoljno je prespojiti putanje sledećeg oblika:

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

Pokažimo ovo sledećim primerom:

Primer 2.5.8

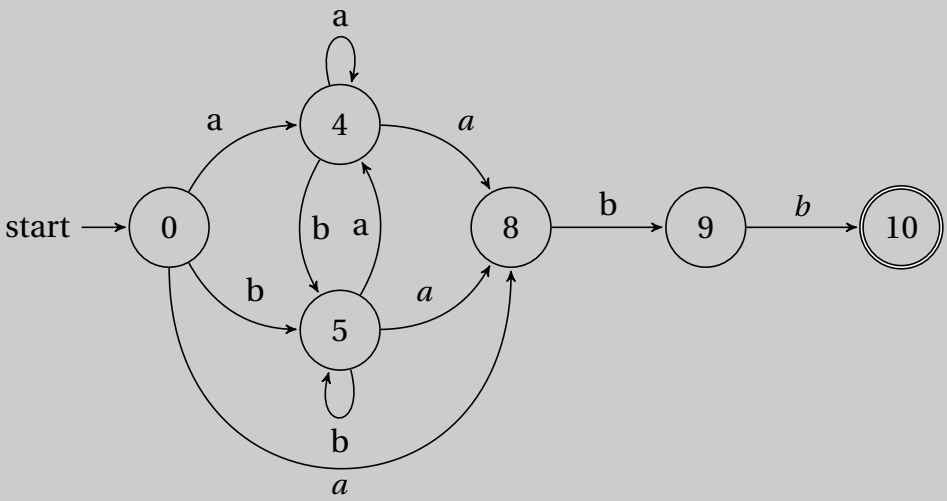
Oslobađanjem ε-prelaza u (N)KA iz primera 2.5.1 dobija se sledeći (N)KA:



Možda nije sasvim jasno zašto su sva tri stanja iz primera 2.5.8 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-ε. Pokažimo da ovo važi na još jednom primeru.

Primer 2.5.9

Oslobađanjem ε-prelaza u (N)KA iz primera 2.5.3 dobija se sledeći (N)KA:



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 narednom poglavlju.

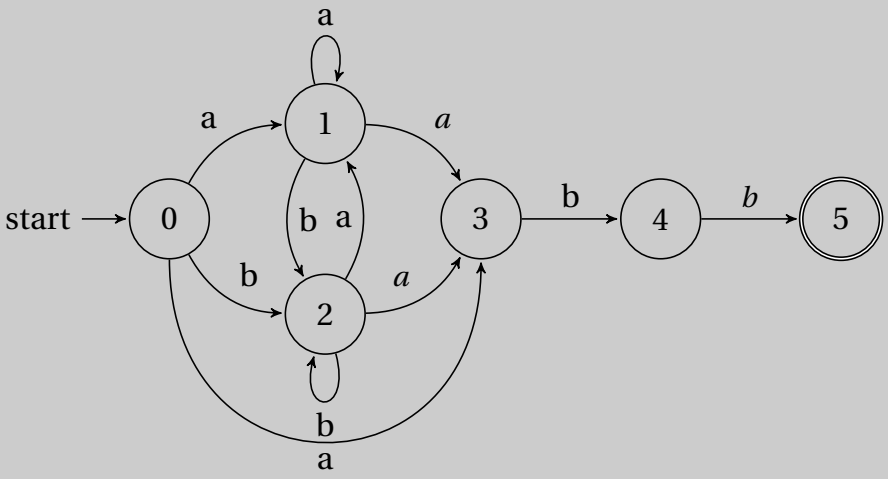
2.5.3 Gluškovljev algoritam

Obeležimo svako pojavljivanje karaktera u regularnom izrazu brojevima od 1 do n, pri čemu je n broj karaktera koji se pojavljuju u regularnom izrazu, a zatim karakter ispred početka izraza obeležimo 0. Ti brojevi predstavljaju stanja do kojih se može doći preko obeleženih karaktera. Zatim gledamo kojim karakterom može da počne reč koja pripada jeziku opisanom datim regularnim izrazom. Stanje koje predstavlja broj kojim smo označili traženi karakter predstavlja početno stanje, a potom posmatramo do kojih sve stanja možemo doći, dodajemo grane preko karaktera, i postupak ponavljamo za svaki broj.

Primer 2.5.10

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

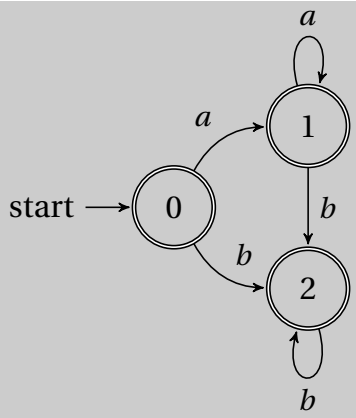
	(	a	|	b	)	$*$	a	b	b
0		1		2			3	4	5



Primer 2.5.11

Konstruisati automat koji odgovara regularnom izrazu $a*b*$.

	a	$*$	b	$*$
0	1		2	



Vidimo da se primenom Tompsonovog algoritma i algoritma oslobađanja od ε-prelaza dobija isti rezultat kao i primenom Gluškovljevog algoritma.

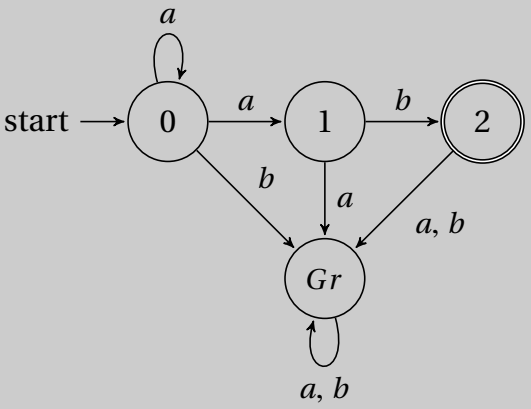
2.5.4 Algoritam konstrukcije podskupova

Algoritam konstrukcije podskupova nam omogućava da od (N)KA dobijemo DKA. Prikažimo postupak konstruisanja podskupova na sledećem primeru:

Primer 2.5.12

Determinizovati automat koji odgovara izrazu $a+b$.

Korak 1: Konstruisati (N)KA koji odgovara zadatom regularnom izrazu. Često se pre same determinizacije automat upotpunjuje. Iako imamo algoritam kojim možemo dobiti potpuno korektan i optimalan (N)KA, ovde ćemo dati jedno ad hoc rešenje. (N)KA koji odgovara zadatom regularnom izrazu je:



Korak 2: Napraviti tabelu prelazaka dobijenog (N)KA:

	a	b
$\longrightarrow 0$	0, 1	Gr
1	Gr	2
(2)	Gr	Gr
Gr	Gr	Gr

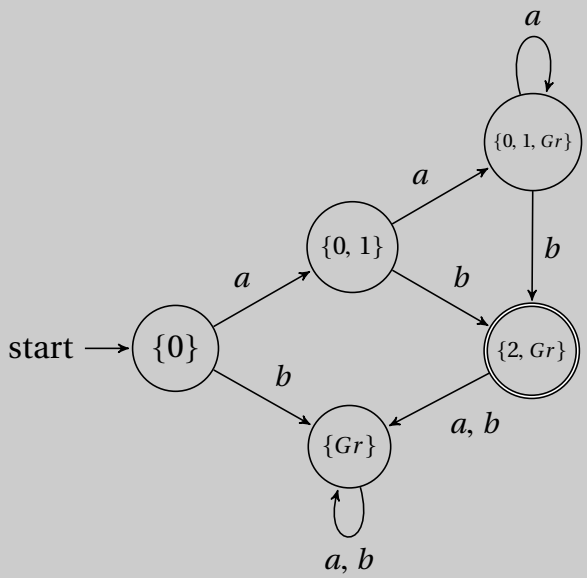
Napomena: Strelicom su obeležena početna stanja, a zaokružena stanja predstavljaju završna stanja.

Korak 3: Konstruisati DKA prema sledećem uputstvu:

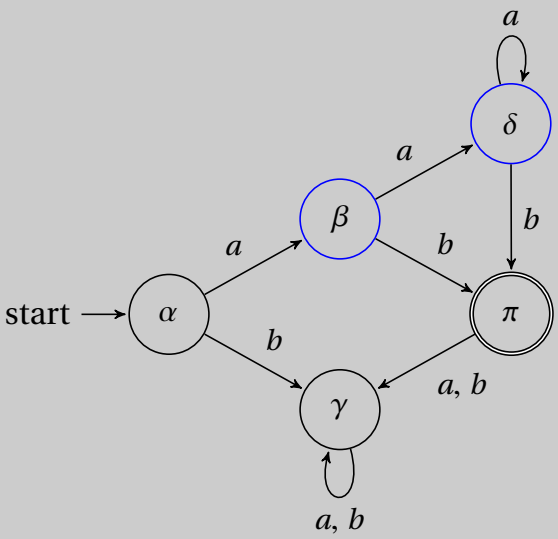
Na početku, skupovi Q, I, F i Δ iz DKA su prazni skupovi. Stanja DKA biće podskupovi stanja (N)KA. Stanja koja su bila početna u (N)KA stavimo u jedan podskup i označimo taj podskup kao početno stanje DKA. Kako je u ovom slučaju jedino stanje 0 bilo početno u (N)KA, onda je traženo početno stanje (tj. 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 može se stići:
  - preko karaktera a, do stanja 0 ili 1, te je $\{0, 1\} = q_2 \in Q$ i $(q_1, a, q_2) \in \Delta$
  - preko karaktera b, do stanja Gr, te je $\{Gr\} = q_3 \in Q$ i $(q_1, b, q_3) \in \Delta$
- Iz stanja q_2 može se stići:
  - preko karaktera a, do stanja 0, 1 ili Gr, te je $\{0, 1, Gr\} = q_4 \in Q$ i $(q_2, a, q_4) \in \Delta$
  - preko karaktera b, do stanja 2 ili Gr, te je $\{2, Gr\} = q_5 \in Q$ i $(q_2, b, q_5) \in \Delta$
- Iz stanja q_3 može se stići:
  - preko karaktera a ili b, do stanja Gr, te je $(q_3, a, q_3) \in \Delta$ i $(q_3, b, q_3) \in \Delta$
- Iz stanja q_4 može se stići:
  - preko karaktera a, do stanja 0, 1 ili Gr, te je $(q_4, a, q_4) \in \Delta$
  - preko karaktera b, do stanja 2 ili Gr, te je $(q_4, b, q_5) \in \Delta$
- Iz stanja q_5 može se stići:
  - preko karaktera a ili b, do stanja Gr, te je $(q_5, a, q_2) \in \Delta$ i $(q_5, b, q_2) \in \Delta$

Ovo nam daje sledeći DKA:

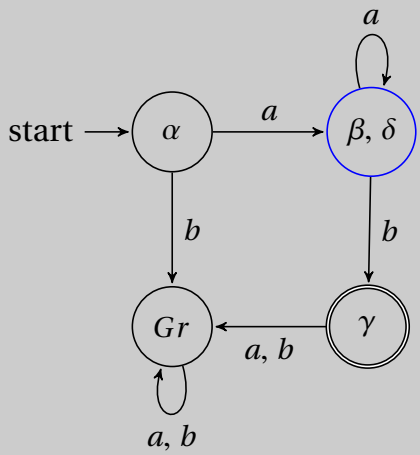


Stanja DKA koja sadrže završna stanja (N)KA označavamo završnim. U ovom slučaju, to je stanje q_5 jer sadrži stanje 2 koje je bilo završno u (N)KA. Broj stanja u najgorem slučaju iznosi 2^n, ali takvi slučajevi su retki. Naravno, stanja ne moramo obeležavati skupovima. Umesto toga možemo ih preimenovati, na primer, slovima grčkog alfabeta:



Primetimo da ovaj automat nije minimalan. Stanja β i δ su identična, jer za oba važi da se preko slova a može stići u stanje δ, a preko slova b se može stići u stanje π. Iako ćemo se u poglavlju [2.4.5] detaljnije pozabaviti algoritmom minimalizacije DKA, nije loše da prikažemo da se nekad i ad hoc metodom može doći do MDKA. Naravno, ovaj metod nije preporučljiv za korišćenje jer ne postoji matematička osnova iza njega, te se ne može formalno dokazati da je korektan. Takođe, ukoliko automati sadrže veliki broj stanja i grana, minimalizacija ad hoc

metodom znatno otežava posao, a i nemoguća je za automatizaciju. Ipak, prikažimo kako izgleda MDKA koji smo minimizovali ad hoc metodom:



Naravno, DKA takođe možemo da predstavimo preko tablice:

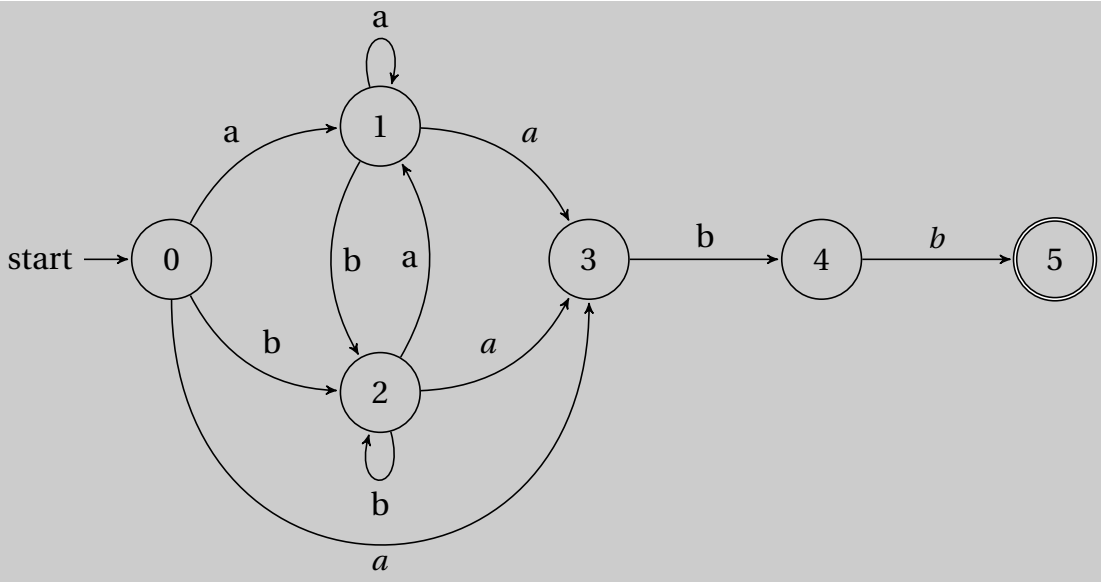
	a	b
$\longrightarrow \{0\}$	$\{0, 1\}$	$\{Gr\}$
$\{0, 1\}$	$\{0, 1, Gr\}$	$\{2, Gr\}$
$\{Gr\}$	$\{Gr\}$	$\{Gr\}$
$\{0, 1, Gr\}$	$\{0, 1, Gr\}$	$\{2, Gr\}$
$\{2,Gr\}$	$\{Gr\}$	$\{Gr\}$

Napomena: Iz tablice vidimo da su stanja $\{0, 1\}$ i $\{0, 1, Gr\}$ ekvivalentna po prelazima i zato ih možemo spojiti. Verovatno bi neko primetio da stanja $\{Gr\}$ i $\{2, Gr\}$ imaju identične prelaske, pa bi ih okarakterisao kao ekvivalentne, ali njih ne smemo da spojimo iako imaju iste prelaske zato što nemaju istu „završnost", tj. jedno stanje je završno, a drugo nije, pa zbog toga ona sigurno nisu ekvivalentna.

Primer 2.5.13

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:

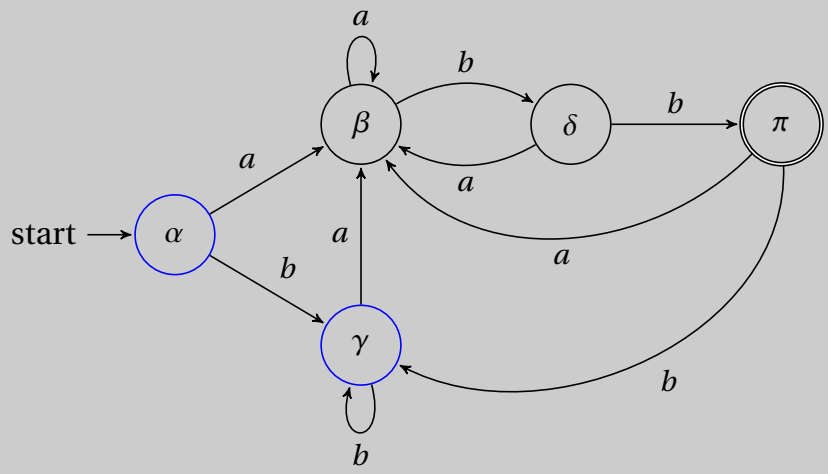
	a	b
$\longrightarrow 0$	1, 3	2
1	1, 3	2
2	1, 3	2
3	–	4
4	–	5
(5)	–	–

Tablica DKA:

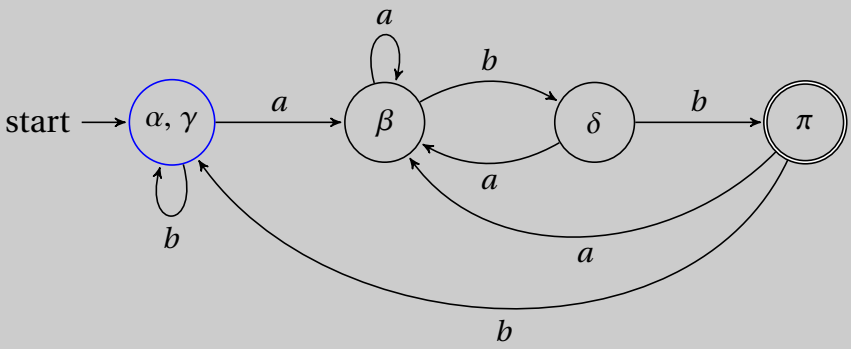
	a	b
$\longrightarrow \{0\}$	$\{1, 3\}$	$\{2\}$
$\{1, 3\}$	$\{1, 3\}$	$\{2, 4\}$
$\{2\}$	$\{1, 3\}$	$\{2\}$
$\{2, 4\}$	$\{1, 3\}$	$\{2, 5\}$
($\{2,5\}$)	$\{1, 3\}$	$\{2\}$

Primetimo da u ovom primeru nismo upotpunili automat pre determinizacije. To se ne smatra lošim postupkom, ali ipak ćemo naglasiti da je zgodnije upotpuniti automat pre determinizacije.

Imenujmo dobijene podskupove $\{0\}$, $\{1, 3\}$, $\{2\}$, $\{2, 4\}$, $\{2, 5\}$ slovima α, β, γ, δ i π, redom. Čitajući tablicu prelaska DKA, dobijeni DKA možemo prikazati grafički:



Minimizovaćemo dobijeni DKA ad hoc metodom:



Formalizam algoritma konstrukcije podskupova

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

Pojasnimo δ sledećim primerom.

Primer 2.5.14

Za automat iz primera 2.5.13 važi:

Tablica (N)KA:

	a	b
	...	
1	1, 3	2
	...	
3	–	4
	...	

Tablica DKA:

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

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.

Teorema 2.1. Za sve skupove stanja q, $q' \in Q_{DKA}$, sva stanja p, $p' \in Q_{(N)KA}$ i reč $\mathtt{w}$ važi:

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

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

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

$$p \cdot\overset{\mathtt{w}}{\cdot}\cdot q,$$

koja označava da se krenuvši od *skupa* stanja p, čitajući reč $\mathtt{w}$, može stići do *skupa* stanja q.

Primer 2.5.15

Iz primera 2.5.13 možemo uočiti da važi sledeće:

$$\{0\} \overset{abb}{\cdots} \{2, 5\},$$

tj. ako se u DKA krene iz stanja $\{0\}$ i pročita reč abb, stići 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 $\mathtt{abb}$ 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:

$$\{1, 3\} \overset{ab}{\cdots} \{2, 4\}.$$

Dokaz teoreme 2.1 se izvodi indukcijom po dužini reči, ali on neće biti prikazan.

2.5.5 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 oznaci L_q, sledeći je skup:

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

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 2.5. 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 $\mathtt{w} = \varepsilon$. Pošto trenutno govorimo o DKA, onda je jasno da neće postojati ε-prelazi, te će ovo uvek biti ispunjeno. ■
Nakon uvođenja ovih oznaka, 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 prepoznaju polazeći iz stanja p i q jednaki. Pišemo:

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

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 2.2. Nerodova ekvivalencija je relacija ekvivalencije.

Jezik stanja q za reči dužine najviše k, u oznaci $L_q^{(k)}$, 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:

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

tj. dva stanja će se razlikovati za reč dužine $k+1$ ako postoji reč dužine $k+1$ koja će jedno stanje odvesti u završno stanje, a drugo stanje u nezavršno stanje. Stanja mogu biti ekvivalentna za sve reči dužine k, ali će 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 2.5.16

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

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

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

   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 $a \in \Sigma$:

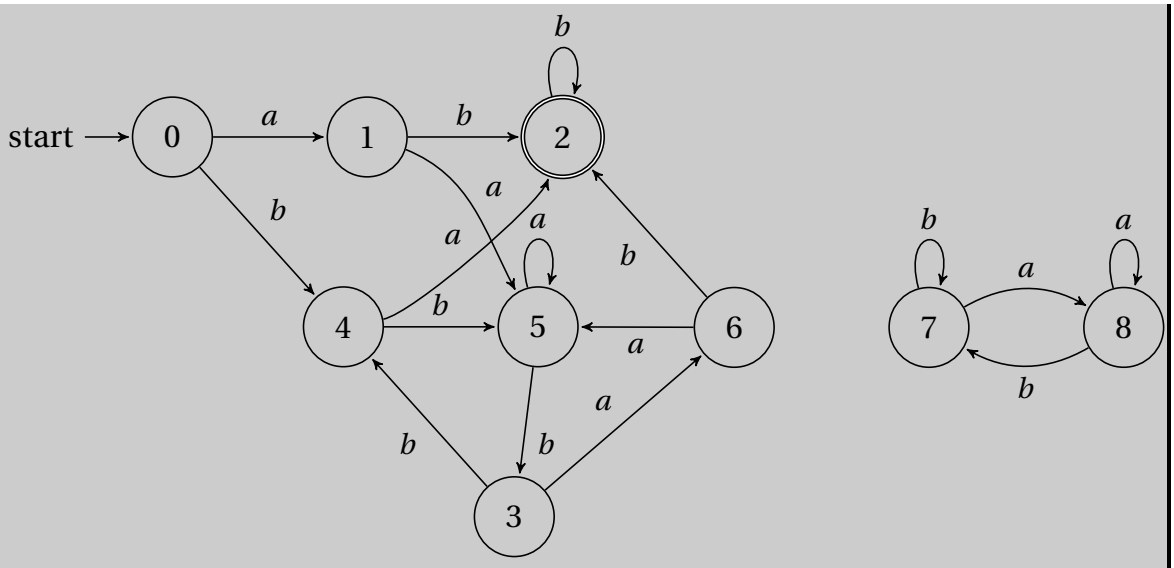
$$ako\ p \xrightarrow{a} p' \ \wedge\ q \xrightarrow{a} q' \ \wedge\ p' \sim_k q',\ onda\ p \sim_{k+1} q.$$

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 $n \in \mathbf{N}$ takav broj da važi $\sim_{n-1} \ \equiv\ \sim_n$. Nakon što nađemo traženo n, dolazimo do zaključka:

$$\sim\ \equiv\ \sim_n .$$

Primer 2.5.17

Minimizovati sledeći DKA:



Primetimo prvo da su stanja 7 i 8 nedostižni iz početnog stanja, te njih možemo da zanemarimo u potpunosti. Prelazimo na 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 klasa 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 ekvivalenciju 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 klasa ekvivalencije prethodno posmatrane relacije.

- razmatramo stanja 0 i 5:

a) $0 \xrightarrow{a} 1 \wedge 5 \xrightarrow{a} 5 \wedge 1 \not\sim_1 5$.

Iz ovoga zaključujemo: $0 \not\sim_2 5$.

- razmatramo stanja 0 i 3:

  a) $0 \xrightarrow{a} 1 \wedge 3 \xrightarrow{a} 6 \wedge 1 \sim_1 6$.

  b) $0 \xrightarrow{b} 4 \wedge 3 \xrightarrow{b} 4 \wedge 4 \sim_1 4$.

  Pošto u azbuci 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:

  a) $1 \xrightarrow{a} 5 \wedge 6 \xrightarrow{a} 5 \wedge 5 \sim_1 5$.

  b) $1 \xrightarrow{b} 2 \wedge 6 \xrightarrow{b} 2 \wedge 2 \sim_1 2$.

  Pošto u azbuci 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 klasa ekvivalencije prethodno posmatrane relacije.

- razmatramo stanja 0 i 3:

  a) $0 \xrightarrow{a} 1 \wedge 3 \xrightarrow{a} 6 \wedge 1 \sim_2 6$.

  b) $0 \xrightarrow{b} 4 \wedge 3 \xrightarrow{b} 4 \wedge 4 \sim_2 4$.

  Pošto u azbuci 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:

  a) $1 \xrightarrow{a} 5 \wedge 6 \xrightarrow{a} 5 \wedge 5 \sim_2 5$.

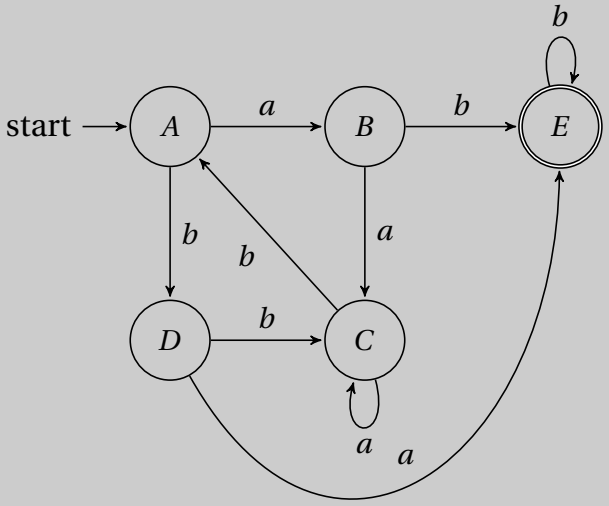
  b) $1 \xrightarrow{b} 2 \wedge 6 \xrightarrow{b} 2 \wedge 2 \sim_2 2$.

  Pošto u azbuci 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:



Videli smo kako nam Murov algoritam omogućava da na ispravan način minimizujemo proizvoljan DKA. U prethodnom primeru nismo upotpunili DKA pre minimizacije, pa se čitaocu prepušta da prvo upotpuni početni DKA, pa onda nad njim da izvrši Murov algoritam i da upotpuni MDKA iz prethodnog primera i da vidi da li postoji razlika između ova dva postupka. Ispravnost Murovog algoritma nam garantuje naredna teorema koju ćemo navesti bez dokaza.

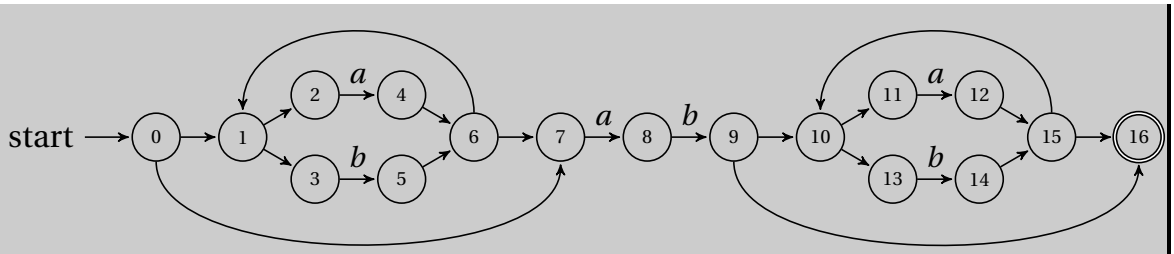
TEOREMA 2.7. Rezultujući MDKA koji se dobija primenom Murovog algoritma na (prethodno upotpunjen) DKA je minimalan do na izomorfizam automata.

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.

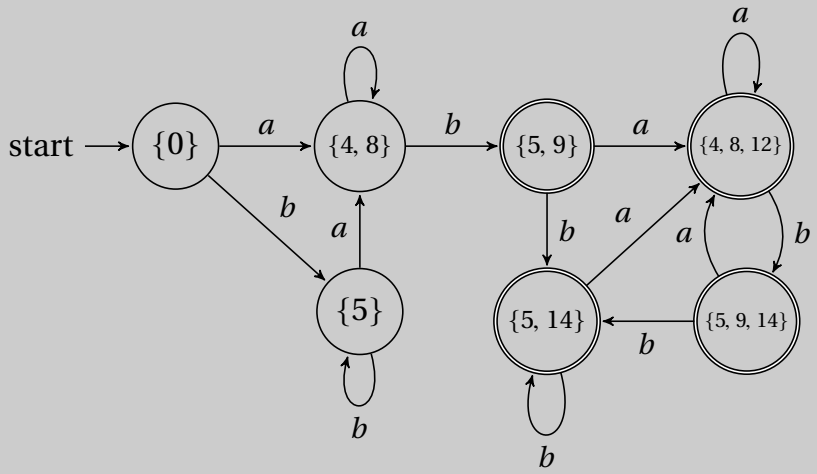
Primer 2.5.18

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

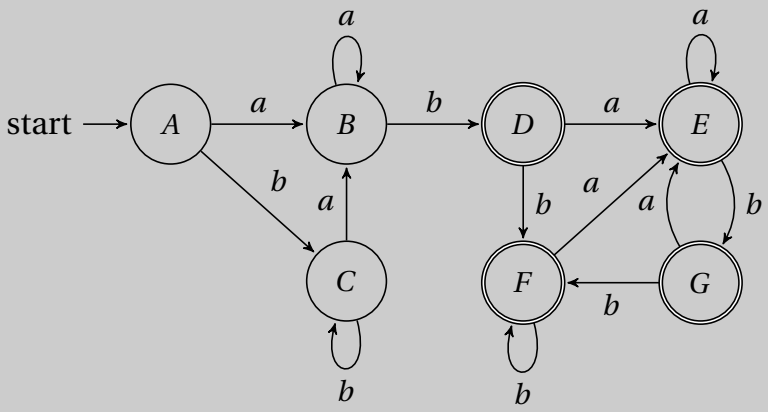
Tompsonov algoritam:



Algoritmi oslobađanja 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 $\sim_1$:

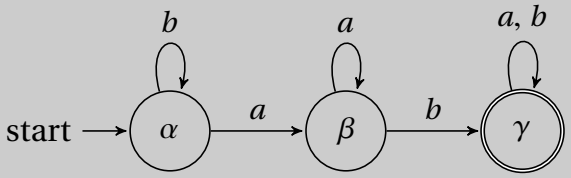
- $A \not\sim_1 B$ ($A \xrightarrow{b} C \wedge B \xrightarrow{b} D \wedge C \not\sim_0 D$).

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:

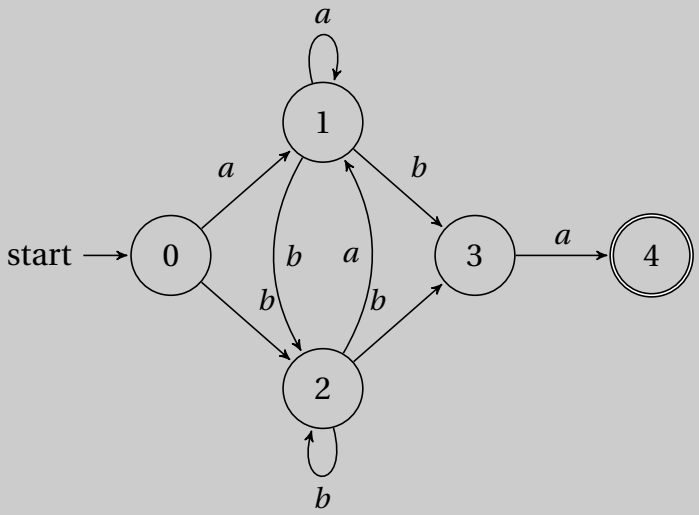


Primer 2.5.19

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

Gluškovljev algoritam:

	(	a	|	b	)	+	b	a
0		1		2			3	4



Algoritam konstrukcije podskupova:

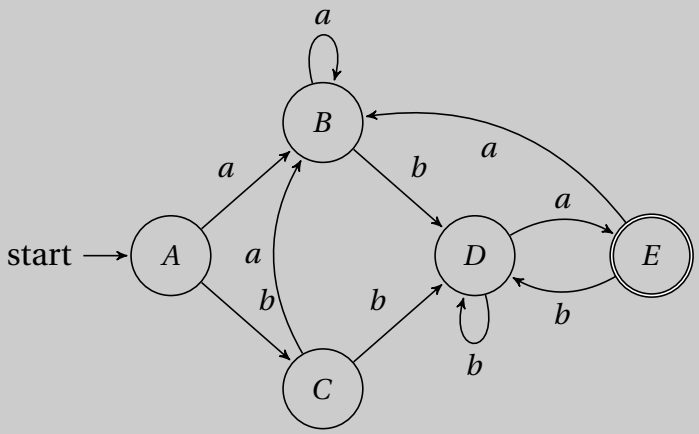
Tablica (N)KA:

	a	b
$\longrightarrow 0$	1	2
1	1	2, 3
2	1	2, 3
3	4	–
(4)	–	–

Tablica DKA:

	a	b
$\longrightarrow \{0\}$	$\{1\}$	$\{2\}$
$\{1\}$	$\{1\}$	$\{2, 3\}$
$\{2\}$	$\{1\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 4\}$	$\{2, 3\}$
($\{1,4\}$)	$\{1\}$	$\{2, 3\}$

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 $\sim_2$:

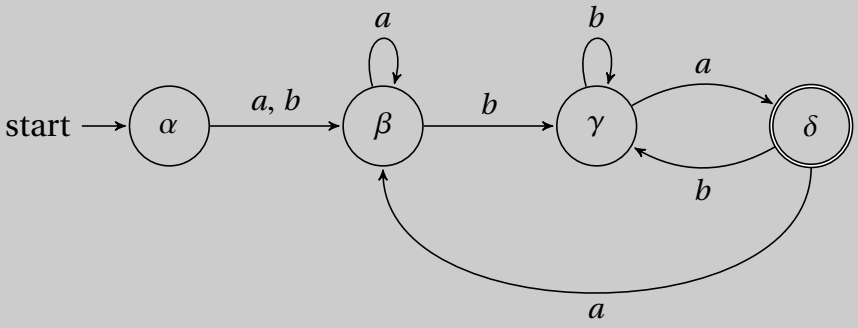
- $A \not\sim_2 B$ $(A \xrightarrow{b} C \wedge B \xrightarrow{b} D \wedge C \not\sim_1 D)$.
- $A \not\sim_2 C$ $(A \xrightarrow{b} C \wedge C \xrightarrow{b} D \wedge C \not\sim_1 D)$.

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:



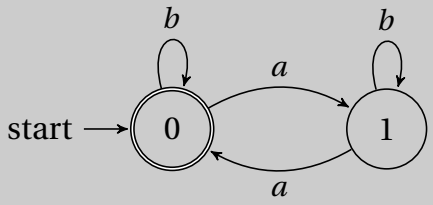
Napomenimo jednu važnu posledicu Klinijeve teoreme: Dva regularna jezika su ekvivalentna akko su odgovarajući PMDKA ekvivalentni.

2.5.6 Metod eliminacije stanja

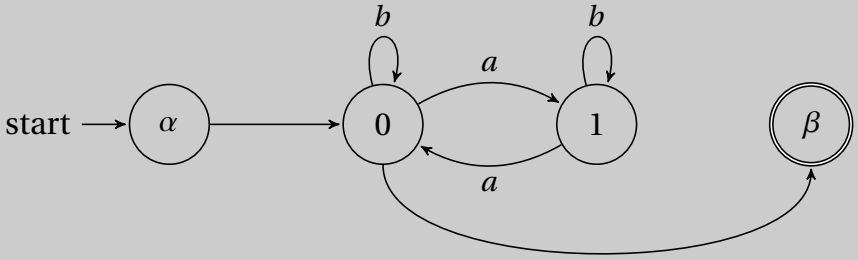
Videli smo kako od regularnog izraza dobijamo PMDKA. Sada nam je zadatak da na osnovu zadatog automata odredimo koji je regularni jezik njime opisan. Primenu metoda ilustrovaćemo na primerima.

Primer 2.5.20

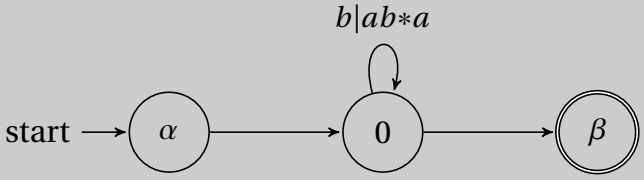
Odrediti koji je regularni jezik opisan sledećim automatom:



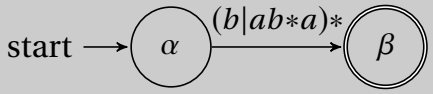
Pripremni korak je dodavanje novih stanja. Dodajemo novo jedinstveno početno stanje α i novo jedinstveno završno stanje β. Cilj je da se oslobodimo ostalih stanja, tj. da nam ostanu samo stanja α i β.
Prvo što radimo jeste povezivanje novih stanja sa starim. Iz stanja α dodajemo grane u sva stara početna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



Da bismo se oslobodili 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 prepoznaje 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, eliminišemo stanje 0:

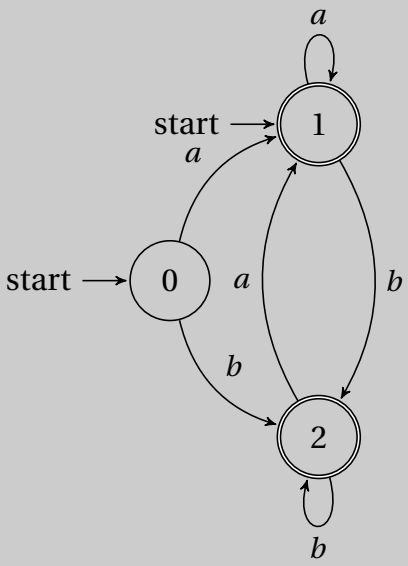


Dakle, zadati automat prepoznaje regularni jezik $(b|ab*a)*$.

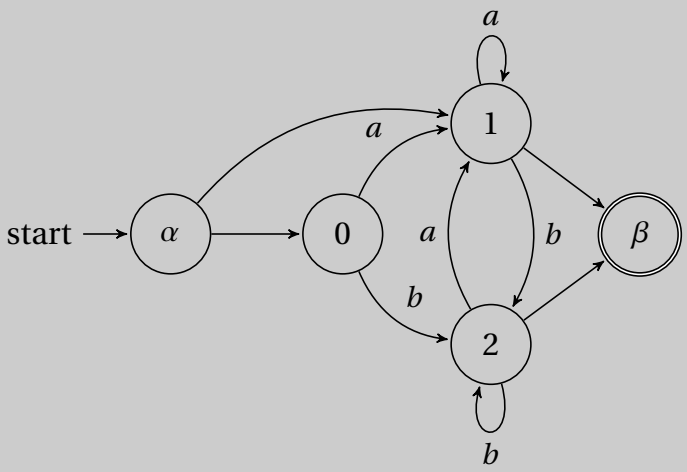
Napomenimo već sada da dužina dobijenog regularnog izraza veoma brzo raste što je automat složeniji. Ponekad je moguće vršiti skraćivanja izraza u hodu, što ćemo ilustrovati sledećim primerom.

Primer 2.5.21

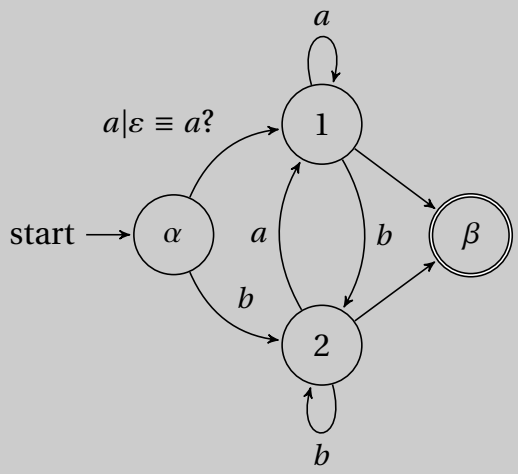
Odrediti koji je regularni jezik opisan sledećim automatom:



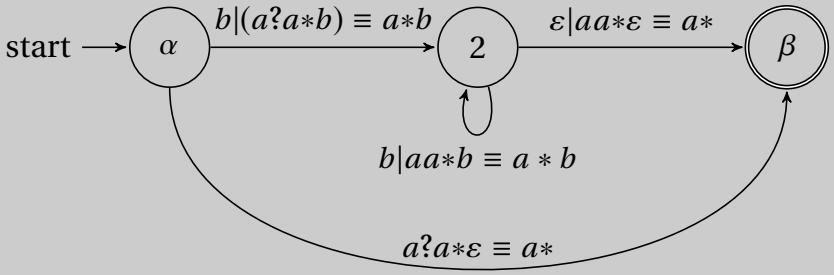
Pripremni korak:



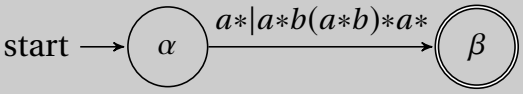
Eliminacija stanja 0:



Eliminacija stanja 1:



Eliminacija stanja 2:



Primetimo sledeće ekvivalentnosti:

$$a*|a*b(a*b)*a* \equiv a*|(a*b)+a* \equiv (\varepsilon|(a*b)+)a* \equiv (a*b)*a*.$$

Dakle, automat prepoznaje regularni jezik $(a*b)*a*$.

2.5.7 Pitanja i zadaci

Zadatak 2.5.7.1. Upotpuniti deterministički konačni automat iz primera 2.4.6.

Zadatak 2.5.7.2. 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 oslobađanja od ε-prelaza, a potom i primenom Gluškovljevog algoritma.

Zadatak 2.5.7.3. Dokazati lemu 2.2.

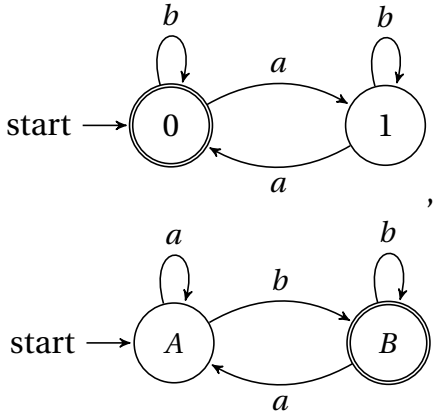
2.6 Određivanje preseka jezika

Po definiciji, dopušteni operatori koje možemo koristiti za konstruisanje novih regularnih jezika jesu spajanje, Klinijevo zatvorenje i unija. Jasno je da ako imamo dva regularna jezika, rezultat njihove unije (tj. njihovog spajanja ili primene Klinijevog zatvorenja) takođe će biti regularan jezik. Ostaje da se razjasni na koji način možemo opisati presek dva regularna jezika.

Neka je $\Sigma = \{a_1, a_2, ..., a_n\}$. Pošto je Σ^* regularan jezik (i znamo da ga možemo predstaviti regularnim izrazom $(a_1|a_2|...|a_n)*$), 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. Pokazaćemo posredno da su regularni izrazi zatvoreni za operaciju preseka.

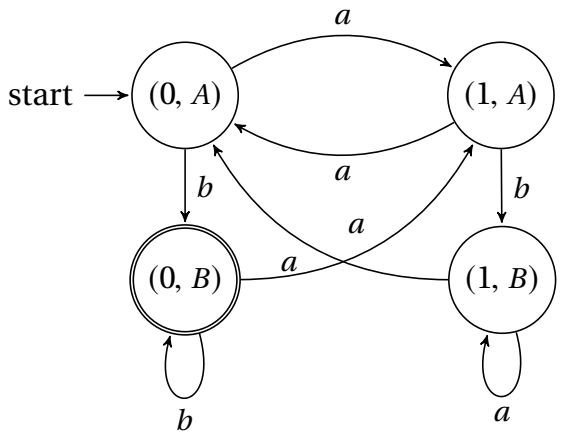
LEMA 2.8. Za dva konačna automata moguće je napraviti treći konačni automat koji prepoznaje njihov presek.

Dokaz. Bez gubitka opštosti, a to nam dosta olakšava dokaz, pretpostavićemo da su automati kojima računamo preseke 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 konstrukcije 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 $(0, A)$. Slično, završna stanja moraju da sadrže završna stanja oba automata, a to je $(0, B)$.



■

Proizvod istovremeno simulira oba automata. Ono zbog čega je proizvod automata koristan jeste što pravim odabirom završnih stanja možemo dobiti različite automate:

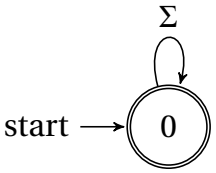
$\mathcal{A} \cap \mathcal{B}$	(0, B)
$\mathcal{A} \cup \mathcal{B}$	(0, A), (0, B), (1, B)
$\mathcal{A}\backslash\mathcal{B}$	(0, A)
$\mathcal{B}\backslash\mathcal{A}$	(1, B)

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:

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

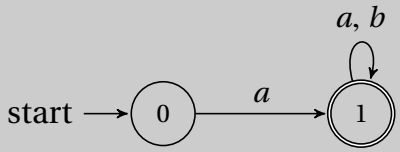
pri čemu je $\Sigma*$ zapis za automat, koji opisuje regularni jezik Σ^*, a koji je predstavljen sledećim automatom:



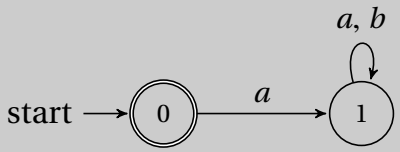
Kada znamo da konstruišemo Dekartov proizvod automata, onda znamo i razliku, pa znamo i komplement (na osnovu prethodne definicije). Ipak, operaciju komplementa automata možemo sprovesti na sledeći način: Neka je zadat PDKA $\mathcal{A} = (\Sigma, Q, I, F, \Delta)$. Komplementarni automat $\mathcal{A}^c$ je automat $(\Sigma, Q, I, Q\backslash F, \Delta)$. 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. Ilustrujmo ovo sledećim primerom:

Primer 2.6.1

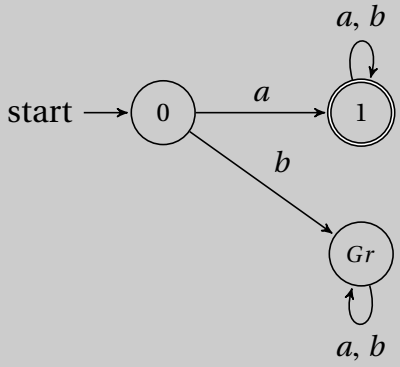
Automat koji prepoznaje regularni jezik $a(a|b)*$ je sledeći automat:



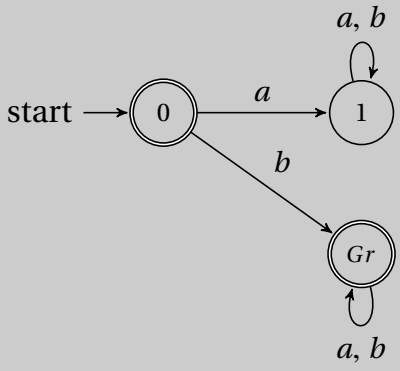
Prateći prethodno uputstvo, automat koji prepoznaje regularni jezik $(a(a|b)*)^c$ trebalo bi da bude sledeći automat:



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



Sada se ispravan komplement dobija primenom prethodnog pravila:



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.

TEOREMA 2.8. Jezik L nije regularan.

Dokaz. Pretpostavimo suprotno, tj. da je jezik L regularan. Na osnovu Klinijeve teoreme, postoji (N)KA koji prepoznaje jezik L, i neka je to automat $\mathcal{A} = (\Sigma, Q, I, F, \Delta)$. 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 $\mathtt{w} \in L$ koja je duža od k važi sledeće: prilikom njenog prihvatanja će se kroz barem jedno stanje (označimo ga q) proći dva puta (na osnovu Dirihleovog principa), tj. za neke $q_0 \in I$ i $q_r \in F$:

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

gde je $\mathtt{w} = \mathtt{xyz}$. Rečima, postojaće neko stanje q tako da ukoliko se krene iz stanja q_0, pročita neka reč $\mathtt{x}$, zatim stigne do stanja q koje se može ponavljati (preko reči $\mathtt{y}$ se opet stigne do stanja q) i nadalje se čita reč $\mathtt{z}$ i stigne do stanja q_r.
Kako $\mathtt{w} \in L$, to i $\mathtt{xz} \in L$ (jer postoji put $q_0 \xrightarrow{\mathtt{x}} q \xrightarrow{\mathtt{z}} q_r$ u automatu). Već znamo da $\mathtt{xyz} \in L$. Takođe, reč $\mathtt{xyyz} \in L$ (jer postoji put $q_0 \xrightarrow{\mathtt{x}} q \xrightarrow{\mathtt{y}} q \xrightarrow{\mathtt{y}} q \xrightarrow{\mathtt{z}} q_r$ u automatu). Lako se vidi da sve reči oblika $\mathtt{xy}^i\mathtt{z}$ pripadaju jeziku L, za $i \geq 0$.

Dakle, postoji reč $\mathtt{w} \in L$, takva da je $|\mathtt{w}| > |Q|$ i koja je oblika:

$$\underbrace{aa\ldots a}_{n}\underbrace{bb\ldots b}_{n}.$$

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

Posmatrajmo razbijanje:

$$\underbrace{aa\ldots a}_{\mathtt{x}}\underbrace{aa\ldots a}_{\mathtt{y}}\underbrace{aa\ldots abbb\ldots bbb}_{\mathtt{z}}.$$

ili simetrično razbijanje:

$$\underbrace{aa\ldots abb\ldots b}_{\mathtt{x}}\underbrace{bb..b}_{\mathtt{y}}\underbrace{bb\ldots b}_{\mathtt{z}}.$$

Primetimo da u ovim slučajevima važi $\mathtt{xz} \notin L$, te ovakva razbijanja nisu moguća.
Posmatrajmo razbijanje:

$$\underbrace{aa\ldots a}_{\mathtt{x}}\underbrace{aa\ldots abb\ldots b}_{\mathtt{y}}\underbrace{bb\ldots b}_{\mathtt{z}}.$$

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

nije moguće.

Zaključujemo da ne možemo da rastavimo reč $\mathtt{w}$ na tri reči. Kontradikcija. ■
Napomenimo da smo pri dokazu ove teoreme koristili lemu koja se poznata pod imenom **Lema pumpanja** (*Pumping lemma*). Ona govori o osobini jezika da se svaka reč može razbiti na tri dela, tj. $\mathtt{w} = \mathtt{xyz}$, tako da važi $\mathtt{xy}^i\mathtt{z} \in L$, za sve $i \geq 0$.

Poglavlje 3

Kontekstnoslobodne gramatike

Kontekstnoslobodne gramatike (češće ćemo koristiti oznaku KSG ili 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 pomoćni simboli).

Posmatrajmo sledeći primer:

Primer 3.0.1

Neka je data gramatika G sa sledećim skupom pravila:

1. A $\longrightarrow$ aAb, i
2. A $\longrightarrow \varepsilon$.

Ovo je jedan jednostavan primer kontekstno slobodne 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:

$$A \overset{1}{\Longrightarrow} aAb \overset{1}{\Longrightarrow} aaAbb \overset{2}{\Longrightarrow} aabb.$$

Kada dobijemo reč u kojoj nema neterminalnih simbola, onda za tu reč možemo reći da je **reč jezika opisane gramatike**. Vidimo da reč $\mathtt{w} = aabb$ 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 $\varepsilon = a^0b^0 \in L$. Kako dobijamo ε koristeći prethodno navedena pravila? Primenom pravila 2 (samo jednom), dobija se tražena pražna reč:

$$A \overset{2}{\Longrightarrow} \varepsilon.$$

Navedimo još dva primera KS gramatike.

Primer 3.0.2

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

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

što se često kraće zapisuje:

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

Primer 3.0.3

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

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

Iz primera 3.0.3 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.

3.1 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 $P \subseteq N \times (NU\Sigma)^*$. 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 3.1.1

Gramatika iz primera 3.0.2, koju smo označili G_2, obuhvata sledeće komponente: $\Sigma = \{a, b\}$, $N = \{A\}$, $S = A$ i $P = \{(A, aAb), (A, ab)\}$.

Napomenimo da pravila nećemo pisati kao uređene parove, već onako kako smo pisali u primerima 3.0.1 do 3.0.3.

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 3.1.2

Neka je zadato pravilo gramatike $A \longrightarrow aABb$ i neka smo nekim izvođenjem nezavršnog simbola S došli do $aAbAb$.
Dalje možemo zameniti prvi simbol A na sledeći način:

$$S \Longrightarrow \ldots \Longrightarrow a\underline{A}bAb \Longrightarrow a\underline{aABb}bAb.$$

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

$$S \Longrightarrow \ldots \Longrightarrow aAb\underline{A}b \Longrightarrow aAb\underline{aABb}b.$$

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 3.1.3

Neka je zadato pravilo gramatike $bAb \longrightarrow aABb$ 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:

$$S \Longrightarrow \ldots \Longrightarrow aAb\underline{Ab} \Longrightarrow aA\underline{aABb}.$$

Usvojimo dodatna pravila zapisivanja:

- $\longrightarrow$ 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
- $\Longrightarrow$ 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 $\Longrightarrow$. Kažemo da **niska** $\alpha X \beta$ **izvodi nisku** $\alpha\gamma\beta$, u oznaci $\alpha X \beta \Longrightarrow \alpha\gamma\beta$, akko postoji pravilo $X \longrightarrow \gamma \in P$, pri čemu su α, β, $\gamma \in (N \cup \Sigma)^*$ i $X \in N$.

Relacija označena simbolom $\Longrightarrow^*$ je refleksivno tranzitivno zatvorenje relacije izvođenja. To je najmanja tranzitivno-refleksivna relacija koja sadrži datu relaciju, tj. ima 0 ili više koraka izvođenja. Relacija označena simbolom $\Longrightarrow^+$ je tranzitivno 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:

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

3.2 Drvo izvođenja

Primer 3.2.1

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}:

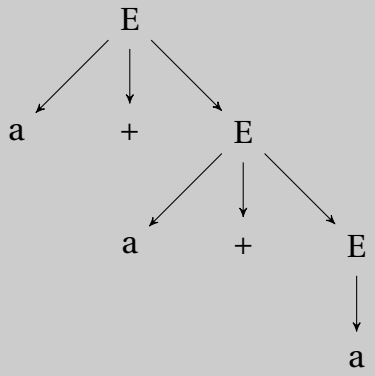
1. $E \longrightarrow a$, i
2. $E \longrightarrow a + E$.

Primer izvođenja izraza $a + a + a$:

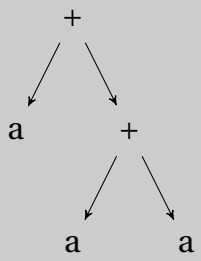
$$E \overset{2}{\Longrightarrow} a + E \overset{2}{\Longrightarrow} a + a + E \overset{1}{\Longrightarrow} a + a + a.$$

Ovakva gramatika, u kojoj se uvek menja desni neterminalni simbol, naziva se **desno rekurzivna gramatika**.

Izvođenje se takođe može predstaviti i **drvetom izvođenja**:



Drvo izvođenja apstrahuje detalje koje se vide u izvođenju, ali koja nisu bitna. Izvođenje se može predstaviti i **drvetom apstraktne sintakse**. Ono ne prikazuje pomoćne simbole.



Ovo drvo predstavlja desno asocijativno drvo (na slici se vidi kako se drvo „nagnulo' na desnu stranu) i zato je u ovoj gramatici operator + desno asocijativan (u gramatici, ovo se ogleda u pravilu $E \longrightarrow a + E$). Dakle, desna rekurzija vodi ka desnoj asocijativnosti.

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

Primer 3.2.2

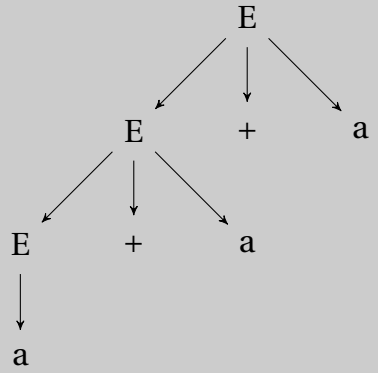
Definišemo pravila gramatike koju ćemo označiti G_{+L}:

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

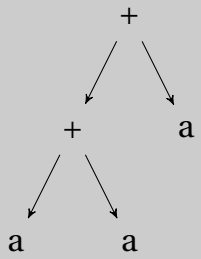
Primer izvođenja izraza $a + a + a$:

$$E \overset{2}{\Longrightarrow} E + a \overset{2}{\Longrightarrow} E + a + a \overset{1}{\Longrightarrow} a + a + a.$$

Ovakva gramatika, u kojoj se uvek menja levi neterminalni simbol, naziva se **levo rekurzivna gramatika**. Drvo izvođenja u ovom slučaju je:



Drvo apstraktne sintakse u ovom slučaju je:



U ovom slučaju, ovo drvo predstavlja levo asocijativno drvo (na slici se vidi kako se drvo „nagnulo" na levu stranu) i zato je u ovoj gramatici operator + levo asocijativan (u gramatici, ovo se ogleda u pravilu $E \longrightarrow E + a$). Dakle, leva rekurzija vodi ka levoj asocijativnosti.

Iako je nama svejedno za ovakve izraze da li ćemo koristiti levu ili desnu asocijativnost, ako bismo hteli da ih opišemo u programskom jeziku C, koristili bismo gramatiku označenu G_{+L}, jer su aritmetički operatori u C-u definisani kao levo asocijativni.

Postoji još jedan način na koji možemo da definišemo gramatiku za izraze iz primera

3.2.1 Prikažimo ga sledećim primerom.

Primer 3.2.3

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

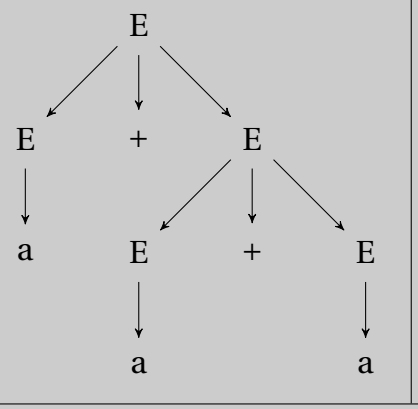
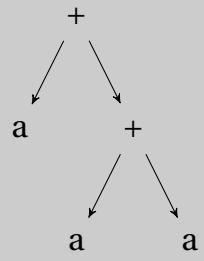
1. $E \longrightarrow a$, i
2. $E \longrightarrow E + E$.

U ovoj gramatici, za izraz $a + a + a$ postoje sledeća izvođenja:

1. najlevlje izvođenje (*leftmost derivation*):

$$I_1 : E \overset{2}{\Longrightarrow} E + E \overset{1}{\Longrightarrow} a + E \overset{2}{\Longrightarrow} a + E + E \overset{1}{\Longrightarrow} a + a + E \overset{1}{\Longrightarrow} a + a + a,$$

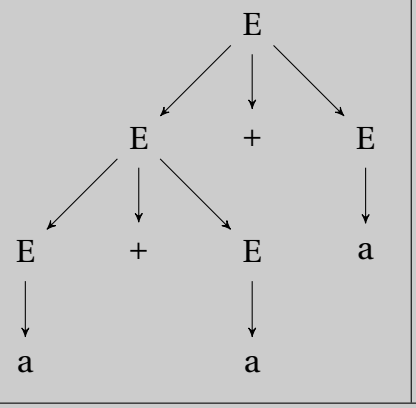
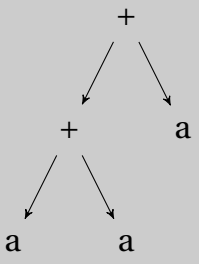
sa svojim drvetom izvođenja i drvetom apstraktne sintakse:

Drvo izvođenja	Drvo apstraktne sintakse
	

2. najdešnje izvođenje (*rightmost derivate*):

$$I_2 : E \overset{2}{\Longrightarrow} E + E \overset{1}{\Longrightarrow} E + a \overset{2}{\Longrightarrow} E + E + a \overset{1}{\Longrightarrow} E + a + a \overset{1}{\Longrightarrow} a + a + a,$$

sa svojim drvetom izvođenja i drvetom apstraktne sintakse:

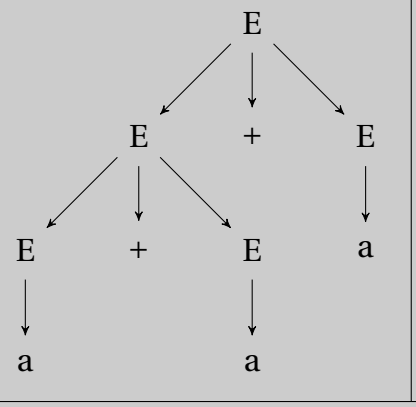
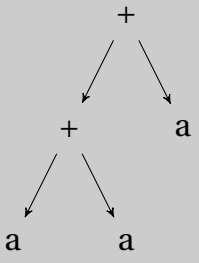
Drvo izvođenja	Drvo apstraktne sintakse
	

3. kombinovana izvođenja:

$$I_3 : E \overset{2}{\Longrightarrow} E+E \overset{1}{\Longrightarrow} E+a \overset{2}{\Longrightarrow} E+E+a \overset{1}{\Longrightarrow} a+E+a \overset{1}{\Longrightarrow} a+a+a,$$

$$I_4 : E \overset{2}{\Longrightarrow} E+E \overset{2}{\Longrightarrow} E+E+E \overset{1}{\Longrightarrow} a+E+E \overset{1}{\Longrightarrow} a+a+E \overset{1}{\Longrightarrow} a+a+a,$$

sa svojim drvetom izvođenja i drvetom apstraktne sintakse:

Drvo izvođenja	Drvo apstraktne sintakse
	

3.3 Višeznačne gramatike

Primetimo u primeru 3.2.3 da smo za izvođenja I_2, I_3 i I_4 dobili ista drveta izvođenja. Ono što je važnije primetiti jeste da se dogodilo da smo za istu nisku generisanu gramatikom G_{+LD} dobili dva različita drveta izvođenja (na primer, I_1 i I_2). Ovaj rezultat nam je bitan zbog sledeće leme i prateće definicije.

LEMA 3.1. Ako je za datu gramatiku G tačan jedan od sledećih iskaza:

a. Svaka reč u gramatici G ima jedinstveno najlevlje izvođenje;

b. Svaka reč u gramatici G ima jedinstveno najdešnje izvođenje;

c. Svaka reč u gramatici G ima jedinstveno drvo izvođenja;

onda su tačna i druga dva iskaza.

Na osnovu ove leme uvodi se sledeća definicija.

Gramatiku u kojoj za barem jednu nisku postoje barem dva različita drveta izvođenja nazivamo **višeznačnom gramatikom**. Inače, gramatika je **jednoznačna**.

Videli smo u primeru 3.2.3 da se u izvođenju I_1 nameće desna asocijativnost, dok se u izvođenjima I_2, I_3 i I_4 sugeriše leva asocijativnost. Dupla rekurzija ostavlja slobodan izbor asocijativnosti, te je zbog toga u praksi treba izbegavati duple rekurzije, a samim tim i višeznačne gramatike.

Napomenimo da je problem određivanja da li je neka gramatika višeznačna nedeterministički problem, tj. ne postoji algoritam koji određuje da li je data gramatika višeznačna ili ne. Naravno, moguće je kontraprimerom pokazati da neka gramatika nije višeznačna, kao što smo prikazali u primeru 3.2.3.

Pogledajmo još jedan primer gramatike aritmetičkih izraza.

Primer 3.3.1

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. Odrediti gramatiku ovakvih (i njima sličnih) aritmetičkih izraza.

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

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

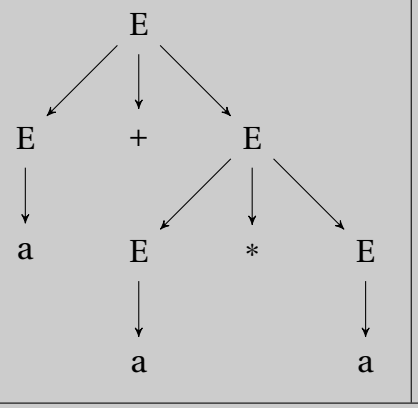
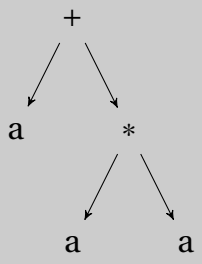
3. $E \longrightarrow E * E$.

Ovde imamo slučaj duple rekurzije, i već smo videli da ona daje višeznačnost što se tiče asocijativnosti. Dodatno se pojavio još jedan problem, a to je problem prioriteta operacija. Na primer, za izraz $a + a * a$ postoje dva načina najlevljeg izvođenja:

1. Korisćenjem prvo drugog pravila:

$$I_1 : E \overset{2}{\Longrightarrow} E + E \overset{1}{\Longrightarrow} a + E \overset{3}{\Longrightarrow} a + E * E \overset{1}{\Longrightarrow} a + a * E \overset{1}{\Longrightarrow} a + a * a,$$

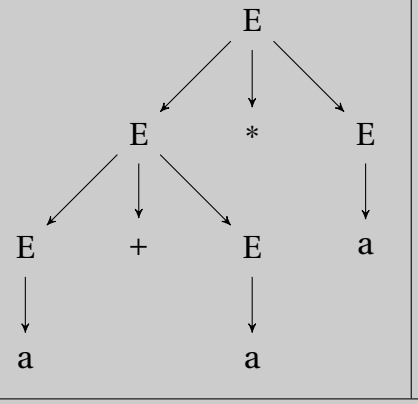
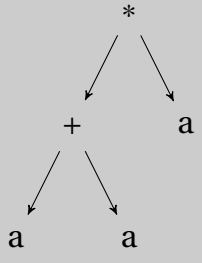
sa svojim drvetom izvođenja i drvetom apstraktne sintakse:

Drvo izvođenja, $\mathcal{D}(I_1)$	Drvo apstraktne sintakse
	

2. Korisćenjem prvo trećeg pravila:

$$E \overset{3}{\Longrightarrow} E * E \overset{2}{\Longrightarrow} E + E * E \overset{1}{\Longrightarrow} a + E * E \overset{1}{\Longrightarrow} a + a * E \overset{1}{\Longrightarrow} a + a * a,$$

sa svojim drvetom izvođenja i drvetom apstraktne sintakse:

Drvo izvođenja, $\mathcal{D}(I_2)$	Drvo apstraktne sintakse
	

Poštovanjem matematičkog dogovora da operacija množenja ima viši prioritet nad

operacijom sabiranja, dolazimo do zaključka da je drvo izvođenja označeno $\mathcal{D}(I_1)$ u prethodnom primeru jedino ispravno. Sada smo uvereni da je višeznačnost gramatike nepogodna za sintaksičku analizu. Navešćemo još i da nametanje asocijativnosti možemo postići odgovarajućom rekurzijom, a nametanje prioriteta razdvajanjem pravila na slojeve. Ilustrujmo ovo neformalno pravilo sledećim primerom.

Primer 3.3.2

Aritmetičke izraze iz primera 3.3.1 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. $E \longrightarrow E + T$,
2. $E \longrightarrow T$,
3. $T \longrightarrow T * a$, i
4. $T \longrightarrow a$.

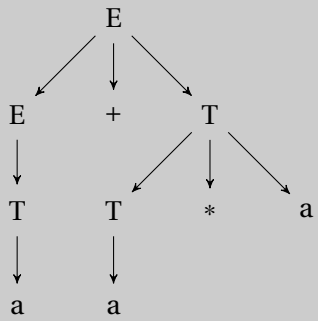
Primetimo da gramatika G_{+*L} nije višeznačna. Njena pravila razdvajaju sabiranje i množenje na dva različita nivoa jer su različitih prioriteta.

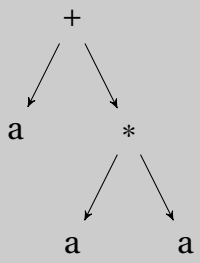
Primer izvođenja izraza $a + a * a$:

$$E \overset{1}{\Longrightarrow} E + T \overset{2}{\Longrightarrow} T + T \overset{4}{\Longrightarrow} a + T \overset{3}{\Longrightarrow} a + T * a \overset{4}{\Longrightarrow} a + a * a,$$

sa svojim drvetom izvođenja i drvetom apstraktne sintakse:

Drvo izvođenja	Drvo apstraktne sintakse





Dopunimo gramatiku iz prethodnog primera tako da obuhvata aritmetičke izraze koji uključuju cele brojeve, zatim sve četiri osnovne operacije nad njima i zagrade sledećim dvama primerima.

Primer 3.3.3

Definišemo pravila gramatike koju ćemo označiti G_{op}:

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

Primetimo da, kako su operacije sabiranja i oduzimanja istog prioriteta, njihovi nivoi su jednaki. Isto važi za operacije množenja i deljenja.

Primer 3.3.4

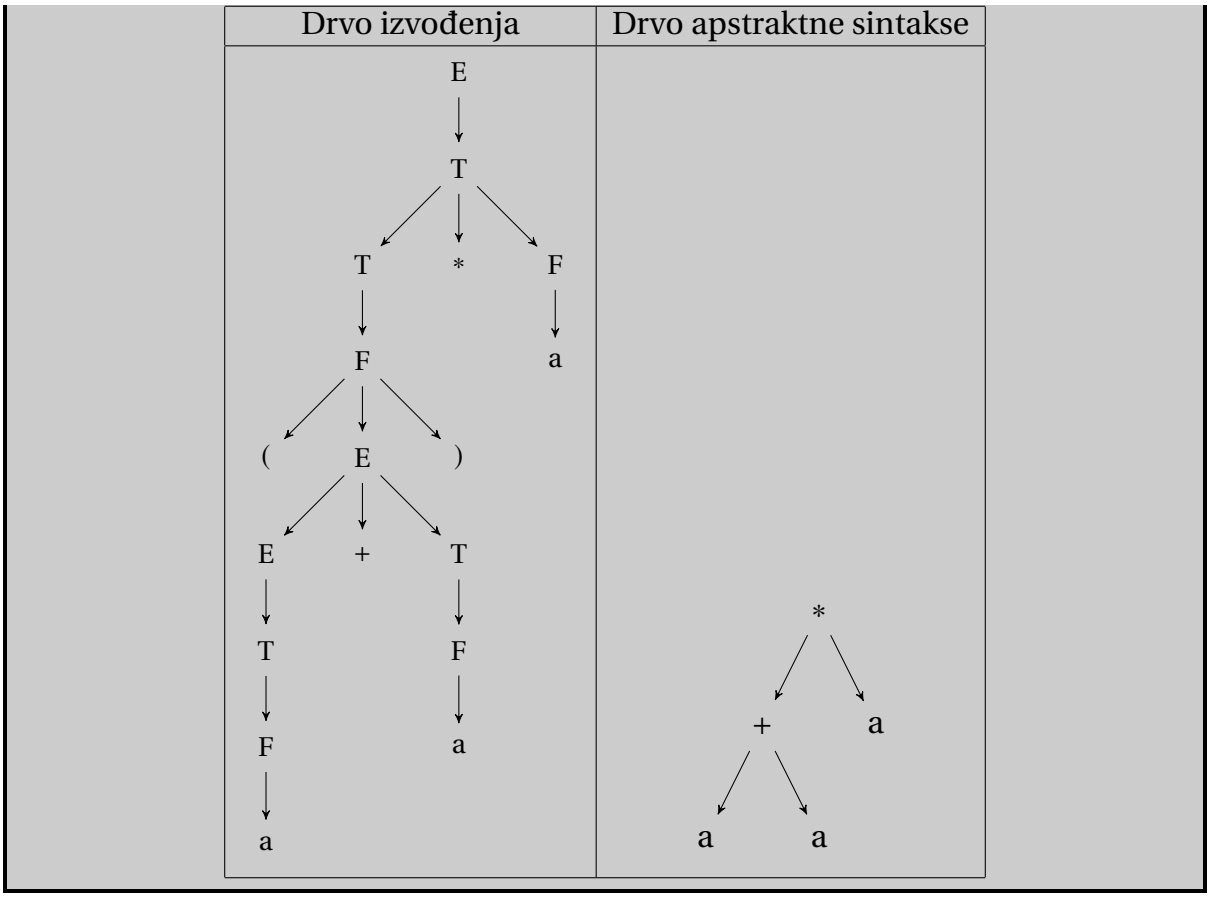
Definišemo pravila gramatika koju ćemo označiti G_{opz}:

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

Primer izvođenja izraza $(a+a)*a$:

$$E \Longrightarrow T \Longrightarrow T*F \Longrightarrow F*F \Longrightarrow (E)*F \Longrightarrow (E+T)*F \Longrightarrow (T+T)*F \Longrightarrow (F+T)*F \\ \Longrightarrow (a+T)*F \Longrightarrow (a+F)*F \Longrightarrow (a+a)*F \Longrightarrow (a+a)*a,$$

sa svojim drvetom izvođenja i drvetom apstraktne sintakse:



3.4 Transformacije gramatika

3.4.1 Ćišćenje suvišnih simbola

- Nedostnižni simboli
  Nedostižni simboli su oni simboli koji nisu dostižni. Kako otkriti dostižne?
  - Nivo 1: Aksioma je dostižan simbol, jer se iz aksiome kreće izvođenje.
  - Nivo 2: Svi simboli koji se nalaze sa desne strane pravila aksiome su dostižni.
  - ...
  - Nivo n: Svi simboli koji se nalaze sa strane pravila iz prethodnog nivoa su dostižni.

  Ovo se postiže backtracking-om. Konstruišemo graf izvođenja pravila i pretragom u širinu ili dubinu obići ćemo sve dostižne simbole. Simboli koje nismo obišli su nedostižni i njih eliminišemo iz gramatike.

- Neproduktivni simboli
  Neproduktivni simboli su oni simboli koji imaju rekurzivno pravilo, a nemaju izlaz iz rekurzije. Neproduktivni simboli su oni koji nisu produktivni. Kako otkriti produktivne?

- Nivo 1: Simboli koji ne izvode neterminale su produktivni simboli.
- ...
- Nivo n: Simboli koji izvode terminale i produktivne simbole iz prethodnog nivoa.

Kada otkrijemo neproduktivne, uklanjamo sva pravila vezana za njih.

Kako bismo izbegli stvaranje neproduktivnih simbola prilikom pisanja gramatike, potrebno je da vodimo računa o sledećim pravilima:

1. Ne treba dodavati novi terminal i pravila koja izvodi dok se ne pojavi sa desne strane nekog pravila.
2. Ako imamo rekurziju u pravilo, obavezno napisati izlaz iz rekurzije.

Primer 3.4.1

Neka je zadata gramatika:

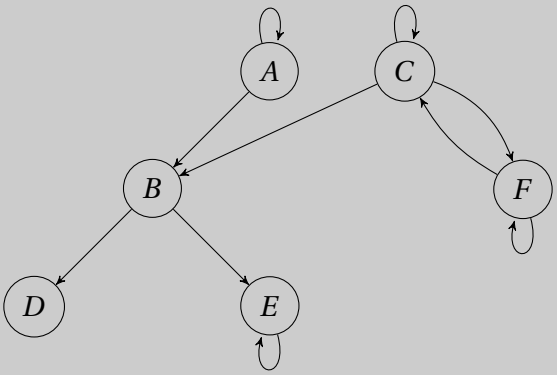
$$
\begin{array}{lcl}
A & \longrightarrow & aB \mid bA \\
B & \longrightarrow & cD \mid E \\
C & \longrightarrow & CA \mid Ba \\
D & \longrightarrow & a \mid b \\
E & \longrightarrow & eE \mid Ee \\
F & \longrightarrow & Ca \mid Fb
\end{array}
$$

Dostižni simboli:

1. nivo: A
2. nivo: A, B
3. nivo: A, B, D, E
4. nivo: A, B, D, E

Pošto su 3. i 4. nivo identični, to su svi dostižni simboli. To znači da su nedostižni simboli C i F.

Drugi način da odredimo nedostižne simbole je preko grafa:



Obilaskom grafa iz čvora A, zaključujemo da su nedostižna stanja C i F. Ne postoji nijedna grana koja vodi do tih čvorova.

Gramatika nakon eliminisanja nedostižnih stanja:
$A \longrightarrow aB \mid bA$
$B \longrightarrow cD \mid E$
$D \longrightarrow a \mid b$
$E \longrightarrow eE \mid Ee$

Sada tražimo neproduktivne simbole, tako što ćemo odrediti skup produktivnih:

1. nivo: D
2. nivo: D, B
3. nivo: D, B, A
4. nivo: D, B, A

Pošto su 3. i 4. nivo identični, to su svi produktivni simboli. Zaključujemo da je E jedini neproduktivan simbol. Zato eliminišemo pravilo E i svako pravilo koje sadrži E. Nakon eliminisanja simbola E, gramatika izgleda ovako:
$A \longrightarrow aB \mid bA$
$B \longrightarrow cD$
$D \longrightarrow a \mid b$

3.4.2 Oslobađanje od jednostrukih pravila

Pravilo je jednostruko ako ima samo jedno izvođenje (samo jednu desnu stranu). Takvog pravila se možemo osloboditi tako što ćemo umesto simbola uvrstiti njihovu desnu stranu u svakom pravilu u kom se pojavljuje.
Uklanjanje jednostrukih pravila može da naruši čitljivost, ali smanjuje zapis.

Primer 3.4.2

Uzmimo gramatiku iz primera 3.4.1 bez suvišnih stanja.
Jedino jednostruko pravilo je B. Njega se oslobađamo tako što svuda umesto B uvrstimo cD. Primenom ove transformacije dobijamo sledeću gramatiku:

$$\begin{array}{lcl} A & \longrightarrow & acD \mid bA \\ D & \longrightarrow & a \mid b \end{array}$$

3.4.3 Oslobađanje od leve rekurzije

Pravilo je levo-rekurzivno ako je neterminal sa leve strane prvi simbol sa desne strane pravila. Nekad će nam leva rekurzija biti pogodna, ali nekad će nam smetati. Zato je veoma važno znati kako da je se oslobodimo. Primenom ove trasformacije oslobađemo se leve rekurzije, ali ne i rekurzije. U suštini, levo-rekurzivna pravila ćemo trasformisati u desno-rekurzivna. Ilustrujmo primenu transformacije na sledećem primeru:

Primer 3.4.3

Neka nam je zadata sledeća gramatika:

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

Jezik opisan ovom gramatikom je $\beta\alpha*$

Leve rekurzije se oslobađamo uvođenjem novog neterminala:

$$\begin{array}{lcl} A & \longrightarrow & \beta A' \\ A' & \longrightarrow & \alpha A' \mid \varepsilon \end{array}$$

Sad imamo desno-rekurzivnu gramatiku koja izvodi iste reči kao i početna. Primetimo da smo uveli ε pravilo. To nam uglavnom ne smeta, ali možemo i bez toga:

$$\begin{array}{lcl} A & \longrightarrow & \beta A' \mid \beta \\ A' & \longrightarrow & \alpha A' \mid \alpha \end{array}$$

Napomena: Ovakva gramatika je levo faktorisana jer pravila imaju iste prefikse.

Hajde da malo uopštimo postupak. Imamo više levo-rekurzivnih pravila i više pravila koja nisu levo-rekurzivna:

$$A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

Jezik opisan ovom gramatikom je $(\beta_1|\beta_2|\ldots|\beta_m)(\alpha_1|\alpha_2|\ldots|\alpha_n)*$
Transformacijom dobijamo sledeću gramatiku, bez leve rekurzije:

$$\begin{array}{lcl} A & \longrightarrow & \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A' \\ A' & \longrightarrow & \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon \end{array}$$

Ako hoćemo bez ε pravila, onda dobijamo sledeće:

$$A \longrightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A' \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$
$$A' \longrightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

Da bi smo dali najopštiji slučaj, moramo uvesti pojam posredne i neposredne leve rekurzije. Rekurzije koje smo videli u ovom primeru su neposredne.
Primer posredne leve rekurzije:

$$A \longrightarrow Ba \mid b$$
$$B \longrightarrow Ab \mid cA$$

Eliminacija posredne leve rekurzije se tesno zasniva na eliminacije neposredne.

Ad hoc metoda: Uvedemo redosled kojim čemo obrađivati simbole. Npr. krenemo od A:

$$A \longrightarrow Ba \mid b$$

Drugo pravilo ćemo prepisati tako što ćemo prethodna pravila uvrstiti u njega, ali ne svuda već samo tamo gde može da se stvori leva rekurzija:

$$B \longrightarrow Bab \mid bb \mid cA$$

Sada imamo samo neposrednu rekurziju, a nje znamo da se rešimo. Dobijamo sledeće:

$$A \longrightarrow Ba \mid b$$
$$B \longrightarrow bbB' \mid cAB'$$
$$B' \longrightarrow abB' \mid \varepsilon$$

3.4.4 Oslobađanje od ε pravila

DEFINICIJA ε-SLOBODNA GRAMATIKA. Gramatika je ε-slobodna ako nema ε pravilo ili ako se ono javlja samo u aksiomi, pri čemu aksioma ne sme da se javlja sa desne strane pravila (time je obezbeđeno da se ε može izvesti samo na početku, kao prvi i jedini korak, nadalje neće biti moguće izvesti ε pravilo).

Da bismo se oslobodili ε pravila, prvo moramo da pronađemo anulirajuće simbole. Anulirajući simboli su oni simboli koji mogu da izvedu ε u 0 ili više koraka ($A \Longrightarrow^* \varepsilon$). Ako je aksioma anulirajuća, možemo izvesti ε, inače nema potrebe za ε pravilima. Nakon što pronađemo anulirajuće simbole, treba da na odgovarajući način transformišemo gramatiku. Prikažimo postupak na primeru:

Primer 3.4.4

Osloboditi se ε pravila u sledećoj gramatici:

$$
\begin{array}{lcl}
S & \longrightarrow & Aa \mid Bba \\
A & \longrightarrow & aBB \mid CC \mid aD \\
B & \longrightarrow & aB \mid b \\
C & \longrightarrow & CD \mid DE \mid a \\
D & \longrightarrow & aB \mid bBa \mid \varepsilon \\
E & \longrightarrow & aD \mid DD
\end{array}
$$

Prvo treba pronaći sve simbole koji izvode ε, a onda i sve simbole koji izvode pronađene anulirajuće simbole.

1. $A_0 = \{D\}$
2. $A_1 = \{D, E\}$
3. $A_2 = \{D, E, C\}$
4. $A_3 = \{D, E, C, A\}$
5. $A_4 = \{D, E, C, A\}$

Pošto su A_3 i A_4 isti skupovi, tu stajemo.

Gramatiku transformišemo na sledeći način - svaku desnu stranu pravila pišem više puta uzimajući u obzir da li anulirajući simbol može da se javi ili ne:

- pravilo S: A je anulirajući simbol pa možemo izvesti a (ako A izvede ε) ili Aa (ako A ne izvede ε)
- pravilo A:
  - C je anulirajući simbol pa možemo izvesti C ili CC
  - D je anulirajući simbol pa možemo izvesti a ili aD
- pravilo B: nema anulirajućih simbola pa ga samo prepisujemo
- pravilo C:
  - C i D su anulirajući pa možemo izvesti D (ako C izvede ε), C (ako D izvede ε), CD (ako ni C ni D ne izvedu ε) i ε (ako oba izvedu ε) s tim što ovo pravilo NE PIŠEMO jer nam je cilj da se oslobodimo ε pravila
  - D i E su anulirajući pa možemo izvesti D, E ili DE, s tim što se pravilo D već javilo pa da ne pišemo ponovo
- pravilo E: D je anulirajući pa možemo da izvedemo a, aD, D ili DD

Time smo dobili ε-slobodnu gramatiku:

$$
\begin{array}{lllllllll}
S & \longrightarrow & a & | & Aa & | & Bba & & \\
A & \longrightarrow & aBB & | & C & | & CC & | & a & | & aD \\
B & \longrightarrow & aB & | & b & & & & \\
C & \longrightarrow & D & | & CD & | & E & | & DE & | & a \\
D & \longrightarrow & aB & | & bBa & & & & \\
E & \longrightarrow & a & | & aD & | & D & | & DD &
\end{array}
$$

Primer 3.4.5

Datu gramatiku osloboditi od ε pravila:

$$
\begin{array}{lll}
X & \longrightarrow & aY \\
Y & \longrightarrow & bY \mid \varepsilon
\end{array}
$$

Odredimo anulirajuće simbole:

1. $A_0 = \{Y\}$
2. $A_1 = \{Y\}$

Dakle, Y je jedini anulirajući simbol. Primenom transformacije dobijamo sledeću ε-slobodnu gramatiku:

$$
\begin{array}{lllll}
X & \longrightarrow & a & | & aY \\
Y & \longrightarrow & b & | & bY
\end{array}
$$

Primer 3.4.6

Datu gramatiku osloboditi od ε pravila:

$$
\begin{array}{lllllll}
S & \longrightarrow & a & | & AA & | & bS \\
A & \longrightarrow & bA & | & \varepsilon & | & Sa
\end{array}
$$

Odredimo anulirajuće simbole:

1. $A_0 = \{A\}$
2. $A_1 = \{A, S\}$
3. $A_2 = \{A, S\}$

Anulirajući su A i S.

Ako idemo slepo po algoritmu, u ovom slučaju, nećemo dobiti tačno rešenje:
$S \longrightarrow a \mid A \mid AA \mid b \mid bS$
$A \longrightarrow b \mid bA \mid a \mid Sa$

Ne možemo nikako da izvedemo praznu reč, zato dodajemo ε pravilo aksiomi:
$S \longrightarrow a \mid A \mid AA \mid b \mid bS \mid \varepsilon$
$A \longrightarrow b \mid bA \mid a \mid Sa$

Sad imamo još jedan problem. Aksioma izvodi ε i nalazi se sa desne strane pravila, a to po definiciji ε-slobodne gramatike ne sme da se dogodi. Zato uvodimo novo pravilo S' koje postaje nova aksioma:
$S' \longrightarrow S \mid \varepsilon$
$S \longrightarrow a \mid A \mid AA \mid b \mid bS$
$A \longrightarrow b \mid bA \mid a \mid Sa$

3.5 Potisni automati (Push-down automata)

Primer 3.5.1

Pred nama je problem uparivanja zagrada:
npr. (()())((())())
Treba napisati program koji proverava da li su zagrade dobro uparene.

U C-u se ovakav problem može rešiti pomoću brojačke promenljive. Svaki put kad se naiđe na otvorenu zagradu uvećava se brojač, kad se naiđe na zatvorenu brojač se umanjuje. Pritom ne dozvoljavamo da brojač postane negativan i na kraju zahtevamo da brojač bude jednak nuli. Ali ako bismo malo zakomplikovali primer:
([])[()]
brojačka promenljiva nam neće mnogo pomoći jer sada treba da brinemo o tome i koja zagrada je kada otvorena, a ne samo koliko ih ima. Treba da bude zatvorena ona zagrada koja je poslednja otvorena, dakle treba nam stek na koji ćemo stavljati otvorene zagrade, a kad se naiđe na zatvorenu zagradu skidamo poslednju otvorenu i proveravamo da li su iste vrste.

Za ovakve probleme zgodni su potisni automati.

Neformalno, potisni automati su Konačni automati sa stekom (potisnom listom).

Svakom Potisnom Automatu (PA) može da se pridruži jedna KSG i obrnuto.

Primer 3.5.2

Zadata je jezik $a^n b^n$ i potreban nam je automat koji će prepoznavati reči tog jezika.

Ideja 1: Slično kao sa zagradama, možemo da stavljamo karakter a na stek pa kad naiđemo na b skinemo a sa steka. Međutim, ovakav automat bi prihvatio reči abab, a to nije ono što tražimo.

Ideja 2: Slično kao ideja 1, samo da zabranimo čitanje nakon pražnjenja steka. Ni to nam ne rešava problem jer će automat prihvatiti reči kao što je aababb.

Ideja 3: Uvodimo stanja:

- u prvom stanju ćemo čitati slovo a i stavljati na stek, a ako naiđemo na b prijavljujemo grešku
- kad završimo sa čitanjem a prelazimo u drugo stanje u kom ćemo čitati b i skidati a sa steka, i ako pročitamo a prijavićemo grešku
- takođe, ako se stek isprazni pre završetka čitanja reči ne smemo prihvatiti reč jer imamo manjak karaktera a u reči
- isto tako, ako nam po čitanju reči ostane još neko slovo a na steku znači da ih imamo viška i to ne smemo prihvatiti
- dakle, reč ćemo prihvatiti samo ako je stek prazan nakon završetka čitanja reči

3.5.1 Definicija (N)PA

(N)PA je uređena sedmorka

$$(\Sigma, Q, \Gamma, I, z_0, K, \Delta)$$

gde je:

- Σ ulazna azbuka
- Q skup stanja automata
- Γ skup simbola steka (često se poklapa sa Σ)
- $I \subseteq Q$ skup početnih stanja
  Konfiguracija potisnog automata predstavlja skup Q x Σ x Γ^* odnosno, sastoji se od stanja u kom se automat nalazi, simbola koji je na ulazu i onoga što je na steku.

  Prateći tu definiciju, početnu konfiguraciju možemo definisati kao

  $$(i, w, z_0)$$

pri čemu je $i \in I$

- $z_0 \in \Gamma$ početni simbol na steku.
  Koristi se da označi dno steka. Ako je stek prazan, automat ne može da radi, to može da bude poslednji korak, ali ako se isprazni u sred čitanja reči onda se to smatra greškom.

- $K \subseteq Q \times \Gamma^*$ skup završnih konfiguracija
  Završnu konfiguraciju definišemo na sledeći način:

  $$(q, \varepsilon, x)$$

  gde je $q \in Q$, $x \in \Gamma^*$ i $(q, x) \in K$

  K ima sledeću strukturu:

  1. $K = F \times \varepsilon$, $F \subseteq Q$ je podskup završnih stanja
     Ovo nam govori da automat prihvata reč ako je ispraznio stek i ulaz, i nalazi se u završnom stanju (kaže se i automat prihvata završnim stanjem i praznim stekom).
  2. $K = F \times \Gamma^*$
     Dovoljno je da bude u završnom stanju kad pročita celu reč, šta je na steku nam nije bitno (kaže se i da prihvata završnim stanjem)
  3. $K = Q \times \varepsilon$
     Nebitno je stanje u kom se nalazimo, bitno je da je stek prazan nakon čitanja reči (kaže se i da prihvata praznim stekom).

  Bilo koji od ova tri uslova je dovoljan. Tu činjenicu nećemo dokazivati.

- $\Delta \subseteq Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \times Q \times \Gamma^*$ relacija prelaska
  U zavisnosti od stanja u kom se automat nalazi, simbola koji je pročitao i simbola koji se nalazi na vrhu steka, automat će preći u neko stanje i upisati 0 ili više simbola na stek. Pri tome može da upiše i onaj simbol koji je skinuo sa steka ako nam je potrebno da se stek ne menja.

Ako se automat nalazi u stanju q, na ulazu je aw, a na steku zx onda prelazi u stanje q', pročitao je slovo a i na ulazu ostaje w, skinuo je z sa steka i upisao y. To je moguće ako i samo ako postoji relacija u skupu Δ koja automat iz stanja q vodi u stanje q' ako se na ulazu pojavi karakter a i na steku z,a zatim na stek upisuje y.

$$(q, aw, zx) \vdash (q', w, yx) \iff (q, a, z, q', y) \in \Delta$$

$q, q' \in Q$, $a \in \Sigma$, $w \in \Sigma^*$, $z \in \Gamma$, $x, y \in \Gamma^*$

$\vdash^*$ je tranzitivno-refleksivna relacija.[1]

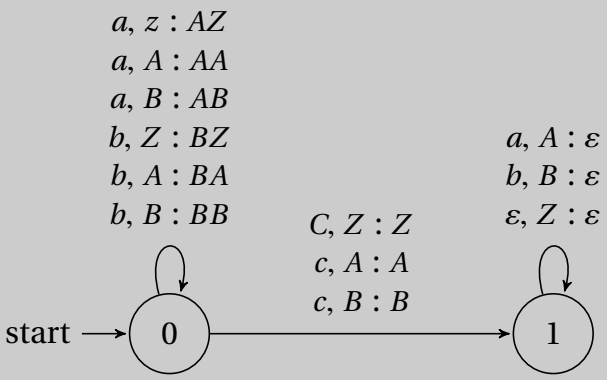
[1] Podsetiti se šta je tranzitivno-refleksivna relacija kod Konačnih automata.

Jezik automata Jezik automata je skup svih reči iz Σ^* takvih da iz početne konfiguracije, u 0 ili više koraka stignem u završnu konfiguraciju.

$$L(\mathcal{A}) = \{w \in \Sigma^* | \exists i \in I, q \in Q, x \in \Gamma^* : (i, w, z_0) \vdash^* (q, \varepsilon, x) \wedge (q, x) \in K\}$$

Primer 3.5.3

Odrediti jezik sledećeg automata:
$\Sigma = \{a, b, c\}$
$Q = \{0, 1\}$
$I = \{0\}$
$\Gamma = \{Z, A, B\}$
$Z = z_0$



Primetimo da smo na granima zapisivali pravila u obliku a,Z:AZ. To je isto kao da smo zapisali na primer $(0, a, Z, 0, AZ)$. Dvotačka nam razdvaja ulaznu komponentu od rezultujuće.
U stanju 0

- ako pročitam karakter a, upisujem A na stek i ostajem u 0
- ako pročitam karakter b, upisujem B na stek i ostajem u 0
- ako procitam karakter c, upisujem praznu reč na stek i prelazim u stanje 1

U stanju 1

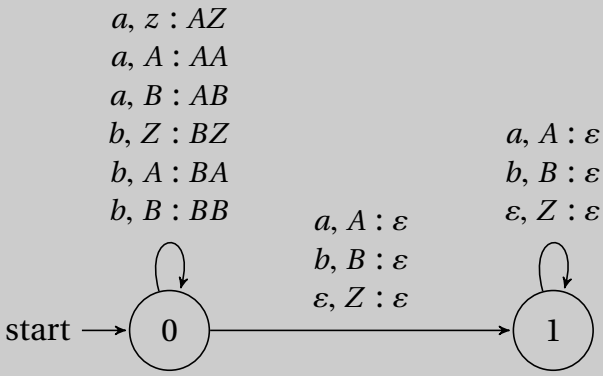
- ako pročitam karakter a i na steku je A, skidam A sa steka, upisujem ε i ostajem u 1
- ako pročitam karakter b i na steku je B, skidam B sa steka, upisujem ε i ostajem u 1

- ako procitam karakter ε i na steku je Z, upisujem praznu reč na stek i prelazim u stanje 1
- za kombinacije za koje ne postoji prelaz automat će zablokirati i neće prihvatiti reč

Zaključujemo, da je ovaj automat prepoznaje reči oblika: $wc\overline{w}$, gde je $\overline{w}$ obrnuta reč w. Pri tome, $w \in \{a, b\}^*$

Primer 3.5.4

Odrediti jezik sledećeg automata:
$\Sigma = \{a, b\}$
$Q = \{0, 1\}$
$I = \{0\}$
$\Gamma = \{Z, A, B\}$
$Z = z_0$



Jezik ovog automata sličan je prethodnom: $w\overline{w}$, $w \in \{a, b\}^*$.

Razlika ovog automata i automata iz prethodnog primera je što je prvi deterministički, dok je automat iz ovog primera nedeterministički.
Primetimo da kod Potisnih automata gledamo kombinaciju ulaza i karaktera na vrhu steka kad gledamo prelaze.

TEOREMA .Za svaku KSG moguće je konstruisati NPA.

Slično kao kod (N)DKA, klasa kontekstno-slobodnih jezika i klasa potisno prepoznatljivih jezika su jednake.
Međutim, nećemo moći svaki Nedeterministički PA da transformišemo u Deterministički PA.

3.5.2 Analiza naniže (Predictive parcing)

Analiza naniže kreće od korena drveta ka listovima, pri čemu bira put gledajući prvi karakter na ulazu.
Automat ima samo jedno stanje (prost potisni automat) i zato ga nećemo crtati.

Postupak:

- prvo što se piše na stek je aksioma
- ako je karakter na vrhu steka neterminal, skidamo ga sa steka i upisujemo desnu stranu pravila za taj neterminal
- ako je karakter na vrhu steka terminal, skidamo ga sa steka i upoređujemo sa prvim simbolom na ulazu
  - ako su isti, vršimo sinhornizaciju steka i ulaza tako što uklanjamo taj karakter sa ulaza, a sa steka je već uklonjen
  - ako su različiti prijavljujemo grešku

Primer 3.5.5

Primeniti analizu naniže na reč abbccd za sledeću gramatiku:

$S \longrightarrow AB$
$A \longrightarrow aAb \mid \varepsilon$
$B \longrightarrow cB \mid d$

stanje steka	ulaz
$\underline{S}$	aabbccd
$\underline{A}B$	aabbccd
$\underline{a}AbB$	aabbccd
$\underline{A}bB$	abbccd
$\underline{a}AbbB$	abbccd
$\underline{A}bbB$	bbccd
$\underline{b}bB$	bbccd
$\underline{b}B$	bccd
$\underline{B}$	ccd
$\underline{c}B$	ccd
$\underline{B}$	cd
$\underline{c}B$	cd
$\underline{B}$	d
$\underline{d}$	d
ε	$\dashv$

Napomena: Podvučeni karakter predstavlja vrh steka.

Pošto su se obe kolone ispraznile, automat prihvata reč.
Napišimo izvođenje koje odgovara ovom postupku:
$S \Longrightarrow AB \Longrightarrow aAbB \Longrightarrow aaAbbB \Longrightarrow aabbB \Longrightarrow aabbcB \Longrightarrow aabbccB \Longrightarrow aabbccd$
Kao što možemo da vidimo prihvatanje je uvek najlevlje (left-most) tj. uvek se menja onaj neterminal koji je najlevlji. To je zbog toga što u tabeli smatramo vrhom steka najlevlji karakter, samim tim prvi karakter koji će biti zamenjen je najlevlji jer se uvek menja vrh steka.

Razmatramo posebnu vrstu gramatika, tzv. $LL(k)$ gramatike.

Gramatika je $LL(k)$ ako na osnovu viđenih prvih k karaktera sa ulaza možemo da konstruišemo DPA.
Najkorišćenije su $LL(1)$ gramatike, koje dozvoljavaju da se gleda samo prvi karakter sa ulaza.

Primer 3.5.6

Odrediti skup izbora svakog pravila sledeće gramatike:[2]

pravilo			skup izbora
S	$\longrightarrow$	AB	$\{a, c, d\}$
A	$\longrightarrow$	aAb	$\{a\}$
	$\mid$	ε	$\{b, c, d\}$
B	$\longrightarrow$	cB	$\{c\}$
	$\mid$	d	$\{d\}$

Kako određujemo skupove? Gledamo koji je prvi karakter koji se može pročitati pravilom.

- B - oba pravila počinju terminalima i skupovi su jasni.
- A - prvo pravilo počinje terminalom i to je jedini izbor. Međutim, drugo pravilo je ε i zato gledamo šta se može naći posle neterminala A. Po prvom pravilu A, to može biti karakter b. Međutim, A se pojavljuje na još jednom mestu sa desne strane, a to je u aksiomi. Nakon A nalazi se B, pa u skup izbora za drugo pravilo A ulaze i svi karakteri iz skupova izbora pravila B.
- S - nema terminale pa uzimamo skupove terminala A. Pošto A može da izvede ε, u skup izbora za S moraju da uđu i karakteri iz skupa izbora za B.

Da bi gramatika bila LL(1) skupovi izbora pravila terminala moraju biti disjunktni, što je ispunjeno u ovom slučaju.

Da bi gramatika bila $LL(1)$ ne sme da bude levo-faktorisana niti da ima levu rekurziju. Ilustrojmo to na primeru.

Primer 3.5.7

$$\begin{array}{rcl} A & \longrightarrow & bB \\ & | & b \end{array}$$

Ova gramatika je levo-faktorisana. Ako na ulazu pročitamo samo jedan karakter b, nije jasno koje pravilo primenjujemo.

$$\begin{array}{rcl} A & \longrightarrow & Aa \\ & | & b \end{array}$$

Ova gramatika je levo-rekurzivna. U daljem tekstu ćemo objasniti zašto nam ova činjenica smeta.

Primer 3.5.8

Osloboditi gramatiku od leve rekurzije:

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

Levu rekurziju uklanjamo dodavanjem novog neterminalnog simbola za svako levo-rekurzivno pravilo.

$$\begin{array}{rcl} E & \longrightarrow & TE' \\ E' & \longrightarrow & +TE' \\ & | & \varepsilon \\ T & \longrightarrow & FT' \\ T' & \longrightarrow & *FT' \\ & | & \varepsilon \\ F & \longrightarrow & a \\ & | & (E) \end{array}$$

Pomoćni skupovi

Da bi smo odredili skupove izbora za svako pravilo, potrebna su nam dva pomoćna skupa. To su:

1. skup Prvi
2. skup Sledi

[2]Kada na testu traži da konstruišemo gramatiku dovoljno je da napišemo pravila i skupove izbora.

Skup Prvi:

Neka je α niz terminala i neterminala. Ako iz njega mogu da izvučem $a\alpha'$ u 0 ili više koraka onda ono što se izvodi sa α može da počne simbolom a i kažemo da a pripada skupu $Prvi(\alpha)$.

$$\alpha \Longrightarrow^* a\alpha' \Longrightarrow a \in Prvi(\alpha)$$

$a \in \Sigma$, $\alpha, \alpha' \in (N \cup \Sigma)^*$

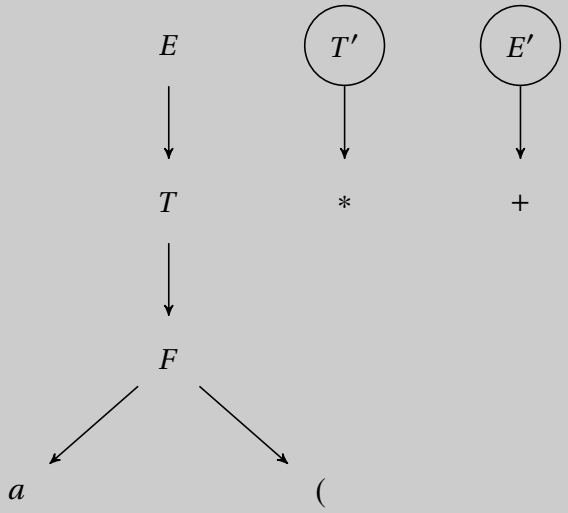
$$Prvi(\alpha) = \{a \in \Sigma \quad | \quad \exists \alpha' : \alpha \Longrightarrow^* a\alpha'\}$$

Primer 3.5.9

Odrediti skup Prvi za gramatiku iz primera 3.5.8.
Možemo ih odrediti intuitivno. Za svako pravilo gledamo kojim terminalnim simbolom može početi.

$Neterminal$	$Prvi$
E	$\{a, (\}$
E'	$\{+\}$
T	$\{a, (\}$
T'	$\{*\}$
F	$\{a, (\}$

Drugi način bio bi crtanje grafa.



Primer 3.5.10

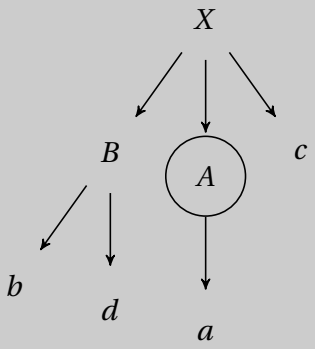
Odrediti skup Prvi za sledeću gramatiku:

$$\begin{aligned} X &\longrightarrow AB \\ &\mid\ c \\ A &\longrightarrow aA \\ &\mid\ \varepsilon \\ B &\longrightarrow bB \\ &\mid\ d \end{aligned}$$

Intuitivno:

$Neterminal$	$Prvi$
X	$\{a, b, c, d\}$
A	$\{a\}$
B	$\{b, d\}$

Graf:



Uopštimo pravila za konstrukciju grafa:

Ako postoji pravilo $X \longrightarrow \alpha Y \beta$, $X \in N$, $Y \in \Sigma \cup N$, $\alpha \in N^*$, $\beta \in (\Sigma \cup N)^*$ u graf treba dodati granu od X do Z ako je α anulirajući simbol ($\alpha \Longrightarrow^* \varepsilon$).

Objasnimo na ovom mestu zašto je leva rekurzija nepoželjna. Ako imamo pravila oblika $A \Longrightarrow A\alpha \mid \beta$, gde su α i β bilo kakvi nizovi terminala i neterminala, desiće se da skup $Prvi(A)$ sadrži sve simbole koji se nalaze u skupu $Prvi(\beta)$. Međutim, tada će se ti simboli naći i u skupovima izbora za oba ta pravila i skupovi izbora neće biti disjunktni.
Slicno, ako imamo pravila oblika $A \Longrightarrow \alpha\beta \mid \alpha\gamma$, tj. ako je gramatika levo faktorisana, tada će simboli iz skupa $Prvi(\alpha)$ biti deo oba skupa izbora i oni opet neće biti disjunktni.

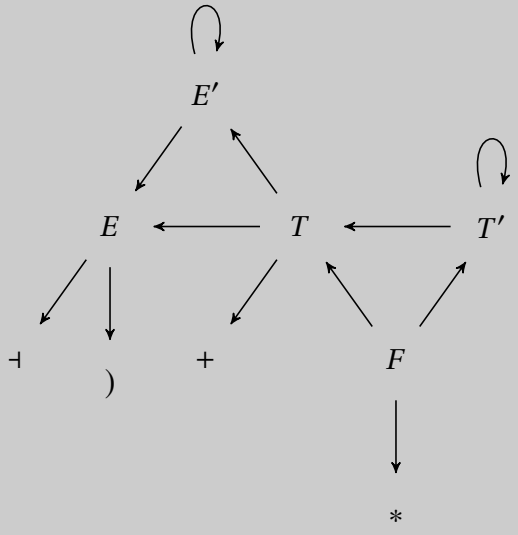
Skup Sledi:

$S \Longrightarrow^* \alpha A a \beta$
Ako krenuvši iz aksiome S iza neterminala A može da se nađe simbol a onda kažemo da a pripada skupu $Sledi(A)$.

$$Sledi(A) = \{a \in \Sigma \quad | \quad \exists \alpha, \beta \in (N \cup \Sigma^*) : S \Longrightarrow^* \alpha A a \beta\}$$

Primer 3.5.11

Odrediti skup Sledi za gramatiku iz primera 3.5.8.



Gledamo neterminale sa desne strane i za njih tražimo simbole koji se mogu naći iza.
Važno je napomenuti da strelicu usmeravamo od desne ka levoj strani, a ne obrnuto!
Sada nam preostaje samo da iz grafa pročitamo simbole koji pripadaju skupu Sledi za svaki neterminal:

$Neterminal$	$Sledi$
E	$\{\dashv,)\}$
E'	$\{\dashv,)\}$
T	$\{+, \dashv,)\}$
T'	$\{+, \dashv,)\}$
F	$\{*, +, \dashv,)\}$

Primer 3.5.12

Odrediti skupove izbora za gramatiku iz primera 3.5.8.
U primerima 3.5.9 i 3.5.11 smo odredili skupove Prvi i Sledi i sad ćemo to da iskoristimo. Za svako pravilo koje izvodi neke neterminale skup izbora je skup Prvi, a za ε-pravila to je skup Sledi:

$Pravilo$			$Skup\ \ izbora$
E	$\longrightarrow$	TE'	$\{a, (\}$
E'	$\longrightarrow$	$+TE'$	$\{+\}$
	$\mid$	ε	$\{\dashv,)\}$
T	$\longrightarrow$	FT'	$\{a, (\}$
T'	$\longrightarrow$	$*FT'$	$\{*\}$
	$\mid$	ε	$\{+, \dashv,)\}$
F	$\longrightarrow$	a	$\{a\}$
	$\mid$	(E)	$\{(\}$

Primetimo da je ovo deterministički automat koji prihvata aritmetičke izraze sa operacijama + i *.

Primer 3.5.13

Proveriti da li izraz (a+a)*a pripada gramatici opisanoj u primeru 3.5.8.

Stek	Ulaz
E	(a+a)*a
TE'	(a+a)*a
FT'E'	(a+a)*a
(E)T'E'	(a+a)*a
E)T'E'	a+a)*a
TE')T'E'	a+a)*a
FT'E')T'E'	a+a)*a
aT'E')T'E'	a+a)*a
T'E')T'E'	+a)*a
E')T'E'	+a)*a
+TE')T'E'	+a)*a
TE')T'E'	a)*a
FT'E')T'E'	a)*a
aT'E')T'E'	a)*a
T'E')T'E'	)*a
E')T'E'	)*a
)T'E'	)*a
T'E'	*a
*FT'E'	*a
FT'E'	a
aT'E'	a
T'E'	⊣
E'	⊣
ε	⊣

Pošto ni na steku ni na ulazu nije ostalo karaktera izraz je prhivaćen.

Uopštimo pravila za konstrukciju grafa:
Ako postoji pravilo $X \longrightarrow \alpha Y \beta$ u graf dodajemo granu od Y:

1. do svakog $a \in Prvi(\beta)$
2. do X, ako je β anulirajući simbol ($\beta \Longrightarrow^* \varepsilon$).

gde je α, $\beta \in (\Sigma \cup N)^*$, $X, Y \in N$.

Primer 3.5.14

LL(1) gramatikom opisati sledeći fragment C koda:
int a;
float *b, a, c[10];
char ** c, b;

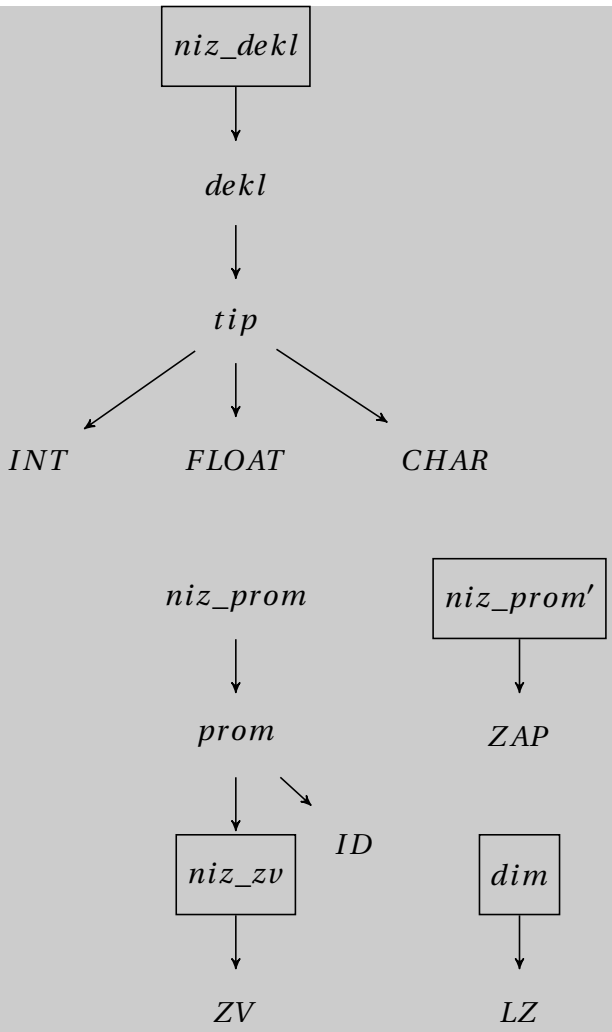
Prvo konstruišemo gramatiku:

$$
\begin{array}{lcl}
niz\_dekl & \longrightarrow & niz\_dekl \quad dekl \\
 & | & \varepsilon \\
 & & \\
dekl & \longrightarrow & tip \quad niz\_prom \quad TZ \\
 & & \\
tip & \longrightarrow & INT \\
 & | & FLOAT \\
 & | & CHAR \\
 & & \\
niz\_prom & \longrightarrow & niz\_prom \quad ZAP \quad prom \\
 & | & prom \\
 & & \\
prom & \longrightarrow & niz\_zv \quad ID \quad dim \\
 & & \\
niz\_zv & \longrightarrow & niz\_zv \quad ZV \\
 & & \\
dim & \longrightarrow & LZ \quad BROJ \quad DZ \\
 & | & \varepsilon
\end{array}
$$

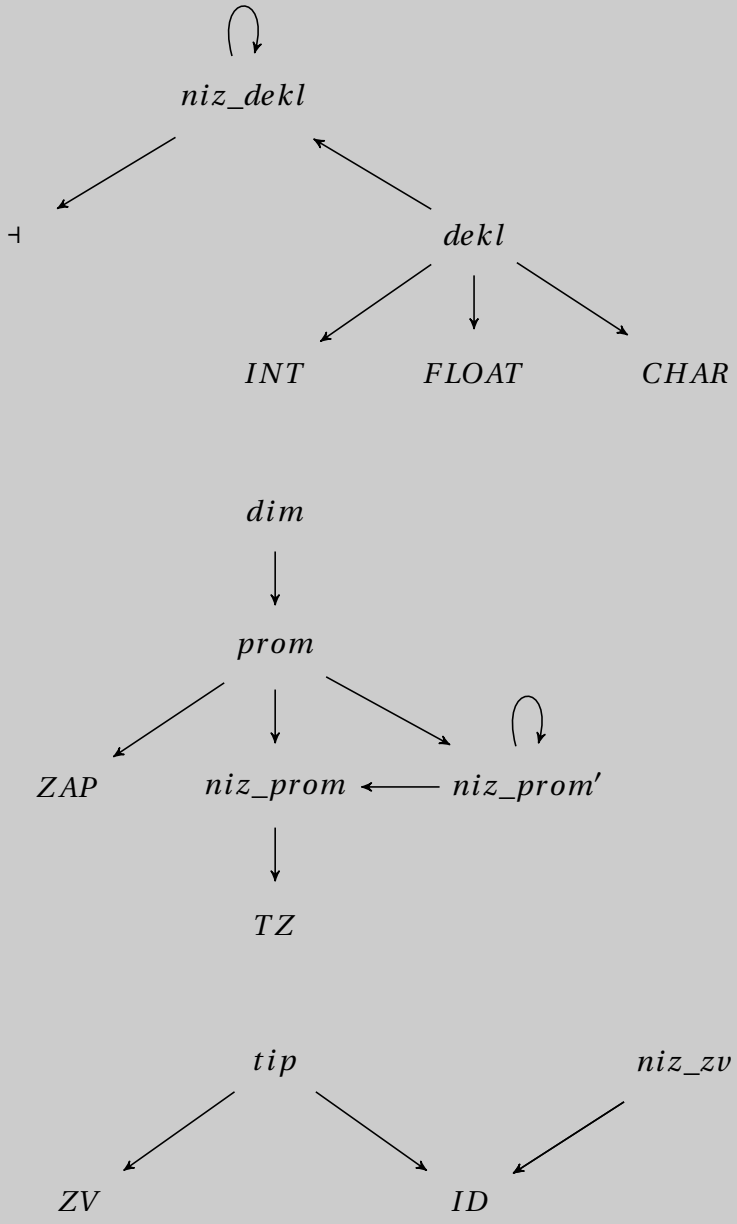
Naša gramatika je levo-rekurzivna, pa je sledeći korak oslobađanje od leve rekurzije:

$$
\begin{array}{lcl}
niz\_dekl & \longrightarrow & dekl \quad niz\_dekl \\
 & | & \varepsilon \\
 & & \\
dekl & \longrightarrow & tip \quad niz\_prom \quad TZ \\
 & & \\
tip & \longrightarrow & INT \\
 & | & FLOAT \\
 & | & CHAR \\
 & & \\
niz\_prom & \longrightarrow & prom \quad niz_p rom' \\
 & & \\
niz\_prom' & \longrightarrow & ZAP \quad prom \quad niz\_prom' \\
 & | & \varepsilon \\
 & & \\
prom & \longrightarrow & niz\_zv \quad ID \quad dim \\
 & & \\
niz\_zv & \longrightarrow & ZV \quad niz\_zv \\
 & | & \varepsilon \\
 & & \\
dim & \longrightarrow & LZ \quad BROJ \quad DZ
\end{array}
$$

Prvi:


niz_dekl
dekl
tip
INT
FLOAT
CHAR
niz_prom
niz_prom'
prom
ZAP
ID
niz_zv
dim
ZV
LZ

Sledeći:



Sada čitamo koji su simboli pripadaju skupovima Prvi i Sledi za svaki neterminal:

	Prvi	Sledi
niz_dekl	INT, FLOAT, CHAR	⊣
dekl	INT, FLOAT, CHAR	⊣, INT, FLOAT, CHAR
tip	INT, FLOAT, CHAR	ZV, ID
niz_prom	ZV, ID	TZ
niz_prom'	ZAP	TZ
prom	ZV, ID	TZ, ZAP
niz_zv	ZV	ID
dim	LZ	TZ, ZAP

Preostaje nam još da odredimo skupove izbora.

$Pravilo$					$Skup\ \ izbora$
niz_dekl	$\longrightarrow$	$dekl$	niz_dekl		$\{INT, FLOAT, CHAR\}$
	$\mid$	ε			$\{\dashv\}$
$dekl$	$\longrightarrow$	tip	niz_prom	TZ	$\{INT, FLOAT, CHAR\}$
tip	$\longrightarrow$	INT			$\{INT\}$
	$\mid$	$FLOAT$			$\{FLOAT\}$
	$\mid$	$CHAR$			$\{CHAR\}$
niz_prom	$\longrightarrow$	$prom$	$niz_p rom'$		$\{ZV, ID\}$
niz_prom'	$\longrightarrow$	ZAP	$prom$	niz_prom'	$\{ZAP\}$
	$\mid$	ε			$\{TZ\}$
$prom$	$\longrightarrow$	niz_zv	ID	dim	$\{ZV, ID\}$
niz_zv	$\longrightarrow$	ZV	niz_zv		$\{ZV\}$
	$\mid$	ε			$\{ID\}$
dim	$\longrightarrow$	LZ	$BROJ$	DZ	$\{LZ\}$

3.5.3 Analiza naviše

Analiza naviše kreće od listova drveta ka korenu.

Postupak:

- na početku je stek prazan
- prebacimo nekoliko karaktera sa ulaza na stek
- ako prepoznamo desnu stranu nekog pravila na steku menjamo je levom (više karaktera menjamo jednim)

- na kraju treba da se nađe aksioma na steku da bi reč bila prihvaćena

Primer 3.5.15

Primeniti analizu naviše na reč abbccd za sledeću gramatiku:

$S \longrightarrow AB$
$A \longrightarrow aAb \mid \varepsilon$
$B \longrightarrow cB \mid d$

stanje steka	ulaz
ε	aabbccd
a	abbccd
aa	bbccd
aaA	bbccd
aaAb	bccd
aA	bccd
aAb	ccd
A	ccd
Ac	cd
Acc	d
Accd	⊣
AccB	⊣
AcB	⊣
AB	⊣
S	⊣

Napomena: Podvučeni karakter predstavlja vrh steka.

Poštoje na steku aksioma, a na ulazu ništa nije ostalo, automat prihvata reč.
Napišimo izvođenje koje odgovara ovom postupku:
$S \Longrightarrow AB \Longrightarrow AcB \Longrightarrow AccB \Longrightarrow Accd \Longrightarrow aAbccd \Longrightarrow aaAbbccd \Longrightarrow aabbccd$
Kao što možemo da vidimo prihvatanje je uvek najdešnje (right-most) tj. uvek se menja onaj neterminal koji je najdešnji. To je zbog toga što u tabeli smatramo vrhom steka najdešnji karakter, samim tim prvi karakter koji će biti zamenjen je najdešnji jer se uvek menja vrh steka.

Kombinacija KA (koji čita stek i bitan je za determinizaciju) i PA.

Primer 3.5.16

Napraviti automat koji prepoznaje sledeću gramatiku:

$$
\begin{array}{lcl}
A & \longrightarrow & Aa \\
 & | & B \\
B & \longrightarrow & bB \\
 & | & b
\end{array}
$$

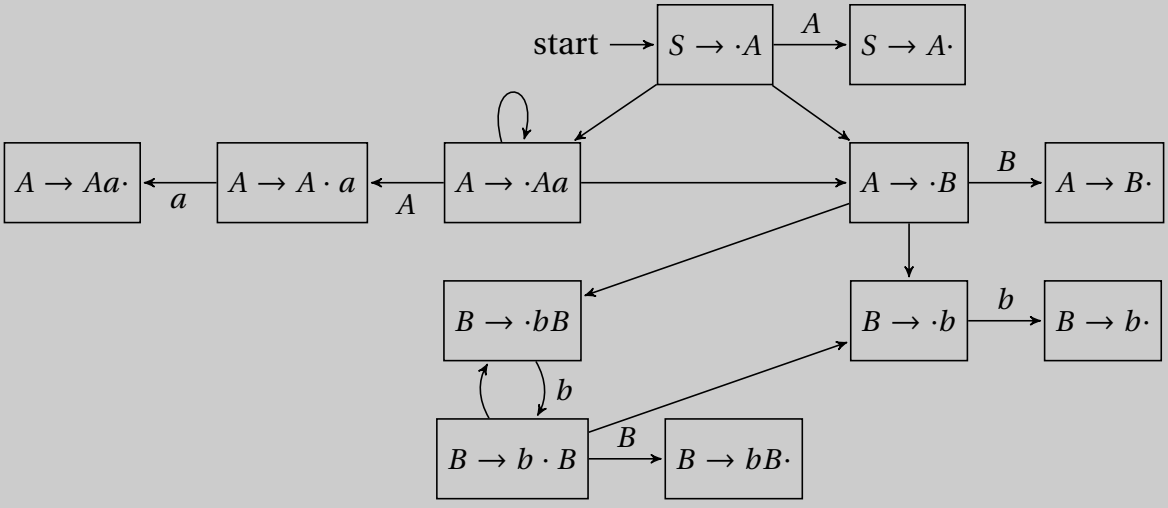
Pripremni korak:
Uvodimo novi neterminal S na početak i numerišemo svako pravilo.

$$
\begin{array}{lcl}
0\ S & \longrightarrow & A \\
1\ A & \longrightarrow & Aa \\
2 & | & B \\
3\ B & \longrightarrow & bB \\
4 & | & b
\end{array}
$$

Ajtem je gram pravilo koje u sebi sadrži tačkicu. Ono je indikator progresa. Ako je na početku, znači da tek treba da vidimo to pravilo, ako je u sredi, znači da smo u procesu čitanja tog pravila, a ako je na kraju, onda smo upravo pročitali to pravilo. Kad je tačkica na kraju, to je znak da treba redukovati.

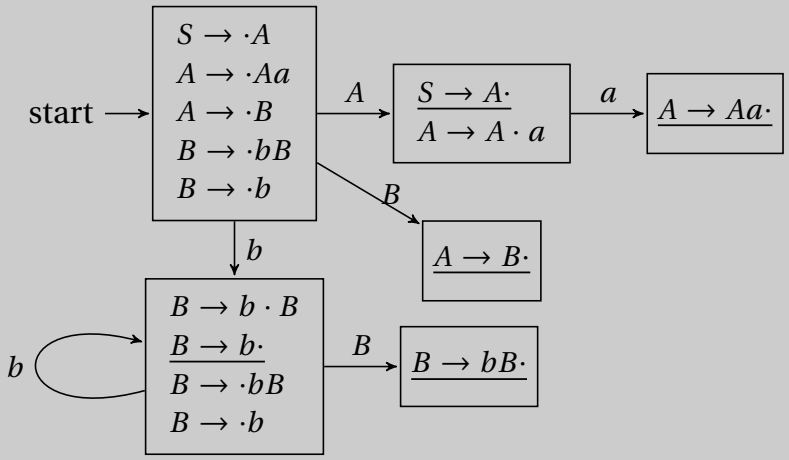
$$
\begin{array}{lcl}
S & \longrightarrow & \cdot A \\
S & \longrightarrow & A\cdot \\
A & \longrightarrow & \cdot Aa \\
A & \longrightarrow & A \cdot a \\
A & \longrightarrow & Aa\cdot \\
A & \longrightarrow & \cdot B \\
A & \longrightarrow & B\cdot \\
B & \longrightarrow & \cdot bB \\
B & \longrightarrow & b \cdot B \\
B & \longrightarrow & bB\cdot \\
B & \longrightarrow & \cdot b \\
B & \longrightarrow & b\cdot
\end{array}
$$

Nedeterministički automat:



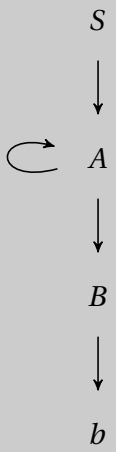
1. Kad god imamo tačkicu ispred neterminala, sa ε je spoj sa svim ajtemima koji odgovaraju tom neterminalu.
2. Kad god imamo tačkicu ispred simbola i vidimo taj simbol, možemoda pomerimo ajtem (tačkicu).

Deterministički automat:

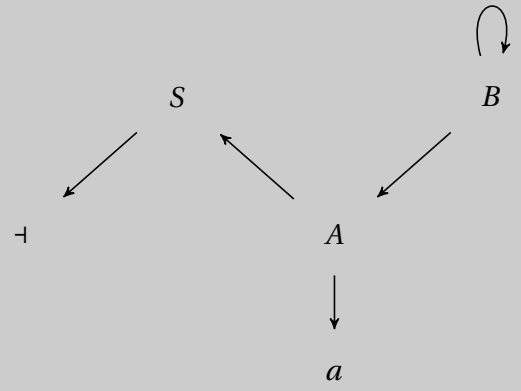


1. Kad imamo tačkicu ispred T u to stanje dodaj pravila sa tačkicom na početku.
2. Kad imamo skup ajtema, gledamo prelaze za sve njih, a ne samo za jedan.

Graf za skup Prvi:

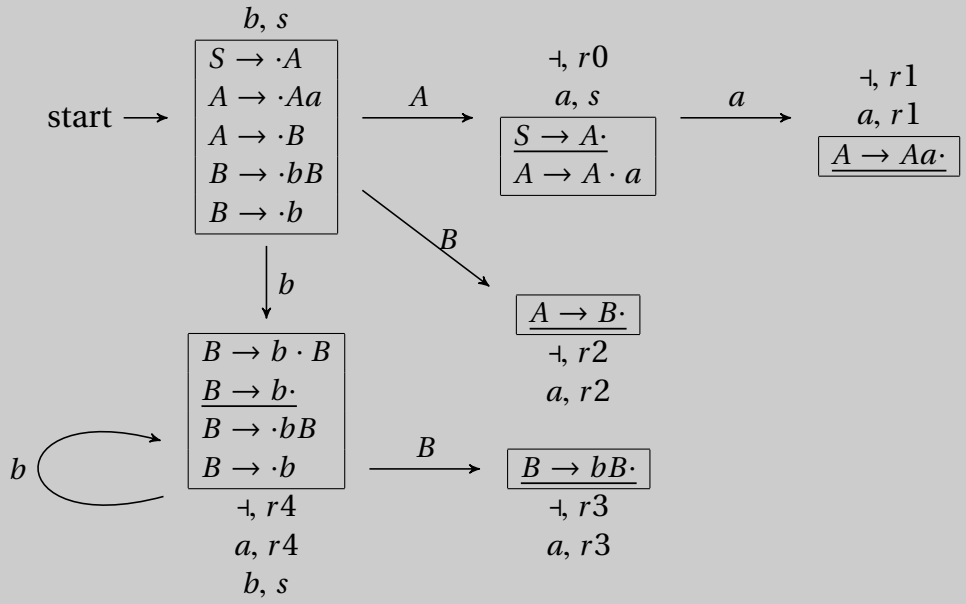


Graf za skup Sledi:



	$Prvi$	$Sledi$
S	b	$\dashv$
A	b	$\dashv$, a
B	b	$\dashv$, a

Dodeljujemo akcije svakom stanju:



Primenimo stek na ulaz bbaa:

$Stek$	$Ulaz$
${}_0\varepsilon$	$bbaa$
${}_0b_4$	baa
${}_0b_4b_4$	aa
${}_0b_4B_5$	aa
${}_0B_3$	aa
${}_0A_1$	aa
${}_0A_1a_2$	a
${}_0A_1$	a
${}_0A_1a_2$	$\dashv$
${}_0A_1$	$\dashv$
${}_0S_0$	$\dashv$

Action/Goto tabela:

	a	b	$\dashv$	A	B
0		$s4$		1	3
1	$s2$		$r0/acc$		
2	$r1$		$r1$		
3	$r2$		$r2$		
4	$r4$	$s4$	$r4$		5
5	$r3$		$r3$		

Prazna mesta u tabeli su greške.
Ove tabele u potpunosti kodiraju sliku.
Primenimo ovu tablicu na ulaz bbaa:

$Stek$	$Ulaz$
0	$bbaa$
04	baa
044	aa
045	aa
03	aa
01	aa
012	a
01	a
012	$\dashv$
01	$\dashv$
acc	$\dashv$

SLR - metode (simple LR)
LR gramatike (prihvatanje odozdo nagore)

Preciznije metode:

- kanonski LR(1)
- LALR(1) - u YACC-u se primenjuje ova metoda

Konflikti nastaju kad na istim simbolima imamo više prelaza (shift-reduce ili reduce-reduce konflikti).
Ovde je leva rekurzija poželjana - veliko parče ulaza se prebacuje na stek pa ga posle deo po deo obrađujemo. Za desnu rekurziju nam je potreban dublji stek.

Primer 3.5.17

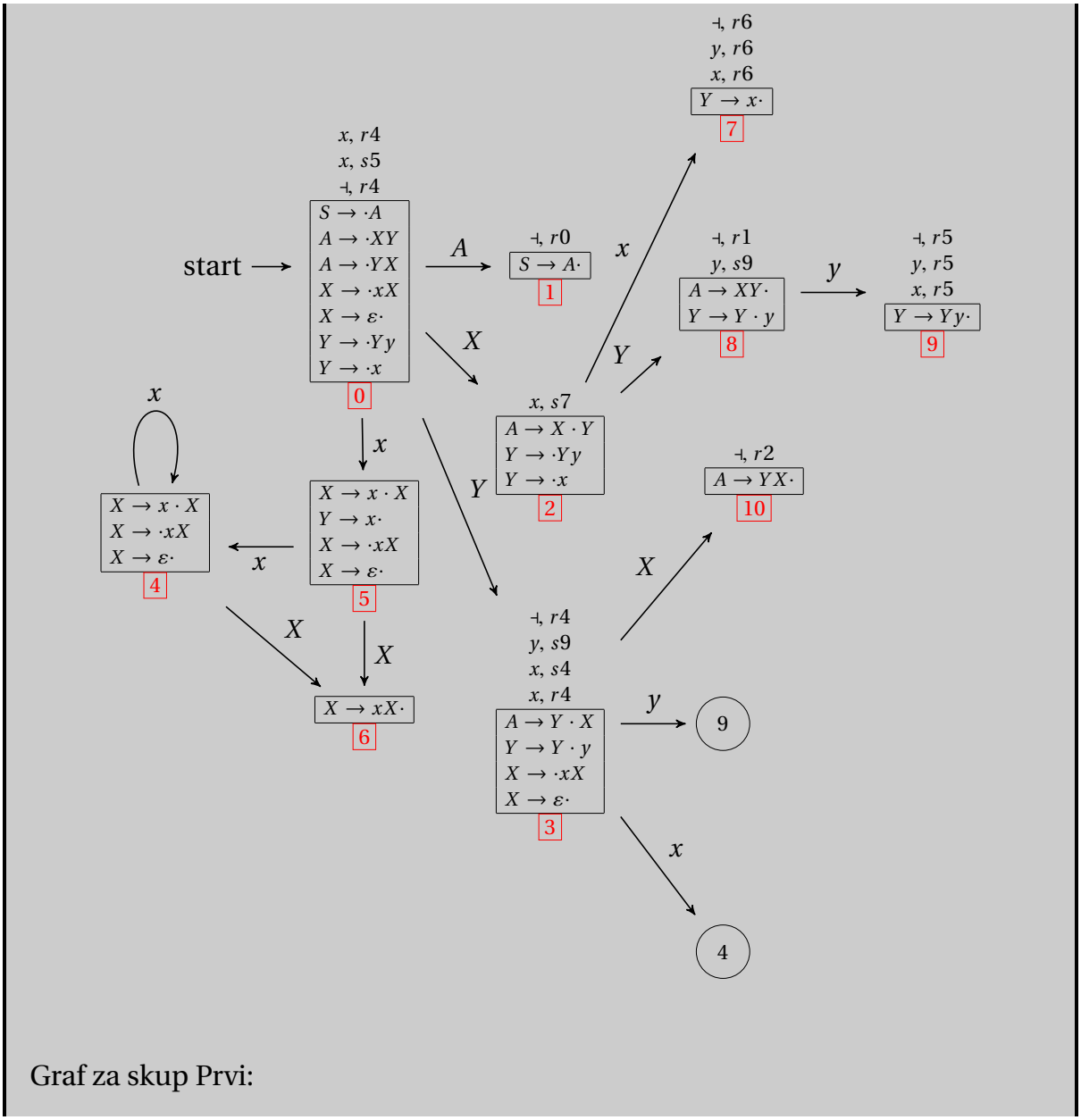
Napraviti automat za sledeću gramatiku:

$$\begin{array}{lcl} A & \longrightarrow & XY \\ & | & YX \\ X & \longrightarrow & xX \\ & | & \varepsilon \\ Y & \longrightarrow & Yy \\ & | & x \end{array}$$

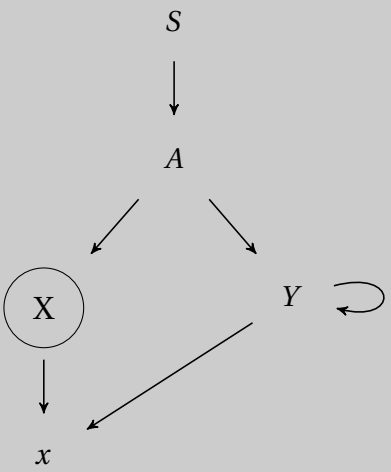
Pripremni korak:

$$\begin{array}{llcl} 0 & S & \longrightarrow & A \\ 1 & A & \longrightarrow & XY \\ 2 & & | & YX \\ 3 & X & \longrightarrow & xX \\ 4 & & | & \varepsilon \\ 5 & Y & \longrightarrow & Yy \\ 6 & & | & x \end{array}$$

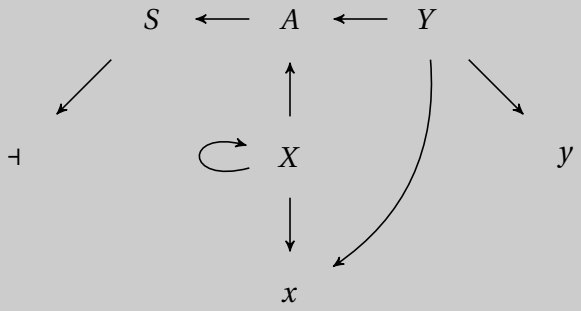
Deterministički automat



Graf za skup Prvi:



Graf za skup Sledi



	Prvi	*Sledi*
S	x	$\dashv$
A	x	$\dashv$
X	x	$x, \dashv$
Y	x	$x, y, \dashv$

Poglavlje 4

Semantička analiza

4.1 Atributske gramatike

Atributske gramatike su KSG koje podrazumevaju da svi tokeni imaju neke vrednosti. Obogaćujemo ih atributima i uz njih stavimo pravila za izračunavanje vrednosti.

Primer 4.1.1

Neka je zadata sledeća gramatika:

$$\begin{array}{rcl} E & \longrightarrow & E+E \\ & | & E*E \\ & | & (E) \\ & | & br \end{array}$$

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:

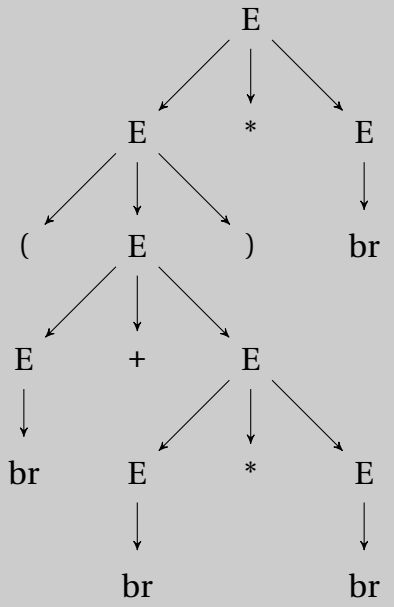
$$(3+5*2)*10.$$

Pretpostavljamo da token br ima atribut v, što se obeležava $br.v$ i pretpostavljamo

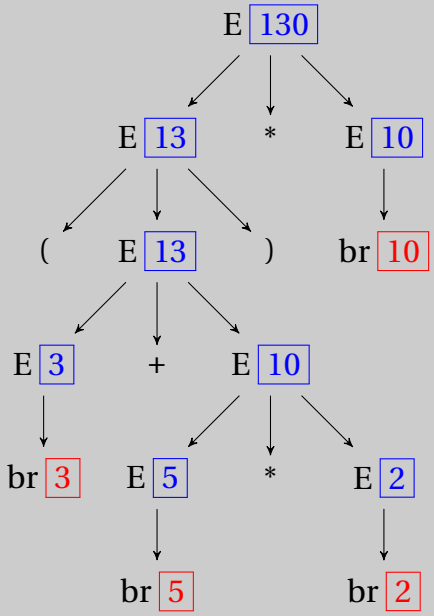
da je leksički analizator taj koji pridružuje vrednosti. Vrednost izraza računamo:

$$\overbrace{\overbrace{\overbrace{\overbrace{\underbrace{(}_{3}\ \underbrace{br}_{3}\ \underbrace{+}_{5}\ \underbrace{br}_{5}\ \underbrace{*}_{2}\ \underbrace{br}_{2}\ \underbrace{)}_{10}}^{10}}^{13}}^{13}\ \underbrace{*}_{10}\ \underbrace{br}_{10}}^{130},$$

što se češće obeležava drvetom izvođenja:



Listovi drveta 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 drvetu 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. Izračunavanja ćemo pisati u vitičastim zagradama pored pravila:

$$\begin{array}{lcll} E^1 & \longrightarrow & E^2 + E^3 & \{E^1.v = E^2.v + E^3.v\} \\ & | & E^2 * E^3 & \{E^1.v = E^2.v * E^3.v\} \\ & | & (E^2) & \{E^1.v = E^2.v\} \\ & | & br & \{E^1.v = br.v\} \end{array}$$

U YACC-u to zapisujemo na sledeći način:

```
E : E '+' E    {$$ = $1 + $3; }
  | E '*' E    {$$ = $1 * $3; }
  | '(' E ')'    {$$ = $2; }
  |   br       {$$ = $1; }
  ;
```

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.

Razlikujemo 2 vrste atributskih gramatika:

1. Sintetizovane gramatike – vrednost atributa simbola sa leve strane računamo na osnovu vrednosti atributa simbola sa desne strane. Nazivamo ih i S-atributske gramatike. U drvetu izvođenja to predstavlja kretanje odozdo nagore.

2. Nasleđene gramatike – vrednost atributa simbola sa desne strane izračunavamo

na osnovu vrednosti atributa simbola koji mu prethode (u stablu, „otac" i „starija braća"). U drvetu izvođenja to predstavlja kretanje odozgo-naniže i sleva nadesno.

Dakle, moguća su tri smera kretanja kroz drvo. Zdesna nalevo nije moguće jer bi to značilo da koristimo vrednost atributa nekog simbola koji još nismo pročitali.

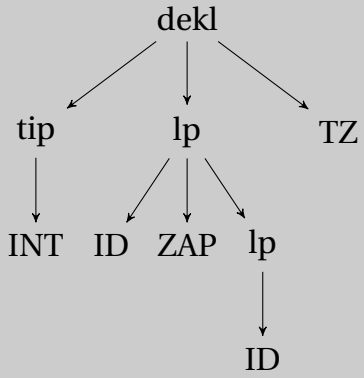
Pogledajmo konkretnu situaciju gde se koriste nasleđeni atributi:

Primer 4.1.2

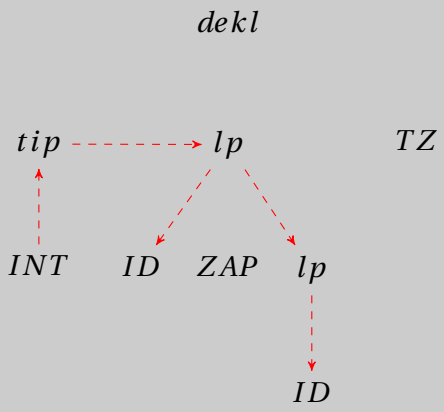
Napisati atributsku gramatiku za sledeći fragment koda:

```
int a, b;
```

Drvo 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 drvetu izvođenja radi detaljnije ilustracije:



Gramatika koja opisuje ovaj kod (pravila sada umećemo u sredinu kada su nasleđena) izgleda:

$$\begin{array}{lcll} dekl & \longrightarrow & tip & \{lp.t + tip.t\} \quad lp \\ lp^1 & \longrightarrow & ID \quad ZAP & \{ID.t = lp^2.t = lp^1.t\} \quad lp^2 \\ & | & ID & \{ID.t = lp^2.t\} \\ tip & \longrightarrow & INT & \\ & | & FLOAT & \end{array}$$

Primer 4.1.3

Neka je zadata gramatika

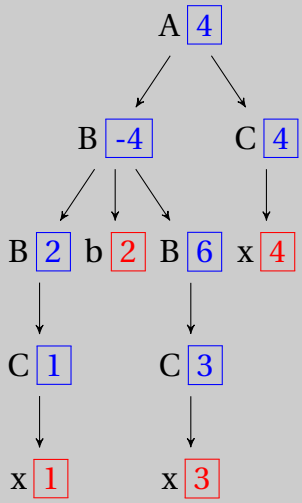
$$\begin{array}{lcll} A & \longrightarrow & BC & \{\$\$ = \$1 + 2 * \$2;\} \\ B & \longrightarrow & BbB & \{\$\$ = \$1 - \$3;\} \\ & | & C & \{\$\$ = 2 * \$1;\} \\ C & \longrightarrow & CaC & \{\$\$ = 3 * \$1 + \$2;\} \\ & | & x & \{\$\$ = \$1;\} \end{array}$$

i niska „`xbxx`", pri čemu tokeni niske imaju redom atribute 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:

$$A \Longrightarrow BC \Longrightarrow BbBC \Longrightarrow CbBC \Longrightarrow CbCC \Longrightarrow^* xbxx.$$

Dalje nije problem. Treba nacrtati drvo izvođenja i propagirati vrednosti atributa odozdo-nagore prema pravilima gramatike. Iz stabla izvođenja:



dobijamo da je vrednost izraza jednaka 4.

Kako se ovo izračunavanje izvršava? YACC radi analizu odozdo-naviše. Na početku je stek prazan. YACC sam pravi Action/GoTo tablicu koja mu govori šta da radi u kom koraku. Konflikti se razrešavaju određivanjem asocijativnosti i prioriteta pomoću *%left* i *%right*.

Stek	*Ulaz*
$\begin{array}{\|c\|}\hline \varepsilon \\ \hline \varepsilon \\ \hline \end{array}$	$br + br * br$
$\begin{array}{\|c\|}\hline br \\ \hline 3 \\ \hline \end{array}$	$+br * br$
$\begin{array}{\|c\|}\hline E \\ \hline 3 \\ \hline \end{array}$	$+br * br$
$\begin{array}{\|c\|c\|}\hline E & + \\ \hline 3 & ? \\ \hline \end{array}$	$br * br$
$\begin{array}{\|c\|c\|c\|}\hline E & + & br \\ \hline 3 & ? & 2 \\ \hline \end{array}$	$*br$
$\begin{array}{\|c\|c\|c\|}\hline E & + & E \\ \hline 3 & ? & 2 \\ \hline \end{array}$	$*br$
$\begin{array}{\|c\|c\|c\|c\|}\hline E & + & E & * \\ \hline 3 & ? & 2 & ? \\ \hline \end{array}$	br
$\begin{array}{\|c\|c\|c\|c\|c\|}\hline E & + & E & * & br \\ \hline 3 & ? & 2 & ? & 5 \\ \hline \end{array}$	$\dashv$
$\begin{array}{\|c\|c\|c\|c\|c\|}\hline E & + & E & * & E \\ \hline 3 & ? & 2 & ? & 5 \\ \hline \end{array}$	$\dashv$

Stek	*Ulaz*
$\begin{array}{\|c\|c\|c\|}\hline E & + & E \\ \hline 3 & ? & 10 \\ \hline \end{array}$	⊣
$\begin{array}{\|c\|}\hline E \\ \hline 13 \\ \hline \end{array}$	⊣

Imamo 2 steka: jedan za simbole, koji se naziva **stek parsiranja**, i drugi za vrednosti, koji se naziva **stek atributa**. Primetimo da smo na steku atributa ponekad imali vrednosti označene znakom '?'. Taj znak obeležava neku vrednost koja nam nije bitna jer je nigde ne koristimo.

Primeri ispitnih rokova

Januar 2016. godine

1. Definisati pojam normalizovanog automata koji se dobija Tompsonovim algoritmom. Uz korišćenje Tompsonovog algoritma, konstruisati MDKA za regularni jezik $a(a|b)*ba$.

2. Definisati anulirajuće terminale u KSG. Opisati kako se oni mogu izračunati. Definisati pojam ε-slobodne gramatike. Odrediti ε-slobodnu gramatiku ekvivalentnu gramatici

$$\begin{aligned} S &\longrightarrow aA \mid bB \mid AB, \\ A &\longrightarrow Aa \mid BB, \\ B &\longrightarrow BC \mid Bb \mid \varepsilon, \\ C &\longrightarrow Cc \mid c. \end{aligned}$$

3. LL(1) gramatikom opisati prototip funkcija u C-u u kojima se javljaju samo promenljive osnovnih tipova. Primeri takvih prototipova su:

```
int f();
float g(int x, char y);
```

   Na osnovu te gramatike u C-u implementirati sintaksički analizator koji rekurzivnim spustom proverava da li je dati prototip ispravan (pretpostaviti da je leksički analizator već implementiran kroz funkciju `yylex`).

4. Za gramatiku $S \longrightarrow xSyS \mid x$ konstruisati SLR(1) automat, ACTION i GOTO tablice i prikazati rad automata na prihvatanju niske $xxxyxyx$.

Februar 2016. godine

1. Kako se određuje presek dva automata? Konstruisati MDKA za sve reči nad azbukom $\Sigma = \{a, b\}$ koje se završavaju na aa, a u sebi sadrže paran broj slova b.

2. Definisati pojam potisnog automata i jezik potisnog automata koji prihvata praznom potisnom listom. Konstruisati potisni automat koji završnim stanjem prihvata reči nad azbukom $\{a, b\}$ koji su palindromi.

3. LL(1) gramatikom opisati definicije nabrojivih tipova u C-u. Na primer:

   ```
   enum boja { CRVENA, PLAVA, ZELENA };
   enum karta { ZANDAR = 12, DAMA, KRALJ };
   ```

   Na osnovu te gramatike u C-u implementirati sintaksički analizator koji rekurzivnim spustom proverava da li je data definicija ispravna (pretpostaviti da je leksički analizator već implementiran kroz funkciju `yylex`).

4. Za gramatiku $S \longrightarrow SxSy \mid x$ konstruisati SLR(1) automat, ACTION i GOTO tablice i prikazati rad automata na prihvatanju niske $xxxyxxy$.

Literatura

[1] Duško Vitas. *Prevodioci i interpretatori: Uvod u teoriju i metode kompilacije programskih jezika*. Univerzitet u Beogradu – Matematički fakultet, 2006. *ISBN 86-7589-056-7*.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006. *ISBN 0-201-10088-6*.

[3] Filip Marić. *Yet another Compiler-Compiler (YACC)*. Univerzitet u Beogradu – Matematički fakultet, 2004.