Zdravko
DOVEDAN HAN

FORMALNI JEZICI I PREVODIOCI

• sintaksna analiza i primjene

Recenzenti
Prof. dr. sc. Damir BORAS, FF, Zagreb
Prof. dr. sc. Mirko MALEKOVIĆ, FOI, Varaždin
Prof. dr. sc. Božidar TEPEŠ, UF, Zagreb

Lektorica
Dr. sc. Vjera LOPINA

Uređenje teksta
Autor

Predgovor

Ovo je druga knjiga od tri koje obrađuju teme iz formalnih jezika i prevodilaca. Posvećena je problemima sintaksne analize beskontekstnih jezika, kontekstnih jezika i jezika sa svojstvima. Svojim sadržajem u potpunosti pokriva sadržaj kolegija *Teorija sintaksne analize i primjene* predmeta *Formalni jezici i prevodioci* koji se predaje na Odsjeku za informacijske i komunikacijske znanosti Filozofskog fakulteta u Zagrebu. Sve teme su obrađene u devet poglavlja koja predstavljaju određene logičke cjeline.

0. *Osnove* je uvodno poglavlje u kojem je dan sažeti pregled definicija i temeljnih pojmova iz prve knjige.

1. *Uvod u teoriju sintaksne analize* sadrži temeljne pojmove teorije sintaksne analize, parsiranje i prepoznavanje, te njihovu klasifikaciju.

2. *Prepoznavanje regularnih jezika* poglavlje je u kojem se definira konačni prepoznavač koji prepoznaje regularne jezike. Veći je dio poglavlja posvećen primjeni regularnih izraza u pretraživanju teksta s posebnim težištem na njihovu realizaciju u jeziku Python.

3. *Prepoznavači beskontekstnih jezika* opisani su u trećem poglavlju. To su stogovni prepoznavač, prepoznavač s praznim stogom i prošireni stogovni prepoznavač.

Sljedeća četiri poglavlja posvećena su sintaksnoj analizi beskontekstnih jezika. Prva dva odnose se na višeprolazne (nedeterminističke) postupke sintaksne analize, a preostala dva na jednoprolazne (determinističke) postupke sintaksne analize.

4. *Višeprolazno parsiranje* prvo je poglavlje u kojem su opisani višeprolazni postupci sintaksne analize beskontekstnih jezika primjenljivi na cijeloj klasi beskontekstnih jezika. Obrađena su dva povratna ("backtrack") postupka, silazni i uzlazni.

5. *Tablični postupci parsiranja* poglavlje je u kojem su opisana dva tablična postupka višeprolazne sintaksne analize: Cock-Younger-Kasamijev i Earleyjev.

6. *LL(k) jezici i sintaksna analiza* obrađuje LL(k) jezike i postupke jednoprolazne sintaksne analize s lijevim parsanjem: predikatnu sintaksnu analizu i rekurzivni spust.

7. *LR(k) jezici i sintaksna analiza* obrađuje LR(k) jezike i postupke jednoprolazne sintaksne analize s desnim parsanjem: LR(0) gramatike i deterministički stogovni automat, sintaksnu analizu LR(1) jezika i sintaksnu analizu jezika s prioritetom operatora.

8. *Prepoznavač kontekstnih jezika* poglavlje je posvećeno sintaksnoj analizi (prepoznavanju) kontekstnih jezika i jezika bez ograničenja. Prikazana su dva prepoznavača kontekstnih jezika: dvostruko-stogovni prepoznavač i dvostruko-stogovni prepoznavač s jednim stanjem, te Turingov stroj kao prepoznavač jezika bez ograničenja.

9. *Prepoznavač jezika sa svojstvima* završno je poglavlje u kojem je definiran deterministički prepoznavač jezika sa svojstvima primjenljiv na svim beskontekstnim i kontekstnim jezicima. Postupak prepoznavanja utemeljen je na dvije tablice: tablici prijelaza i tablici akcija. Posebno je podesan za implementaciju na računalima.

U svim je poglavljima dano puno primjera koji upotpunjuju teorijska razmatranja, posebno pojedine definicije i algoritme. Na kraju poglavlja su pitanja i zadaci.

Kao što je već rečeno na početku, knjiga je namijenjena, u prvom redu, studentima studija društveno-humanističke informatike i opće informatologije na Odsjeku za informacijske i komunikacijske znanosti, ali sam siguran da će knjiga s ovakvim sadržajem zainteresirati studente sličnih usmjerenja, inženjere informatike i računarskih znanosti, napredne srednjoškolce i mnoge druge samouke informatičare. Vjerujem da će programi dani u prilozima biti dovoljna motivacija znatiželjnijim čitateljima da ih u prvom koraku usavrše, a potom pristupe realizaciji nekih drugih algoritama danih u knjizi.

Posebno ću biti zahvalan svima onima koji budu pažljivo pročitali ovu knjigu, prihvatili barem jedan njezin djelić i primijenili u svojoj praksi. Sve primjedbe i pitanja možete mi poslati na e-mail adresu:

zdovedan@hotmail.com.

U Zagrebu, rujna 2012. godine

Autor

Sadržaj

Mojoj dragoj Ines

Koja mi je obojila život
prekrasnim jezikom cvijeća...

mojoj vjernoj pratilji
na putovima zemaljskim...

i na putovima duhovnosti!

0.
OSNOVE

sa svojom njuškom mješanca
židovskog lutalice i grčkog pastira
i sa svojim kosama
sa svih strana svijeta
s očima potpuno orošenim
koje mi daju izgled sanjalice
meni koji više ne sanjam često
s rukama kradljivca muzičara i skitnice
koje su opljačkale tolike vrtove
s ustima koja su pila
ljubila i grizla
a da nikad nisu zasitila svoju glad

sa svojom njuškom mješanca
židovskog lutalice i grčkog pastira
razbojnika i vagabunda
s grudima koje su grlile
pod suncem ljeta
sve ono što je suknju nosilo
sa srcem koje je znalo patiti
onoliko koliko je patilo
a da od toga ne pravi drame
s dušom koja nema
ni najmanje šanse da se spasi
i izbjegne čistilište

sa svojom njuškom mješanca
židovskog lutalice i grčkog pastira
doći ću moja nježna zarobljenice
sestro moje duše
izvoru mog života
doći ću da upijem
tvojih dvadeset godina
bit ću princ krvi
sanjalica ili maloljetnik
već kako ti izabereš
i stvorit ćemo od svakog dana
cijelu jednu vječnost ljubavi
da bismo živjeli od nje umirući

mješanac
le métèque

(georges moustaki/
nepoznati student, 1973)

Izučavanje teorije sintaksne analize temelji se na definicijama danim u prvoj knjizi. To je definicija jezika, regularnih izraza, gramatika i automata koja je ovdje dana u sažetom pregledu, bez primjera.

0.1 JEZIK

Znak je jedinstven, nedjeljiv element. Alfabet je konačan skup znakova. Najčešće ćemo ga označivati sa $\mathcal{A}$. Ako se znakovi alfabeta $\mathcal{A}$ poredaju jedan do drugog dobije se niz znakova (engl. *string*) ili "povorka", ili "nizanica". Operacija dopisivanja znaka iza znaka, ili niza iza niza, naziva se nadovezivanje ili konkatenacija nizova.

Duljina niza znakova jest broj znakova sadržanih u njemu. Često se duljina niza znakova x označuje s $d(x)$ ili $|x|$. Niz znakova $aaaa\ldots a$, sačinjen od n jednakih znakova, piše se kao a^n.

Neka su x i y nizovi znakova nad alfabetom $\mathcal{A}$. Kaže se da je x prefiks ("početak"), a y sufiks ("dočetak") niza xy i da je y podniz niza xyz (x kao prefiks i y kao sufiks niza xy istodobno su i njegovi podnizovi). Ako je $x \neq y$ i x je prefiks (sufiks) niza y, kaže se da je x svojstveni prefiks (sufiks) od y.

Definira se i niz znakova koji ne sadrži nijedan znak. Naziva se prazan niz. Označivat ćemo ga s ε ili λ. Za bilo koji niz w vrijedi $\varepsilon w = w\varepsilon = w$. Duljina praznog niza jednaka je 0, $|\varepsilon| = 0$, odnosno, ako je a bilo koji znak, vrijedi $a^0 = \varepsilon$. Ako je $x = a_1 \ldots a_n$ niz, obrnuti niz (ili "reverzni" niz) jest x^R, $x^R = a_n \ldots a_1$. Umjesto x^R može se pisati x^{-1}. Vrijedi $x = (x^{-1})^{-1}$.

Ako je $\mathcal{A} = \{a_1, \ldots, a_n\}$ alfabet i $x \in \mathcal{A}^+$ s $N_{a_i}(x)$ označit ćemo broj pojavljivanja znaka a_i u nizu x. Ako su $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ i $\mathcal{B} = \{b_1, b_2, \ldots, b_n\}$ dva alfabeta, definira se njihov produkt:

$$\mathcal{AB} = \{a_i b_j : a_i \in \mathcal{A}, b_j \in \mathcal{B}\}$$

a to je skup svih nizova znakova duljine 2 u kojima je prvi znak iz alfabeta $\mathcal{A}$, a drugi iz alfabeta $\mathcal{B}$. Ako je $\mathcal{A} \neq \mathcal{B}$ tada je i $\mathcal{AB} \neq \mathcal{BA}$ (produkt dvaju alfabeta nije komutativan). Operacija potenciranja alfabeta, $\mathcal{A}^n$, $n \geq 0$, definirana je rekurzivno:

1) $\mathcal{A}^0 = \{\varepsilon\}$
2) $\mathcal{A}^n = \mathcal{A}\mathcal{A}^{n-1}$ za $n > 0$

Dakle, zaključujemo da će $\mathcal{A}^n$ biti skup svih nizova znakova nad alfabetom duljine n.

Skup svih nizova znakova koji se mogu izgraditi nad alfabetom $\mathcal{A}$, uključujući i prazan niz ε i sam alfabet, označivat ćemo sa $\mathcal{A}^*$. Vrijedi:

$$\mathcal{A}^* = \bigcup_{n=0}^{\infty} \mathcal{A}^n$$

Sa $\mathcal{A}^+$ označivat ćemo skup $\mathcal{A}^* \setminus \{\varepsilon\}$. Primijetimo da su $\mathcal{A}^*$ i $\mathcal{A}^+$ beskonačni ali prebrojivi skupovi! Sa $\mathcal{A}^{*k}$ označivat ćemo konačan skup (podskup od $\mathcal{A}^*$) svih nizova znakova nad $\mathcal{A}$ duljine od 0 do k, a sa $\mathcal{A}^{+k}$ označivat ćemo konačan skup (podskup od $\mathcal{A}^+$) svih nizova znakova nad $\mathcal{A}$ duljine od 1 do k. Unarna operacija * poznata je i pod nazivom Kleeneova zvjezdica, jer ju je prvi put definirao Stephen Kleene. Operacija $^+$ je Kleeneov plus.

Ako je $\mathcal{A}$ alfabet i $\mathcal{A}^*$ skup svih nizova znakova nad $\mathcal{A}$, jezik $\mathcal{L}$ nad alfabetom $\mathcal{A}$ jest bilo koji podskup od $\mathcal{A}^*$, tj.

$$\mathcal{L} \subseteq \mathcal{A}^*$$

Često se piše $\mathcal{L}(\mathcal{A})$ da se naznači definiranost nekog jezika $\mathcal{L}$ nad alfabetom $\mathcal{A}$. Nizovi znakova koji čine elemente jezika nazivamo rečenice. Za jezik $\mathcal{L}$ u kojem za sve njegove rečenice w vrijedi da nisu svojstveni prefiks (sufiks) ni jednoj rečenici x, $x \in \mathcal{L}$ i $x \neq w$, kaže se da ima svojstvo prefiksa (sufiksa).

Operacije nad jezicima

S obzirom na to da je jezik skup, primjenom poznatih operacija nad skupovima mogu se iz definiranih graditi novi jezici. Elementi jezika su nizovi znakova pa se može definirati i operacija nadovezivanja.

Ako su $\mathcal{L}_1$ i $\mathcal{L}_2$ jezici, $\mathcal{L}_1 \subseteq \mathcal{A}_1^*$ i $\mathcal{L}_2 \subseteq \mathcal{A}_2^*$, tada je $\mathcal{L}_1\mathcal{L}_2$ nadovezivanje (ulančavanje ili konkatenacija) ili produkt jezika $\mathcal{L}_1$ i $\mathcal{L}_2$:

$$\mathcal{L}_1\mathcal{L}_2 = \{xy : x \in \mathcal{L}_1,\ y \in \mathcal{L}_2\}$$

Simboli i nizovi simbola

Često se promatraju nizovi znakova konačne duljine koji se mogu smatrati jedinstvenom, nedjeljivom cjelinom. Takvi nizovi znakova nazivaju se simboli ili riječi.

Skup svih simbola definiran nad alfabetom $\mathcal{A}$ označivat ćemo s $\mathcal{V}$ i nazivati rječnik. To je uvijek konačan skup. Budući je $\mathcal{V} \subseteq \mathcal{A}^*$, zaključujemo da je $\mathcal{V}$ jezik. Definira se i jezik nad rječnikom, tj. $\mathcal{L}_\mathcal{V} \subseteq \mathcal{V}^*$. Rečenice takvog jezika i dalje su nizovi znakova iz $\mathcal{A}^*$, ali se mogu promatrati i kao nizovi simbola rječnika $\mathcal{V}$.

Klasifikacija jezika

Prema hijerarhiji Chomskog jezike klasificiramo u četiri skupine (ili tipa), kao što je prikazano u sljedećoj tablici:

tip	naziv
0	bez ograničenja
1	kontekstni
2	beskontekstan
3	linearan

Da bismo odredili kojoj klasi jezika pripada neka rečenica korisno je znati lemu ili svojstvo napuhavanja ("*pumping lemma*") koje kaže da svaki jezik dane klase može biti "napuhan" i pritom još uvijek pripadati danoj klasi. Jezik može biti napuhan ukoliko se dovoljno dugi niz znakova jezika može rastaviti na podnizove, od kojih neki mogu biti ponavljani proizvoljan broj puta u svrhu stvaranja novog, duljeg niza znakova koji je još uvijek u istom jeziku. Stoga, ukoliko vrijedi svojstvo napuhavanja za danu klasu jezika, bilo koji neprazni jezik u klasi će sadržavati beskonačan skup konačnih nizova znakova izgrađenih jednostavnim pravilom koje daje svojstvo napuhavanja.

Regularni skupovi

Neka je $\mathcal{A}$ alfabet. Regularni skup (regularni jezik) nad $\mathcal{A}$ definiran je rekurzivno na sljedeći način:

1) $\varnothing$ (prazan skup) je regularni skup nad $\mathcal{A}$.
2) $\{\varepsilon\}$ je regularni skup nad $\mathcal{A}$.
3) $\{\mathtt{a}\}$ je regularni skup nad $\mathcal{A}$, za sve $\mathtt{a}$ iz $\mathcal{A}$.
4) Ako su P i Q regularni skupovi nad $\mathcal{A}$, regularni skupovi su i:

   a) $P \cup Q$ b) PQ c) P^* d) (P)

Dakle, podskup od $\mathcal{A}^*$ jest regularan ako i samo ako je $\varnothing$, $\{\varepsilon\}$ ili $\{\mathtt{a}\}$, za neki $\mathtt{a}$ u $\mathcal{A}$, ili se može dobiti iz njih konačnim brojem primjene operacija unije, produkta i (Kleeneove) operacije *. Izraz s regularnim skupovima može imati i zagrade da bi se naznačio prioritet izvršavanja ove tri operacije, pa najviši prioritet ima podizraz unutar zagrada, potom Kleenova operacija (potenciranje), produkt i, na kraju, unija.

SVOJSTVA REGULARNIH SKUPOVA

Sada ćemo dati jedno svojstvo regularnih skupova čime će biti određeno je li dani skup regularan. To je "svojstvo napuhavanja", a definirano je u slijedećoj lemi napuhavanja regularnih skupova: Neka je $\mathcal{R}$ regularni skup. Tada postoji konstanta k takva da ako je $w \in \mathcal{R}$ i $|w| \geq k$, tada se w može napisati kao xyz, gdje je $0 < |y| \leq k$ i $xy^i z \in \mathcal{R}$ za sve $i \geq 0$.

Lema napuhavanja definira osnovno svojstvo nizova regularnog skupa da svi proizvoljno dugi nizovi (rečenice) regularnog jezika mogu biti "napuhane", tj. postoji središnji dio niza koji se ponavlja proizvoljan broj puta da bi proizveo novi niz koji je u istom jeziku. U praksi se lema napuhavanja često koristi da bi se dokazalo da dani jezik nije regularan.

0.2 REGULARNI IZRAZI

Regularni izrazi (još i "pravilni izrazi") na alfabetu $\mathcal{A}$ označuju (generiraju) određene regularne skupove. Regularni izraz definira se rekurzivno, na sljedeći način:

(1) $\varnothing$ je regularni izraz koji označuje regularni skup $\varnothing$.
(2) ε je regularni izraz koji označuje regularni skup $\{\varepsilon\}$.

(3) a iz $\mathcal{A}$ je regularni izraz koji označuje regularni skup {a}.
(4) Ako su p i q regularni izrazi koji označuju regularne skupove P i Q, redom, tada su:

   (a) $(p+q)$ ili $(p|q)$ regularni izraz koji označuje regularni skup $P \cup Q$.
   (b) (pq) regularni izraz koji označuje regularni skup PQ.
   (c) $(p)^*$ regularni izraz koji označuje regularni skup P^*.

Operaciju "+" ili "|" čitamo "ili". Za regularni izraz p^* koristit ćemo i notaciju $\{p\}$ (što ne treba poistovjećivati sa skupom). Ako je pp^* regularni izraz, može se napisati kao p^+ ili $p\{p\}$. Također ćemo koristiti i notaciju $\{p\}_m^n$ koja ima značenje dopisivanje izraza p najmanje m, najviše n puta, $m \le n$. Izostavljeno m ima značenje 0, a izostavljeno n ima značenje *. Ako ne postoji dvoznačnost u nekom regularnom izrazu, suvišne se zagrade mogu izbaciti. Može se zamisliti da operacija * (i $^+$) ima najviši prioritet, potom operacija nadovezivanja i, na kraju, operacija + (ili |).

Algebarska svojstva regularnih izraza

Reći ćemo da su dva regularna izraza jednaka (=) ako označuju isti regularni skup. Ako su α, β i γ regularni izrazi, tada vrijede sljedeća algebarska svojstva:

1) $\alpha\|\beta = \beta\|\alpha$	2) $\alpha\|(\beta\|\gamma) = (\alpha\|\beta)\|\gamma$	3) $\alpha(\beta\gamma) = (\alpha\beta)\gamma$
4) $\alpha(\beta\|\gamma) = \alpha\beta\|\alpha\gamma$	5) $(\alpha\|\beta)\gamma = \alpha\gamma\|\beta\gamma$	6) $\alpha\varepsilon = \varepsilon\alpha = \alpha$
7) $\alpha^* = \alpha\|\alpha^*$	8) $(\alpha^*)^* = \alpha^*$	9) $\alpha+\alpha = \alpha$

0.3 GRAMATIKE

Gramatika je četvorka $G=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$, gdje su:

$\mathcal{N}$ konačan skup neterminalnih znakova,

$\mathcal{T}$ konačan skup terminalnih znakova (alfabet) uz uvjet da je

$$\mathcal{T} \cap \mathcal{N} = \varnothing$$

$\mathcal{P}$ konačan skup parova nizova:

$$\{(\alpha, \beta): \alpha=\alpha_1\gamma\alpha_2;\ \alpha_1, \alpha_2, \beta \in (\mathcal{N}\cup\mathcal{T})^*,\ \gamma\in\mathcal{N}\}$$

(niz α je iz $(\mathcal{N}\cup\mathcal{T})^+$ i mora sadržati bar jedan znak iz skupa $\mathcal{N}$),

$\mathcal{S}$ poseban znak iz $\mathcal{N}$, $\mathcal{S}\in\mathcal{N}$, nazvan početni znak (ili *početni simbol*).

Element (α,β) iz $\mathcal{P}$ piše se $\alpha\to\beta$ i naziva produkcija. Simbol "$\to$" čita se "producira", "može biti zamijenjeno s" ili "preobličuje se u". Ako $\mathcal{P}$ u nekoj gramatici sadrži produkcije:

$$\alpha \to \beta_1 \quad \ldots \quad \alpha \to \beta_n$$

piše se

$$\alpha \rightarrow \beta_1 | \ \beta_2 | \ldots | \beta_n$$

Znak "|" čita se "ili". β_i su alternative za α. Ako u $\mathcal{P}$ postoji produkcija oblika

$$\alpha \rightarrow \varepsilon | \ \beta | \ \beta\beta | \ \beta\beta\beta | \ \ldots$$

piše se $\alpha \rightarrow \{\beta\}$. Vitičaste zagrade omeđuju niz koji može biti izostavljen ili napisan jedanput, dvaput, triput, itd. Produkcija oblika:

$$\alpha \rightarrow \varepsilon | \ \beta$$

piše se $\alpha \rightarrow [\beta]$. Dakle, uglate zagrade omeđuju niz koji može biti izostavljen ili napisan jedanput. U daljnjem ćemo tekstu neterminale označivati velikim slovima engl. abecede. Terminali će biti mala slova engl. abecede i ostali znakovi (brojke, +, -, *, /, (,), itd.). Neterminal na početku prve produkcije bit će početni simbol.

Gramatika kao generator jezika

Rečenična forma gramatike $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ definirana je rekurzivno:

1) Početni znak je rečenična forma.
2) Ako je $\alpha\delta\gamma$, gdje su $\alpha,\gamma \in (\mathcal{N} \cup \mathcal{T})^*$, rečenična forma i $\delta \rightarrow \beta$ produkcija u $\mathcal{P}$, tada je $\alpha\beta\gamma$ također rečenična forma.

Rečenična forma koja ne sadrži nijedan neterminal naziva se rečenica. Nad skupom $(\mathcal{N} \cup \mathcal{T})^*$ gramatike $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ definira se relacija $\Rightarrow$, čita se izravno izvodi, na sljedeći način: Ako je $\alpha\delta\gamma$ niz iz $(\mathcal{N} \cup \mathcal{T})^*$ i $\delta \rightarrow \beta$ produkcija iz $\mathcal{P}$, tada

$$\alpha\delta\gamma \Rightarrow \alpha\beta\gamma$$

Ako za $\alpha_0, \ \alpha_1, \ \ldots, \ \alpha_n, \alpha_i \in (\mathcal{N} \cup \mathcal{T})^*, \ n \geq 1$, vrijedi

$$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$$

tada je $\alpha_0 \overset{n}{\Rightarrow} \alpha_n$ niz izvođenja duljine n. Općenito se piše

$$\alpha_0 \overset{*}{\Rightarrow} \alpha_n, \ n \geq 0, \ \alpha_0 \overset{+}{\Rightarrow} \alpha_n, \ n > 0$$

i kaže da α_0 izvodi α_n.

Shodno dvjema prethodnim definicijama jezik generiran gramatikom $\mathcal{G}$ može se napisati kao

$$\boxed{\mathcal{L}(\mathcal{G}) \ = \ \{\omega \in \mathcal{T}^* : \ \mathcal{S}^* \Rightarrow \omega\}}$$

što čitamo: "Jezik $\mathcal{L}$ generiran gramatikom $\mathcal{G}$ jest skup rečenica dobivenih nizom svih mogućih izvođenja krenuvši od početnog simbola $\mathcal{S}$".

Klasifikacija gramatika

Gramatike se mogu klasificirati prema obliku svojih produkcija. Za gramatiku $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ kaže se da je:

1) Tipa 3 ili linearna zdesna ako je svaka produkcija iz $\mathcal{P}$ oblika

   $A \to xB$ ili $A \to x$ $\quad A,B\in\mathcal{N},\ x\in\mathcal{T}^*$

   ili linearna slijeva ako je svaka produkcija iz $\mathcal{P}$ oblika

   $A \to Bx$ ili $A \to x$ $\quad A,B\in\mathcal{N},\ x\in\mathcal{T}^*$

   Gramatika linearna zdesna naziva se regularna gramatika ako je svaka produkcija oblika

   $A \to aB$ ili $A \to a$ $\quad A,B\in\mathcal{N},\ a\in\mathcal{T}$

   i jedino je dopuštena produkcija $\mathcal{S}\to\varepsilon$, ali se tada $\mathcal{S}$ ne smije pojavljivati niti u jednoj alternativi ostalih produkcija.

2) Tipa 2 ili beskontekstna ako je svaka produkcija iz $\mathcal{P}$ oblika:

   $A \to \alpha$ $\quad A\in\mathcal{N},\ \alpha\in(\mathcal{N}\cup\mathcal{T})^*$

3) Tipa 1 ili kontekstna ako je svaka produkcija iz $\mathcal{P}$ oblika

   $\alpha \to \beta$ uz uvjet da je $|\alpha|\le|\beta|$

4) Bez ograničenja ili tipa 0 ako produkcije ne zadovoljavaju nijedno od navedenih ograničenja.

Sada možemo reći da je jezik bez ograničenja ako je generiran gramatikom tipa 0, kontekstan ako je generiran gramatikom tipa 1, beskontekstan ako je generiran gramatikom tipa 2 i linearan (ili regularan) ako je generiran gramatikom tipa 3 (ili regularnom gramatikom). Četiri tipa gramatika i jezika uvedenih prethodnom definicijom nazivaju se hijerarhija Chomskog.

Svaka regularna gramatika istodobno je beskontekstna, beskontekstna bez ε–produkcija je kontekstna i, konačno, kontekstna gramatika je istodobno gramatika bez ograničenja. Ako s $\mathcal{L}_i$ označimo jezik tipa i, vrijedi $\mathcal{L}_{i+1}\subseteq\mathcal{L}_i$, $0\le i<3$. Regularne gramatike generiraju najjednostavnije jezike koji mogu biti generirani regularnim izrazima.

Prikaz gramatika

U prethodnim definicijama i primjerima razlikovali smo neterminalne i terminalne simbole prema vrsti znakova: velika slova bila su rezervirana za neterminale, a mala slova i ostali znakovi za terminale. Takvim dogovorom nije bilo neophodno uvijek posebno navoditi skupove neterminala i terminala. Bilo je dovoljno napisati produkcije i zadati početni simbol. Na taj način zadana je sintaksa jezika. Dva su najčešća načina prikaza gramatika: Backus-Naurovom formom i sintaksnim dijagramima.

Backus-Naurova forma (BNF)

Formalizam pisanja produkcija (ili "pravila zamjenjivanja") kojim smo dosad zadavali produkcije gramatike modifikacija je formalizma poznatog kao Backus-Naurova forma ili BNF. Prvi je put bio primijenjen u definiciji jezika ALGOL 60, 1963. godine i još uvijek je prisutan u praksi. Piše se prema sljedećim pravilima:

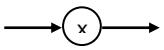
1) Neterminalni simboli pišu se između znakova "<" i ">".
2) Umjesto "$\rightarrow$" koristi se simbol "::=" i čita "definirano je kao".

Pišući neterminale između znakova "<" i ">" moguće je izborom njihovih imena uvesti "značenje" u produkcije, jer će nas imena podsjećati na vrstu rečenica koja će se generirati u nekom podjeziku.

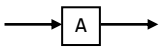
Sintaksni dijagrami

Produkcije gramatike $\mathcal{G}$ mogu biti prikazane i u obliku koji se naziva sintaksni dijagram. Sve je veća prisutnost sintaksnih dijagrama u novijoj literaturi, prije svega zato što se njihovom uporabom bolje uočava struktura jezika. Pravila konstruiranja sintaksnih dijagrama su sljedeća:

1) Terminalni simbol x prikazan je kao



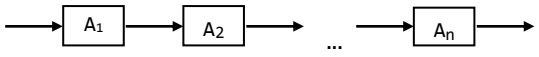
2) Neterminalni simbol A prikazan je kao



3) Produkcija oblika

$A \rightarrow A_1A_2 \ldots A_n \qquad A_i \in (\mathcal{N} \cup \mathcal{T})$

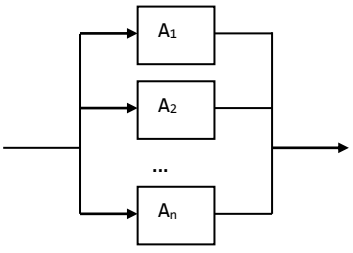
prikazuje se dijagramom



4) Produkcija oblika

$A \rightarrow A_1 | A_2 | \ldots | A_n \qquad A_i \in (\mathcal{N} \cup \mathcal{T})$

prikazuje se dijagramom

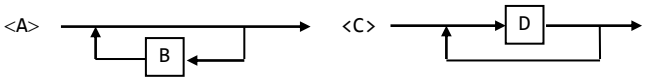


gdje je svaki A_i prikazan prema pravilima od (1) do (4). Ako je $A_i=\varepsilon$, ta se alternativa prikazuje punom crtom.

5) Produkcije oblika

$A \rightarrow \{B\}$ i $C \rightarrow [D]$ $B, D \in (\mathcal{N} \cup \mathcal{T})$

prikazuju se dijagramima:



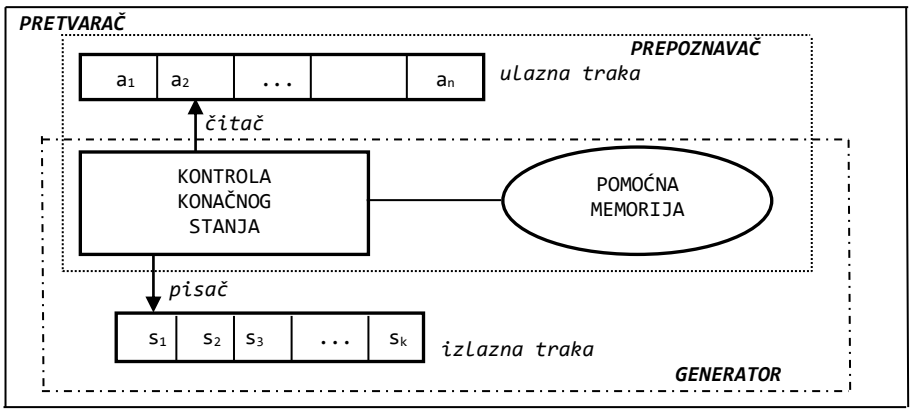
gdje su B i D prikazani dijagramima prema pravilima (1) do (4).

Često se u praksi pojednostavljuje pisanje sintaksnih dijagrama, posebno ako je nedvojbena razlika u pisanju terminala i neterminala. Na primjer, neterminali su riječi napisane velikim slovima, a terminali su riječi napisane malimm slovima ili su to brojevi i ostali znakovi.

0.4 AUTOMATI

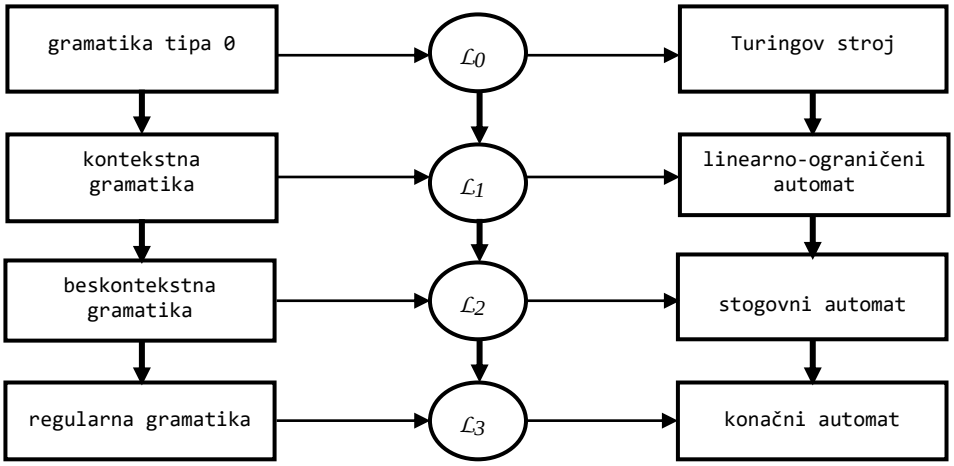
Osim gramatika, uvodimo automate kao važnu klasu generatora, prepoznavača i prevodilaca jezika, posebno pogodnih za implementaciju na računalima. Teorija je konačnih automata koristan instrument za razvoj sustava s konačnim brojem stanja čije mnogobrojne primjene nalazimo i u informatici. Programi, kao što je npr. tekstovni editor, često su načinjeni kao sustavi s konačnim brojem stanja. Na primjer, računalo se zasebno također može promatrati kao sustav s konačno mnogo stanja. Upravljačka jedinica, memorija i vanjska memorija nalaze se teoretski u svakom trenutku u jednom od vrlo velikog broja stanja, ali još uvijek u konačnom skupu stanja. Iz svakodnevnog je života upravljački mehanizam dizala još jedan dobar primjer sustava s konačno mnogo stanja. Prirodnost koncepta sustava s konačno mnogo stanja je razlog primjene tih sustava u različitim područjima, pa je i to vjerojatno najvažniji razlog njihova proučavanja.

Opći model automata dan je na sl. 0.1. Automat koji sadrži sve navedene "dijelove" naziva se pretvarač, automat bez ulazne vrpce je generator, a automat bez izlazne vrpce je prepoznavač. U ovom ćemo pogavlju proučiti konačne generatore, koji generiraju regularne jezike, i pokazati njihovu vezu s regularnim izrazima.



Sl. 0.1 – *Opći model automata.*

Automati najčešće imaju ulogu prepoznavača. Ovisno o jeziku koji se prepoznaje, odnosno o tipu gramatike koja generira takav jezik, postoje i vrste prepoznavača dane na sljedećem crtežu.



Sl. 0.2 - *Chomskyjeva hijerarhija gramatika, njihovi odgovarajući jezici i prepoznavači.*

Konačni automat

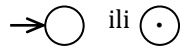
Konačni automat nema pomoćne memorije. To je matematički model sustava koji se nalazi u jednom od mnogih konačnih stanja. Stanje sustava sadrži obavijesti koje su dobivene na temelju dotadašnjih podataka i koje su potrebne da bi se odredila reakcija sustava iz idućih podataka. Drugim riječima, radi se o prijelaznim stanjima koje izvodi dani ulazni znak iz danog alfabeta pod utjecajem funkcije prijelaza. Svaki se ulazni znak može nalaziti samo u jednom prijelaznom stanju, pri čemu je dopušten povratak na prethodno stanje.

Konačni automat je uređena petorka $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$, gdje su:

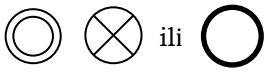
- Q konačni skup stanja
- Σ alfabet
- δ funkcija prijelaza, definirana kao $\delta: Q \times \Sigma \to \mathcal{P}(Q)$ gdje je $\mathcal{P}(Q)$ particija od Q
- q_0 početno stanje, $q_0 \in Q$
- F skup završnih stanja, $F \subseteq Q$

Dijagram prijelaza

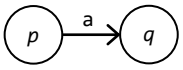
Iz definicije funkcije prijelaza zaključujemo da je to označeni, usmjereni graf čiji čvorovi odgovaraju stanjima automata, a grane su označene znakovima iz alfabeta. Ako, dakle, funkciju prijelaza prikažemo kao graf, dobivamo dijagram prijelaza. Označimo li u tom dijagramu početno stanje i skup završnih stanja, dobivamo konačni automat prikazan grafički. Početno ćemo stanje označivati s:



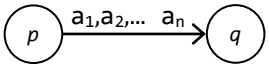
a završno s:



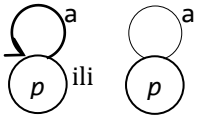
Ako je $p=\delta(q,a)$, tada postoji prijelaz iz stanja q u stanje p pa će grana u dijagramu prijelaza koja spaja q i p, s početkom u q i završetkom u p, biti označena s a:



Ako postoji više grana s početkom u q i završetkom u p, tj. ako je $p=\delta(q,a_1)=...=\delta(q,a_n)$, to će biti prikazano s:



Povratak nekim prijelazom a u isto stanje, tj. ako je $p=\delta(p,a)$, dijagram prijelaza je:



Tablica prijelaza

Funkcija prijelaza može se prikazati tablično. Tada se kaže da je to tablica prijelaza. Redovi tablice predstavljaju stanja, a stupci su označeni znakovima iz alfabeta i predstavljaju prijelaze. Na mjestu u redu označenom s q, $q\in Q$, i stupcu označenom s x, $x\in\Sigma$, upisan je skup narednih stanja (bez vitičastih zagrada) ako je funkcija $\delta(q, x)$ definirana, odnosno nije ništa upisano ako $\delta(q, x)$ nije definirano. Početno ćemo stanje označiti s $\rightarrow$ ili >, a konačno s $\otimes$ ili *.

Deterministički i nedeterministički automat

Iz definicije funkcije prijelaza vidimo da je moguć prijelaz iz tekućeg stanja u više narednih stanja s istim prijelazom a, $a\in\Sigma$, tj. da je ponašanje automata općenito nedeterminističko. Neka je $\mathcal{M}=(Q, \Sigma, \delta, q_0, F)$ konačni automat. Kažemo da je automat deterministički ako za sve $a\in\Sigma$ i sva stanja q, q' i q'' iz F i

$$\delta(q,a) = \{q', q''\}$$

slijedi da je $q'=q''$. Ako postoji barem jedan $a\in\Sigma$ tako da je $q'\neq q''$, automat je nedeterministički.

Stogovni automat

Stogovni automat PDA (Push Down Automat) jest uređena sedmorka, $\mathcal{P}=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, gdje su:

- Q konačan skup stanja (kontrole konačnog stanja)
- Σ ulazni alfabet
- Γ alfabet znakova stoga (potisne liste)
- δ funkcija prijelaza, definirana kao

$$\delta: \ Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma \ \rightarrow \ Q\times\Gamma^*$$

- q_0 početno stanje, $q_0\in Q$
- Z_0 početni znak stoga (potisne liste), $Z_0\in\Gamma$
- F skup završnih stanja, $F\subseteq Q$

Dvostruko-stogovni automat

Dvostruko-stogovni automat definiran je kao uređena sedmorka

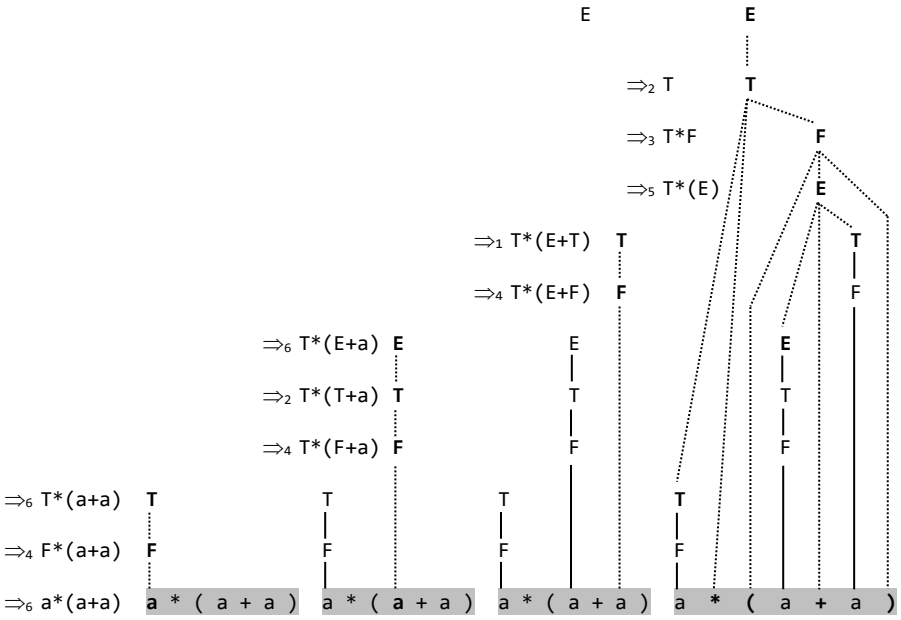
$$Pt=(Q, \Sigma, \Gamma_1, \Gamma_2, \Delta, s, F)$$

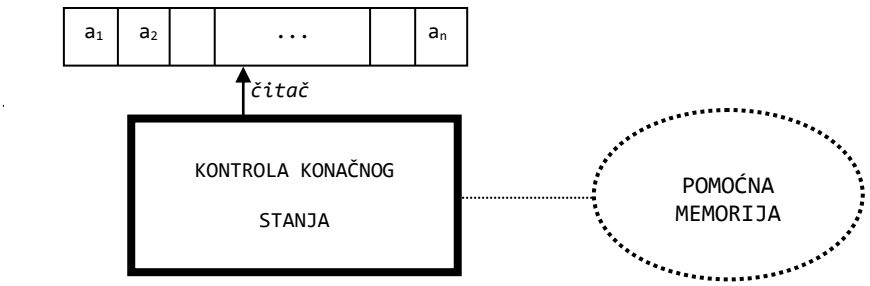
gdje su:

- Q konačan skup stanja (kontrole konačnog stanja)
- Σ ulazni alfabet
- Γ_1, Γ_2 alfabet prvog i drugog stoga
- Δ funkcija prijelaza, definirana kao $\Delta: Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma_1\times\Gamma_2 \ \rightarrow \ Q\times\Gamma_1^*\times\Gamma_2^*$
- s početno stanje, $s\in Q$
- F skup konačnih stanja, $F\subseteq Q$

Vidimo da se ovaj automat razlikuje od stogovnog automata u definiciji funkcije prijelaza Δ koja koristi dva stoga kao pomoćnu memoriju. Kaže se da je Pt linearno-ograničen jer je duljina oba stoga linearno proporcionalna duljini generiranog niza.

1.
UVOD U TEORIJU SINTAKSNE ANALIZE





U praksi se često susrećemo s problemom da je poznata gramatika ili generator nekog jezika i zadan niz znakova, a postavlja se pitanje je li to rečenica jezika generiranog danom gramatikom ili generatorom. Takav se postupak naziva "sintaksna analiza" ("sintaktička analiza").

Ako je jezik definiran gramatikom problem se svodi na nalaženje niza izvođenja, počevši od S, koji bi rezultirao tim nizom (rečenicom). Takav se postupak sintaksne analize naziva "parsiranje". Ustrojbu postupka parsiranja na računalu (program u izabranom jeziku za programiranje) nazivat ćemo "parser". Pri učenju nekog jezika gramatika je najčešće u ulozi prepoznavača. To je njezin pravi smisao. Bilo bi besmisleno znajući gramatiku nekog jezika prijeći odmah na izvođenje rečenica. Takav proces bi možda trajao nekoliko godina! I u teoriji formalnih jezika gramatika je češće u ulozi prepoznavača nego generatora jezika.

Ako je jezik definiran automatom (generatorom), postavljamo pitanje: može li dani niz biti generiran danim generatorom. Tada je automat u ulozi prepoznavača jezika koji analizira ulazni niz i poslije konačno mnogo promjena svojih konfiguracija, krenuvši od početnog stanja, doseže konačno stanje ako je niz u jeziku i odgovara "da", ili se postupak prekida i odgovara "ne" ako ulazni niz nije u jeziku. Takav se postupak sintaksne analize naziva "prepoznavanje", a automat koji to radi naziva se "prepoznavač".

U ovom ćemo poglavlju dati temeljne definicije postupka sintaksne analize: parsiranja i prepoznavanja.

1.1 PARSIRANJE

Vidjeli smo kako gramatika $G=(N,T,P,S)$ generira jezik $L(G)$. To je skup rečenica w dobivenih svim mogućim izvođenjima iz S, tj.

$$L(G) = \{w \in T^* : S \Rightarrow^* w\}$$

U praksi, poslije izvođenja gramatike nekog jezika i njezine verifikacije, dalje će njezina uporaba biti u provjeri je li dani niz w rečenica jezika $L(G)$. Tada se problem svodi na nalaženje niza izvođenja, počevši od S, koji bi rezultirao tim nizom (rečenicom). Takav se postupak općenito naziva sintaksna analiza (sintaktička analiza), odnosno parsiranje (*parsing*), ako se u postupku sintaksne analize koristi gramatika. Tada ćemo implementaciju postupka parsiranja na računalu (program u izabranom jeziku za programiranje) nazivati parser. Stablo izvođenja dobiveno iz niza izvođenja $S \Rightarrow^* w$ sada ćemo zvati stablo sintaksne analize ili stablo parsiranja.

♦ Lijevo i desno parsiranje

Neka je $G=(N,T,P,S)$ beskontekstna gramatika u kojoj su produkcije iz P označene (numerirane) s $1,2,...,p$ i neka je $\alpha \in (N \cup T)^*$. Tada je

(1) lijevo parsiranje za α niz produkcija rabljenih u krajnjem izvođenju slijeva krenuvši od S: $S \Rightarrow_{lm}^{*} \alpha$

(2) desno parsiranje za α reverzni niz produkcija rabljenih u krajnjem izvođenju zdesna krenuvši od S: $S \Rightarrow_{rm}^{*} \alpha$

Ako i predstavlja i-tu produkciju, pisat ćemo $\alpha \Rightarrow_{i} \beta$ ako je $\alpha \Rightarrow_{lm} \beta$, odnosno $\alpha \Rightarrow_{i} \beta$ ako je $\alpha \Rightarrow_{rm} \beta$. Također ćemo pisati $\alpha \Rightarrow_{i_1 i_2} \gamma$ ako je $\alpha \Rightarrow_{i_1} \beta$ i $\beta \Rightarrow_{i_2} \gamma$, odnosno $\alpha \Rightarrow_{i_1 i_2} \gamma$ ako je $\alpha \Rightarrow_{i_1} \beta$ i $\beta \Rightarrow_{i_2} \gamma$.

♣ Primjer 1.1

Neka je G_E gramatika s numeriranim produkcijama

(1) E → E+T (2) E → T (3) T → T*F (4) T → F (5) F → (E) (6) F → a

Lijevo parsiranje rečenice a*(a+a) je 23465124646, što je dobiveno iz niza izvođenja slijeva:

E $\Rightarrow_2$ T $\Rightarrow_3$ T*F $\Rightarrow_4$ F*F $\Rightarrow_6$ a*F $\Rightarrow_5$ a*(E) $\Rightarrow_1$ a*(E+T) $\Rightarrow_2$ a*(T+T) $\Rightarrow_4$ a*(F+T)
$\Rightarrow_6$ a*(a+T) $\Rightarrow_4$ a*(a+F) $\Rightarrow_6$ a*(a+a)
E $\Rightarrow_{23465124646}$ a*(a+a)

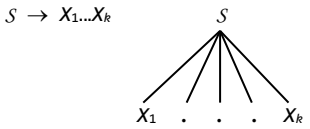
a desno parsiranje je 64642641532, što je dobiveno iz (reverznog) niza izvođenja zdesna:

E $\Rightarrow_2$ T $\Rightarrow_3$ T*F $\Rightarrow_5$ T*(E) $\Rightarrow_1$ T*(E+T) $\Rightarrow_4$ T*(E+F) $\Rightarrow_6$ T*(E+a) $\Rightarrow_2$ T*(T+a) $\Rightarrow_4$ T*(F+a)
$\Rightarrow_6$ T*(a+a) $\Rightarrow_4$ F*(a+a) $\Rightarrow_6$ a*(a+a)
E $\Rightarrow_{64642641532}$ a*(a+a)

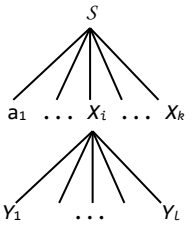
Lijevo i desno parsiranje primjenljivi su samo nad beskontekstnim (i linearnim) jezicima. U lijevom parsiranju stablo sintaksne analize izvodi se s vrha ili silazno, od korijena prema listovima (top-down). U desnom parsiranju stablo sintaksne analize izvodi se odozdo prema vrhu ili uzlazno, od listova prema korijenu (bottom-up).

Silazna sintaksna analiza

Znajući lijevo parsiranje $\pi = i_1 i_2 \ldots i_n$ rečenice $w = a_1 a_2 \ldots a_n$ iz $L(G)$ može se izvesti stablo parsiranja (sintaksne analize) u značenju "s vrha" ili silazno, pa kažemo da je to silazna sintaksna analiza (SA). Počinje se s korijenom stabla, označenim sa S. Parsiranje i_1 daje produkciju koja je upotrijebljena u ekspanziji stabla. Pretpostavimo da i_1 označuje produkciju $S \rightarrow X_1 \ldots X_k$ pa se može kreirati k slijednika iz čvora S koji su označeni s $X_1, X_2, \ldots, X_k$



Ako su $X_1, X_2, \ldots, X_{i-1}$ terminali, tada prvih i-1 simbola (znakova) ulaznog niza w moraju biti $X_1 X_2 \ldots X_{i-1}$. Produkcija i_2 mora biti oblika $X_i \rightarrow Y_1 \ldots Y_l$ pa se nastavlja ekspanzija stabla iz čvora X_i:



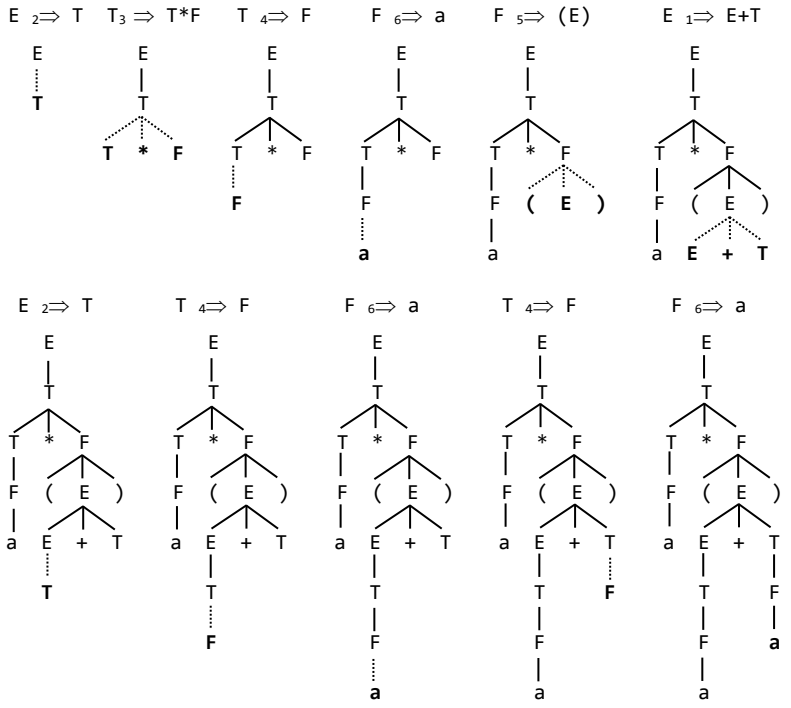
Postupak se nastavlja ekspanzijom na opisani način sve dok se ne izgradi stablo čiji će listovi biti znakovi niza *w*.

♣ Primjer 1.2

Lijevo parsiranje rečenice a*(a+a) jezika definiranog gramatikom $\mathcal{G}_E$

(1) E→E+T (2) E→T (3) T→T*F (4) T→F (5) F→(E) (6) F→a

iz primjera 1.1 bilo je 23465124646. Tome odgovara stablo sintaksne analize dobiveno ekspanzijom od korijena označenoga s E:



Uzlazna sintaksna analiza

Analizirajmo sada desno parsiranje. Ako pogledamo krajnje desno izvođenje a*(a+a) krenuvši od startnog simbola E, gramatike $\mathcal{G}_E$ iz primjera 1.1 imamo:

E ⇒₂ T
⇒₃ T*F
⇒₅ T*(E)
⇒₁ T*(E+T)
⇒₄ T*(E+F)
⇒₆ T*(E+a)
⇒₂ T*(T+a)
⇒₄ T*(F+a)
⇒₆ T*(a+a)
⇒₄ F*(a+a)
⇒₆ a*(a+a)

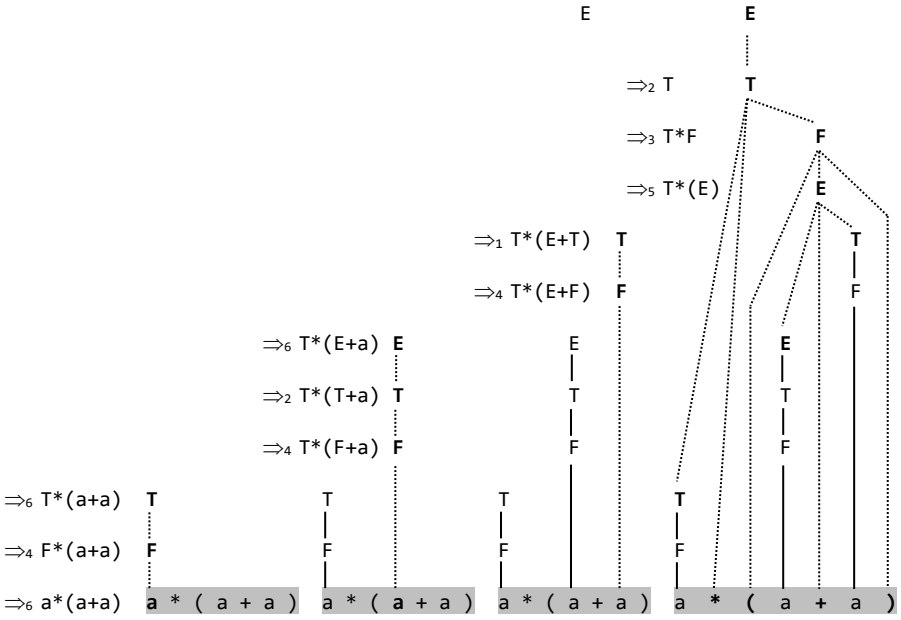
Pišući niz upotrijebljenih produkcija u rečeničnim formama, u obrnutom redosljedu, imamo desno parsiranje 64642641532, pa zaključujemo da je općenito desno parsiranje niza w u gramatici $G=(N,T,P,S)$ zapravo niz produkcija koje su bile upotrijebljene za reduciranje (pretvorbu) rečenice w u startni simbol, u našem primjeru E. Ako to promatramo kao stablo sintaksne analize, zaključujemo da se desno parsiranje rečenice $w=a_1...a_n$ može predočiti kao izvođenje stabla od listova, znakova $a_1,...,a_n$, prema korijenu, startnom simbolu S. Otud i naziv uzlazna sintaksna analiza ili uzlazno parsiranje (eng. "bottom-up" – od dna prema vrhu, uzlazno).

♣ Primjer 1.3

Desno parsiranje rečenice a*(a+a) jezika definiranog gramatikom G_E

(1) E→E+T (2) E→T (3) T→T*F (4) T→F (5) F→(E) (6) F→a

iz primjera 1.1 bilo je 64642641532. Tome odgovara stablo sintaksne analize dobiveno reduciranjem od listova a * (a + a) do korijena označenoga s E:



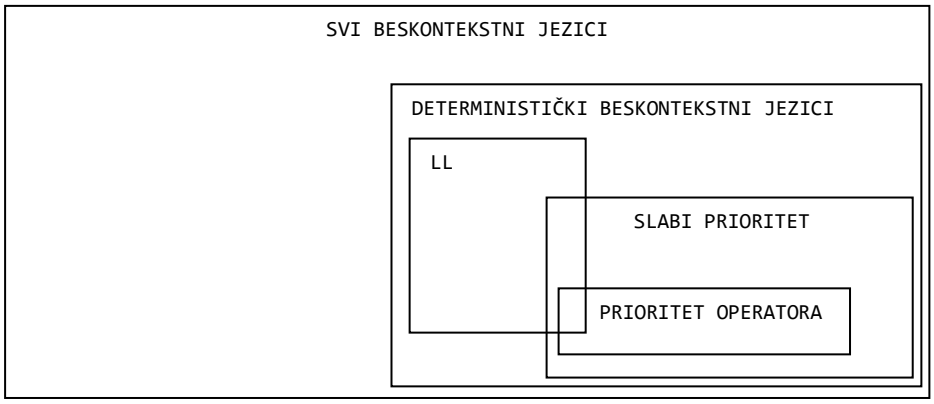
Hijerarhija beskontekstnih jezika

Osim toga što su postupci sintaksne analize (parsiranja) silazni ili uzlazni, općenito se mogu podijeliti na:

- višeprolazne (nedeterminističke) i
- jednoprolazne (deterministične)

Višeprolazni postupci parsiranja, gdje se "višeprolaznost" odnosi na višestruko pretraživanje ulaznog niza, kao što ćemo pokazati, općeniti su, univerzalni postupci parsiranja beskontekstnih jezika, primjenljivi nad cijelom klasom beskontekstnih jezika, ali su najčešće prilično neefikasni. Glavni im je nedostatak vrijeme trajanja (ili broj koraka), posebno ako ulazni niz nije u jeziku.

Jednoprolazni su postupci definirani samo za određene klase beskontekstnih jezika, sl. 1.1. To su jezici (gramatike) tipa LL(k) i LR(k), za koje je moguće konstruirati jednoprolazni postupak sintaksne analize slijeva (za L<u>L</u>) ili zdesna (za L<u>R</u>), koji će raditi deterministički ako im se dopusti da "pogledaju" najviše k ulaznih znakova slijeva nadesno (prvo slovo L u <u>L</u>L i <u>L</u>R to naznačuje) od neke tekuće pozicije, te gramatike s prioritetom operatora za koje je moguće napisati deterministički postupak sintaksne analize upravljan tablicom prioriteta relacije. Osnovni nedostatak jednoprolaznih postupaka sintaksne analize jest ograničena primjenjivost, nad relativno malom klasom beskontekstnih jezika.



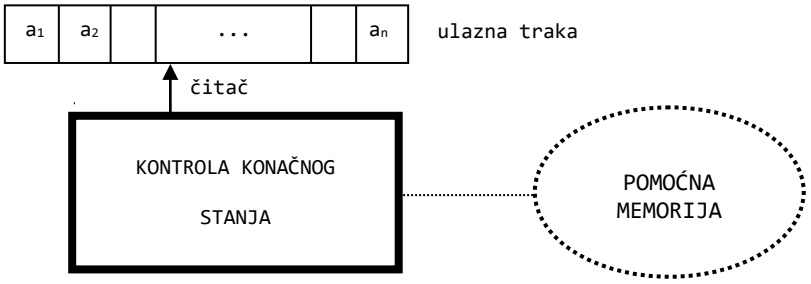
Sl. 1.1 – *Hijerarhija beskontekstnih jezika*.

U poglavlju "Višeprolazno parsiranje" opisane su dvije povratne ("back-track") metode nedeterminističke sintaksne analize slijeva i zdesna koje se, uz određene transformacije gramatika, mogu primijeniti na cijeloj klasi beskontekstnih jezika. U poglavlju "Tablični postupci parsiranja" opisana su još dva općenita postupka sintaksne analize.

Jednoprolazna sintaksna analiza opisana je u poglavljima "LL(k) jezici i sintaksna analiza", te "LR(k) jezici i sintaksna analiza".

1.2 PREPOZNAVANJE

Automati su drugi formalizam za generiranje jezika. Time smo se bavili u prvoj knjizi. No, kao i gramatike, tako su i automati češće u ulozi prepoznavanja rečenica danog jezika. Kažemo da je to prepoznavač jezika. Čine ga ulazna traka, čitač, kontrola konačnog stanja i pomoćna memorija, sl.1.2.



Sl. 1.2 – *Opći model prepoznavača.*

Ulazna traka se može promatrati kao da se sastoji od linearnog niza ćelija koje sadrže po jedan simbol (znak) alfabeta jezika koji se prepoznaje. Početak i kraj mogu biti označeni posebnim znakovima (markerima) koji nisu u alfabetu. Najčešće se marker nalazi na kraju ulaznog niza.

Čitač (ili "ulazna glava") čita jedan znak s pozicije na kojoj se nalazi. Može se pomicati za jedno mjesto lijevo ili desno, ili pak ostati na istom mjestu. Prepoznavač koji nikad ne može pomaknuti čitač ulijevo naziva se *jednosmjerni* prepoznavač.

Uobičajeno je da se ulazna traka promatra kao traka u kojoj se ne mijenja sadržaj (*read-only*). Međutim, postoje prepoznavači (Turingov stroj) u kojima je dopuštena promjena sadržaja ulazne trake, pa se u tom slučaju kaže da je to ulazno-izlazna traka, a ulazna glava je čitač - pisač.

Kontrola konačnog stanja, ili kraće samo kontrola, glavni je dio ili srce prepoznavača. Općenito se sastoji od pet komponenti $(Q, \Sigma, \delta, q_0, F)$, gdje su:

- Q konačni skup stanja
- Σ alfabet
- δ funkcija prijelaza
- q_0 početno stanje, $q_0 \in Q$
- F skup završnih stanja, $F \subseteq Q$

Pomoćna memorija, ili kraće samo memorija, prepoznavača može sadržavati podatke bilo kojeg tipa. Pretpostavlja se da postoji konačni *alfabet memorije*, označen s Γ, i da podaci memorije sadrže znakove alfabeta organiziranih u nekoj strukturi podataka, na primjer mogu biti sa strukturom stoga, $Z_1 Z_2 \ldots Z_n$, gdje je svaki $Z_i \in \Gamma$ i Z_1 je na vrhu.

Za rad s memorijom u većini prepoznavača koriste se dvije funkcije – *funkcija pohranjivanja* i *funkcija dohvata* podataka. Pretpostavlja se da je funkcija za dohvat podataka preslikavanje iz skupa svih mogućih konfiguracija memorije u konačni skup informacijskih simbola, koji mogu biti jednaki simbolima alfabeta memorije.

Na primjer, jedina dohvatljiva informacija memorije sa strukturom stoga jest simbol na njegovom vrhu, pa je funkcija dohvata f definirana kao preslikavanje

$$f\colon \Gamma^{+} \rightarrow \Gamma$$

tako da je

$$f(Z_1Z_2\ldots Z_n) = Z_1$$

Funkcija pohranjivanja jest preslikavanje koje opisuje način promjene sadržaja memorije, tj. preslikava tekući sadržaj memorije i *kontrolni niz* u memoriju. Ako pretpostavimo da memorija ima strukturu stoga, funkcija pohranjivanja g u tom slučaju može biti definirana tako da zamjenjuje simbol Z_1 na vrhu stoga kontrolnim nizom $Y_1\ldots Y_k$, pa se funkcija g može definirati kao

$$g\colon \Gamma^{+} \times \Gamma^{*} \rightarrow \Gamma^{*}$$

tako da je

$$g(Z_1Z_2\ldots Z_n, Y_1\ldots Y_k) = Y_1\ldots Y_kZ_2\ldots Z_n$$

Ako je kontrolni niz prazan, vrh će stoga Z_1 biti zamijenjen s ε, što je ekvivalentno operaciji *pop* ("pucanje", izbacivanje) nad stogom. Poslije toga je simbol Z_2 na vrhu stoga.

Kontrola se može zamisliti kao program koji opisuje ponašanje prepoznavača. Prikazana je konačnim skupom stanja zajedno s preslikavanjem (funkcijom prijelaza) koje opisuje kako se mijenjaju stanja ovisno o tekućem stanju, tekućem simbolu (znaku) ulaznog niza i tekućoj informaciji dohvaćenoj iz pomoćne memorije. Kontrola također odlučuje u kojem će se slučaju ulazna glava pomaknuti i koja će informacija bit pohranjena u memoriju.

Prepoznavač analizira (prepoznaje) ulazni niz čineći niz *pomaka*, mijenjajući svoje konfiguracije, od početnog stanja i početne konfiguracije do dosezanja konačnog stanja i konačne konfiguracije. Konfiguracija prepoznavača jest njegova "slika" koja se općenito sastoji od

(1) stanja kontrole,
(2) sadržaja ulazne trake, zajedno s pozicijom čitača i
(3) sadržaja memorije.

Početna konfiguracija prepoznavača jest ona u kojoj je kontrola u početnom stanju, čitač je pozicioniran na prvi znak ulaznog niza i memorija ima početni sadržaj (može biti prazna ili, na primjer, sadržavati oznaku dna stoga).

Konačna konfiguracija prepoznavača jest ona u kojoj je kontrola u jednom od konačnih stanja, čitač je pozicioniran iza posljednjeg znaka ulaznog niza (na markeru kraja, ako ga ima). Često sadržaj memorije dosezanjem konačne konfiguracije također mora zadovoljavati određene uvjete.

Reći ćemo da prepoznavač prihvaća ulazni niz *w* ako je, krenuvši od početne konfiguracije s *w* na ulaznoj traci, konačnim nizom pomaka dosegnuo jednu od konačnih konfiguracija. Pritom, općenito, prepoznavač može biti deterministički ili nedeterministički, što će ovisiti o definiciji njegove funkcije prijelaza.

Jezik definiran prepoznavačem jest skup prihvatljivih nizova. Prema Chomskyjevoj hijerarhiji za svaku klasu gramatika i jezika koje generiraju postoje ekvivalentni automati koji generiraju odnosno prepoznaju istu klasu jezika. To su konačni, stogovni i linearno ograničeni prepoznavač, te Turingov stroj. Dakle, vrijedi slijedeća klasifikacija prepoznavača ovisno o jeziku kojeg prepoznaju, pa sada možemo reći da je jezik:

(1) desno-linearan (tipa 3) ako i samo ako se može definirati jednosmjernim (determinističkim) konačnim automatom,
(2) beskontekstan (tipa 2) ako i samo ako se može definirati stogovnim (jednosmjerno nedeterminističkim) automatom,
(3) kontekstan (tipa 1) ako i samo ako se može definirati linearno ograničenim (dvosmjerno nedeterminističkim) automatom, i
(4) rekurzivno prebrojiv (tipa 0) ako i samo ako se može definirati Turingovim strojem.

Svaki od navedenih automata (prepoznavača) detaljno ćemo obraditi u narednim poglavljima.

Pitanja i zadaci

1) Koja je razlika između parsiranja i prepoznavanja?

2) Dana je gramatika G s produkcijama:

(1) E → T+E (2) E → T (3) T → F*T (4) T → F (5) F → (E) (6) F → a

Pokažite lijevim, a potom desnim parsiranjem da je niz (a+a)*(a+a) *u jeziku L(G).*

3) Dana je gramatika G s produkcijama:

(1) E → E+E (2) E → E*E (3) E → (E) (4) E → a

Pokažite da je G općenito dvoznačna za lijevo parsiranje, a jednoznačna za desno parsiranje.

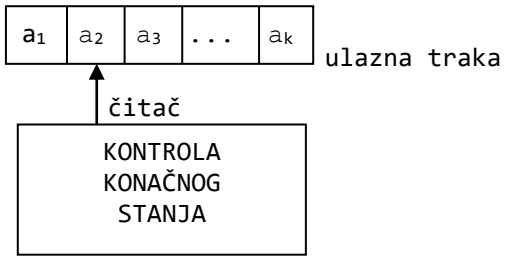
2.
PREPOZNAVANJE REGULARNIH JEZIKA

```
import re

def displaymatch(match):

   if match is None:

     return None

   return '<Match: %r, groups=%r>' % (match.group(), match.groups())


Rim = ['(^M?M?M?)(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$',
       '(^M{0,3})(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$' ]


for x in Rim:

  Re = re.compile (r''+x)

  print displaymatch (Re.match("MMMDCCCLXXXVIII"))


<Match: 'MMMDCCCLXXXVIII', groups=('MMM', 'DCCC', 'LXXX', 'VIII')>
<Match: 'MMMDCCCLXXXVIII', groups=('MMM', 'DCCC', 'LXXX', 'VIII')>
```

Sintaksnu analizu linearnih jezika obradit ćemo u dijelu koji se bavi problemima sintaksne analize beskontekstnih jezika, jer su linearni jezici podskup beskontekstnih jezika. Ovdje ćemo prvo opisati prepoznavač linearnih jezika, a to je (dobro nam poznati) konačni automat. Potom se vraćamo regularnim izrazima i dajemo njihove primjene u analizi (prepoznavanju) teksta.

2.1 KONAČNI PREPOZNAVAČ

Konačni automat u svojstvu prepoznavača, sl. 2.1, sastoji se od ulazne trake, čitača i kontrole konačnog stanja. Prepoznaje desno linearni ili regularni jezik.



Sl. 2.1 – *Opći model konačnog prepoznavača*.

Da bismo shvatili kako konačni prepoznavač prepoznaje rečenice desno linearnog (regularnog) jezika uvodimo definiciju njegove konfiguracije i pomaka.

♦ Konfiguracija prepoznavača

Ako je $\mathcal{M}=(Q, \Sigma, \delta, q_0, F)$ konačni automat, par $(q, w) \in Q \times \Sigma^*$ nazivat ćemo konfiguracija prepoznavača $\mathcal{M}$. (q_0, w) je početna konfiguracija prepoznavača, a (q, ε), $q \in F$, $w \in \Sigma^*$, jest završna konfiguracija prepoznavača.

♦ Pomak prepoznavača

Pomak prepoznavača u $\mathcal{M}$ je prelazak iz jedne u drugu (narednu) konfiguraciju. Prikazan je binarnom relacijom $\vdash$ na konfiguracijama prepoznavača. Ako $\delta(q, a)$ sadrži q', tada vrijedi:

$$(q, aw) \vdash (q', w)$$

za sve $a \in \Sigma$. Ako postoje konfiguracije $C_0, C_1, C_2, \ldots, C_k$, tako da je

$$C_i \vdash C_{i+1} \quad 0 \le i < k$$

tada je

$C_0 \vdash^k C_k$

niz pomaka duljine k. Ako nije bitan broj pomaka, pišemo:

$C \vdash^* C'$

što ima značenje

$C \vdash^k C' \quad k \geq 0$

a ako je postojao najmanje jedan pomak

$C \vdash^+ C' \quad k > 0$

Kaže se da je ulazni niz w prihvatljiv s $\mathcal{M}$ ako

$(q_0, w) \vdash^* (q, \varepsilon)$

za neki $q \in \mathcal{F}$. Jezik definiran s $\mathcal{M}$, $\mathcal{L}(\mathcal{M})$, jest skup nizova prepoznatih s $\mathcal{M}$, tj.

$$\mathcal{L}(\mathcal{M}) = \{ w : \; w \in \Sigma^*, \; (q_0, w) \vdash^* (q, \varepsilon), \; q \in \mathcal{F} \}$$

♣ Primjer 2.1

Tablice prijelaza konačnog automata koji generira jezik prirodnih brojeva djeljivih s 3 dana je s

		0	1	2	3	4	5	6	7	8	9
→	A		B	C	D	B	C	D	B	C	D
	B	B	C	D	B	C	D	B	C	D	B
	C	C	D	B	C	D	B	C	D	B	C
⊗	D	D	B	C	D	B	C	D	B	C	D

Provjerimo je li ulazni niz w=2322543 u jeziku:

$$\begin{aligned}(A, 2322543) &\vdash (C, 322543)\\ &\vdash (C, 22543)\\ &\vdash (B, 2543)\\ &\vdash (D, 543)\\ &\vdash (C, 43)\\ &\vdash (D, 3)\\ &\vdash (D, \varepsilon)\end{aligned}$$

Dakle,

$(A, 2322543) \vdash^* (D, \varepsilon)$

pa zaključujemo da niz 2322543 jest rečenica danog jezika (broj djeljiv s 3).

Slijedi konačni prepoznavač **KP.py** realiziran u Pythonu. Modul fa.py dan je u prilogu knjige.

🖫 KP.py *Prepoznavač regularnih jezika*

```
# -*- coding: cp1250 -*-
from fa  import *
```

```
def Prepoznaj_RJ (w):
  q = q0; i = 1; Err = False
  C = (q, w)
  print '(', q +',', w, ')'
  while not Err and w != '#':
    a = w[0]; j = pos(a, Pr) +1
    if j > 0:
      q = Tp[i][j]
      if q != ' ' :
        i = pos (q, Q) +1; w = w[1:]; C = (q, w)
        if w == '' : w = '#'
        print '(', q +',', w, ')'
      else:
        Err = True
    else:
      Err = True
      print 'Ilegalan znak!'

  print
  if q in F and w == '#':
    print 'Niz je u jeziku.'
  else:
    print 'Niz nije u jeziku!'

  return

Ime_TP, Ok, FA = Ucitaj_FA ()
Q, Pr, Tp, q0, F = FA

if '\t' in Q: Q = Q.replace('\t', '')

print 'Q  =', list (Q)
print 'Pr =', list (Pr)
print 'F  =', list (F)
print '\n', 'TABLICA PRIJELAZA'

if Ok:
  Ispisi_TP (Ime_TP, Tp)
  while True:
    print
    w = Ucitaj_W ( )
    if w == '': break
    Prepoznaj_RJ (w)
```

♣ Primjer 2.2

Prikažimo kako će `Prepoznaj_RJ()` prepoznavati rečenice jezika rimskih brojeva. Najprije definirajmo generator (tablicu prijelaza) jezika rimskih brojeva, `Rimski.TP`:

```
    mdclxvi
  >ABIELMSQ
  xBCIELMSQ
  xCDIELMSQ
  xD IELMSQ
  xEHHFLMSQ
  xF  GLMSQ
  xG   LMSQ
```

```
xH   LMSQ
xI  JLMSQ
xJ  KLMSQ
xK  GLMSQ
xL    NSQ
xM  PPOSQ
xN    OSQ
xO    PSQ
xP     SQ
xQ    VVR
xR      V
xS      T
xT      U
xU      V
xV
```

Prijelazi (alfabet) su dani u prvom redu, poslije dva razmaka. Stanja su u drugom stupcu. U prvom stupcu je oznaka > za početno stanje, x za konačno stanje. Pogledajmo kako će biti analizirani nizovi `mcxi` i `mili`:

```
Q  = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
      'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V']
Pr = ['m', 'd', 'c', 'l', 'x', 'v', 'i']
F  = ['B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
      'P', 'Q', 'R', 'S', 'T', 'U', 'V']

TABLICA PRIJELAZA
...

Upiši ulazni niz: mcxi
( A, mcxi )
( B, cxi )
( E, xi )
( M, i )
( Q, # )

Niz je u jeziku.

Upiši ulazni niz: mili
( A, mili )
( B, ili )
( Q, li )

Niz nije u jeziku!
```

2.2 REGULARNI IZRAZI I PRETRAŽIVANJE

U prvoj smo knjizi regularne izraze rabili kao generatore regularnih skupova. No, njihova je glavna primjena u pretraživanju teksta i pronalaženju i ekstrahiranju podnizova u njemu koji se mogu generirati danim izrazom. Dakle, cijeli tekst nije rečenica jezika, već se promatraju samo pojedini njegovi dijelovi – riječi.

Danas su regularni izrazi dio programa za uređenje i obradu teksta, sistemskih alata, sustava za rad s bazama podataka itd. Međutim, regularni izrazi sve više postaju i važan alat u rješavanju problema obrade prirodnih jezika za što je razvijen veliki broj

funkcija i procedura koje sadrže svi poznatiji jezici za programiranje, kao, na primjer, jezici Java i JScript, Visual Basic i VBScript, JavaScript i ECMAScript, C, C++, C#, elisp, Perl, Python, Tcl, Ruby, PHP itd.

Za nas su važne primjene regularnih izraza u problemima teorije formalnih jezika. Tu je njihovo izučavanje dvojako: pripadaju teoriji formalnih jezika, a uz njihovu primjenu pojednostavljuje se pisanje algoritama za rješavanje problema leksičke analize i prevođenja.

Napisane su mnoge knjige o primjeni regularnih izraza, što opet ukazuje na njihovu sve veću važnost. Mi ćemo ovdje opisati funkcije i procedure regularnih izraza u Pythonu koji će nam omogućiti potpuniji uvid u mogućnosti njihove primjene u rješavanju problema teorije formalnih jezika, ali i šire, u pretraživanju teksta.

Često se regularni izrazi koji se koriste u analizi (prepoznavanju) nizova znakova u nekom ulaznom nizu (tekstu) nazivaju uzorak (ili *pattern*). Realizacija programa za pretraživanje ulaznog niza upravljana regularnim izrazom (RE) naziva se motor (*engine*) regularnih izraza. U biti je to konačni automat, deterministički (DFA) ili nedeterministički (NFA), odnosno, konačni prepoznavač.

P R I M J E N E

Povijest regularnih izraza dio je rane povijesti formalnih jezika i dugo su vremena bili predmetom teorijskih istraživanja vezanih za regularne skupove (jezike). Matematičar Stephen Kleene je 1950-ih opisao ove modele koristeći matematičku notaciju zvanu *regularni skupovi*. Ken Thompson je ugradio ovu notaciju u uređivač *QED*, a potom i u Unixovom editoru *ed*, što je s vremenom dovelo do uporabe regularnih izraza u *grep*-u. Otad se regularni izrazi naširoko koriste u Unixu i Unixoidnim pomoćnim programima kao što su *expr*, *awk*, *Emacs*, *vi*, *lex* i *Perl*.

Rekli smo da su regularni izrazi bili popularni u početku razvoja teorije formalnih jezika, pedesetih i početkom šezdesetih godina prošloga stoljeća. Razvojem teorije gramatika i automata, šezdesetih godina prošloga stoljeća, regularni su izrazi bili malo "zaboravljeni". Tako je bilo dvadesetak godina, do pojave interneta.

Sada svi jezici (Phyton, PHP, Perl, Java Script, ...) imaju procedure i funkcije za rad s regularnim skupovima, odnosno, regularnim izrazima. Značenje regularnih izraza je sparivanje određenih nizova (riječi), u skladu s određenim pravilima. Koriste ih mnogi programi za uređivenje teksta, programi za pretragu i manipuliranje nizovima. Može se reći da danas "regularni izraz", koji se još naziva "uzorak" (engl. *pattern*), rabi za opis (označivanje) skupa nizova znakova bez davanja njegova precizna značenja. Na primjer, skup koji sadrži četiri niza `Mirko`, `Miro`, `Marko` i `Maro` može se opisati regularnim izrazom ili uzorkom `M(i|a)r(k?)o`. Ovdje su "`|`" i "`?`" metaznakovi koji imaju značenje "`+`" i "`+`".

S obzirom na to da smo Python izabrali kao osnovni jezik za opis algoritama u našim knjigama, ovdje ćemo kao primjer primjene regularnih izraza opisati regularne izraze u Pythonu. S druge strane, sintaksa i semantika regularnih izraza Pythona ne razlikuje se bitno od sintakse i semantike regularnih izraza u drugim danas rabljenim jezicima za programiranje.

REGULARNI IZRAZI I PYTHON

Rad s regularnim izrazima (nazvanim *REs*, *regular expressions*, *regexes* ili *regex patterns*) u Pythonu omogućen je uporabom modula `re`. Iako u uputama Pythona piše da je to „mali jezik", može se slobodno reći da je to jedan od boljih motora regularnih izraza, s velikim brojem funkcija i procedura. Koristeći taj „mali jezik" moguće je definirati pravila (regularni izraz) koja generiraju skup nizova (riječi) koje želimo prepoznati ("spariti") u danom tekstu. Mogu to biti, na primjer, neke riječi hrvatskoga teksta, e-mail adresa, telefonski broj ili bilo što slično. U naprednoj uporabi regularnih izraza moguće je modificirati neki niz ili ga podijeliti na podnizove, te uvesti kontekstne aspekte u prepoznavanju rečenica kontekstnog jezika i jezika bez ograničenja.

Postoje mnogi problemi koji se mogu riješiti uz pomoć regularnih izraza i Pythona. No, treba napomenuti da bez obzira na to, rješenje može biti složeno pa je tada možda bolje problem riješiti tradicionalnim postupcima, bez obzira što će izvršavanje programa biti duže.

Mi ćemo se u ovom kratkom prikazu sintakse i semantike regularnih izraza Pythona ograničiti na onaj dio koji je bliži teoriji formalnih jezika. Analizirat ćemo nizove znakova ograničene duljine i odgovoriti jesu li ili nisu rečenica regularnog jezika kojeg dani regularni izraz označuje (generira).

SINTAKSA

Kao što znamo, kompozicija *xy* dvaju regularnih izraza *x* i *y* također je regularni izraz. Općenito, ako *p* sparuje *x*, a *q* sparuje *y*, niz *pq* sparuje *xy*. To će vrijediti uvijek, bez obzira na sadržaj podizraza *x* i *y*. Dakle, složeni izrazi mogu biti izgrađeni kompozicijom jednostavnih izraza, što će nam olakšati postupak izgradnje (definiranja) regularnog izraza za prepoznavanje uzoraka u nekom tekstu.

Počnimo opisom najjednostavnijih regularnih izraza: sparivanjem znakova. Mnogi znakovi jednostavno sparuju sami sebe. Na primjer, regularni izraz `123` sparit će egzaktno niz `123`. Ako niz sadrži slova, velika ili mala, sparivanje će također biti jednoznačno. Na primjer, regularni izraz `Python` sparit će niz `Python`, ali neće `PYTHON` niti `python`. Ako želimo da spari i te nizove koristit ćemo opciju (*case-insensitive*) u kojoj je značenje velikih i malih slova engleskog alfabeta jednako.

Postoji izuzetak u primjeni tog pravila. Neki znakovi `ASCII` skupa znakova imaju posebno značenje i ne sparuju sami sebe. Nazivamo ih *meta-simboli* . To su:

```
. ^ $ * + ? { } [ ] \ | ( )
```

Većinu ovih meta-simbola koristili smo ranije i poznato nam je njihovo značenje. Na primjer, (i) imaju značenje kao i u osnovnoj definiciji ili pravilima pisanja regularnih izraza. Označuju podizraz („blok“) regularnog izraza.

Ako neterminale označimo velikim kosim slovima, gdje je *R* startni simbol, općenito se pisanje regularnih izraza može prikazati sljedećim skupom produkcija:

R → *S*| *B*| *Z*| *Y*| [*N*]| [-*N*]| [*N*-]| (*R*)| *RK*| *RA*| *PR*| *RR*

S → a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|*SS*

B → 0|1|2|3|4|5|6|7|8|9|*BB*

Z → -| _| "| #| $| %| =| @| .

Y → [|]| \| ^| $| .| || ?| *| +| (|)| {| }| d| n| w| E| Q| W

N → *S*-*S*| *B*-*B*| *NN*| *X*

X → *S*| *B*| *Z*| *Y*

K → *| ?| +| $

A → {*B*}| {*B*,*B*}| {,*B*}| {*B*,}

P → ^| ?

Ako je ASCII skup znakova, u sljedećoj je tablici dano značenje meta-simbola u pisanju regularnih izraza:

Metaznak	Značenje	Primjeri	Sparuje
.	Bilo koji znak iz ASCII osim oznake nove linije. Unutar [ ] ima svoje uobičajeno značenje.	(a.cd) (a..d)	abcd axcd a2cd aXcd abcd axyd a6xd aADd
^	“caret” – znak za umetanje, sparuje početak linije (bilo koje linije, u višelinijskom načinu rada)		
$	Sparuje kraj linije (bilo koje linije, u višelinijskom načinu rada)		
(*R*)	“označeni” podizraz (ili “blok”)	(a)	{a}
|	Alternativa (“ili”)	0|1|2|99 Mous(tak|kour)i	{0, 1, 2, 99} {Moustaki, Mouskouri}
*R**	Ponavljanje izraza *R* 0, 1 ili bilo koji veći broj puta	(a)* ili a* (a|b)* go*gle	{ε,a,aa,aaa,...} {ε,a,b,aa,ab,ba,bb,...} {ggle,gogle,google,... }

Metaznak	Značenje	Primjeri	Sparuje
R?	Izostavljanje prethodnog izraza ili pojavljivanje jedanput.	colou?r M(i|a)r(k?)o ((great)?grand)? ((fa|mo)ther)	{color,colour} {Miro, Mirko, Maro, Marko} {father,mother,grand father,grand mother,great grand father,great grand mother}
R+	Ponavljanje prethodnog izraza 1 ili bilo koji veći broj puta	go+gle	{gogle,google,gooogle,...}
[$a_1a_2...a_n$]	=(a_1|a_2|...|a_n)	[abc]	{a,b,c}
[a_i-a_j]	=(a_i| a_{i+1}| ...| a_j) Skup znakova od a_i do a_j prema ASCII uređenju. Ako je redni broj a_i veći od rednog broja a_j, skup je prazan.	[A-F] (-|+)*[0-9]+.[0-9]+ ([0-9]| [1-9][0-9]| 1[0-9][0-9]| 2[0-4][0-9]| 25[0-5])	{A,B,C,D,E,F} Realni brojevi u Pascalu {0,1,2,...9, 10,...,99, 100,...,199, 200,...,249, 250,...,255}
[$a_1...a_nb_i$-b_j]	=(a_1|a_2|...|a_n|b_i|...|b_j)	[abcq-z] [a-fu-z] [-abc]=[abc-] [][0123]	{a,b,c,q,r,s,t,u,v,w,x,y,z} {a,b,c,d,e,f,u,v,w,x,y,z} {-,a,b,c} {],[,0,1,2,3}
[^...]	Sparuje jedan znak koji nije sadržan unutar uglatih zagrada.	[^abc] [^a-z]	Bilo koji skup \ {a,b,c} Bilo koji skup bez malih slova
R{*m*}	Sparuje točno *m* znakova izraza *R*.	(0|1){2}	{00,01,10,11}
R{*m*,*n*}	Sparuje najmanje *m* i najviše *n* znakova izraza *R*.	A{1,8}(.|A{0,3})? gdje je A=[a-zA-Z0-9-_"#$%&= +^'@]	Imena DOS datoteka
R{*m*,*n*}?	Nepohlepna verzija kvalifikatora {*m*,*n*}. Sparuje prvih *m* znakova.	a{2,5} a{2,5}?	{aa,aaa,aaaa,aaaaa} {aa}
R{,*n*}	Sparuje 0 do *n* znakova.	1{,3}2	
R{*m*,}	Sparuje najmanje *m* znakova.	1{3,}2 = 1111*2	{2,12,112,1112} {1112,11112,111112,...}
\d \D	=[0-9] (skup brojki) Skup znakova bez brojki.		{0,1,2,3,4,5,6,7,8,9}
n	*n* je znamenka od 1 do 9. Sparuje *n*-ti spareni označeni podizraz. Ovaj konstrukt je **neregularan** i nije prihvaćen u proširenoj sintaksi regularnih izraza.		

Metaznak	Značenje	Primjeri	Sparuje
\A	Sparuje samo početak (prefiks) niza.		
\b	Sparuje razmak (blank) samo na početku ili kraju riječi.		
\B	Sparuje razmak ako nije na početku ili kraju riječi.		
\t \n \r	- tab (oznaka tabulatora) -newline (oznaka za novu liniju) -return (oznaka za povratak)		
\s	Sparuje nevidljive znakove: \b, \t, \n i \r.		
\S	Sparuje bilo koji znak osim nevidljivih, \s.		
\w	=[a-zA-Z0-9_] (alfanumerički znakovi)		{0,...,9,_,A,...,Z,a,...,z}
\W	Skup svih znakova bez alfanumeričkih znakova ([a-zA-Z0-9_])		Sve znakove osim alfanumeričkih
\Z	Sparuje samo kraj (sufiks) niza.		
\xFF	Znak čiji je ASCII kod jednak FF, gdje je FF heksadecimalni broj.	\xA9	Znak © ako je kodna stranica Latin-1

"POHLEPNO" I "NEPOHLEPNO" SPARIVANJE

Kaže se da su kvantifikatori `'*'`, `'+'` i `'?'` "pohlepni" jer sparuju što je moguće više teksta. To ponekad može dati neočekivan rezultat. Na primjer, ako imamo RE `'<.*>'` sparit će `'<H1>title</H1>'`, a ne podniz `'<H1>'`. Ako dodamo `'?'` iza kvantifikatora `'*'` ili `'+'`, dobit ćemo "nepohlepni" izraz koji će spariti prvi podniz između znakova `'<'` i `'>'`. Dakle, RE `'<.*?>'` u ovom primjeru bi spario samo `'<H1>'`.

U sljedećem primjeru, da bi se dohvatilo `'title'` u nizu `'<H1>title</H1>'`, treba koristiti uzorak `'<.+?>(.+?)<.+?>'` ili `'<[^>]+>([^<]+)<'`.

Općenito ćemo imati "nepohlepno" sparivanje ako iza svakog od kvantifikatora `*`, `+` i `?` dodamo još jedan meta-simbol `?`, tj . `*?`, `+?` i `??`.

PROŠIRENA SINTAKSA REGULARNIH IZRAZA

Python ima još neke mogućnosti u pisanju regularnih izraza u proširenoj notaciji, s posebnim značenjem. Proširena se notacija piše prema pravilu:

```
(?...)
```

Prvi znak poslije upitnika određuje značenje koje vrijedi unutar grupe (podizraza omeđenog zagradama). Evo pregleda svih znakova i njihova značenja:

Izraz	Značenje	Primjeri	Sparuje
(?*O*)	Uključenje jedne ili više opcija *O*→a|i|L|m|s|u|x. Za naše primjene važna je opcija i sa značenjem da velika i mala slova imaju jednako značenje	P(?i)ython	{Python, PYthon, PYTHON, ...}
(?-*O*)	Isključenje navedenih opcija.		
(?:*R*)	Grupiranje izraza bez pamćenja sparenog teksta.		
(?*O*:*R*)	Doseg uključenih opcija *O* odnosi se samo na *R* (do zatvorene zagrade).	(?i:don)	{don,doN,dOn,dON,Don,DoN,DOn, DON}
(?-*O*:*R*)	Doseg isključenih opcija *O* odnosi se samo na *R* (do zatvorene zagrade).	P(?-i:ython)	Python
(?P<*ime*>...)			
(?P=*ime*)			
(?=*R*)	Specificira poziciju koristeći uzorak. Nema doseg.		
(?!*R*)	Specificira poziciju koristeći negaciju uzorka. Nema doseg.		
(?>*R*)	Sparuje neovisan uzorak, bez povratka.		
(?#...)	Komentar.	Ag(?#zlato)	Ag

Dodatna objašnjenja:

(?:...)
Sparuje bilo koji regularni izraz unutar zagrada, ali spareni podniz ne smije biti dohvatljiv

(?#...)
Komentar. Sadržaj unutar zagrada se ignorira.

(?=...)
Sparuje niz prethodnog izraza samo ako ga niz **...** slijedi.

```
'Georges (?=Moustaki)'
```

'Georges ' će biti sparen samo ako ga 'Moustaki' slijedi.

(?!...)
Sparuje niz prethodnog izraza samo ako ga niz **...** ne slijedi.

```
'Georges (?!Moustaki)'
```

'Georges ' će biti sparen samo ako ga 'Moustaki' ne slijedi.

MODUL re.py

Motor regularnih izraza Pythona realiziran je u modulu re.py koji je dio standardne biblioteke Pythonovih modula (nalazi se u folderu LIB). Podržava 8-bitni skup US

ASCII znakova. Ograničit ćemo se na primjenu regularnih izraza u teoriji formalnih izraza i dati solidne osnove da ih možete primijeniti i šire. Osim toga, vidjet ćemo da je jezik regularnih izraza relativno malen i ograničen, tako da nije moguće rješavati sve probleme obrade nizova njihovom uporabom. Stalno treba imati na umu veliki broj funkcija i procedura Pythona za rad s nizovima. Za potpunije značenje i mogućnosti primjene regularnih izraza informirajte se u uputama uz Python, posebno u dijelu „Regular Expression HOWTO“.

Regularni izrazi se definiraju kao nizovne vrijednosti, tj. nizovi znakova koji započinju i završavaju s `'` ili s `"` (tip `str`). Evo nekoliko primjera regularnih izraza:

1) Identifikator u Pythonu i još nekim jezicima za programiranje (npr. u Pascalu)

```
'(_|[a-zA-Z])([a-zA-Z0-9_])*'
```

ili

2) Telefonski brojevi u fiksnoj mreži RH:

```
'0(1|2(0|1|2|3)|3(1|2|3|4|5)|4(0|2|3|4|7|8|9)|5(1|2|3))$'
```

3) Rimski brojevi:

```
'(^M?M?M?)(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

ili

```
'(^M{0,3})(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
```

Modul **`re.py`** sadrži procedure za rad s regularnim izrazima.

Meta-simboli nisu aktivni unutar klasa. Na primjer, `[ahn$]` će spariti bilo koji znak '`a`', '`h`', '`n`' ili '`$`'. Dakle, '`$`' ovdje predstavlja samog sebe.

Može se reći da je uporaba meta-simbola `\` („backslash“) možda najvažnija u Pythonu. Osim u nizovima znakova Pythona, iza simbola `\` mogu slijediti različiti znakovi koji imaju posebna značenja. Također se koristi ako neki od meta-simbola mora spariti samog sebe. Na primjer, `[` ili `\` će spariti sami sebe ako se napiše `\[` ili `\\`.

Modul re.py nalazi se u biblioteci Pythona, pa će aplikacije za rad s regularnim izrazima imati:

```
import re
```

UZORAK

Uzorak (“pattern”) se u Pythonu piše prema pravilu:

```
<uzorak>        → r'<regularni izraz>' | r'' +<nizovni izraz>
<nizovni izraz> → '<regularni izraz>' | <nizovna varijabla> |
                  <nizovni izraz> + <nizovni izraz>
```

gdje je `<nizovna varijabla>` varijabla Pythona tipa `str` koja sadrži niz znakova koji predstavlja regularni izraz. Evo nekoliko primjera pravilno napisanih uzoraka:

```
r'(P|p)ython'
C = '[a-z0-9.]+';  email_com = "(?i)" +C +"@" +C +".com$"
r'' +email_com
```

RIMSKI BROJEVI

Na kraju ovoga poglavlja evo programa koji prepoznaje rimske brojeve. Uzorak (regularni izraz) za prepoznavanje rimskih brojeva opisali smo na dva načina (lista `Rim`). Rezultat, grupiranje po tisućama, stoticama, deseticama i jedinicama, je isti.

```
import re

def displaymatch(match):
    if match is None:
      return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())

Rim = ['(^M?M?M?)(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$',
       '(^M{0,3})(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$' ]

for x in Rim:
  Re = re.compile (r''+x)
  print displaymatch (Re.match("MMMDCCCLXXXVIII"))

<Match: 'MMMDCCCLXXXVIII', groups=('MMM', 'DCCC', 'LXXX', 'VIII')>
<Match: 'MMMDCCCLXXXVIII', groups=('MMM', 'DCCC', 'LXXX', 'VIII')>
```

Pitanja i zadaci

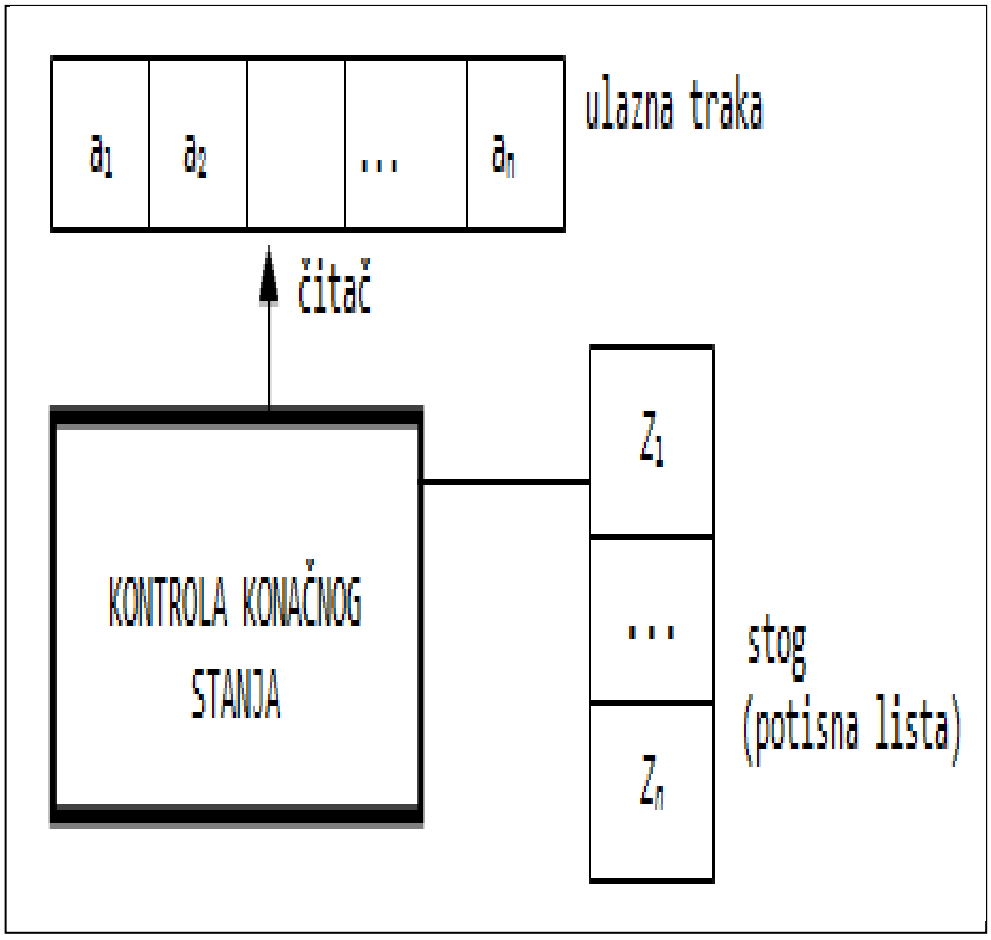
1) Definirajte konačni prepoznavač sljedećih jezika:

2) Godina je prijestupna ako je djeljiva s 4 i nije djeljiva sa 100 ili ako je djeljiva s 400. Na primjer, 1900. godina nije bila prijestupna, a 2000. godina je bila prijestupna. Definirajte konačni prepoznavač koji će prepoznati je li ulazni niz iz intervala od 1600 do 9996 prijestupna godina.

3) Napišite program u Pythonu s uzorkom koji će prepoznati je li ulazni niz registarska oznaka vozila u Bosni i Hercegovini. Ako je s veliko slovo A, E, J, K, M, O *ili* T, *na Wikipediji se može naći da su tri vrste registarskih oznaka u BiH:*

   a) Stare registarske oznake, od 001-s-001 *do* 999-s-999, *na primjer* 234-J-333
   b) Nove registarske oznake, od s01-s-001 *do* s99-s-999, *na primjer* A77-K-007
   c) Taksi vozila od TA-000001 *do* TA-999999, *na primjer* TA-222543

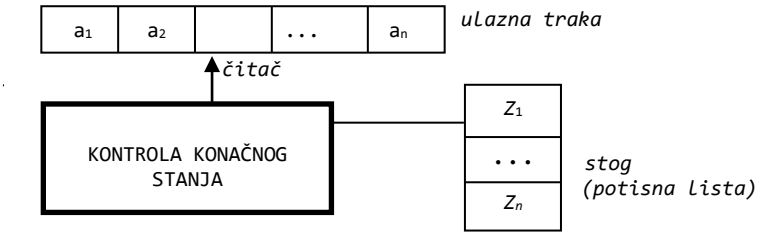
3.
PREPOZNAVAČI BESKONTEKSTNIH JEZIKA



Znamo da su stogovni (potisni) automati generatori beskontekstnih jezika. No, kao i u slučaju gramatika, njihova upotreba je češća u analizi ili prepoznavanju je li ulazni niz rečenica danog beskontekstnog jezika. Tada je automat u ulozi prepoznavača jezika kojeg generira. S obzirom na to da je ovdje riječ o beskontekstnim jezicima koji mogu biti generirani stogovnim automatima, njihovo prepoznavanje također je uz pomoć stogovnih automata koje sada nazivamo "stogovnim prepoznavačima".

3.1 STOGOVNI PREPOZNAVAČ

Automat koji ima pomoćnu memoriju sa strukturom stoga (stack) naziva se stogovni automat, odnosno stogovni prepoznavač ako je bez izlazne trake, sl. 3.1.



Sl. 3.1 – *Stogovni prepoznavač.*

Stogovni automat definirali smo kao uređenu sedmorku $\mathcal{R}=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, gdje su:

- Q konačan skup stanja (kontrole konačnog stanja)
- Σ ulazni alfabet
- Γ alfabet znakova stoga (potisne liste)
- δ funkcija prijelaza, definirana kao

$$\delta: Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma \rightarrow Q\times\Gamma^*$$

- q_0 početno stanje, $q_0\in Q$
- Z_0 početni znak potisne liste, $Z_0\in\Gamma$
- F skup završnih stanja, $F\subseteq Q$

♦ Konfiguracija stogovnog prepoznavača

Konfiguracija stogovnog prepoznavača $\mathcal{R}$ jest (q, w, α) iz $Q\times\Sigma^*\times\Gamma^*$, gdje su:

- q tekuće stanje
- w preostali dio ulaznog niza
- α niz znakova koji predstavlja sadržaj potisne liste; vrh je prvi znak niza

Početna konfiguracija je (q_0, w, Z_0), a završna konfiguracija (q, ε, α), $q\in F$, $\alpha\in\Gamma^*$.

♦ Pomak stogovnog prepoznavača

Pomak stogovnog prepoznavača $\mathcal{R}$ jest relacija $\vdash_{\mathcal{R}}$ (ili samo $\vdash$ ako se $\mathcal{R}$ podrazumijeva):

$$(q,aw,Z\alpha) \vdash (q',w,\gamma\alpha)$$

ako $\delta(q,a,Z)$ sadrži (q',γ) za $q\in Q$, $a\in\Sigma\cup\{\varepsilon\}$, $w\in\Sigma^*$, $Z\in\Gamma$. Kaže se da je ulazni niz w prihvatljiv s $\mathcal{R}$ ako

$$(q_0, w, Z_0) \vdash^*(q, \varepsilon, \alpha)$$

Jezik definiran s $\mathcal{R}$, označen s $\mathcal{L}(\mathcal{R})$, jest skup nizova w prihvatljivih s $\mathcal{R}$. To je općenito beskontekstni jezik:

$$\mathcal{L}(\mathcal{R})=\{w:\ w\in\Sigma^*\ \wedge\ (q_0, w, Z_0) \vdash^*(q, \varepsilon, \alpha),\ q\in\mathcal{F},\ \alpha\in\Gamma^*\}$$

Općenito je stogovni automat nedeterministički. U tom slučaju, da bi automat prihvatio ulazni niz, mora proći kroz sve moguće prijelaze dok ne dosegne završno stanje.

Stogovni smo prepoznavač realizirali u Pythonu (program **SP.py** dan na kraju poglavlja).

♣ Primjer 3.1

Stogovni automat koji prepoznaje beskontekstni jezik $\mathcal{L}=\{0^n1^n:\ n>0\}$ jest

$$\mathcal{P} = (\{q_0,q_1,q_2\},\{0,1\},\{\$,0\},\delta,q_0,\$,\{q_0\})$$

gdje je funkcija prijelaza δ definirana sa:

$$\delta(q_0,0,\$) = \{(q_1,0\$)\} \quad \delta(q_1,0,0) = \{(q_1,00)\}$$
$$\delta(q_1,1,0) = \{(q_2,\varepsilon)\} \quad \delta(q_2,1,0) = \{(q_2,\varepsilon)\}$$
$$\delta(q_2,\varepsilon,\$) = \{(q_0,\varepsilon)\}$$

Provjerimo uz pomoć programa **SP.py**, u kojem su stanja q_0=0, q_1=1 i q_2=2, je li ulazni niz w=000111 prihvatljiv:

```
(0, '000111', '$')
  |-- (1, '00111', '0$')
  |-- (1, '0111', '00$')
  |-- (1, '111', '000$')
  |-- (2, '11', '00$')
  |-- (2, '1', '0$')
  |-- (2, '', '$')
  |-- (0, '', '')
  |-- accept
```

Dakle, niz 000111 je prihvatljiv.

Općenito je stogovni automat nedeterministički. U tom slučaju, da bi automat prihvatio ulazni niz, mora proći kroz sve moguće prijelaze dok ne dosegne završno stanje.

♣ Primjer 3.2

Konstruirajmo automat koji će prepoznavati beskontekstni jezik $\mathcal{L}=\{ww^R: w\in\{a,b\}^+\}$. To će biti

$$\mathcal{R}=(\{q_0,q_1,q_2\}\{a,b\}\{Z,a,b\},\delta,q_0,Z,\{q_2\})$$

gdje je:

$(1)\delta(q_0,a,Z)=\{(q_0,aZ)\}$ $(4)\delta(q_0,a,b)=\{(q_0,ab)\}$ $(7)\delta(q_1,a,a)=\{(q_1,\varepsilon)\}$
$(2)\delta(q_0,b,Z)=\{(q_0,bZ)\}$ $(5)\delta(q_0,b,a)=\{(q_0,ba)\}$ $(8)\delta(q_1,b,b)=\{(q_1,\varepsilon)\}$
$(3)\delta(q_0,a,a)=\{(q_0,aa),(q_1,\varepsilon)\}$ $(6)\delta(q_0,b,b)=\{(q_0,bb),(q_1,\varepsilon)\}$ $(9)\delta(q_1,\varepsilon,Z)=\{(q_2,\varepsilon)\}$

$\mathcal{R}$ će inicijalno prenijeti dio svog ulaza u stog, prema pravilima (1), (2), (4) i (5) i prvim alternativama pravila (3) i (6). Nedeterminizam je u pravilima (3) i (6). Na primjer, ako je ulazni niz abba, $\mathcal{R}$ može načiniti sljedeće nizove premještanja:

$(1)(q_0,abba,Z)\vdash(q_0,bba,aZ)\vdash(q_0,ba,baZ)\vdash(q_0,a,bbaZ)\vdash(q_0,\varepsilon,abbaZ)$
$(2)(q_0,abba,Z)\vdash(q_0,bba,aZ)\vdash(q_0,ba,baZ)\vdash(q_1,a,aZ)\vdash(q_1,\varepsilon,Z)\vdash(q_2,\varepsilon,\varepsilon)$

Budući da niz premještanja (2) okončava u završnoj konfiguraciji, odnosno u završnom stanju q_2, $\mathcal{R}$ prihvaća ulazni niz abba.

3.2 PREPOZNAVAČ S PRAZNIM STOGOM

U prvoj knjizi, [16], u poglavlju 6, definiran je stogovni automat s praznim stogom i pokazana (propozicija 6.1) njegova ekvivalentnost s beskontekstnom gramatikom. Prisjetimo se, ako je $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ beskontekstna gramatika, nedeterministički stogovni automat s praznim stogom jest uređena šestorka $\mathcal{R}=(\{q_0\},\mathcal{T},\mathcal{T}\cup\mathcal{N},\delta,q_0,\mathcal{S})$, gdje je funkcija prijelaza δ definirana kao

$$\delta(q_0,\varepsilon,A)=\{(q_0,\alpha):(A\to\alpha)\in\mathcal{P},A\in\mathcal{N},\alpha\in(\mathcal{T}\cup\mathcal{N})^*\}$$
$$\delta(q_0,a,a)=\{(q_0,\varepsilon):a\in\mathcal{T}\}$$

Vidimo da stogovni automat s praznim stogom ima samo jedno stanje, q_0, nema konačno stanje (odnosno ima skup konačnih stanja, ali prazan, pa smo ga zbog toga izostavili) i $\mathcal{S}$ je početni znak stoga. Kaže se da je niz $w\in\Sigma^*$ <u>prihvatljiv</u> automatom s praznim stogom ako je

$$(q_0,w,\mathcal{S})\ \vdash^+\ (q_0,\varepsilon,\varepsilon)$$

Skup prihvatljivih nizova označujemo s $\mathcal{L}_e(\mathcal{R})$.

♣ Primjer 3.3

Stogovni automat s praznim stogom ekvivalentan gramatici $\mathcal{G}_E$

(1) $E\to E+T$ (2) $E\to T$ (3) $T\to T*F$ (4) $T\to F$ (5) $F\to(E)$ (6) $F\to a$

bit će $\mathcal{R}=(\{q\},\{a,+,*,(,)\},\{E,T,F,a,+,*,(,)\},\delta,q,E)$, gdje su:

(1) $\delta(q,\varepsilon,E)=\{(q,E+T),(q,T)\}$ (5) $\delta(q,+,+)=\{(q,\varepsilon)\}$
(2) $\delta(q,\varepsilon,T)=\{(q,T*F),(q,F)\}$ (6) $\delta(q,*,*)=\{(q,\varepsilon)\}$
(3) $\delta(q,\varepsilon,F)=\{(q,(E)),(q,a)\}$ (7) $\delta(q,(,()=\{(q,\varepsilon)\}$
(4) $\delta(q,a,a)=\{(q,\varepsilon)\}$ (8) $\delta(q,),))=\{(q,\varepsilon)\}$

Na primjer, za prepoznavanje ulaznog niza a*(a+a) $\mathcal{R}$ će načiniti sljedeći niz pomaka:

```
(q,a*(a+a),E)
   ⊢ (q, a*(a+a), T    ) ⊢ (q, a*(a+a), T*F  )   ⊢ (q, a*(a+a), F*F  )
   ⊢ (q, a*(a+a), a*F  ) ⊢ (q, *(a+a),  *F   )   ⊢ (q, (a+a),   F    )
   ⊢ (q, (a+a),   (E)  ) ⊢ (q, a+a),    E)   )   ⊢ (q, a+a),    E+T) )
   ⊢ (q, a+a),    T+T) ) ⊢ (q, a+a),    F+T) )   ⊢ (q, a+a),    a+T) )
   ⊢ (q, +a),     +T)  ) ⊢ (q, a),      T)   )   ⊢ (q, a),      F)   )
   ⊢ (q, a),      a)   ) ⊢ (q, ),       )    )   ⊢ (q, ε,       ε    )
```

U prethodnom je primjeru niz pomaka koje je načinio stogovni automat s praznim stogom prihvaćajući ulazni niz ekvivalentan krajnjem izvođenju slijeva rečenice a*(a+a) počevši s početnim simbolom E:

$$E \underset{2}{\Rightarrow} T \underset{3}{\Rightarrow} T*F \underset{4}{\Rightarrow} F*F \underset{6}{\Rightarrow} a*F \underset{5}{\Rightarrow} a*(E) \underset{1}{\Rightarrow} a*(E+T) \underset{2}{\Rightarrow} a*(T+T) \underset{4}{\Rightarrow} a*(F+T) \underset{6}{\Rightarrow} a*(a+T) \underset{4}{\Rightarrow} a*(a+F) \underset{6}{\Rightarrow} a*(a+a)$$

Znamo da se takva vrsta analize naziva "silazna (top-down) sintaksna analiza".

3.3 PROŠIRENI STOGOVNI AUTOMAT

Osim automata s praznim stogom definira se i prošireni stogovni automat.

♦ Prošireni stogovni automat

Prošireni stogovni automat je uređena sedmorka $\mathcal{R}=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, gdje je funkcija prijelaza definirana kao $\delta: Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma^* \rightarrow Q\times\Gamma^*$. Značenje ostalih simbola je nepromijenjeno. Kaže se da je niz $w\in\Sigma^*$ prihvatljiv proširenim stogovnim automatom $(q_0,w,Z_0)^+\vdash(q,\varepsilon,\varepsilon)$ za neki $q\in Q$. Skup prihvatljivih nizova označujemo s $\mathcal{L}_e(\mathcal{R})$.

♣ Primjer 3.4

Definirajmo prošireni stogovni automat jezika $\mathcal{L}=\{ww^R:\ w\in\{a,b\}^*\}$. To će biti

$$\mathcal{R}=(\{p,q\}\{a,b\}\{a,b,S,Z\},\delta,q,Z,\{p\})$$

gdje je funkcija prijelaza:

```
(1) δ(q,a,ε)   = {(q,a)}   (2) δ(q,b,ε)   = {(q,b)}   (3) δ(q,ε,ε)  = {(q,S)}
(4) δ(q,ε,aSa) = {(q,S)}   (5) δ(q,ε,bSb) = {(q,S)}   (6) δ(q,ε,SZ) = {(p,ε)}
```

Za ulazni niz aabbaa $\mathcal{R}$ će načiniti sljedeći niz premještanja:

```
(q,aabbaa,Z) ⊢(q,abbaa, aZ)      ⊢(q,bbaa,  aaZ)     ⊢(q,baa,  baaZ)
             ⊢(q,baa,   SbaaZ)   ⊢(q,aa,    bSbaaZ)  ⊢(q,aa,   SaaZ)
             ⊢(q,a,     aSaaZ)   ⊢(q,a,     SaZ)     ⊢(q,ε,    aSaZ)
             ⊢(q,ε,     SZ)      ⊢(p,ε,     ε)
```

Najprije se prednji dio ulaznog niza premješta u stog. Potom se oznaka sredine, S, postavlja na vrh stoga. $\mathcal{R}$ zatim smješta sljedeći simbol ulaznog niza u stog i zamjenjuje aSa ili bSb sa S u listi. $\mathcal{R}$ nastavlja na isti način sve dok se ne iskoristi cijeli ulazni niz. Ako je SZ ostalo u listi, $\mathcal{R}$ briše SZ i unosi završno stanje.

♦ Gramatika i ekvivalentni prošireni stogovni automat

Ako je $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},S)$ beskontekstna gramatika, može se izvesti ekvivalentni nedeterministički prošireni stogovni automat $\mathcal{L}(\mathcal{R})=\mathcal{L}(\mathcal{G})$. $\mathcal{R}$ će biti uređena sedmorka, $\mathcal{R}=(\{q,r\}, \mathcal{T}, \mathcal{T}\cup\mathcal{N}\cup\{\$\}, \delta, q, \$, \{q,r\},)$, gdje je funkcija prijelaza δ definirana kao

(1) $\delta(q,a,a) = \{(q,a): a\in\mathcal{T}\}$
(2) $\delta(q,\varepsilon,\alpha) = \{(q,a): (A\rightarrow\alpha)\in\mathcal{P}, A\in\mathcal{N}, \alpha\in(\mathcal{T}\cup\mathcal{N})^+\}$
(3) $\delta(q,\varepsilon,\$S) = \{(r,\varepsilon)\}$

Vrh stoga je na desnoj strani.

♣ Primjer 3.5

Prošireni stogovni automat s praznim stogom ekvivalentan gramatici $\mathcal{G}_E$

(1) E→E+T (2) E→T (3) T→T*F (4) T→F (5) F→(E) (6) F→a

bit će $\mathcal{R}$=({q,r}, {a,+,*,(,)}, {E,T,F,a,+,*,(,),$},δ,q,$,{r}), gdje su:

(1) $\delta(q,x,\varepsilon)$ = {(q,x)} za sve x∈{a,+,*,(,)}
(2) $\delta(q,\varepsilon,E+T)$ = {(q,E)}
$\delta(q,\varepsilon,T)$ = {(q,E)}
$\delta(q,\varepsilon,T*F)$ = {(q,T)}
$\delta(q,\varepsilon,F)$ = {(q,T)}
$\delta(q,\varepsilon,(E))$ = {(q,F)}
$\delta(q,\varepsilon,a)$ = {(q,F)}
(3) $\delta(q,\varepsilon,\$E)$ = {(r,ε)}

Na primjer, za prepoznavanje ulaznog niza a*(a+a) $\mathcal{R}$ će načiniti sljedeći niz pomaka:

(q,a*(a+a),$)	⊢ (q, *(a+a), $a)	⊢ (q, *(a+a), $F)
	⊢ (q, *(a+a), $T)	⊢ (q, (a+a), $T*)
	⊢ (q, a+a), $T*( )	⊢ (q, +a), $T*(a)
	⊢ (q, +a), $T*(F)	⊢ (q, +a), $T*(T)
	⊢ (q, +a), $T*(E)	⊢ (q, a), $T*(E+)
	⊢ (q,), $T*(E+a)	⊢ (q,), $T*(E+F)
	⊢ (q,), $T*(E+T)	⊢ (q,), $T*(E)
	⊢ (q, ε, $T*(E))	⊢ (q, ε, $T*F)
	⊢ (q, ε, $T)	⊢ (q, ε, $E)
	⊢ (r, ε, ε)	

dakle, desno parsiranje je:

E $\Rightarrow_2$ T $\Rightarrow_3$ T*F $\Rightarrow_5$ T*(E) $\Rightarrow_1$ T*(E+T) $\Rightarrow_4$ T*(E+F) $\Rightarrow_6$ T*(E+a) $\Rightarrow_2$ T*(T+a) $\Rightarrow_4$ T*(F+a) $\Rightarrow_6$ T*(a+a) $\Rightarrow_4$ F*(a+a) $\Rightarrow_6$ a*(a+a)

Valja primijetiti da je u prethodnom primjeru bilo moguće načiniti veliki broj pomaka s ulaznim nizom a*(a+a). Međutim, jedino se danim nizom pomicanja iz početne konfiguracije može okončati u prihvatljivoj konačnoj konfiguraciji. Dakle, može se zaključiti da je općenito prošireni stogovni prepoznavač nedeterministički i ekvivalentan je uzlaznom (povratnom) postupku sintaksne analize. U sljedećem smo primjeru pokazali kako se može konstruirati prošireni deterministički prepoznavač iz gramatike $\mathcal{G}_E$, ekvivalentnoj gramatici $\mathcal{G}_E$ koji će uvijek raditi deterministički.

♣ Primjer 3.6

Prošireni stogovni automat s praznim stogom ekvivalentan gramatici $\mathcal{G}_E$:

```
(1) E→E+E   (2) E→E*E   (3) E→(E)   (4) E→a
```

bit će $\mathcal{R}$=({q,r}, {a,+,*,(,)}, {E,a,+,*,(,),$},δ,q,$,{r}), gdje su:

```
(1) δ(q,x,ε)   = {(q,x)} za sve x∈{a,+,*,(,)}
(2) δ(q,ε,E+E) = {(q,E)}
    δ(q,ε,E*E) = {(q,E)}
    δ(q,ε,(E)) = {(q,E)}
    δ(q,ε,a)   = {(q,E)}
(3) δ(q,ε,$E)  = {(r,ε)}
```

Na primjer, za prepoznavanje ulaznog niza a*(a+a) $\mathcal{R}$ će načiniti sljedeći niz pomaka:

```
(q,a*(a+a),$) ⊢ (q, *(a+a), $a       )   ⊢ (q, *(a+a), $E       )
              ⊢ (q, (a+a),  $E*      )   ⊢ (q, a+a),   $E*(     )
              ⊢ (q, +a),    $E*(a    )   ⊢ (q, +a),    $E*(E    )
              ⊢ (q, a),     $E*(E+   )   ⊢ (q, ),      $E*(E+a  )
              ⊢ (q, ),      $E*(E+E  )   ⊢ (q, ),      $E*(E    )
              ⊢ (q, ε,      $E*(E)   )   ⊢ (q, ε,      $E*E     )
              ⊢ (q, ε,      $E       )   ⊢ (r, ε,      ε        )
```

Desno parsiranje je:

```
E ⇒2 E*E ⇒3 E*(E) ⇒1 E*(E+E) ⇒4 E*(E+a) ⇒4 E*(a+a) ⇒4 a*(a+a)
```

U sedmom ćemo poglavlju obraditi nekoliko postupaka jednoprolazne uzlazne sintaksne analize koji se još nazivaju "sintaksna analiza s reduciranjem".

P R O G R A M I

STOGOVNI PREPOZNAVAČ

Program stogovnog prepoznavača učitava iz posebne tekstualne datoteke koja će predstavljati inicijalizaciju komponenti prepoznavača – varijabli u Pythonu. Imena su sljedeća:

- `Q` - lista stanja
- `A` - alfabet (lista znakova)
- `St` - alfabet stoga (`['#'] + A +` lista "neterminala")
- `_1` - početni znak stoga `St`
- `s` - početno stanje
- `F` - lista konačnih stanja
- `D` - funcija prijelaza definirana kao n-torka para n-torki:

```
( ( (q, a, t), (q', k) ), ( (…), (…) ) ... )
```

Na primjer, definicija prepoznavača jezika $\mathcal{L}$={0^n1^n: n>0} iz primjera 3.1 može se napisati kao

```
Q  = [0, 1, 2];  A = ['0', '1'];  St = ['$', '0'];  _1 = '$';  s = 0; F = [0]
```

```
D  = ( ( (0,'0','$'), (1,'0$') ),
       ( (1,'0','0'), (1,'00') ),
       ( (1,'1','0'), (2,'')   ),
       ( (2,'1','0'), (2,'')   ),
       ( (2,'','$'),  (0,'')   )  )
```

SP.py *Stogovni prepoznavač*

```
# -STOGOVNI PREPOZNAVAČ BESKONTEKSTNIH JEZIKA
import os
from   fun import *
from   easygui import *

NL = '\n'
def Ucitaj_W (): # Učitavanje ulaznog niza
  return komp (raw_input ('Upiši ulazni niz: '))

def Ispisi_SP (Ime):
  print Ime
  print NL, 'SP = (Q, A, St, _1, D, s, F)', NL
  print 'Q  =', Q, ' A =', A,  ' St =', St, ' _1 =', _1, ' s =', s, ' F =', F
  print NL, 'D:'
  for d in D:
    print ' ', d[0], '=', d[1]
  print

def Ucitaj_SP (): # Ucitavanje tablice prijelaza
  Ok = True
  Ime = fileopenbox ('UČITAVANJE STOGOVNOG PREPOZNAVAČA (SP)', None,'*.SP','*')
  if os.path.exists(Ime):
    for line in open (Ime, 'r') :
      exec (line)
  else:
    print 'Ne postoji SP s danim imenom!'
    Ok = False
  return Ime, Ok, (Q, A, St, _1, D, s, F)

def Ispisi_C (y, C): print y, C

def SP (x):
  global Q, A, St, _1, D, s, F
  Ok = True; End = False; q = s; alfa = '$'; C = (q, x, alfa); Ispisi_C ('', C)
  while len(x)>=0 and Ok and not End:
    X = '';  a = ''
    if len(x) > 0    : X = x[0]; x = x[1:]
    if len(alfa) > 0 : a = alfa[0]
    Ok = False
    for d in D:
      d0, d1 = d
      q0, e, g1 = d0
      if q == q0 and e == X and g1 in a:
        q, g2 = d1
        if g2 == '' and a != '': alfa = alfa[1:]
        if g2 != ''            : alfa = g2 +alfa[1:]
        Ok = True
        break
```

```
      if Ok:
        C = (q, x, alfa);
        Ispisi_C ('   |--', C)
        if q in F and alfa == '' : End = True
        if End and x != ''       : Ok  = False
    Ok = Ok and End
    return Ok

Ime, Ok, DSP = Ucitaj_SP ()
Q, A, St, _1, D, s, F = DSP
Ispisi_SP (Ime)
w = Ucitaj_W(); print
while len(w) > 0:
  Ok = SP (w)
  if Ok: Ispisi_C ('   |--', 'accept')
  else : Ispisi_C ('   |--', 'error')
  print
  w = Ucitaj_W(); print
```

Pitanja i zadaci

1) Definirajte stogovni prepoznavač jezika "aritmetičkih" izraza generiranog gramatikom s produkcijama:

A → AOA| (A)| a
O → +| -| *| /

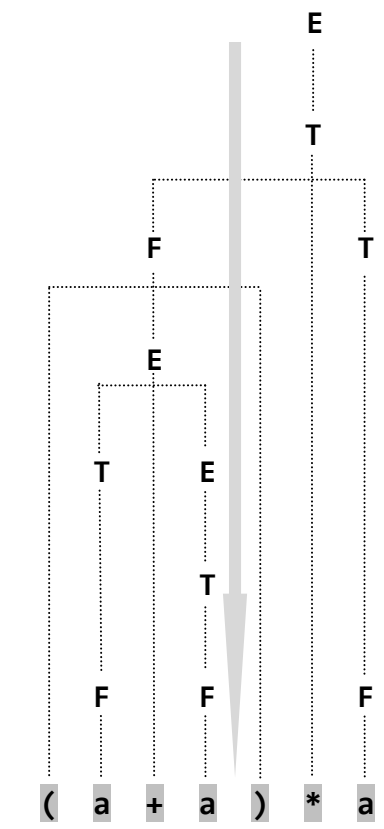
2) Definirajte stogovni prepoznavač jezika $L=\{wcw^R: w\in\{a,b\}^*\}$.

3) SP.py je deterministički stogovni prepoznavač. Preuredite ga tako da bude nedeterministički.

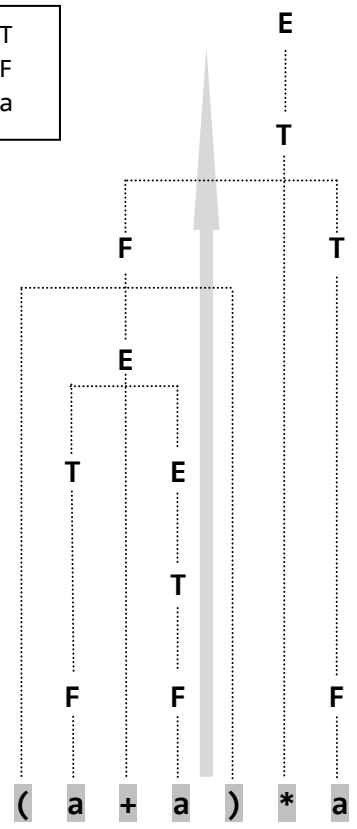
4) Napišite program koji će predstavljati realizaciju nedeterminističkog prepoznavača s praznim stogom.

5) Napišite program koji će predstavljati realizaciju proširenog stogovnog prepoznavača.

4.
VIŠEPROLAZNO PARSIRANJE



E → T+E| T
T → F*T| F
F → (E)| a



E ⇒ T ⇒ F*T⇒ (E)*T
⇒ (T+E)*T ⇒ (F+E)*T
⇒ (a+E)*T ⇒ (a+T)*T
⇒ (a+F)*T ⇒ (a+a)*T
⇒ (a+a)*F ⇒ (a+a)*a

E ⇒ T ⇒ F*T ⇒ F*F
⇒ F*a ⇒ (E)*a
⇒ (T+E)*a ⇒ (T+T)*a
⇒ (T+F)*a ⇒ (T+a)*a
⇒ (F+a)*a ⇒ (a+a)*a

Sintaksna analiza (parsiranje) beskontekstnih jezika važan je dio teorije formalnih jezika. Mnoge knjige i znanstveni radovi bave se tim problemom. Postupci općenite sintaksne analize primjenljivi su nad širokom klasom beskontekstnih jezika. Postoji nekoliko takvih algoritama. Ovdje su opisana dva višeprolazna postupka koja pripadaju klasi povratnih ("backtrack") algoritama: silazna (top-down) sintaksna analiza i uzlazna (bottom-up) sintaksna analiza.

4.1 SILAZNA SINTAKSNA ANALIZA

Naziv "silazna sintaksna analiza" ili "sintaksna analiza s vrha" (top-down) dolazi od ideje da se pokuša izvesti stablo sintaksne analize ulaznog niza započeto od korijena (vrha) gradeći ga prema dolje, prema listovima. Neformalno je postupak silazne sintaksne analize jezika generiranog nekom gramatikom G sljedeći:

1) Najprije se alternativama produkcija gramatike G pridruže, proizvoljno, indeksi. Na primjer, ako su α_1, α_2, ..., α_n sve alternative produkcije za A, onda je α_1 prva, α_2 druga alternativa, itd.
2) Neka je $w=a_1a_2...a_n$ ulazni niz za koji treba utvrditi je li u jeziku generiranom s G. Bit će upotrijebljena kazaljka koja će pokazivati na tekući znak ulaznog niza (na početku je to znak a_1).
3) Početni simbol S korijen je stabla sintaksne analize i istovremeno aktivni čvor.

Sljedeća dva koraka izvode se rekurzivno:

4) Ako je aktivni čvor neterminalni simbol X, bira se njegova prva alternativa i generira k njegovih izravnih slijednika. Neka su označeni s X_1, ..., X_k. X_1 je novi aktivni čvor. Ako je $k=0$, aktivni čvor postaje simbol koji slijedi X.
5) Ako je aktivni čvor terminalni simbol, treba ga usporediti s tekućim simbolom ulaznog niza (simbolom na koji pokazuje kazaljka). Ako su jednaki, kazaljka se pomiče za jedan simbol udesno. Aktivni čvor postaje simbol desno od terminalnog simbola.

Ako terminalni simbol nije jednak tekućem simbolu ulaznog niza, vraća se na čvor gdje je bila upotrijebljena prethodna produkcija i izvodi sljedeća alternativa. Ako nijedna alternativa nije moguća, vraća se na prethodni čvor itd. Ako su upotrijebljene sve alternative i nije se uspjelo s izjednačivanjem ulaznog niza w, ulazni niz nije u jeziku generiranom gramatikom G. Na primjer, neka je dana gramatika G s produkcijama:

$S \rightarrow aSbS| aS| c$

Treba provjeriti je li ulazni niz $w=aacbc$ u jeziku kojeg G generira, tj. postoji li, počevši sa S, niz izvođenja

$S \Rightarrow^* aacbc$

Najprije se može usvojiti da je

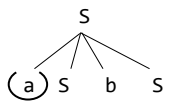
`aSbS` prva,
`aS` druga i
`c` treća alternativa za `S`.

Postupak silazne sintaksne analize bio bi:

a) 

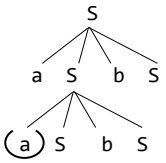

```
aacbc
↑
```

S je korijen i inicijalni aktivni čvor. Kazaljka pokazuje na simbol a.

b) 

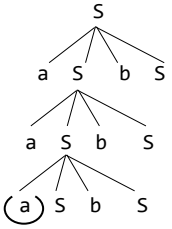

```
aacbc
 ↑
```

Generirana su četiri izravna slijednika od S (prva alternativa). Terminal a je aktivni čvor i jednak je tekućem simbolu. Kazaljka se pomiče za jedno mjesto.

c) 

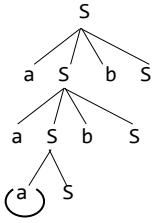

```
aacbc
  ↑
```

Aktivni čvor je neterminal S, pa se ponovo generiraju četiri izravna slijednika (prva alternativa). Aktivni čvor je terminal a, jednak je simbolu ulaznog niza, pa se kazaljka pomiče za jedno mjesto.

d) 

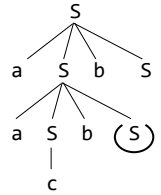

```
aacbc
  ↑
```

Neterminal S ponovno je aktivni čvor. Generiraju se slijednici od S prema prvoj alternativi.

e) 

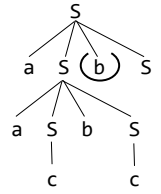

```
aacbc
  ↑
```

Terminal a je aktivni čvor i nije jednak tekućem simbolu. Pokušava se s drugom alternativom.

f) 

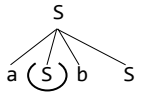

```
aacbc
    ↑
```

Terminal a je aktivni čvor i nije jednak tekućem simbolu. Pokušava se s trećom alternativom. Prvo je terminal c aktivni čvor i jednak je tekućem simbolu. Kazaljka se pomiče za jedno mjesto. Sada je terminal b aktivni čvor i jednak je tekućem simbolu. Kazaljka se pomiče za jedno mjesto.

g) 

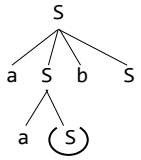

```
aacbc
    ↑
```

Neterminal S je aktivni čvor. Generiraju se slijednici prema prvoj alternativi. Poslije toga je terminal a aktivni čvor i nije jednak tekućem znaku. Pokušava se s drugom alternativom. Opet je terminal a aktivni čvor i nije jednak tekućem znaku. Pokušava se s trećom alternativom. Tada je terminal c aktivni čvor i jednak je tekućem znaku. Kazaljka se pomiče za jedno mjesto.

h) 

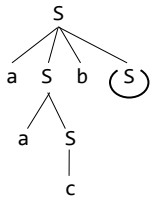
aacbc
↑

Kazaljka je iza posljednjeg znaka ulaznog niza, a u stablu su ostala još dva znaka, b i S. Vraćamo se do mjesta gdje je posljednji put ekspandirano stablo. Ne postoji više alternativa, pa se vraćamo na prethodno mjesto ekspanzije stabla. Ni tu nije moguća nova ekspanzija, pa se dalje vraćamo, do koraka (b).

i) 

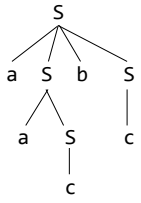
aacbc
↑

Pokušava se s drugom alternativom. Aktivni čvor je terminal a i jednak je tekućem znaku. Kazaljka se pomiče za jedno mjesto.

j) 

aacbc
↑

Aktivni čvor je S. Generira se podstablo iz prve alternative. Novi aktivni čvor je terminal a i nije jednak tekućem znaku. Pokušava se s drugom alternativom. Opet je aktivni čvor terminal a. Pokušava se s trećom alternativom. Aktivni čvor je terminal c i jednak je tekućem znaku. Kazaljka se pomiče za jedno mjesto. Aktivni čvor je terminal b i jednak je tekućem znaku. Kazaljka se pomiče za jedno mjesto.

k) 

aacbc
↑

Aktivni čvor je S. Generira se podstablo iz prve alternative. Novi aktivni čvor je terminal a i nije jednak tekućem znaku. Pokušava se s drugom alternativom. Opet je aktivni čvor terminal a. Pokušava se s trećom alternativom. Aktivni čvor je terminal c i jednak je tekućem znaku. Kazaljka se pomiče za jedno mjesto i pokazuje iza posljednjeg znaka ulaznog niza. Više nema aktivnih čvorova. Ulazni niz je u jeziku.

Primjenljivost silazne sintaksne analize

Iz prethodnog primjera mogu se uočiti osnovni nedostaci postupka silazne (povratne) sintaksne analize.

Prvo, postoji veoma opasna klopka u toj proceduri. Ako je gramatika rekurzivna slijeva proces se ne bi nikada završio! Naime, u tom slučaju bi se smjenom nekog neterminala (aktivnog čvora) dobio isti aktivni čvor, pa njegovom smjenom opet isti itd.

Drugi je nedostatak povratne silazne sintaksne analize to što uvodi određene semantičke aspekte. Na primjer, treba znati koji će dio tablice simbola biti izbrisan u slučaju neuspješnog izjednačivanja; koja je alternativa na redu da se ekspandira stablo sintaksne analize i na koji simbol ulaznog niza treba vratiti kazaljku.

Treći je nedostatak indeksiranje (uređenje) alternativa produkcija. Stavljanje neke alternative na prvo mjesto u nekoj produkciji može ili povećati ili smanjiti ukupno vrijeme potrebno za izvršavanje sintaksne analize. To znači da bi trebalo znati vjerojatnosti pojavljivanja svake od alternativa u nizovima izvođenja rečenica jezika i potom ih urediti stavljajući na prva mjesta one alternative koje imaju veću vjerojatnost pojavljivanja.

Intuitivno se može zaključiti da bi se sintaksna analiza s vrha mogla ubrzati ako sve alternative počinju terminalnim simbolima. Ako bi, pored toga, terminalni simboli bili različiti, moglo bi se jednoznačno, na osnovu tekućeg simbola ulaznog niza, odrediti koja će alternativa biti upotrijebljena. Ovakva razmišljanja, kao što će se vidjeti u sljedećem poglavlju, vode k pojmu `LL(k)` gramatika i jednoprolaznih sintaksnih analiza.

Da zaključimo: silazna sintaksna analiza može se primijeniti samo za gramatike koje nemaju rekurzija slijeva. Gramatika iz uvodnog primjera silazne sintaksne analize nije rekurzivna slijeva (no jest zdesna), pa je moguće u konačno mnogo prolazaka iscrpsti sva izvođenja.

Eliminiranje rekurzija slijeva

Transformacija gramatike koja je rekurzivna slijeva u ekvivalentnu gramatiku koja to nije, česta je u teoriji formalnih jezika, posebno u teoriji sintaksne analize.

• Propozicija 4.1

Svaki beskontekstni jezik ima gramatiku nerekurzivnu slijeva.

Ovo je poznati teorem teorije formalnih jezika čiji nas dokaz ne interesira već nam je važno znati i upamtiti tu činjenicu pa koristimo riječ "propozicija". Sada možemo reći da je silazna sintaksna analiza primjenljiva nad cijelom klasom beskontekstnih jezika. Također je korisno znati i sljedeću propoziciju.

• Propozicija 4.2

Ako je $\mathcal{G}$ gramatika $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ u kojoj su:

$$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

sve A-produkcije u $\mathcal{P}$ i nijedan β_i ne počinje s A, tada je $\mathcal{G}'$ gramatika ekvivalentna gramatici $\mathcal{G}$, $\mathcal{G}'=(\mathcal{N}'\cup\{A'\}, \mathcal{T}, \mathcal{P}', \mathcal{S})$, gdje je A' novi neterminal, a $\mathcal{P}'$ je dobiveno iz $\mathcal{P}$ u kojem su produkcije zamijenjene sa:

$$A \rightarrow \beta_1 \mid \ldots \mid \beta_n \mid \beta_1 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 \mid \ldots \mid \alpha_m \mid \alpha_1 A' \mid \ldots \mid \alpha_m A'$$

♣ Primjer 4.1

Primijenimo transformaciju iz propozicije 4.2. na gramatiku $\mathcal{G}$ s produkcijama:

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid a$$

Dobit ćemo gramatiku $\mathcal{G}'$ s produkcijama:

$$E \rightarrow T \mid TE'$$
$$E' \rightarrow +T \mid +TE'$$
$$T \rightarrow F \mid FT'$$
$$T' \rightarrow *F \mid *FT'$$
$$F \rightarrow (E) \mid a$$

odnosno, ako E' zamijenimo s X, a T' s Y, imamo:

E → T | TX
X → +T | +TX
T → F | FY
Y → *F | *FY
F → (E) | a

♥ Algoritam 4.1 *Eliminiranje rekurzija slijeva.*

Ulaz
Svojstvena beskontekstna gramatika $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$.

Izlaz
Ekvivalentna gramatika $\mathcal{G}'=(\mathcal{N}', \mathcal{T}, \mathcal{P}', S)$ nerekurzivna slijeva.

Postupak
Propozicija 4.2 je temeljna za eliminiranje rekurzija slijeva. Skup produkcija $\mathcal{P}'$ ćemo izgraditi na sljedeći način:

(1) Neka je $\mathcal{N}=\{A_1, ..., A_n\}$. Treba transformirati $\mathcal{G}$ tako da ako je $A_i \to \alpha$ produkcija, tada α počinje ili terminalom ili neterminalom A_j tako da je $j>i$. Postaviti $i=1$.
(2) Neka je A_i-produkcija oblika $A_i \to A_i\alpha_1|...|A_i\alpha_m|\beta_1|...|\beta_p$ gdje nijedan β_j ne počinje s A_k ako je $k \le i$. (Uvijek će to biti moguće.) Zamijeniti A_i-produkcije sa:

$$A_i \to \beta_1|...|\beta_p|\beta_1A'_i|...|\beta_pA'_i$$
$$A'_i \to \alpha_1|...|\alpha_m|\alpha_1A'_i|...|\alpha_mA'_i$$

gdje je A'_i novi neterminal. Sada sve A_i-produkcije počinju terminalom ili s A_k, $k>i$.
(3) Ako je $i=n$, $\mathcal{G}'$ će biti rezultirajuća gramatika i prekida se daljnje izvođenje postupka. Inače, $i=i+1$ i $j=1$.
(4) Zamijeniti sve produkcije oblika $A_i \to A_j\alpha$, gdje je $A_j \to \beta_1|...|\beta_m$, produkcijama $A_i \to \beta_1\alpha|...|\beta_m\alpha$. Ako je A_j-produkcija počinjala terminalom ili s A_k, $k>j$, isto svojstvo će sada imati A_i-produkcija.
(5) Ako je $j=i-1$, ide se na korak (2). Inače, $j=j+1$ i ide se na korak (4).

♣ Primjer 4.2

Primijenimo algoritam 4.1 na gramatiku $\mathcal{G}$ s produkcijama:

A → BC| a B → CA| Ab C → AB| CC| a

Odmah uočavamo da je C-produkcija rekurzivna slijeva, dok A i B imaju skrivene rekurzije, također slijeva. Neka je A_1=A, A_2=B i A_3=C. Prikazana je promjena gramatika poslije svake primjene koraka (2) ili (4), ali samo nove produkcije neterminala čije su produkcije promijenjene:

(2) i=1: nema promjene
(4) i=2, j=1: B → CA|BCb|ab
(2) i=2: B → CA|ab|CAB'|abB'
B'→ CbB'|Cb
(4) i=3, j=1: C → BCB|aB|CC|a
(4) i=3, j=2: C → CACB|abCB|CAB'CB|abB'CB|aB|CC|a

(2) i=3: C → abCB|abB'CB|aB|a|abCBC'|abB'CBC'|aBC'|aC'
C'→ ACBC'|AB'CBC'|CC'|ACB|AB'CB|C

Konačno, uvodeći smjenu X za B' i Y za C', rezultirajuća gramatika bez rekurzija slijeva ima produkcije:

A → BC| a
B → CA| ab| CAX| abX
X → CbX| Cb
C → abCB| abXCB| aB| a| abCBY| abXCBY| aBY| aY
Y → ACBY| AXCBY| CY| ACB| AXCB| C

Eliminiraj_R.py *Eliminiranje rekurzija slijeva*

```
def Eliminiraj_R (G):
  N, T, P, S = G; P2 = []; N2 = []
  for x in P: P2.append(x)
  for x in N: N2.append(x)

  Nf = list(A)
  for x in N : Nf.remove(x)
  n  = len(N);

  for i in range (len(P)):
    Ai = P[i][0];  X = P[i][1:]; Y = []; k = 0
    while k < len(X):
      Beta = X[k]; NT = Beta[0]
      if NT in T or NT in N and N.index(NT) > i : Y.append(Beta)
      k += 1

    for Beta in Y:
      if Beta in X: X.remove (Beta)

    if len(X) != 0:
      A2 = []; P2[i] = [Ai] +Y +X
      for j in range(0,i+1):
        Y = P2[i][1:]; P2[i] = P2[i][:1];  k = 0
        while k < len (Y):
          Beta = Y[k]; NT = Beta[0]
          if NT == N[j]:
            if j < i:
              Z = P2[j];  m = len(Z)
              for p in range (1, m): P2[i].append(Z[p] +Beta[1:])
            else :
              Nn = Nf[0];  N2.append(Nn);  Nf = Nf[1:]
              Y = Y[k:]; A2 = []
              while len (Y) > 0:
                Beta = Y[0]; NT = Beta[0]; Alfa = Beta[1:]
                if NT != Ai : P2[i].append(Beta)
                else        : A2 = A2 +[Alfa] +[Alfa +Nn]
                Y.remove (Beta)
              m = len (P2[i])
              for p in range (1, m): P2[i].append (P2[i][p] +Nn)
              P2.append([Nn] +A2)
          else:
            P2[i].append(Y[k])
          k += 1

  return (N2, T, P2, S)
```

♣ Primjer 4.3

U sljedećoj tablici dane su tri gramatike koje su rekurzivne slijeva i njihove ekvivalentne gramatike bez rekurzija slijeva dobivene primjenom procedure **`Eliminiraj_R()`**.

Ulazna gramatika	Ekvivalentna gramatika bez rekurzija slijeva
p-3-2.grm (primjer 4.2)	p-3-2.gr2
N = ['A', 'B', 'C'] T = ['a', 'b'] S = A P: A -> BC| a B -> CA| Ab C -> AB| CC| a	N = ['A', 'B', 'C', 'D', 'E'] T = ['a', 'b'] S = A P: A -> BC| a B -> CA| ab| CAD| abD C -> a| abCB| abDCB| aB| aE| abCBE| abDCBE| aBE D -> Cb| CbD E -> ACB| ACBE| ADCB| ADCBE| C| CE
ETF1.grm	ETF1.gr2
N = ['E', 'T', 'F'] T = ['+', '*', '(', ')', 'a'] S = E P: E -> E+T| T T -> T*F| F F -> (E)| a	N = ['E', 'T', 'F', 'A', 'B'] T = ['+', '*', '(', ')', 'a'] S = E P: E -> T| TA T -> F| FB F -> (E)| a A -> +T| +TA B -> *F| *FB
ETF0.grm	ETF0.gr2
N = ['E'] T = ['+', '*', '(', ')', 'a'] S = E P: E -> E+E| E*E| (E)| a	N = ['E', 'A'] T = ['+', '*', '(', ')', 'a'] S = E P: E -> (E)| a| (E)A| aA A -> +E| +EA| *E| *EA

Algoritam silazne sintaksne analize

Poslije neformalnog opisa postupka silazne sintaksne analize i analize primjenljivosti takvog postupka, evo i njegova algoritma.

Algoritam koristi dva stoga (dvije "potisne" liste), L_1 i L_2, i kazaljku koja pokazuje na tekući simbol ulaznog niza. U opisu algoritma bit će korištena notacija slična onoj koja je bila upotrijebljena u opisu konfiguracije potisnog prepoznavača.

♥ Algoritam 4.2 *Silazna sintaksna analiza.*

Ulaz

Gramatika $G=(N,T,P,S)$ koja nije rekurzivna slijeva i ulazni niz $w=a_1a_2\ldots a_n$, $n\geq 1$. Pretpostavka je da su produkcije u P numerirane od 1 do p.

Izlaz

Stablo sintaksne analize za w, ako je w u jeziku $L(G)$; inače "pogreška".

Postupak

(1) Urediti alternative za svaki neterminal A iz $\mathcal{N}$. Na primjer, ako su $A\rightarrow\alpha_1|\alpha_2|\ldots|\alpha_k$ sve A-produkcije u $\mathcal{P}$ i alternative su uređene kao što je pokazano, tada je A_1 indeks za α_1, A_2 indeks za α_2, itd.

(2) Četvorka (s,i,L_1,L_2) označivat će konfiguraciju algoritma, gdje su

- s stanje algoritma: q - napredovanje, b – povrat i t – kraj,
- i položaj ulazne kazaljke. Ako je $i=n+1$ znači da je dosegnut kraj ulaznog niza (bit će označen s $),
- L_1 potisna lista koja predstavlja povijest izbora alternativa i simbole ulaznog niza koji su bili jednaki aktivnim čvorovima stabla sintaksne analize. Vrh liste je zdesna,
- L_2 potisna lista koja predstavlja tekuću lijevu rečeničnu formu. Simbol na vrhu (vrh je slijeva) označuje aktivni čvor stabla koje će biti generirano.

(3) Početna konfiguracija algoritma je $(q,1,\varepsilon,S\$)$.

(4) Postoji šest vrsta koraka. Bit će opisani konfiguracijom algoritma. Notacija $(s,i,\alpha,\beta)\vdash(s',i',\alpha',\beta')$, jer se ovdje radi o vrsti automata, ima značenje prelaska iz konfiguracije (s,i,α,β) u konfiguraciju (s',i',α',β'). Moguća su sljedeća premještanja:

(a) *Ekspanzija stabla*

$$(q,i,\alpha,A\beta)\vdash(q,i,\alpha A_1,\alpha_1\beta)$$

gdje je $A\rightarrow\alpha_1$ produkcija u $\mathcal{P}$; α_1 je prva alternativa za A. Ovaj korak odgovara ekspanziji parcijalno izvedenog stabla koristeći prvu alternativu za prvi neterminal s lijeva u stablu.

(b) *Uspješno izjednačenje ulaznog i izvedenog simbola*

$$(q,i,\alpha,a\beta)\vdash(q,i+1,\alpha a,\beta)$$

pod uvjetom da je $a_i=a$, $i\leq n$. Terminalni se simbol premješta s vrha liste L_2 na vrh liste L_1. Kazaljka se pomiče za jedno mjesto.

(c) *Neuspješno izjednačenje ulaznog i izvedenog simbola*

$$(q,i,\alpha,a\beta)\vdash(b,i,\alpha,a\beta)$$

ako je $a_i\neq a$. Lijeva rečenična forma koja je bila upravo izvedena nije konzistentna s ulazom. Prelazi se u "povratni" mod.

(d) *Uspješno okončanje*

$$(q,n+1,\alpha,\$)\vdash(t,n+1,\alpha,\varepsilon)$$

Dosegnut je kraj ulaznog niza i nađena lijeva rečenična forma koja odgovara ulazu.

(e) *Povrat kazaljke*

$$(b,i,\alpha a,\beta)\vdash(b,i-1,\alpha,a\beta)$$

za sve a iz $\mathcal{T}$. Svaki se put ulazni simbol premješta iz L_1 u L_2.

(f) *Pokušaj sljedeće alternative*

$(b,i,\alpha A_j,\alpha_j\beta)\vdash$

(1) $(q,i,\alpha A_{j+1},\alpha_{j+1}\beta)$, ako je α_{j+1} $(j+1)$-alternativa za A.

(2) Ne postoji sljedeća konfiguracija, ako je $i=1$, $A=S$ i postoji samo j alternativa za S. Taj uvjet pokazuje da su iscrpljene sve moguće lijeve rečenične forme konzistentne s ulazom w, a da se za njega nije uspjelo izvesti stablo sintaksne analize. Ulazni niz w nije u jeziku pa se prekida daljnje izvršavanje postupka.

(3) $(b,i,\alpha,A\beta)$, inače. Sve su alternative za A iscrpljene. Vraća se i izbacuju se sve alternative A_j iz L_1 i zamjenjuje α_j s A u L_2.

Izvršavanje algoritma je kao što slijedi:

(1) Krenuvši s početnom konfiguracijom, izračunaju se slijedeće uzastopne konfiguracije:

$C_0 \vdash C_1 \vdash \ldots \vdash C_i \vdash \ldots$

sve dok je definirana naredna konfiguracija.

(2) Ako je posljednje izračunata konfiguracija

$(t,n+1,\alpha,\varepsilon)$

prekida se daljnje izvršavanje. Ulazni je niz u jeziku generiranom gramatikom G. Inače, dojavljuje se pogreška.

♣ Primjer 4.4

Primijenimo algoritam u sintaksnoj analizi jezika jednostavnih aritmetičkih izraza generiranog gramatikom:

$E \to T+E\ (1)\quad E \to T\ \ (2)\quad T \to F*T\ (3)\quad T \to F\ (4)\quad F \to a\ (5)$

gdje brojevi u zagradi označuju uređenje produkcija. Neka je E1, jednako T+E, prva alternativa za E, a E2, jednako T, druga alternativa za E. T1 jednako F*T i T2 jednako F su prva i druga alternativa za T, a F1 jednako a je prva alternativa za F. Ako je ulazni niz w jednak a+a, algoritam će izračunati slijedeće konfiguracije:

```
(q, 1, ε, E$)
    ⊢ (q, 1, E1,               T+E$) ⊢ (q, 1, E1T1,            F*T+E$)
    ⊢ (q, 1, E1T1F1,         a*T+E$) ⊢ (q, 2, E1T1F1a,          *T+E$)
    ⊢ (b, 2, E1T1F1a,         *T+E$) ⊢ (b, 1, E1T1F1,          a*T+E$)
    ⊢ (b, 1, E1T1,           F*T+E$) ⊢ (q, 1, E1T2,             F+E$)
    ⊢ (q, 1, E1T2F1,           a+E$) ⊢ (q, 2, E1T2F1a,           +E$)
    ⊢ (q, 3, E1T2F1a+,           E$) ⊢ (q, 3, E1T2F1a+E1,        T+E$)
    ⊢ (q, 3, E1T2F1a+E1T1,   F*T+E$) ⊢ (q, 3, E1T2F1a+E1T1F1, a*T+E$)
    ⊢ (q, 4, E1T2F1a+E1T1F1a, *T+E$) ⊢ (b, 4, E1T2F1a+E1T1F1a, *T+E$)
    ⊢ (b, 3, E1T2F1a+E1T1F1, a*T+E$) ⊢ (b, 3, E1T2F1a+E1T1,   F*T+E$)
    ⊢ (q, 3, E1T2F1a+E1T2,     F+E$) ⊢ (q, 3, E1T2F1a+E1T2F1,   a+E$)
    ⊢ (q, 4, E1T2F1a+E1T2F1a,   +E$) ⊢ (b, 4, E1T2F2a+E1T2F1a,   +E$)
    ⊢ (b, 3, E1T2F1a+E1T2F1,   a+E$) ⊢ (b, 3, E1T2F1a+E1T2,     F+E$)
    ⊢ (b, 3, E1T2F1a+E1,       T+E$) ⊢ (q, 3, E1T2F1a+E2,       T$)
    ⊢ (q, 3, E1T2F1a+E2T1,     F*T$) ⊢ (q, 3, E1T2F1a+E2T1F1,  a*T$)
    ⊢ (q, 4, E1T2F1a+E2T1F1a,   *T$) ⊢ (b, 4, E1T2F1a+E2T1F1a,  *T$)
```

```
    ⊢ (b, 3, E1T2F1a+E2T1F1,    a*T$) ⊢ (b, 3, E1T2F1a+E2T1,      F*T$)
    ⊢ (q, 3, E1T2F1a+E2T2,        F$) ⊢ (q, 3, E1T2F1a+E2T2F1,      a$)
    ⊢ (q, 4, E1T2F1a+E2T2F1a,      $) ⊢ (t, 4, E1T2F1a+E2T2F1a,      ε)
```

Lista L_1 sadrži stablo sintaksne analize (lijeve rečenične forme):

```
E ⇒ T+E ⇒ F+E ⇒ a+E ⇒ a+T ⇒ a+F ⇒ a+a
```

Algoritam silazne sintaksne analize (parser) realiziran je u Pythonu i dan u nastavku. Program može prikazati svaku promjenu konfiguracije, pa istovremeno predstavlja lekciju za učenje postupka silazne sintaksne analize.

TopDown.py *Silazna sintaksna analiza*

```
def TopDown (G, w, ISP = True): # Silazna (top-down) sintaksna analiza
  """ G - ulazna gramatika; w - ulazni niz; ISP - ispis konfiguracija """
  def Niz_I (Alfa): # Ispis niza izvođenja
    Net = []
    for i in range (len(P)): Net.append (P[i][0])
    SF = S; print SF,
    D = '';
    for L in range (len(Alfa)):
      C = Alfa[L]
      if C in N: D += C +Alfa[L+1]
    L = 0
    while L < len(D):
      C  = D [L]; i = Net.index(C);  j = int(D[L+1])
      SF = SF.replace (C, P[i][j], 1)
      print '=>', SF,
      L += 2
    print '\n'

  def Forw (C, Sym): # Napredovanje
    s, i, Alfa, Beta = C
    t = Beta[0]; k = pos (t, N)
    if k != -1:
      """ ekspanzija stabla """
      Alfa = Alfa +t +'1';  Beta = P[k][1] +Beta[1:]
    else:
      """ izjednačivanje """
      if t != Sym: s = 'b'
      else:
        i += 1
        if i != n+1 : Sym = w[i-1]
        else        : Sym = ''; s = 'b'
        Alfa += t; Beta = Beta[1:]
        if Beta == '$' and i == n+1 : s = 't'
    return (s, i, Alfa, Beta), Sym

  def Back (C, Sym): # Vraćanje
    s, i, Alfa, Beta = C
    t = Alfa [-1];
    if t in T and pos (Alfa [len(Alfa) -2], N) == -1:
      """ vraćanje kazaljke """
      Alfa = Alfa[:len(Alfa)-1];  Beta = t +Beta;  i = i-1;  Sym = w[i-1]
```

```
    else:
      """ pokušaj slijedeće alternative """
      RB = int (t); t = Alfa[len(Alfa)-2]; k = pos(t, N); m = len(P[k])-1
      if RB == m :
        """ iscrpljene su sve alternative """
        Alfa = Alfa [:len(Alfa)-2];  Beta = t +Beta[len(P[k][m]):]
        Err = Alfa == '' and i == 0
      else:
        """ sljedeća alternativa """
        m = RB +1
        Alfa = Alfa[:len(Alfa)-2] +t +str(m)
        Beta = P[k][m] +Beta[len(P[k][m-1]):]
        s = 'q'
    return (s, i, Alfa, Beta), Sym

  def Ispis (C): # Ispis konfiguracije
    if ISP: print '\n%6d' %BC, C

  N, T, P, S = G
  Err = False;  s = 'q';  i = 1; Alfa = '';  Beta = S +'$'; BC = 0
  n = len(w);  Sym = w[0]
  C   = (s, i, Alfa, Beta)
  while not Err and s != 't':
    BC += 1;  Ispis (C)
    s = C[0]
    if   s == 'q': C, Sym = Forw (C, Sym)
    elif s == 'b':
      Err = C[2] == ''
      if not Err : C, Sym = Back (C, Sym)
  print BC, 'konfiguracija \n'
  if not Err:
    print w, ' je rečenica jezika! Može se dobiti nizom izvođenja: \n'
    Niz_I (C[2]); print
  else:
    print 'Niz ', w, ' nije u jeziku!'
  return
```

♣ Primjer 4.5

Primijenimo parser **`TopDown()`** u sintaksnoj analizi jezika jednostavnih aritmetičkih izraza generiranog gramatikom `ETF.grm`,

$E \to T+E\ (1)\quad E \to T\ (2)\quad T \to F*T\ (3)\quad T \to F\ (4)\quad F \to (E)\ (5)\quad F \to a\ (6)$

za nekoliko ulaznih nizova $w=(^{i}a)^{i}$, $i=1,\ldots,9$. U sljedećoj su tablici dani rezultati analize (BC - broj konfiguracija), dok su nizovi izvođenja izostavljeni.

i	BC	i	BC	i	BC
1	175	4	12,922	7	830,053
2	776	5	51,827	8	3,320,414
3	3,201	6	207,468	9	13,281,879

Vjerujemo da ste iznenađeni brojem konfiguracija koje rastu geometrijskom progresijom! I autor ove knjege bio je iznenađen kad je prije 30 godina realizirao prvi parser silazne analize u jeziku MBASIC, u operacijskom sustavu CP/M. Analiza ulaznog niza $(^{5}a)^{5}$ trajala je oko 45 minuta! Dakako, razlog tomu bio je ne samo veliki broj promjena konfiguracija (51,827), već i znatno sporiji procesor od današnjih.

Već smo u uvodnom dijelu o silaznoj sintaksnoj analizi rekli da se njezina efikasnost može povećati ako bismo znali vjerojatnosti pojavljivanja svake od alternativa u nizovima izvođenja rečenica jezika i potom ih uredili stavljajući na prva mjesta one alternative koje imaju veću vjerojatnost pojavljivanja. Također treba staviti alternative koji počinju terminalom na prva mjesta, jer se tada aktivni čvor – terminal izravno uspoređuje s tekućim simbolom i pokušava sa sljedećom alternativom ako nisu jednaki.

♣ Primjer 4.6

Usporedimo analizu istih nizova s tri uređenja gramatike `ETF.grm`:

```
ETF.grm   E→T+E (1)  E→T   (2)  T→F*T (3)  T→F   (4)  F→(E) (5)  F→a   (6)

ETF-1.grm E→T   (1)  E→T+E (2)  T→F   (3)  T→F*T (4)  F→(E) (5)  F→a   (6)

ETF-2.grm E→T   (1)  E→T+E (2)  T→F   (3)  T→F*T (4)  F→a   (5)  F→(E) (6)
```

Rezultati su prikazani u sljedećoj tablici.

i	w_i	BC		
		ETF	ETF-1	ETF-2
1	(((((((a)))))))	13,281,879	47	61
2	(a+a)*a	252	127	123
3	(a+a)*(a+a)*a	474	247	241
4	a+a+a+a+a+a+a+a+a	150	207	189
5	a*a*a*a*a*a*a*a*a	222	119	101
6	a+a*a+a*a+a*a+a*a	132	217	199
7	(((a+a)*a)+a)*a	3907	2520	2516
8	((((((a))))))	207,518	207,518	207,518

Vidimo da je uređenje (1) i (2) mnogo efikasnije od osnovnog uređenja. Posljednji niz, w_8, nije u jeziku pa je jednak broj konfiguracija u sva tri uređenja.

4.2 UZLAZNA SINTAKSNA ANALIZA

Postoji općenita metoda sintaksne analize koja je po smjeru suprotna od silazne sintaksne analize. To je uzlazna (“bottom-up”) sintaksna analiza.

U silaznoj se sintaksnoj analizi stablo sintaksne analize gradi od korijena prema dolje, do listova. Nasuprot tome, u uzlaznoj sintaksnoj analizi, počinje se od listova i pokušava izgraditi stablo prema gore, do korijena. Ovdje će biti izložen algoritam uzlazne sintaksne analize koji se naziva sintaksna analiza "s reduciranjem premještanja". Analiza se odvija koristeći najvažnije svojstvo sintaksne analize zdesna, prolazeći kroz sva moguća krajnja izvođenja zdesna koja su konzistentna s ulazom.

Premještanje se sastoji od ispitivanja niza na vrhu potisne liste i utvrđivanja postoji li desna strana produkcije koja može biti izjednačena s vrhom liste. Ako postoji, vrh potisne liste treba zamijeniti njome. Ako postoji više desnih strana produkcija koje se mogu upotrebiti za reduciranje vrha postisne liste, uz pretpostavku

prethodnog proizvoljnog uređenja produkcija, bit će upotrebljena prva. Ako nije moguća nijedna redukcija, sljedeći se ulazni simbol premješta u listu i ponovi postupak. Uvijek će se pokušati s reduciranjem prije premještanja. Ako se dosegne kraj ulaznog niza i nije moguća nijedna redukcija, vraća se na posljednje premještanje gdje je bila učinjena redukcija i pokuša se s nekom drugom redukcijom.

Ulazni niz je u jeziku generiranom zadanom gramatikom ako lista sadrži početni simbol i dosegnut je kraj ulaznog niza. Ako se pokušalo sa svim redukcijama i nije se postiglo takvo stanje, postupak se prekida uz zaključak da ulazni niz nije rečenica jezika. Razmotrimo to na sljedećem primjeru. Neka je dana gramatika s produkcijama:

S→AB A→ab B→aba

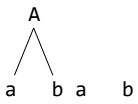
i neka je ulazni niz *w*=ababa. Pogledajmo kako će se odvijati uzlazna sintaksna analiza:

a)

ababa
↑

Najprije se prvi znak ulaznog niza, a, premješta u listu. Nije moguća nijedna redukcija, pa se i b prebacuje u listu.

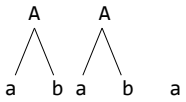
b)



ababa
↑

ab se može reducirati u A. Premješta se sljedeći znak u listu. Sada je u listi Aa i nije moguća redukcija. Premješta se sljedeći znak. Sadržaj je liste Aab.

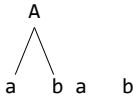
c)



ababa
↑

ab se može reducirati u A. Posljednji se znak premješta u listu. Sadržaj liste je AAa.

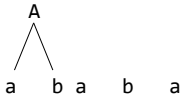
d)



ababa
↑

Nije moguća nijedna redukcija. Vraća se na prethodni korak kad je u listi bilo Aab (b je na vrhu).

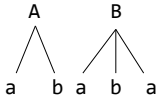
e)



ababa
↑

Nije moguća nijedna redukcija. Premješta se posljednji znak u listu. Sadržaj je liste: Aaba.

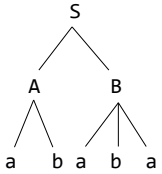
f)



ababa
↑

Sada se aba može reducirati u B. Sadržaj je liste: AB.

g)



ababa
↑

Konačno se AB može zamijeniti sa S. Time je stablo sintaksne analize izgrađeno. Niz je u jeziku koji dana gramatika generira i može se dobiti izvođenjem:

S ⇒ AB ⇒ Aaba ⇒ ababa

Primjenljivost uzlazne sintaksne analize

Izložena metoda može se razmatrati kao niz svih mogućih premještanja u nedeterminističkoj sintaksnoj analizi dane gramatike zdesna. Međutim, kao što je bilo u slučaju analize s vrha, i ovdje postoje izvjesna ograničenja u primjeni.

Prvo, postoje situacije u kojima bi broj mogućih premještanja bio beskonačan. Takva klopka može se pojaviti ako gramatika ima petlje, tj. postoji izvođenje $A^+ \Rightarrow A$ za neki neterminal A. Tada bi broj podstabala bio beskonačan.

Drugu poteškoću stvaraju ε-produkcije u gramatici. Tada se može načiniti proizvoljan broj redukcija u kojima bi prazan niz bio "reduciran" u neki neterminal. Postupak sintaksne analize može se proširiti da prihvaća i gramatike sa ε-produkcijama, ali je to izostavljeno u ovoj knjizi. Osim toga, u trećem je poglavlju dan algoritam za izbacivanje ε-produkcija, iz čega slijedi da je ovaj postupak primjenljiv za sve beskontekstne jezike.

Algoritam uzlazne sintaksne analize

Poslije neformalnog opisa postupka uzlazne sintaksne analize evo i algoritma koji dajemo u notaciji sličnoj onoj koja je upotrijebljena u algoritmu silazne sintaksne analize.

♥ Algoritam 4.3 *Uzlazna sintaksna analiza.*

Ulaz

Gramatika $G=(\mathcal{N},\mathcal{T},\mathcal{P},S)$ bez petlji i ε-produkcija (svojstvena ili primjerena gramatika) i ulazni niz $w=a_1a_2\ldots a_n$, $n\geq 1$. Uređene produkcije od 1 do p.

Izlaz
Stablo sintaksne analize za w, ako je w u $\mathcal{L}(G)$. Inače, "pogreška".

Postupak

(1) Urediti alternative neterminala iz $\mathcal{N}$ počevši od startnog simbola i njegovih alternativa koje će imati indeks 1, 2, itd. sve do posljednjeg neterminala i njegovih alternativa koje će imati indekse ... $p-1$, p.

(2) Četvorka (s,i,L_1,L_2) označivat će konfiguraciju algoritma, gdje su:

- s stanje algoritma: q - napredovanje, b – povrat i t – kraj,
- i lokacija ulazne kazaljke. Ako je $i=n+1$ znači da je dosegnut kraj ulaznog niza (bit će označen s $),
- L_1 potisna lista (vrh zdesna) koja će sadržavati niz terminala i neterminala koji izvodi dio ulaznog niza lijevo od pozicije kazaljke,
- L_2 potisna lista (vrh slijeva) koja će sadržavati povijest premještanja i reduciranja neophodnih za dobivanje sadržaja liste L_1.

(3) Početna je konfiguracija algoritma $(q,1,\$,\varepsilon)$.

(4) Izvršavanje počinje prvim korakom:

1. korak: *premještanje*

$(q,i,\alpha,\gamma) \vdash (q,i+1,\alpha a_i,s\gamma)$

pod uvjetom da je $i \neq n+1$. Potom se ide na drugi korak. Ako je $i=n+1$, prelazi se na treći korak. Ako je prvi korak bio moguć, upisuje se i-ti simbol na vrh liste L_1, pomiče se ulazna kazaljka i upisuje s na vrh liste L_2, što je znak da je bilo obavljeno premještanje.

2. korak: *pokušaj reduciranja*

$(q,i,\alpha\beta,\gamma) \vdash (q,i,\alpha A,j\gamma)$

pod uvjetom da je $A \to \beta$ j-ta produkcija u $\mathcal{P}$. β je prva desna strana u linearnom uređenju (1), tj. sufiks niza $\alpha\beta$. Broj produkcije upisan je u listu L_2. Ako je ovaj korak bio moguć, ponovo se vraća na njega.

3. korak: *prihvaćanje*

$(q,n+1,\$\mathcal{S},\gamma) \vdash (t,n+1,\$\mathcal{S},\gamma)$

Ako ovaj korak nije prihvatljiv, ide se na četvrti korak.

4. korak: *početak povratka*

$(q,n+1,\alpha,\gamma) \vdash (b,n+1,\alpha,\gamma)$

pod uvjetom da je $\alpha \neq \$\mathcal{S}$. Ide se na peti korak.

5. korak: *povratak*

a) $(b,i,\alpha A,j\gamma) \vdash (q,i,\alpha' B,k\gamma)$

ako je $A \to \beta$ j-ta produkcija u $\mathcal{P}$, a $B \to \beta'$ je sljedeća produkcija, označena s k prema uređenju (1), u kojoj je desna strana sufiks od $\alpha\beta$ (vrijedi $\alpha\beta = \alpha'\beta'$). Ide se na drugi korak.

b) $(b,n+1,\alpha A,j\gamma) \vdash (b,n+1,\alpha\beta,\gamma)$

ako je $A \to \beta$ j-ta produkcija u $\mathcal{P}$ i nije preostala nijedna alternativa za reduciranje αß. Ide se ponovo na peti korak.

c) $(b,i,\alpha A,j\gamma) \vdash (q,i+1,\alpha\beta a,s\gamma)$

ako je $i \neq n+1$, j-ta produkcija u $\mathcal{P}$ je $A \to \beta$, i nije preostala nijedna alternativa za reduciranje $\alpha\beta$. Ovdje je $a=a_i$ premješteno u L_1, a s u L_2. Ide se na drugi korak.

d) $(b,i,\alpha a,s\gamma) \vdash (b,i-1,\alpha,\gamma)$

ako je na vrhu liste L_2 simbol koji označuje premještanje (simbol s).

♣ Primjer 4.7

Primijenimo algoritam u sintaksnoj analizi jezika generiranog gramatikom ETF1.grm:

```
E → E + T   (1)    E → T   (2)
T → T * F   (3)    T → F   (4)
F → a       (5)    F →(E)  (6)
```

Primijetimo da je $\mathcal{G}$ rekurzivna slijeva, što u slučaju primjene algoritma uzlazne sintaksne analize ne predstavlja nikakve probleme.

Prema danom uređenju produkcija pojavom E+T na vrhu liste L_1 najprije će se pokušati reduciranje koristeći E→E+T, potom E→T. Također će se uvijek prije pokušati reducirati s T→T*F nego s T→F. Razmotrimo promjenu konfiguracije algoritma u sintaksnoj analizi niza a*a:

```
(q, 1, $, ε)
   ⊢ (q, 2, $a,   s)              ⊢ (q, 2, $F,   5s)
   ⊢ (q, 2, $T,   45s)            ⊢ (q, 2, $E,   245s)
   ⊢ (q, 3, $E*,  s245s)          ⊢ (q, 4, $E*a, ss245s)
   ⊢ (q, 4, $E*F, 5ss245s)        ⊢ (q, 4, $E*T, 45ss245s)
   ⊢ (q, 4, $E*E, 245ss245s)      ⊢ (b, 4, $E*E, 245ss245s)
   ⊢ (b, 4, $E*T, 45ss245s)       ⊢ (b, 4, $E*F, 5ss245s)
   ⊢ (b, 4, $E*a, ss245s)         ⊢ (b, 3, $E*,  s245s)
   ⊢ (b, 2, $E,   245s)           ⊢ (q, 3, $T*,  s45s)
   ⊢ (q, 4, $T*a, ss45s)          ⊢ (q, 4, $T*F, 5ss45s)
   ⊢ (q, 4, $T,   35ss45s)        ⊢ (q, 4, $E,   235ss45s)
   ⊢ (t, 4, $E,   235ss45s)
```

Niz a*a je u jeziku $\mathcal{L}(\mathcal{G})$. Može se dobiti nizom izvođenja (desnih rečeničnih formi):

```
E ⇒ T ⇒ T*F ⇒ T*a ⇒ F*a ⇒ a*a
```

🖫 BottomUp.py *Uzlazna sintaksna analiza*

```
def Bottom_Up (G, w, ISP = True):

  def Niz_I (L2) :
    while 's' in L2: L2 = L2.replace ('s', '')
    SF = Alt[0][0]
    print SF
    for k in range (0,len(L2),2):
      i = int (L2[k:k+2])
      j = SF.rfind(Alt[i-1][0])
      SF = SF[:j] +Alt[i-1][1] +SF[j+1:]
      print '  =>', SF
    print

  def Ispis (C, BC) :
    if ISP: print '%6d' %BC, C

  def Premjesti (C, Sym):
    s, i, L1, L2 = C
    L1 += Sym;  L2 = 's' +L2
    i  += 1
    return (s, i, L1, L2)

  def Reduciranje (C):
    s, i, L1, L2 = C
    m = 1
```

```
  while m <= Br_Alt:
    Beta = Alt[m-1][1]
    L = len (Beta)
    j = L1.rfind (Beta)
    if j != -1 and L == len(L1) -j:
      L1 = L1[: j] +Alt[m-1][0]; L2 = '%02d' %m +L2
      return (s, i, L1, L2)
    else:
      m += 1
  if i == n+1:
    if L1 == '$' +Alt[0][0]: s = 't'
    else                   : s = 'b'
  else:
    Sym = w[i-1]; C = (s, i, L1, L2);
    s,i,L1,L2 = Premjesti (C, Sym)
  return (s,i,L1,L2)

def Vracanje (C):

  def X (C, T): # Pomoćna procedura
    s, i, L1, L2 = C
    L2 = L2[2:];  L = int(T);  L1 = L1[:-1] +Alt[L-1][1]; L += 1
    while L <= Br_Alt:
      Beta = Alt[L-1][1]; j = L1.rfind (Beta)
      if j != -1 and j == len(L1) -len(Beta):
        L1 = L1[: j] +Alt[L-1][0];  L2 = '%02d' %L +L2;  s  = 'q';
        break
      L += 1
    return (s, i, L1, L2)

  def Y (C):
    s, i, L1, L2 = C
    if i > 1:
      i -= 1;  L1 = L1[:-1]
      if L2[0] == 's': L2 = L2[1:]
      else           : L2 = L2[2:]
    return (s, i, L1, L2)

  s, i, L1, L2 = C
  if i < n +1:
    T = ''
    if len(L2) > 0:
      if L2[0] == 's'  : T = L2[0]
      elif len (L2) > 0: T = L2[:2]

    if T == '': return True, (s,i,L1,L2)

    if T == 's':
      C = Y (C)
      s,i,L1,L2 = C
      return False, (s,i,L1,L2)

    s, i, L1, L2 = X (C, T)
    if s == 'b' : s = 'q'
    Sym = w[i-1];  C = (s, i, L1, L2);
    s,i,L1,L2 = Premjesti (C, Sym)
    return False, (s, i, L1, L2)
```

```
    else:
      if L2[0] == 's': T = L2[0]
      else           : T = L2[:2]
      if T == 's': C = Y (C)
      else       : C = X (C, T)
      s,i,L1,L2 = C
      return False, (s, i, L1, L2)

N, T, P, S = G;  Alt = Uredi_P(P);  Br_Alt = len(Alt)
Err = False;  Sym = w[0];  n = len(w);
s = 'q';  i = 1; L1 = '$';  L2 = ''
BC = 0;  C = (s, i, L1, L2);  Ispis (C, BC)

C = Premjesti(C, Sym)
while not Err and s != 't':
  BC += 1; Ispis (C, BC)
  if   s == 'q' : C      = Reduciranje (C)
  else          : Err, C = Vracanje (C)
  s, i, L1, L2 = C
if s == 't': BC += 1; Ispis (C, BC)

print '\n', BC, 'konfiguracija \n'
if not Err:
  print w, ' je rečenica jezika! Može se dobiti nizom izvođenja:\n'
  Niz_I (L2)
else:
  print 'Niz ', w, ' nije u jeziku!'

return
```

♣ Primjer 4.8

Analizirajmo ulazne nizove kao u primjeru 4.7 s tri ekvivalentne gramatike:

```
ETF.grm          ETF4.grm                      ETF0.grm

E -> T+E | T     E -> CD | a | CDI | aI         E -> E+E | E*E | (E) | a
T -> F*T | F     A -> FE
F -> (E) | a     B -> GE
                 C -> (
                 D -> (DH | aH | (DIH | aIH
                 F -> +
                 G -> *
                 H -> )
                 I -> A | AI | B | BI
```

Rezultati su prikazani u sljedećoj tablici.

i	w_i	BC		
		ETF	ETF4	ETF0
1	(((((((a)))))))	45	390,440	27
2	(a+a)*a	105	346	14
3	(a+a)*(a+a)*a	1944	42,561	25
4	a+a+a+a+a+a+a+a+a	586,527	20,437,971	35
5	a*a*a*a*a*a*a*a*a	977,133	20,437,971	35
6	a+a*a+a*a+a*a+a*a	627,466	20,437,971	35
7	(((a+a)*a)+a)*a	5158	423,440	28
8	((((((a))))))	239	97,917	99

Vidimo da je gramatika ETF0 znatno efikasnija od gramatike ETF, a ona znatno efikasnija od gramatike ETF4. I ne samo to, gramatika ETF0 je najefikasnija u sintaksnoj analizi jezika jednostavnih "aritmetičkih izraza" koje generira. Podrazumijeva se da se može koristiti samo u uzlaznoj sintaksnoj analizi. Postupak je tada bez povratka (jednoprolazan), ako je ulazni niz u jeziku. Na primjer, pogledajmo promjenu konfiguracija u analizi niza w_3:

```
 0 ('q', 1, '$', '')
 1 ('q', 2, '$(', 's')
 2 ('q', 3, '$(a', 'ss')
 3 ('q', 3, '$(E', '04ss')
 4 ('q', 4, '$(E+', 's04ss')
 5 ('q', 5, '$(E+a', 'ss04ss')
 6 ('q', 5, '$(E+E', '04ss04ss')
 7 ('q', 5, '$(E', '0104ss04ss')
 8 ('q', 6, '$(E)', 's0104ss04ss')
 9 ('q', 6, '$E', '03s0104ss04ss')
10 ('q', 7, '$E*', 's03s0104ss04ss')
11 ('q', 8, '$E*(', 'ss03s0104ss04ss')
12 ('q', 9, '$E*(a', 'sss03s0104ss04ss')
13 ('q', 9, '$E*(E', '04sss03s0104ss04ss')
14 ('q', 10, '$E*(E+', 's04sss03s0104ss04ss')
15 ('q', 11, '$E*(E+a', 'ss04sss03s0104ss04ss')
16 ('q', 11, '$E*(E+E', '04ss04sss03s0104ss04ss')
17 ('q', 11, '$E*(E', '0104ss04sss03s0104ss04ss')
18 ('q', 12, '$E*(E)', 's0104ss04sss03s0104ss04ss')
19 ('q', 12, '$E*E', '03s0104ss04sss03s0104ss04ss')
20 ('q', 12, '$E', '0203s0104ss04sss03s0104ss04ss')
21 ('q', 13, '$E*', 's0203s0104ss04sss03s0104ss04ss')
22 ('q', 14, '$E*a', 'ss0203s0104ss04sss03s0104ss04ss')
23 ('q', 14, '$E*E', '04ss0203s0104ss04sss03s0104ss04ss')
24 ('q', 14, '$E', '0204ss0203s0104ss04sss03s0104ss04ss')
25 ('t', 14, '$E', '0204ss0203s0104ss04sss03s0104ss04ss')
```

Pitanja i zadaci

1) *Definirajte gramatiku iz uvodnog primjera silazne sintaksne analize, učitajte je parserom* **TopDown.py** *i analizirajte ulazni niz* aacbc *te usporedite ispisane konfiguracije s izvođenjem danim na stranicama 52 i 53.*

2) *Definirajte gramatiku cijelih brojeva i ispišite, bez upotrebe programa silazne sintaksne analize, prvih deset konfiguracija ako je ulazni niz* -54321. *Potom provjerite rezultat primjenom parsera* **TopDown.py**.

3) *Usporedite vremena izvršavanja postupka silazne analize ako je ulazna gramatika kao u primjeru 4.3 mijenjajući ulazni niz* $w=({}^{n}a)^{n}$ *za* $n=1,2,...,20$. *Što zaključujete?*

4) *Usporedite silaznu i ulaznu sintaksnu analizu za isti ulazni niz koristeći dvije ekvivalentne gramatike, gramatiku iz primjera 4.4 za silaznu, a gramatiku s produkcijama:*

   $E \rightarrow E+E \mid E*E \mid (E) \mid a$

   za uzlaznu sintaksnu analizu. Što zaključujete? Zašto?

5) *Dan je stogovni automat* $R=(\{q,r\},\Sigma,\Gamma,\delta,q,\$,\{r\})$, *gdje je funkcija prijelaza* δ *definirana sa:*

```
(1) δ(q,α,ε)          = {(q,α)}  za α∈{a,+,*,(,)}
(2) δ(q,ε,E+T)        = {(q,E)}
    δ(q,ε,T)          = {(q,E)}
    δ(q,ε,T*F)        = {(q,T)}
    δ(q,ε,F)          = {(q,T)}
    δ(q,ε,(E))        = {(q,F)}
    δ(q,ε,a)          = {(q,F)}
(3) δ(q,ε,$E)         = {(r,ε)}
```

Pokazati na primjeru ulaznog niza a+a*a *da* R *radi kao uzlazni postupak sintaksne analize.*

5.
TABLIČNI POSTUPCI PARSIRANJA

CYK

```
E -> TA| FB| CD| a   ( 1), ( 2), ( 3), ( 4)
T -> FB| CD| a       ( 5), ( 6), ( 7)
F -> CD| a           ( 8), ( 9)
A -> GE              (10)
B -> HT              (11)
C -> (               (12)
D -> EI              (13)
G -> +               (14)
H -> *               (15)
I -> )               (16)
```

```
Pi = [2, 8, 12, 13, 1, 7, 10,
      14, 4, 16, 11, 15, 7]

E => FB     => CDB     => (DB
  => (EIB   => (TAIB   => (aAIB
  => (aGEIB => (a+EIB  => (a+aIB
  => (a+a)B => (a+a)HT => (a+a)*T
  => (a+a)*a
```

7	T,E						
6							
5	F,T,E						
4		D					
3		E					
2			A	D		B	
j↑ 1	C	E,T,F	G	E,T,F	I	H	E,T,F
i→	1	2	3	4	5	6	7

Earley

```
E → T + E  (1)
E → T      (2)
T → F * T  (3)
T → F      (4)
F → (E)    (5)
F → a      (6)

Pi = [6, 4, 6, 4, 2,
    1, 5, 6, 4, 3, 2]
E
 => T => F*T => F*F
 => F*a => (E)*a
 => (T+E)*a
 => (T+T)*a
 => (T+F)*a
 => (T+a)*a
 => (F+a)*a
 => (a+a)*a
```

I_0
[E → •T+E, 0]
[E → •T, 0]
[T → •F*T, 0]
[T → •F, 0]
[F → •(E), 0]
[F → •a, 0]

I_1
[F → (•E), 0]
[E → •T+E, 1]
[E → •T, 1]
[T → •F*T, 1]
[T → •F, 1]
[F → •(E), 1]
[F → •a, 1]

I_2
[F → a•, 1]
[T → F•*T, 1]
[T → F•, 1]
[E → T•+E, 1]
[E → T•, 1]
[F → (E•), 0]

I_3
[E → T+•E, 1]
[E → •T+E, 3]
[E → •T, 3]
[T → •F*T, 3]
[T → •F, 3]
[F → •(E), 3]
[F → •a, 3]

I_4
[F → a•, 3]
[T → F•*T, 3]
[T → F•, 3]
[E → T•+E, 3]
[E → T•, 3]
[E → T+E•, 1]
[F → (E•), 0]

I_5
[F → (E)•, 0]
[T → F•*T, 0]
[T → F•, 0]
[E → T•+E, 0]
[E → T•, 0]

I_6
[T → F*•T, 0]
[T → •F*T, 6]
[T → •F, 6]
[F → •(E), 6]
[F → •a, 6]

I_7
[F → a•, 6]
[T → F•*T, 6]
[T → F•, 6]
[T → F*T•, 0]
[E → T•+E, 0]
[E → T•, 0]

Osim dvaju općenitih postupka sintaksne analize (parsiranja) danih u prethodnom poglavlju ovdje dajemo još dva, koji pripadaju tzv. "tabličnim postupcima parsiranja". To su Cocke-Younger-Kasamijeva i Earleyjeva metoda. Oba postupka zahtijevaju oko n^3 vremena za analizu ulaznog niza duljine n.

5.1 COCKE-YOUNGER-KASAMIJEV ALGORITAM (CYK)

Vidjeli smo da silazni i uzlazni povratni postupak sintaksne analize imaju vrijeme trajanja analize koje raste eksponencijalno s povećanjem duljine ulaznog niza. Cocke-Younger-Kasamijev (CYK) algoritam, međutim, treba n^3 vremena i n^2 memorijskog prostora. Ovaj je postupak sintaksne analize primjenljiv samo za beskontekstne jezike generirane gramatikama čije su produkcije u "Chomskyjevoj normalnoj formi" (CNF). Neformalno, postupak je sljedeći:

Neka je $G=(N, T, P, S)$ gramatika u Chomskyjevoj normalnoj formi (CNF), bez ε–produkcija. Neka je $w=a_1...a_n$ ulazni niz kojeg treba analizirati. Pretpostavimo da je svaki a_i u T, $1\le i\le n$. Bit CYK postupka jest da se izgradi tablica sintaksne analize **T** čiji će elementi biti označeni s t_{ij}, $1\le i\le n$ i $1\le j\le n-i+1$. Svaki t_{ij} ima vrijednost koja je podskup od N. Neterminal A će biti u t_{ij} ako i samo ako je $A^+\Rightarrow a_i...a_{i+j-1}$, tj. ako A izvodi j ulaznih simbola koji počinju od pozicije i. U posebnom slučaju, w je u $L(G)$ ako i samo ako je S u t_{1n}.

Dakle, da bi se utvrdilo da je niz w u jeziku mora se izgraditi tablica **T** za w i provjeriti je li S na mjestu t_{1n}. Potom, ako želimo niz izvođenja koji završava s w, moramo upotrijebiti tu tablicu. Ovdje će prvo biti dan algoritam 5.1 koji će izgraditi tablicu, a algoritam 5.2 "čita" niz izvođenja.

♥ **Algoritam 5.1** Cocke-Younger-Kasamiev postupak parsiranja (CYK).

Ulaz

Gramatika $G=(N, T, P, S)$ u Chomskyjevoj normalnoj formi, bez ε–produkcija i ulazni niz $w=a_1...a_n$, $w\in T^+$.

Izlaz

Tablica sintaksne analize **T** za w tako da t_{ij} sadrži A ako i samo ako je $A^+\Rightarrow a_i...a_{i+j-1}$.

Postupak

(1) Staviti $t_{i1}=\{A: A\to a_i \text{ u } P\}$ za svaki i. Poslije ovog koraka, ako t_{i1} sadrži A, sigurno postoji niz izvođenja $A^+\Rightarrow a_i$.

(2) Pretpostavimo da je $t_{ij'}$ bilo izračunato za sve i, $1\le i\le n$, i sve j', $1\le j'\le j$. Staviti:

t_{ij}={A: za neki k, $1 \le k < j$, A→BC je u $\mathcal{P}$, B je u t_{ik}, C je u $t_{i+k,j-k}$}

Budući da je $1 \le k < j$, i k i $j-k$ su manji od j. Dakle, i t_{ik} i $t_{i+k,j-k}$ bili su izračunati prije nego je t_{ij} bilo izračunato. Poslije ovoga koraka ako t_{ij} sadrži A, tada je:

$$A \Rightarrow BC \overset{+}{\Rightarrow} a_i \ldots a_{i+k-1} C \overset{+}{\Rightarrow} a_i \ldots a_{i+k-1} a_{i+k} \ldots a_{i+j-1}$$

(3) Ponavljati korak (2) sve dok t_{ij} ne postane poznato za sve $1 \le i \le n$ i $1 \le j \le n-i+1$.

♣ Primjer 5.1

Razmotrimo gramatiku $\mathcal{G}$ s produkcijama:

S → AA| AS| b
A → SA| AS| a

Ulazni je niz abaab. Primjenjujući algoritam 5.1, tablica sintaksne analize je:

5	A,S				
4	A,S	A,S			
3	A,S	S	A,S		
2	A,S	A	S	A,S	
j↑ 1	A	S	A	A	S
i→	1	2	3	4	5

Iz koraka (1) je t_{11}={A} budući je A→a u $\mathcal{P}$ i a_1=a. U koraku (2) se dodaje S u t_{32}, budući je S→AA u $\mathcal{P}$ i A je i u t_{31} i t_{41}. Primijetiti, općenito, da ako su t_{ij} prikazani kao što je pokazano, možemo izračunati t_{ij}, $j>1$, ispitujući neterminale u sljedećim parovima ulaza:

$$(t_{i1}, t_{i+1,j-1}), (t_{i2}, t_{i+2,j-2}), \ldots, (t_{i,j-1}, t_{i+j-1,1})$$

Tada, ako je B u t_{ik} i C u $t_{i+k,j-k}$ za neki k tako da je $1 \le k < j$ i A→BC je u $\mathcal{P}$, treba dodati A u t_{ij}. Budući je S u t_{1s}, abaab je u jeziku $\mathcal{L}(\mathcal{G})$.

U sljedećem je algoritmu opisano kako se iz tablice sintaksne analize može dobiti sintaksna analiza slijeva.

♥ Algoritam 5.2 *Parsiranje slijeva iz CYK tablice sintaksne analize.*

Ulaz

Gramatika $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$ u Chomskyjevoj normalnoj formi, u kojoj su produkcije numerirane od 1 do p, ulazni niz $w=a_1 \ldots a_n$ i tablica sintaksne analize **T** izgrađena algoritmom 5.1.

Izlaz

Sintaksna analiza slijeva (niz lijevih rečeničnih formi) ili "pogreška".

Postupak

Najprije opišimo rekurzivnu proceduru **gen**(i,j,A) koja će generirati sintaksnu analizu slijeva sukladnu izvođenju $A \overset{+}{\Rightarrow} a_i a_{i+1} \ldots a_{i+j-1}$, također slijeva. Procedura **gen**(i,j,A) definirana je kako slijedi:

(1) Ako je `j=1` i `m`-ta produkcija u $\mathcal{P}$ jest $A \to a_i$, ispisati broj produkcije `m`.
(2) Ako je `j>1`, `k` je najmanji cijeli broj, $1 \le k < j$, tako da je za neki `B` u t_{ik} i `C` u $t_{i+k,j-k}$, $A \to BC$ produkcija u $\mathcal{P}$, recimo numerirana s `m` (može biti više izbora za $A \to BC$, ali ćemo uzeti prvu). Ispisati broj `m` i izvršiti **gen**(i,k,B), potom **gen**(i+k,j-k,C).

Postupak počinje s **gen**(1,n,S), ispitujući je li S u t_{1n}. Ako nije, dojavljuje se pogreška, ako jest, nastavlja se na opisani način.

♣ Primjer 5.2

Razmotrimo gramatiku $\mathcal{G}$ iz primjera 5.1. Numerirajmo produkcije:

```
(1) S → AA
(2) S → AS
(3) S → b
(4) A → SA
(5) A → AS
(6) A → a
```

Ulazni je niz abaab, kao u primjeru 5.1, pa je tablica sintaksne analize također kao u primjeru 5.1. Budući da je S u t_{15}, ulazni je niz u jeziku $\mathcal{L}(\mathcal{G})$. Da bismo našli sintaksnu analizu slijeva, poziva se procedura **gen**(1,5,S). Pronađeno je A u t_{11} i u t_{24}, te produkcija $S \to AA$ u skupu produkcija. Dakle, ispisuje se 1 (broj produkcije $S \to AA$).

Dalje se poziva **gen**(1,1,A) i **gen**(2,4,A). **gen**(1,1,A) daje produkciju broj 6. Budući da je S u t_{21} i A je u t_{33} i $A \to SA$ je četvrta produkcija, **gen**(2,4,A) ispisuje 4 i poziva **gen**(2,1,S), pa **gen**(3,3,A). Nastavljajući, na kraju bismo dobili 164356263. Dakle, rečenica abaab dobiva se u nizu izvođenja:

$$S \Rightarrow AA \Rightarrow aA \Rightarrow aSA \Rightarrow abA \Rightarrow abAS \Rightarrow abaS \Rightarrow abaAS \Rightarrow abaaS \Rightarrow abaab$$

Budući da je gramatika $\mathcal{G}$ dvoznačna, moguće je dobiti više sintaksnih analiza slijeva, što je ovdje izostavljeno.

Ustroj algoritma CYK sintaksne analize dan je u drugom dijelu ovoga poglavlja.

♣ Primjer 5.3

CYK sintaksna analiza nije primjenljiva za gramatiku `ETF.grm`,

```
E -> T+E | T
T -> F*T | F
F -> (E) | a
```

pa je prvo transformirajmo u ekvivalentnu gramatiku `ETF-CNF.GRM` (u `CNF`-u) i uredimo produkcije:

```
E -> TA | FB | CD | a  ( 1), ( 2), ( 3), ( 4)
T -> FB | CD | a       ( 5), ( 6), ( 7)
F -> CD | a            ( 8), ( 9)
A -> GE                (10)
B -> HT                (11)
C -> (                 (12)
D -> EI                (13)
G -> +                 (14)
H -> *                 (15)
I -> )                 (16)
```

Ako je ulazni niz `(a+a)*a`, tablica sintaksne analize je (program **`CYK.py`**):

```
7      T,E
6
5    F,T,E
4                D
3                E
2                          A        D                B
1        C    E,T,F        G    E,T,F        I        H    E,T,F
-------------------------------------------------------------------
         1        2        3        4        5        6        7
```

lijevo parsiranje je

```
Pi = [2, 8, 12, 13, 1, 7, 10, 14, 4, 16, 11, 15, 7]

E
  => FB
  => CDB
  => (DB
  => (EIB
  => (TAIB
  => (aAIB
  => (aGEIB
  => (a+EIB
  => (a+aIB
  => (a+a)B
  => (a+a)HT
  => (a+a)*T
  => (a+a)*a
```

5.2 EARLEYJEV POSTUPAK PARSIRANJA

Završavamo ovo poglavlje s još jednom tabličnom metodom sintaksne analize poznatom kao "Earleyjev postupak sintaksne analize (parsiranja)". Neformalno, postupak je sljedeći. Neka je $G=(\mathcal{N},\mathcal{T},\mathcal{P},S)$ beskontekstna gramatika i $w=a_1\ldots a_n$ ulazni niz, $w\in\mathcal{T}^*$. Objekt oblika

$$[A \rightarrow X_1X_2\ldots X_k \bullet X_{k+1}\ldots X_m,i]$$

naziva se <u>stavka</u> za w ako je $A\rightarrow X_1X_2\ldots X_m$ produkcija u $\mathcal{P}$ i $0\le i\le n$. Točka između X_k i X_{k+1} je metasimbol i nije u $\mathcal{N}\cup\mathcal{T}$. Cijeli broj k može biti bilo koji broj od 0 (tada je • prvi simbol) do m (tada je • posljednji simbol). Ako je produkcija $A\rightarrow\varepsilon$, tada je stavka jednaka $[A\rightarrow\bullet,i]$.

Za svaki j, $0\le j\le n$, treba izgraditi listu stavaka $\mathbf{I}_j$ tako da je $[A\rightarrow\alpha\bullet\beta,i]$ u $\mathbf{I}_j$, za $0\le i\le j$, ako i samo ako za neke γ i δ vrijedi $S^*\Rightarrow\gamma A\delta$, $\gamma^*\Rightarrow a_1\ldots a_i$ i $\alpha^*\Rightarrow a_{i+1}\ldots a_j$. Dakle, druga komponenta stavke i broj liste u kojoj se pojavljuje stavljaju u zagradu dio ulaznog niza izvedenog iz α. Drugi uvjeti stavke samo nas osiguravaju od mogućnosti da se produkcija $A\rightarrow\alpha\beta$ može upotrijebiti na način kako se pojavljuje u nekom ulaznom nizu konzistentnom s w do pozicije j.

Niz lista $\mathbf{I}_0, \mathbf{I}_1, ..., \mathbf{I}_n$ može se nazvati lista sintaksne analize ulaznog niza w. Primijetimo da je w u jeziku $\mathcal{L}(\mathcal{G})$ ako i samo ako postoji stavka oblika $[\mathcal{S}\rightarrow\alpha\bullet,0]$ u $\mathbf{I}_n$.

♥ Algoritam 5.3 *Earleyjev postupak parsiranja.*

Ulaz
Beskontekstna gramatika $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ i ulazni niz $w=a_1...a_n$ iz $\mathcal{T}^*$.

Izlaz
Sintaksna analiza slijeva (niz lijevih rečeničnih formi) ili "pogreška".

Postupak
Prvo se konstruira $\mathbf{I}_0$ na sljedeći način:

(1) Ako je $\mathcal{S}\rightarrow\alpha$ produkcija u $\mathcal{P}$, dodati $[\mathcal{S}\rightarrow\bullet\alpha,0]$ u $\mathbf{I}_0$. Potom izvršiti korak (2) i (3) sve dok se nova stavka može dodati u $\mathbf{I}_0$.
(2) Ako je $[B\rightarrow\gamma\bullet,0]$ u $\mathbf{I}_0$, gdje γ može biti i ε, što znači da se ovaj korak bio izvršio inicijalno, dodati $[A\rightarrow\alpha B\bullet\beta,0]$ u $\mathbf{I}_0$ za sve $[A\rightarrow\alpha\bullet B\beta,0]$.
(3) Pretpostavimo da je $[A\rightarrow\alpha\bullet B\beta,0]$ stavka u $\mathbf{I}_0$. Dodati u $\mathbf{I}_0$, za sve produkcije iz $\mathcal{P}$ oblika $B\rightarrow\gamma$, stavku $[B\rightarrow\bullet\gamma,0]$ (uz pretpostavku da ta stavka nije već bila u $\mathbf{I}_0$). Sada se može konstruirati $\mathbf{I}_j$ iz $\mathbf{I}_0, \mathbf{I}_1, ..., \mathbf{I}_{j-1}$.
(4) Za svaki $[B\rightarrow\alpha\bullet a\beta,i]$ u $\mathbf{I}_{j-1}$ tako da je $a=a_j$ dodati $[B\rightarrow\alpha a\bullet\beta,i]$ u $\mathbf{I}_j$. Potom izvršavati korake (5) i (6) sve dok se može dodati nova stavka.
(5) Neka je $[A\rightarrow\alpha\bullet,i]$ stavka u $\mathbf{I}_j$. Provjeriti nalaze li se stavke oblika $[B\rightarrow\alpha\bullet A\beta,k]$ u $\mathbf{I}_i$. Za sve nađene stavke dodati $[B\rightarrow\alpha A\bullet\beta,k]$ u $\mathbf{I}_j$.
(6) Neka je $[A\rightarrow\alpha\bullet B\beta,i]$ stavka u $\mathbf{I}_j$. Za sve $B\rightarrow\gamma$ u $\mathcal{P}$ dodati $[B\rightarrow\bullet\gamma,j]$ u $\mathbf{I}_j$.

Primijetiti da se pojavom stavke s terminalom desno od točke ne tvori nova stavka u koracima (2), (3), (5) i (6). Algoritam, dakle, tvori $\mathbf{I}_j$ za $0\leq j\leq n$.

♣ Primjer 5.4

Primijenimo algoritam 5.3 u sintaksnoj analizi jezika jednostavnih aritmetičkih izraza generiranog gramatikom `Exp.GRM`:

```
E → T + E   (1)
E → T       (2)
T → F * T   (3)
T → F       (4)
F → (E)     (5)
F → a       (6)
```

i neka je `(a+a)*a` ulazni niz. Iz koraka (1) dodajemo dvije nove stavke, $[E\rightarrow\bullet T+E,0]$ i $[E\rightarrow\bullet T,0]$ u $\mathbf{I}_0$. Te će stavke biti uzete u obzir pri dodavanju stavki $[T\rightarrow\bullet F*T,0]$ i $[T\rightarrow\bullet F,0]$ u $\mathbf{I}_0$ prema pravilu (3). Nastavljajući, dodaju se $[F\rightarrow\bullet(E),0]$ i $[F\rightarrow\bullet a,0]$. Više se nijedna stavka ne može dodati u $\mathbf{I}_0$.

Sada gradimo $\mathbf{I}_1$. Prema (4) dodajemo $[F\rightarrow(\bullet E),0]$, budući da je $a_1=(\bullet$. Potom se, prema pravilu (6), dodaju $[E\rightarrow\bullet T+E,1]$, $[E\rightarrow\bullet T,1]$, $[T\rightarrow\bullet F*T,1]$, $[T\rightarrow\bullet F,1]$, $[F\rightarrow\bullet(E),1]$ i $[F\rightarrow\bullet a,1]$. Sada se više nijedna stavka ne može dodati u $\mathbf{I}_1$.

Da bismo izgradili $\mathbf{I}_2$, primijetimo da je $a_2=a$ i da se prema pravilu (4) stavka $[F\rightarrow a\bullet,1]$ može dodati u $\mathbf{I}_2$. Dalje, prema pravilu (5), promatramo tu stavku odlazeći u $\mathbf{I}_1$ i tražeći stavke s F koje slijedi

točka. Pronalazimo dvije stavke, [T→F•*T,1] i [T→F•,1], i dodajemo ih u $\mathbf{I}_2$. Promatrajući prvu od njih, ništa, ali druga nas navodi da ponovno pregledamo $\mathbf{I}_1$, ovog puta stavke sa •T u sebi. Najviše se dvije stavke mogu dodati u $\mathbf{I}_2$, [E→T•+E,1] i [E→T•,1]. Opet druga stavka uzrokuje da se [F→(E•),0] može dodati u $\mathbf{I}_2$. Sada se više nijedna stavka ne može dodati u $\mathbf{I}_2$. Nastavljajući, na kraju bismo dobili liste (program **Earley.py**):

```
I0                  I1                  I2                  I3
[E → •T+E, 0]       [F → (•E), 0]       [F → a•,   1]       [E → T+•E, 1]
[E → •T,   0]       [E → •T+E, 1]       [T → F•*T, 1]       [E → •T+E, 3]
[T → •F*T, 0]       [E → •T,   1]       [T → F•,   1]       [E → •T,   3]
[T → •F,   0]       [T → •F*T, 1]       [E → T•+E, 1]       [T → •F*T, 3]
[F → •(E), 0]       [T → •F,   1]       [E → T•,   1]       [E → •F,   3]
[F → •a,   0]       [F → •(E), 1]       [F → (E•), 0]       [F → •(E), 3]
                    [F → •a,   1]                           [F → •a,   3]

I4                  I5                  I6                  I7
[F → a•,   3]       [F → (E)•, 0]       [T → F*•T, 0]       [F → a•,   6]
[T → F•*T, 3]       [T → F•*T, 0]       [T → •F*T, 6]       [T → F•*T, 6]
[T → F•,   3]       [T → F•,   0]       [T → •F,   6]       [T → F•,   6]
[E → T•+E, 3]       [E → T•+E, 0]       [F → •(E), 6]       [T → F*T•, 0]
[E → T•,   3]       [E → T•,   0]       [F → •a,   6]       [E → T•+E, 0]
[E → T+E•, 1]                                               [E → T•,   0]
[F → (E•), 0]
```

Budući da je [E→T•,0] na kraju liste, niz (a+a)*a je u jeziku $\mathcal{L}(\mathcal{G})$.

♥ Algoritam 5.4 *Izvođenje desnog parsanja iz lista stavaka Earleyjeve SA*

Ulaz

Beskontekstna gramatika $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ bez ciklusa, ulazni niz $w=a_1\ldots a_n$ iz $\mathcal{T}^*$ i liste stavaka $\mathbf{I}_0,\mathbf{I}_1,\ldots,\mathbf{I}_n$ za w.

Izlaz

π, globalna varijabla – desno parsanje za w, ili poruka "pogreška" ako ulazni niz nije rečenica jezika generiranog gramatikom $\mathcal{G}$.

Postupak

Ako ne postoji stavka $[\mathcal{S}\to\alpha\bullet,0]$ u listi $\mathbf{I}_n$, tada w nije u $\mathcal{L}(\mathcal{G})$, dojavljuje se "pogreška" i prekida postupak. Inače, inicijalizirati π s ε i izvesti proceduru $\mathbf{R}([\mathcal{S}\to\alpha\bullet,0],n)$ gdje je procedura

$\mathbf{R}([A\to\beta\bullet,i],j)$

definirana kao što slijedi:

1) Neka je π jednako h kojeg slijedi prethodna vrijedost od π, gdje je h broj produkcije $A\to\beta$.
2) Ako je $\beta=X_1X_2\ldots X_m$, postaviti $k=m$ i $l=j$.
3) (a) Ako je $X_k\in\mathcal{T}$, umanjiti k i l za 1.
   (b) Ako je $X_k\in\mathcal{N}$, pronaći stavku $[X_k\to\gamma\bullet,r]$ u I_l za neki r tako da je

   $[A\to X_1X_2\ldots X_{k-1}\bullet X_k\ldots X_m,i]$

u I_r. Tada izvršiti $\mathbf{R}([X_k \to \gamma\ \bullet, r], l)$. Umanjiti k za 1 i postaviti $l=r$.

4) Ponavljati korak (3) sve dok k ne postane jednako 0. Stati.

Earleyjev postupak sintaksne analize, algoritme 5.3 i 5.4, realizirali smo u programu danom na kraju poglavlja.

♣ Primjer 5.5

Evo rezultata sintaksne analize niza (a+a)*a iz primjera 5.4:

```
R ([E->T·,0], 7)
R ([T->F*T·,0], 7)
R ([T->F·,6], 7)
R ([F->a·,6], 7)
R ([F->(E)·,0], 5)
R ([E->T+E·,1], 4)
R ([E->T·,3], 4)
R ([T->F·,3], 4)
R ([F->a·,3], 4)
R ([T->F·,1], 2)
R ([F->a·,1], 2)

Pi = [6, 4, 6, 4, 2, 1, 5, 6, 4, 3, 2]

E
  => T
  => F*T
  => F*F
  => F*a
  => (E)*a
  => (T+E)*a
  => (T+T)*a
  => (T+F)*a
  => (T+a)*a
  => (F+a)*a
  => (a+a)*a
```

P R O G R A M I

Slijedi ustroj algoritama CYK i Earlaye sintaksne analize u Pythonu.

🖫 CYK.py *CYK parser*

```
def CYK (G, w):
  Net, Ter, P, S = G;  P2 = Uredi_P(P);  Tcyk = [];  O = chr(183)
  P2 = Uredi_P (P);  M = len(P2);

  def Ispisi_T (): # Ispis tablice
    for j in range (N-1, -1, -1):
      print j+1,
      for i in range (N-j): print '%8s' % Tcyk[j][i],
      print
    print '--' +'---------'*N
    print ' ',
    for i in range (N): print '%8d' %(i+1),
    print NL
```

```
def gen (i, j, A, Pi = []):
  if j==0:
    L = [A, w[i]];  p = P2.index(L) +1;  Pi.append (p)
    return Pi
  else:
    Ok = False; k = 1
    while not Ok and k < j+1:
      x = Tcyk[k-1][i];  y = Tcyk[j-k][i+k]
      p = 0
      while not Ok and p < len(x):
        q = 0
        while not Ok and q < len(y):
          B = x[p]; C = y[q];  BC = B +C;  L = [A, BC]
          Ok = L in P2
          q += 2
        p += 2
      k += 1
    Pi.append (P2.index(L) +1); k -= 1
    Pi = gen (i, k-1, B, Pi);  Pi = gen (i+k, j-k, C, Pi)
    return Pi

Ok = True;  i = 0
"Provjera ulazne gramatike"
while Ok and i < M:
  alfa = P2[i][1]
  Ok = Ok and (len(alfa) == 2 and alfa[0] in Net and alfa[1] in Net or
               len(alfa) == 1 and alfa in Ter)
  i += 1

if not Ok :
  print 'Gramatika nije u CNF-u!'
  return False, Tcyk

N = len(w); i = 0
for j in range (N):
  Tcyk.append ([])
  for i in range (N-j): Tcyk[j].append('')

'(1)'
j = 0
for i in range (N):
   c = w[i];  k = 0
   while k < M:
     A = P2[k][0]
     if c == P2[k][1]:
       if Tcyk[j][i] == '': Tcyk[j][i] = A
       else               : Tcyk[j][i] += ',' +A
     k += 1

'(2)'
for j in range (1, N):
  for i in range (N-j):
    for k in range (1, j+1):
      x = Tcyk[k-1][i];  y = Tcyk[j-k][i+k]
      for p in range (0, len(x), 2):
        for q in range (0, len(y), 2):
          B = x[p]; C = y[q];  BC = B +C
```

```
            for m in range(M-1,-1,-1):
              A = P2[m][0]
              if P2[m][1] == BC:
                if A not in Tcyk[j][i]:
                  if Tcyk[j][i] == '': Tcyk[j][i] = A
                  else               : Tcyk[j][i] += ',' +A


  Ispisi_T ()
  Ok = S in Tcyk[N-1][0]

  if not Ok: print w, ' NIJE rečenica jezika! \n';  return

  Pi = [];  Pi = gen (0, N-1, S)
  print w, ' je rečenica jezika! Može se dobiti nizom izvođenja:\n'
  print 'Pi =', Pi, '\n'
  LSF (Pi, P2)

  return
```

Earley.py *Earleyjev parser*

```
def Earley (G, w):
  N, T, P, S = G;  I = [[]];  O = chr(183)
  P2 = Uredi_P (P);  m = len(P2);  n = len(w)

  def Ispisi_I ():
    for i in range (len(I)):
      print 'I', i
      for j in range (len(I[i])):
        L = I[i][j]
        print '[', L[0], '->', '%6s' %L[1] +',', L[2], ']'
      print
    return

  def Dodaj_St (L, q):
    if L not in I[q] : I[q].append (L)

  def R (Ij, j, Pi = []): # Izvođenje desnog parsanja
    '(1)'
    A, beta, i = Ij
    print 'R ([' +A +'->' +beta +',' +str(i)+'],', str(j) +')'
    beta = beta [:-1]
    L    = [A, beta]
    Pi   = [P2.index(L) +1] +Pi

    '(2)'
    k = len(beta) -1; l = j
    while k > -1:
      if beta[k] in T:
        '(3a)'
        k -= 1; l -= 1
      else:
        '(3b)'
        Xk = beta[k];  alfa = beta[:k] +O +beta[k:];  p = 0;  Ok = False
        while not Ok and p < len(I[l]):
          B, gama, r = I[l][p]
```

```
            if B == Xk and O == gama[-1]:
              q = 0
              while not Ok and q < len(I[r]):
                X, x, j = I[r][q];  Ok = X == A and alfa == x
                q += 1
            p +=1
          Pi = R ([Xk, gama, r], l, Pi);  k -= 1;  l = r
      return Pi

    '(1)'
    for i in range (m):
      if P2[i][0] == S: I[0].append ([S, O+P2[i][1], 0])

    '(3)'
    i = 0
    while i < len(I[0]):
      x = I[0][i][1];  j = pos (O, x)
      if j != -1:
        j = pos(x[j+1], N)
        if j != -1:
          B = N[j]
          for k in range (m):
            if P2[k][0] == B: Dodaj_St ([B, O+P2[k][1], 0], 0)
      i += 1

    '(4)'
    for i in range (len(w)):
      a = w[i];  j = i+1; I.append([]);  k = 0
      while k < len(I[j-1]):
        B, x, y = I[j-1][k];  l = x.find (O+a)
        if l != -1:  x = x.replace(O+a, a+O);  Dodaj_St ([B, x, y], j)
        k += 1

      '(5)'
      k = 0
      while k < len(I[j]):
        A, x, y = I[j][k]
        if O == x[-1]:
          for l in range (len(I[y])):
            B, x1, y1 = I[y][l]
            if x1.find (O) != -1:
              p = x1.find(O)
              if p < len(x1)-1:
                C  = x1[p+1]
                if A == C: x1 = x1.replace (O+C, C+O); Dodaj_St ([B, x1, y1], j)
        k += 1

      '(6)'
      k = 0
      while k < len(I[j]):
        A, x, y = I[j][k];  l = pos (O, x)
        if l != -1 and O != x[-1]:
          if x[l+1] in N:
            A = x[l+1]
            for p in range(len (P2)):
              B, gama = P2[p]
              if B == A: Dodaj_St ([B, O+gama, j], j)
        k += 1
```

```
k = 0;  Ok = False
while k < len (I[-1]) and not Ok :
  A, alfa, i = I[-1][k]
  Ok = A == S and O == alfa[-1] and i == 0
  k += 1
if not Ok :
  print 'Niz ', w, ' nije u jeziku!'
  return

Ispisi_I ()
print; print w, ' je rečenica jezika! Može se dobiti nizom izvođenja: \n'

Pi = R (I[-1][k-1], n)
print '\n', 'Pi =', Pi, '\n'
RSF (Pi, P2)

return
```

Pitanja i zadaci

1) Dana je gramatika G s produkcijama u CNF-U:

A → a| BC
B → b| BB| CC
C → CB| a

Provjerite koji je od ulaznih nizova:

a aaaaaaa bb bc babb babba

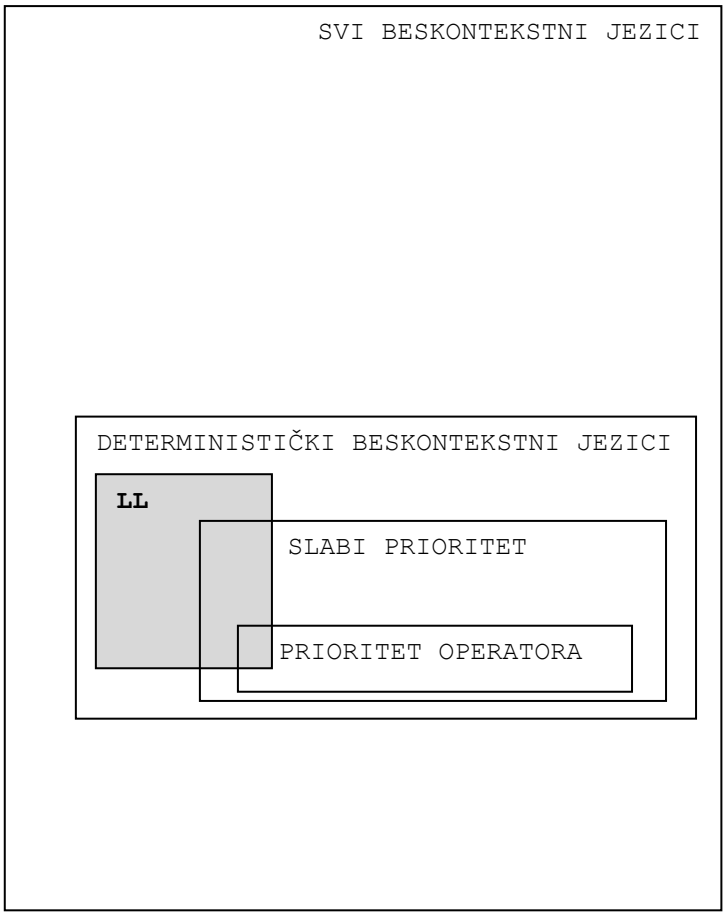
rečenica jezika L(G).

2) Usporedite analizu istih rečenica Earleyjevim parserom koristeći gramatiku Exp.GRM *iz primjera 5.4 i ekvivalentnu gramatiku* Exp0.GRM *s produkcijama:*

```
E -> E+E | E*E | (E) | a
```

Što zaključujete?

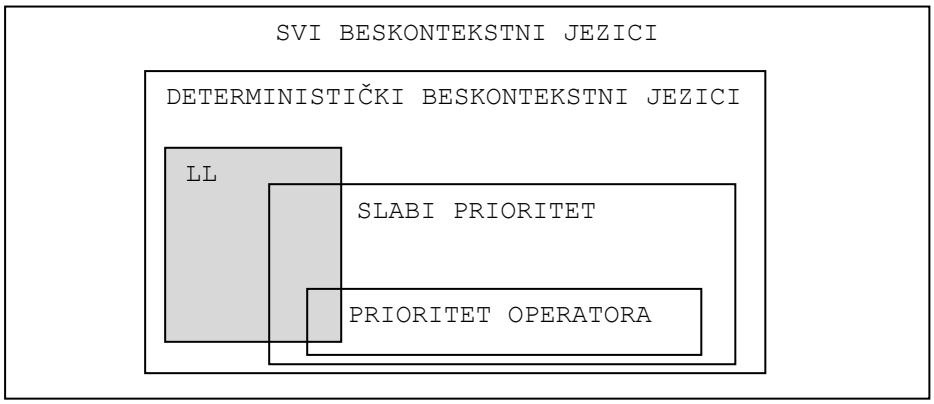
6.
LL(k) JEZICI I SINTAKSNA ANALIZA



U poglavlju "Višeprolazno parsanje" opisane su dvije povratne ("backtrack") metode nedeterminističkog parsanja slijeva i zdesna koje se, uz određene transformacije gramatika, mogu primijeniti na cijeloj klasi beskontekstnih gramatika, odnosno jezika koje one generiraju. Bilo je riječi o ograničenjima primjenljivosti takvih postupaka i o njihovim nedostacima, a glavni im je nedostatak bio vrijeme trajanja (ili broj koraka), posebno ako ulazni niz nije u jeziku.

U ovom će poglavlju biti riječi o klasi beskontekstnih jezika za koje je moguće konstruirati efikasne programe sintaksne analize koji čine $c_1 n$ operacija i koriste $c_2 n$ memorijskog prostora u obradi niza duljine n, gdje su c_1 i c_2 konstante. Takvi postupci rade deterministički, pretražujući ulazni niz samo jedanput. To su jezici (gramatike) tipa LL(k) i LR(k), za koje je moguće konstruirati jednoprolazni postupak (program) sintaksne analize (parsanja) slijeva (za LL) ili zdesna (za LR), koji će raditi deterministički ako im se dopusti da "pogledaju" najviše k ulaznih znakova slijeva nadesno (prvo slovo L u LL i LR to naznačuje) od neke tekuće pozicije, te gramatike s prioritetom operatora za koje je moguće napisati deterministički postupak sintaksne analize upravljan tablicom prioriteta relacije.

Osnovni nedostatak jednoprolaznih postupaka sintaksne analize jest ograničena primjenljivost, nad relativno malom klasom beskontekstnih jezika, sl. 6.1.



Sl. 6.1 – *Hijerarhija beskontekstnih jezika.*

U ovom ćemo poglavlju najprije definirati jezik tipa LL(k). Potom ćemo opisati postupke njegove sintaksne analize.

6.1 JEZICI TIPA *LL*(*k*)

Počinjemo s najvećom "prirodnom" klasom jezika za koje je moguća (silazna) analiza slijeva. To su *LL*(*k*) jezici. Najprije dajemo definiciju generatora tih jezika - gramatika tipa *LL*(*k*).

Definicija gramatike tipa *LL*(*k*)

Neka je $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ jednoznačna gramatika i $w=a_1a_2...a_n$ rečenica jezika $\mathcal{L}(\mathcal{G})$. Tada postoji jedinstven niz lijevih rečeničnih formi $\alpha_0,\alpha_1,...,\alpha_m$ tako da vrijedi:

$$\mathcal{S} = \alpha_0, \qquad \alpha_i \overset{p_i}{\underset{lm}{\Rightarrow}} \alpha_{i+1}$$

za $0 \le i < m$ i $\alpha_m = w$. Sintaksna analiza slijeva (lijevo parsanje) za w je $p_0p_1...p_{m-1}$. Pretpostavimo da je potrebno naći taj niz lijevih rečeničnih formi pretražujući ulazni niz w slijeva nadesno samo jednom. To se može pokušati uraditi konstruirajući niz lijevih rečeničnih formi. Ako je

$$\alpha_i = a_1...a_jA\beta$$

tada na tom mjestu treba učitati prvih j znakova ulaznog niza i usporediti ih s prvih j znakova od α_i. Bilo bi dobro kad bi se α_{i+1} moglo odrediti znajući samo $a_1...a_j$ (dio ulaznog niza koji treba pretražiti do tog mjesta), nekoliko slijedećih ulaznih znakova ($a_{j+1}a_{j+2}...\ a_{j+k}$ za neki fiksni k) i neterminal A. Ako te tri stvari jedinstveno određuju produkciju koja će biti upotrijebljena za ekspanziju neterminala A, može se precizno odrediti α_{i+1} iz α_i i k ulaznih simbola $a_{j+1}a_{j+2}...\ a_{j+k}$.

Gramatika u kojoj svako krajnje izvođenje slijeva ima to svojstvo je tipa *LL*(*k*). Ovdje prvi "*L*" označuje da se ulazni niz pretražuje slijeva, a drugi "*L*" da se pri tome izvode lijeve rečenične forme, koristeći k tekućih znakova za donošenje odluke koja će produkcija biti upotrijebljena. Vidjet ćemo da je za gramatike tipa *LL*(*k*) moguće konstruirati deterministički postupak sintaksne analize. Prije potpune definicije gramatike tipa *LL*(*k*) dajemo definiciju skupa FIRST.

♦ Međa

Neka je $\alpha=x\beta$ lijeva rečenična forma gramatike $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ tako da je $x\in\mathcal{T}^*$, a β počinje ili neterminalom ili je jednako ε. Kažemo da je x zatvoreni dio od α. Granica između x i β jest međa.

♣ Primjer 6.1

Neka je α=abacAaB. Zatvoreni dio od α jest abac, otvoreni AaB. Ako je α=abc, abc je zatvoreni dio, ε otvoreni. Međa je na desnom kraju.

♦ FIRST_k

Neka je $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ beskontekstna gramatika i neka su $\alpha,\beta\in(\mathcal{N}\cup\mathcal{T})^*$. Definira se:

$$\text{FIRST}_k(\alpha) = \{x:\ \alpha^*\Rightarrow x\beta \text{ i } |x|=k, \text{ ili } \alpha^*\Rightarrow x \text{ i } |x|\le k\}$$

tj. $\text{FIRST}_k(\alpha)$ se sastoji od svih prefiksnih terminala duljine k (ili manje, ako α izvodi niz terminala duljine manje od k), niza terminala koji može biti izveden iz α. Ako je $\text{FIRST}_k(\alpha)=\{w\}, w\in\mathcal{T}^*$, pisat ćemo $\text{FIRST}_k(\alpha)=w$.

♦ Gramatika tipa *LL*(*k*)

Neka je $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ beskontekstna gramatika. Kaže se da je $\mathcal{G}$ tipa *LL*(*k*), za neki fiksni cijeli broj k, ako u slučaju da postoje dva krajnja izvođenja slijeva:

1) $\mathcal{S} \underset{lm}{*\Rightarrow} w\mathtt{A}\alpha \underset{lm}{*\Rightarrow} w\beta\alpha \underset{lm}{*\Rightarrow} wx$ i

2) $\mathcal{S} \underset{lm}{*\Rightarrow} w\mathtt{A}\alpha \underset{lm}{*\Rightarrow} w\gamma\alpha \underset{lm}{*\Rightarrow} wy$

tako da je $\mathtt{FIRST}_k(x)=\mathtt{FIRST}_k(y)$, vrijedi $\beta=\gamma$. Kaže se da je gramatika tipa LL ako je tipa $LL(k)$ za neki k.

Neformalno, $\mathcal{G}$ je tipa $LL(k)$ ako za danu lijevu rečeničnu formu $w\mathtt{A}\alpha$ iz $(\mathcal{N}\cup\mathcal{T})^+$ i prvih k terminalnih znakova (ako egzistiraju) koji će biti izvedeni iz $\mathtt{A}\alpha$ postoji najviše jedna produkcija koja će se upotrijebiti za A da bi se izveo niz znakova koji počinje s w iza kojeg slijedi k takvih znakova.

♣ Primjer 6.2

Neka je $\mathcal{G}_1$ gramatika s produkcijama:

$\mathtt{S} \rightarrow \mathtt{aAS} \mid \mathtt{b}$
$\mathtt{A} \rightarrow \mathtt{a} \mid \mathtt{bSA}$

Intuitivno, $\mathcal{G}_1$ je gramatika tipa $LL(1)$ zato što za dani C, prvi neterminal u bilo kojoj lijevoj rečeničnoj formi, i c, slijedeći ulazni znak, postoji najviše jedna produkcija za C sposobna izvesti niz terminala započet s c. Prema definiciji gramatike tipa $LL(1)$, ako je

$\mathtt{S} \underset{lm}{*\Rightarrow} w\mathtt{S}\alpha \underset{lm}{*\Rightarrow} w\beta\alpha \underset{lm}{*\Rightarrow} wx$ i

$\mathtt{S} \underset{lm}{*\Rightarrow} w\mathtt{S}\alpha \underset{lm}{*\Rightarrow} w\gamma\alpha \underset{lm}{*\Rightarrow} wy$

i ako x i y počinju istim znakom, mora biti $\beta=\gamma$. Posebno, ako x i y počinju s a, tada se mora uporabiti produkcija $\mathtt{S}\rightarrow\mathtt{aAS}$ i $\beta=\gamma=\mathtt{aAS}$. Alternativu $\mathtt{S}\rightarrow\mathtt{b}$ nije moguće upotrijebiti. Ako x i y počinju s b, mora se uporabiti alternativa $\mathtt{S}\rightarrow\mathtt{b}$ i vrijedi $\beta=\gamma=\mathtt{b}$ ($x=y=\varepsilon$ je nemoguće jer se ne izvodi ε u gramatici $\mathcal{G}_1$). Slično, ako se promatraju dva izvođenja:

$\mathtt{S} \underset{lm}{*\Rightarrow} w\mathtt{A}\alpha \underset{lm}{*\Rightarrow} w\beta\alpha \underset{lm}{*\Rightarrow} wx$ i

$\mathtt{S} \underset{lm}{*\Rightarrow} w\mathtt{A}\alpha \underset{lm}{*\Rightarrow} w\gamma\alpha \underset{lm}{*\Rightarrow} wy$

i $x=y=\mathtt{a}$, onda je za A morala biti upotrijebljena produkcija $\mathtt{A}\rightarrow\mathtt{a}$, a ako je $x=z=\mathtt{b}$, produkcija $\mathtt{A}\rightarrow\mathtt{bSA}$.

Gramatika $\mathcal{G}_1$ iz prethodnog primjera primjer je primitivne gramatike tipa LL(1). Slijedi definicija:

♦ Primitivna *LL*(1) gramatika

Beskontekstna gramatika $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ bez ε-produkcija, tako da za sve $\mathtt{A}\in\mathcal{N}$ svaka alternativa od A počinje različitim terminalom, naziva se primitivna $LL(1)$ gramatika.

Dakle, u primitivnoj $LL(1)$ gramatici za dani par $(\mathtt{A},\mathtt{a})$, gdje je $\mathtt{A}\in\mathcal{N}$, $\mathtt{a}\in\mathcal{T}$, postoji najviše jedna produkcija oblika $\mathtt{A}\rightarrow\mathtt{a}\alpha$.

♣ Primjer 6.3

Razmotrimo kompliciraniji slučaj gramatike $\mathcal{G}_2$ definirane sa:

$$S \rightarrow \varepsilon \mid abA \quad A \rightarrow Saa \mid b$$

Može se pokazati da je $\mathcal{G}_2$ tipa $LL(2)$. Da bi se to uradilo, treba pokazati da ako je wBα bilo koja lijeva rečenična forma od $\mathcal{G}_2$ i wx je rečenica u $\mathcal{L}(\mathcal{G}_2)$, tada postoji najviše jedna produkcija B$\rightarrow\beta$ u $\mathcal{G}_2$ tako da $FIRST_2(\beta\alpha)$ sadrži $FIRST_2(x)$. Pretpostavimo da je:

$$S \underset{lm}{*\Rightarrow} w S\alpha \underset{lm}{*\Rightarrow} w\beta\alpha \underset{lm}{*\Rightarrow} wx \text{ i}$$

$$S \underset{lm}{*\Rightarrow} w S\alpha \underset{lm}{*\Rightarrow} w\gamma\alpha \underset{lm}{*\Rightarrow} wy$$

gdje su prva dva znaka x i y jednaka, ako postoje. Ili je $w=\alpha=\varepsilon$, ili je produkcija A$\rightarrow$Saa bila upotrijebljena najmanje jedanput u izvođenju $S \underset{lm}{*\Rightarrow} wS\alpha$. Dakle, ili je $\alpha=\varepsilon$, ili počinje s aa.

Pretpostavimo da je S$\rightarrow\varepsilon$ bilo upotrijebljeno u izravnom izvođenju wSα u w$\beta\alpha$. Tada je $\beta=\varepsilon$ i x je ili ε ili počinje s aa. Slično, ako je S$\rightarrow\varepsilon$ bilo upotrijebljeno u izravnom izvođenju wSα u w$\gamma\alpha$, tada je $\alpha=\varepsilon$ i $y=\varepsilon$, ili y počinje s aa. Ako je u izravnom izvođenju wSα u w$\beta\alpha$ bilo upotrijebljeno S$\rightarrow$abA, tada je β=abA i x počinje s ab. Slično, ako je S$\rightarrow$abA bilo upotrijebljeno u izravnom izvođenju wSα u w$\gamma\alpha$, tada je γ=abA i y počinje s ab. Prema tome, ne postoji neka druga mogućnost osim da je x=y=ε, x i y počinju s aa, ili oba počinju s ab.

Posljedice definicije *LL*(*k*)

Iz definicije $LL(k)$ gramatike slijedi da, ako je dana lijeva rečenična forma wAα, tada w i k znakova koji slijede w jedinstveno određuju koja će produkcija biti upotrijebljena za ekspanziju od A. Na prvi pogled, moraju se pamtiti svi w da bi se zaključilo koja će produkcija biti upotrijebljena kao sljedeća. Međutim, nije tako. Idući je teorem fundamentalan za razumijevanje $LL(k)$ gramatika.

• Propozicija 6.1

Neka je $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ beskontekstna gramatika. $\mathcal{G}$ je tipa $LL(k)$ ako i samo ako vrijedi sljedeći uvjet: ako su A$\rightarrow\beta$ i A$\rightarrow\gamma$ dvije različite produkcije u $\mathcal{P}$, tada je:

$$FIRST_k(\beta\alpha) \cap FIRST_k(\gamma\alpha) = \varnothing$$

za sve wAα tako da je $\mathcal{S} \underset{lm}{*\Rightarrow} wA\alpha$.

Dakle, prema propoziciji 6.1, beskontekstna je gramatika $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ tipa $LL(1)$ ako i samo ako je za sve A iz $\mathcal{N}$ svaki skup A-produkcija $A\rightarrow\alpha_1|\alpha_2|\ldots|\alpha_n$ iz $\mathcal{P}$ takav da su svi parovi $FIRST_1(\alpha_1), FIRST_1(\alpha_2), \ldots, FIRST_1(\alpha_n)$, disjunktni skupovi.

♣ Primjer 6.4

Gramatika $\mathcal{G}$ s produkcijama S$\rightarrow$aS|a ne može biti tipa $LL(1)$ jer je:

$$FIRST_1(aS) = FIRST_1(a) = a$$

Intuitivno, u sintaksnoj analizi niza koji počinje s a, gledajući samo prvi ulazni znak, nije moguće odrediti hoće li za ekspanziju od S biti upotrijebljeno S$\rightarrow$aS ili S$\rightarrow$a. S druge strane, $\mathcal{G}$ je tipa $LL(2)$. Prema teoremu 6.1, ako je S$\Rightarrow$wAα, tada je A=S i $\alpha=\varepsilon$. Za S su dane dvije produkcije, tako da je β=aS i $\gamma=\varepsilon$. Budući da je $FIRST_2(aS)=aa$ i $FIRST_2(a)=a$, $\mathcal{G}$ je na temelju propozicije 6.1 tipa $LL(2)$.

Razmotrimo sada $LL(1)$ gramatike s ε-produkcijama. Najprije uvodimo definiciju funkcije $FOLLOW_k$.

- **$FOLLOW_k$**

  Neka je $G=(N,T,P,S)$ beskontekstna gramatika. Za $\beta \in (N \cup T)^*$ definira se funkcija:

  $$FOLLOW_k(\beta) = \{w: S^* \Rightarrow \alpha\beta\gamma \text{ i } w \in FIRST_k(\gamma)\}$$

Dakle, $FOLLOW_1(A)$ uključuje skup terminalnih znakova koji se mogu pojaviti neposredno iza A u bilo kojoj rečeničnoj formi. Ako je αA rečenična forma, tada je ε također u $FOLLOW_1(A)$.

Proširimo funkcije $FIRST$ i $FOLLOW$ do domene u kojoj će se umjesto nizova pojavljivati skupovi nizova, tj. ako je $G=(N,T,P,S)$ beskontekstna gramatika i $X \subseteq (N \cup T)^*$, tada je:

$$FIRST_k(X) = \{w: \text{za neki } \alpha \text{ u } X,\ w \in FIRST_k(\alpha)\}$$
$$FOLLOW_k(X) = \{w: \text{za neki } \alpha \text{ u } X,\ w \in FOLLOW_k(\alpha)\}$$

- **Propozicija 6.2**

  Beskontekstna gramatika $G=(N,T,P,S)$ je tipa $LL(1)$ ako i samo ako za svaki A u N, gdje su $A \to \beta$ i $A \to \gamma$ dvije različite produkcije, vrijedi:

  $$FIRST_1(\beta\ FOLLOW_1(A)) \cap FIRST_1(\gamma\ FOLLOW_1(A)) = \varnothing$$

Prema tome, gramatika G je tipa $LL(1)$ ako i samo ako za svaki par A-produkcija $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$ vrijede sljedeći uvjeti:

1) $FIRST_1(\alpha_1), FIRST_1(\alpha_2), \ldots, FIRST_1(\alpha_n)$ su disjunktni po parovima.
2) Ako je $\alpha_i \Rightarrow^* \varepsilon$, tada je $FIRST_1(\alpha_j) \cap FOLLOW_1(A) = \varnothing$, za $1 \le j \le n$, $i \ne j$.

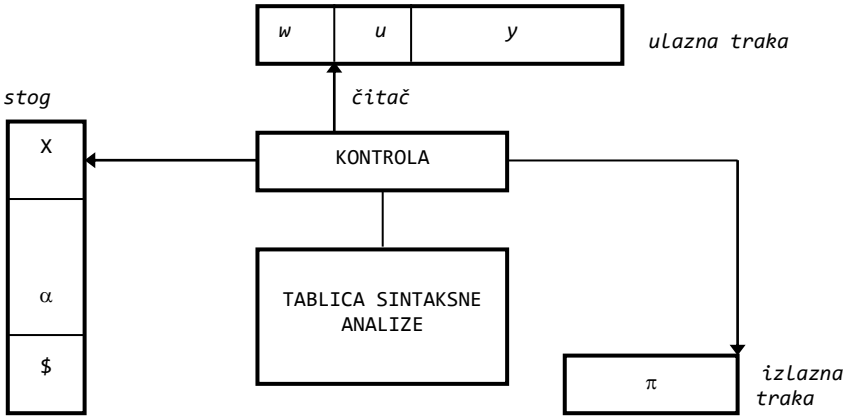
6.2 PREDIKATNA SINTAKSNA ANALIZA

Jezici $LL(k)$ mogu se analizirati veoma efikasno uporabom k-predikatnih postupaka. Postupak predikatne ("k-predikatne" ili "k-predvidljive") sintaksne analize $LL(k)$ jezika generiranog gramatikom $G=(N,T,P,S)$ shematski je prikazan na sl. 6.2.

Ulazna traka sadrži ulazni niz. Čitač može učitati do k slijedećih znakova, nazvat ćemo ih tekući niz. Na slici je to niz u.

Stog sadrži niz $X\alpha\$$, gdje je $\$$ poseban znak za oznaku kraja stoga. Simbol X je na vrhu stoga. Sa Γ će biti označen alfabet znakova stoga.

Izlazna traka sadrži niz indeksa π produkcija gramatike G koje su bile upotrijebljene za izvođenje ulaznog niza.



Sl. 6.2 - *Model k-predikatnog postupka SA.*

Konfiguracija postupka predikatne sintaksne analize, jer se očigledno radi o prepoznavaču jezika $LL(k)$, definirana je uređenom trojkom:

$(x, X\alpha, \pi)$

gdje su:

- x neuporabljeni dio ulaznog niza
- $X\alpha$ niz stoga, X je na vrhu
- π izlazni niz

Akcija postupka predikatne SA, $\mathbb{A}$, upravljana je tablicom sintaksne analize, $\mathbb{M}$, i predstavlja preslikavanje iz skupa $(\Gamma \cup \{\$\}) \times T^{*k}$ u skup koji sadrži sljedeće elemente:

1) (β, i), gdje je β u Γ^*, a i je broj produkcije. Pretpostavlja se da je β ili desna strana produkcije s indeksom i, ili njezina reprezentacija
2) pop
3) accept
4) error

Postupak SA analizira ulazni niz čineći niz premještanja. U jednom premještanju utvrđuju se tekući niz, *u*, i simbol X na vrhu stoga. Tada se konzultira ulaz $\mathbb{M}(X,u)$ tablice sintaksne analize da bi se utvrdilo aktualno premještanje (akcija). I ovdje ćemo za premještanje koristiti relaciju $\vdash$ na skupu konfiguracija. Neka je *u* jednako $FIRST_k(x)$. Piše se:

1) $(x,X\alpha,\pi) \vdash (x,\beta\alpha,\pi i)$

   ako je $\mathbb{M}(X,u)=(\beta,i)$. Ovdje se znak X na vrhu stoga zamjenjuje nizom $\beta \in \Gamma^*$ i broj produkcije i dodaje izlazu. Čitač se ne pomiče.

2) $(x,a\alpha,\pi) \vdash (x',\alpha,\pi)$

   ako je $\mathbb{M}(a,u)$=pop i $x=ax'$, tj. kad je znak na vrhu stoga jednak tekućem znaku (prvom znaku tekućeg niza), stog "puca" i čitač se pomiče za jedno mjesto udesno.

3) Ako postupak dosegne konfiguraciju $(\varepsilon,\$,\pi)$, analiza je završena. Izlazni niz π sadrži indekse produkcija koje su bile upotrijebljene u izvođenju lijevih rečeničnih formi, počevši sa S i završavajući s ulaznim nizom. Može se pretpostaviti da je $\mathbb{M}(\$,\varepsilon)$ uvijek jednako accept. Konfiguracija $(\varepsilon,\$,\pi)$ je prihvatljiva.

4) Ako postupak dosegne konfiguraciju $(x,X\alpha,\pi)$ i $\mathbb{M}(X,u)$=error, prekida se daljnja analiza i dojavljuje pogreška. Konfiguracija $(x,X\alpha,\pi)$ je neprihvatljiva.

Ako je $w \in \mathcal{T}^*$ niz koji treba analizirati, tada je početna konfiguracija postupka sintaksne analize $(w,X_0\$,\varepsilon)$, gdje je X_0 početni znak. Ako je:

$(w,X_0\$,\varepsilon) \ ^*\vdash (\varepsilon,\$,\pi)$

piše se $\mathbb{A}(w)=\pi$ i π se naziva izlaz od $\mathbb{A}$ za ulaz w. Ako $(w,X_0\$,\varepsilon)$ ne dosegne prihvatljivu konfiguraciju, kaže se da je $\mathbb{A}(w)$ nedefinirano.

Kaže se da je $\mathbb{A}$ valjani postupak (algoritam) k-predikatne sintaksne analize beskontekstne gramatike $\mathcal{G}$ ako:

1) $\mathcal{L}(\mathcal{G}) = \{w: \mathbb{A}(w) \text{ je definirano}\}$, i
2) Ako je $\mathbb{A}(w)=\pi$, tada je π sintaksna analiza slijeva za w.

U ovom se slučaju za $\mathbb{M}$ kaže da je valjana tablica sintaksne analize za $\mathcal{G}$.

♣ Primjer 6.5

Konstruirajmo 1-predikatni postupak sintaksne analize $\mathbb{A}$ gramatike:

(1) S → aAS (2) S → b (3) A → a (4) A → bSA

	a	b	ε
S	aAS, 1	b, 2	error
A	a, 3	bSA, 4	error
a	pop	error	error
b	error	pop	error
$	error	error	accept

Stupci su označeni znakovima iz alfabeta uz dodatak praznog niza, ε, i predstavljaju tekući znak, a redovi su označeni znakovima iz $\mathcal{N}$ i $\mathcal{T}$, uz dodatak znaka $, i predstavljaju znak na vrhu stoga. Koristeći tu tablicu, $\mathbb{A}$ će analizirati ulazni niz abbab kao što slijedi:

```
(abbab, S$, ε) ⊢ (abbab, aAS$, 1)     ⊢ (bbab,   AS$, 1)
               ⊢ (bbab, bSAS$, 14)    ⊢ (bab,   SAS$, 14)
               ⊢ (bab,   bAS$, 142)   ⊢ (ab,     AS$, 142)
               ⊢ (ab,     aS$, 1423)  ⊢ (b,       S$, 1423)
               ⊢ (b,       b$, 14232) ⊢ (ε,        $, 14232)
```

Dakle, abbab je u jeziku generiranom danom gramatikom. Taj se niz može dobiti nizom izvođenja:

$S \Rightarrow aAS \Rightarrow abSAS \Rightarrow abbAS \Rightarrow abbaS \Rightarrow abbab$

Niz π=14232 sadrži indekse (brojeve) produkcija koje su bile upotrijebljene u izravnim izvođenjima lijevih rečeničnih formi.

6.3 SINTAKSNA ANALIZA *LL*(1) JEZIKA

Središnji dio postupka k-predikatne sintaksne analize jest tablica sintaksne analize, **M**. U prethodnom primjeru smo je napisali intuitivno. Pogledajmo kako se ta tablica tvori u posebnom slučaju, kada je gramatika tipa LL(1).

♥ Algoritam 6.1 *Tvorba tablice sintaksne analize LL(1) gramatike.*

Ulaz
Beskontekstna gramatika $G=(\mathcal{N},\mathcal{T},\mathcal{P},S)$ tipa *LL*(1).

Izlaz
M, valjana tablica sintaksne analize za G.

Postupak
Pretpostavimo da je $ na dnu stoga. **M** je definirano na $(\mathbf{\Gamma}\cup\{\$\})\times(\mathcal{T}\cup\{\varepsilon\})$, gdje je $\mathbf{\Gamma}=\mathcal{N}\cup\mathcal{T}$, kao što slijedi:

1) Ako je $A\to\alpha$ i-ta produkcija u $\mathcal{P}$, tada je $\mathbf{M}(A,a)=(\alpha,i)$ za sve a u $FIRST_1(\alpha)$, $a\neq\varepsilon$. Ako je ε također u $FIRST_1(\alpha)$, tada je $\mathbf{M}(A,b)=(\alpha,i)$ za sve b u $FOLLOW_1(A)$.
2) $\mathbf{M}(a,a)$=pop za sve $a\in\mathcal{T}$.
3) $\mathbf{M}(\$,\varepsilon)$=accept.
4) Inače, $\mathbf{M}(X,a)$=error, $X\in(\mathbf{\Gamma}\cup\{\$\})$, $a\in(\mathcal{T}\cup\{\varepsilon\})$.

♣ Primjer 6.6

Razmotrimo tvorbu tablice sintaksne analize za gramatiku G s produkcijama:

(1) E → TA (2) A → +TA (3) A → ε (4) T → FB
(5) B → *FB (6) B → ε (7) F → (E) (8) F → a

Gramatika G dobivena je transformacijom gramatike G_0 definirane sa:

E → E+T| T T → T*F| F F → (E)| a

G_0 nije tipa *LL*(1) (rekurzivna je slijeva). Primjenom algoritma za eliminiranje rekurzija slijeva dobiva se gramatika G za koju se, primjenjujući teorem 6.2, može pokazati da je tipa *LL*(1). Izračunajmo najprije E-red koristeći korak (1) algoritma 6.1. Ovdje je $FIRST_1(TA)=\{(,a\}$, pa je:

M(E,() = M(E,a) = TA,1

Svi ostali ulazi u E-redu su error. Sada izračunajmo ulaze A-reda. Uočavamo da je $FIRST_1(+TA)=+$, pa je:

M(A,+) = +TA,2

Budući je A→ε produkcija, mora se izračunati $FOLLOW_1(A)$, a jednak je {ε,)}, pa je :

$M(A,\varepsilon) = M(A,)) = \varepsilon,3$

Ostali ulazi za A su error. Nastavljajući s izračunavanjem redova ostalih produkcija, na koncu bismo dobili tablicu sintaksne analize (radi preglednosti su na mjestima gdje je trebalo stajati error ostale praznine):

	a	(	)	+	*	ε
E	TA, 1	TA, 1				
A			ε, 3	+TA, 2		ε, 3
T	FB, 4	FB, 4				
B			ε, 6	ε, 6	*FB, 5	ε, 6
F	a, 8	(E), 7				
a	pop					
(		pop				
)			pop			
+				pop		
*					pop	
$						accept

Ako, na primjer, treba analizirati niz (a*a), algoritam 1-predikatne sintaksne analize učinit će sljedeći niz premještanja (radi izbjegavanja dvoznačnosti, konfiguracije su napisane u uglatim zagradama):

```
[(a*a),E$,ε]
     ⊢ [(a*a),      TA$, 1]          ⊢ [(a*a),    FBA$, 14]
     ⊢ [(a*a),   (E)BA$, 147]        ⊢ [ a*a),   E)BA$, 147]
     ⊢ [ a*a),   TA)BA$, 1471]       ⊢ [ a*a), FBA)BA$, 14714]
     ⊢ [ a*a),  aBA)BA$, 147148]     ⊢ [  *a),  BA)BA$, 147148]
     ⊢ [  *a), *FBA)BA$, 1471485]    ⊢ [   a), FBA)BA$, 1471485]
     ⊢ [   a),  aBA)BA$, 14714858]   ⊢ [    ),  BA)BA$, 14714858]
     ⊢ [    ),    A)BA$, 147148586]  ⊢ [    ),    )BA$, 1471485863]
     ⊢ [    ε,      BA$, 1471485863] ⊢ [    ε,      A$, 14714858636]
     ⊢ [    ε,        $, 147148586363]
```

Dakle, ulazni niz (a*a) je u jeziku $\mathcal{L}(\mathcal{G})$ i može se dobiti nizom izvođenja:

$E \Rightarrow TA \Rightarrow FBA \Rightarrow (E)BA \Rightarrow (TA)BA \Rightarrow (FB)BA \Rightarrow (aB)BA$
$\Rightarrow (a*FB)BA \Rightarrow (a*aB)BA \Rightarrow (a*a)BA \Rightarrow (a*a)A \Rightarrow (a*a)$

6.4 REKURZIVNI SPUST

Često se koristi sintaksna analiza "s rekurzivnim spustom", koju je publicirao Wirth u sintaksnoj analizi jezika PL/0. Moglo bi se reći da je to pravi "školski primjer" realizacije problema SA $LL(1)$ jezika. Ako pretpostavimo da su produkcije neterminala A, $A \rightarrow \alpha_1|\ldots|\alpha_n$, i ako je Sym tekući simbol, dio postupka sintaksne analize s rekurzivnim spustom koji se odnosi na A možemo općenito prikazati ovako:

```
def A():
  if   Sym in FIRST(alfa1 +FOLLOW(A())):
    " Kod za alfa1 "
  elif Sym in FIRST(alfa2 +FOLLOW(A())):
    " Kod za alfa2 "
    ...
  elif Sym in FIRST(alfaN +FOLLOW(A())):
    " Kod za alfaN "
```

To znači da će se sparivanjem tekućeg simbola s jednim od očekujućih pozvati odgovarajuća procedura. Ako to nije moguće, bit će dojavljena pogreška.

Postupak SA s rekurzivnim spustom, kao što slijedi iz njegova imena, sadrži rekurzije. Otuda i zaključak da će se najbolji efekti doseći njegovom ustrojbom u nekom jeziku za programiranje koji dopušta rekurzije (npr. Python ili Pascal). Tada je moguće iz sintaksnih dijagrama izravno izvesti program sintaksne analize danog jezika. Ako se postupak ustroji u jeziku koji ne dopušta rekurzije, treba uložiti dodatni napor za eliminiranje rekurzija, za što je potrebno uvesti pomoćne varijable i strukture podataka.

P R O G R A M I

U ovom dijelu dajemo programe koji prikazuju kako se mogu implementirati dva postupka sintaksne analize *LL*(1) jezika: 1-predikatnu sintaksnu analizu i rekurzivni spust.

1-PREDIKATNA SINTAKSNA ANALIZA

Da bi se izvela sintaksna analiza ulaznog niza i zaključilo pripada li jeziku kojeg generira zadana *LL*(1) gramatika G, program 1-predikatne sintaksne analize najprije, uz pomoć procedure `Tablica_SA (G)`, gradi tablicu sintaksne analize `M` učitane gramatike `G`. Tablicu ispisuje procedura `Ispis_M (M,I,J)`, gdje je `J` lista znakova stupaca, a `I` lista znakova redaka. Evo kompletnog programa i prikaza njegovog rada u sintaksnoj analizi jezika danog u primjeru 6.6.

PREDIKATNA-SA.py

```
from gramatika import *
def Ispis_M (M, I, J):
  print ' ',
  for j in range (len(J)): print '%10s' %J[j],
  c = '--' +'-'*11*len(J)
  print NL, c
  for i in range (len(I)):
    print I[i],
    for j in range (len(J)):  print '%10s' % M[i][j],
    print
  print c
  return
def Tablica_SA (G): # Tablica sintaksne analize
  N, T, P, S = G;
  if '#' in T: T.remove ('#')
  I = N +T +['$'];  n = len(I) -1;  J = T +['#'];  m = len(J) -1;  M = []
  for i in range (len(I)):
    M.append ([''])
    for j in range (len(J)): M[i].append ('')
  P2 = Uredi_P (P)
  def first (alfa, Fi=[]):
    z = alfa[0]
    if z in T+['#']:
      if z not in Fi: Fi.append (z)
    else:
      for x in P2:
        if x[0] == z: Fi = first (x[1], Fi)
    return Fi
```

```
    def follow (A, Fo=[]):
      for x, y in P2:
        if y[-1] == A and x != A:
          for a, b in P2:
            if x in b and x != b[-1]:
              b  = b[b.find(x)+1:]
              Fo = first(b, Fo)
              if [b[0], '#'] in P2: Fo = follow (b[0], Fo)
      return Fo

    def upisi (z):
      j = J.index(z); p = P2.index ([A, X[k]]); M[i][j] = X[k] +',' +str(p+1)

    '(1)'
    for i in range (len(P)):
      X = P[i];  A = X[0]
      for k in range (1, len(X)):
        Fi = first (X[k], [])
        for fi in Fi: upisi (fi)
        if '#' in Fi:
          Fo = follow (A, [])
          for fo in Fo: upisi (fo)

    '(2)'
    i0 = I.index (T[0])
    for i in range (i0, n, 1): M[i][i-i0] = 'pop'

    '(3)'
    M[n][m] = 'accept';  Ispis_M (M, I, J)
    return (M, I, J)

  def _1_PSA (T, w):

    def ispis (s, C):
      def f (c, d):
        return c +' '*(d -len(c))
      print s +'(', f(C[0], len(w)), f(C[1], 12), C[2], ')'

    M, I, J = T
    x = w;  X = I[0] +'$';  Pi = [];  C = (x, X, Pi);  ispis ('', C)
    Err = False; Accept = False
    while not Err and not Accept:
      i = I.index(X[0]);  j = J.index(x[0]);  s = M[i][j];  Err = s == ''
      if not Err:
        Accept = s == 'accept'
        if not Accept:
          if s == 'pop':
             x = x[1:]
             if x == '': x = '#'
             X = X[1:]
          else:
            k = s.find(',');  alfa = s[:k]
            if alfa == '#': alfa = ''
            X = X.replace (X[0], alfa, 1);  Pi.append(int(s[k+1:]))
          C = (x, X, Pi);  ispis ('  |- ', C)
      else: Err = True
    return Accept, Pi
```

```
def Niz_Izv (P, Pi, w):
  P2 = Uredi_P (P)
  print NL, w, ' je rečenica jezika! Može se dobiti nizom izvođenja:', NL
  print 'Pi =', Pi, NL
  LSF (Pi, P2)
  return

Grm, Ok, G = Ucitaj_G ('1-PREDIKATNA SINTAKSNA ANALIZA', '*.grm')
if Ok :
  Ispisi_G (Grm, G);  T = Tablica_SA (G)
  w = Ucitaj_W()
  while len(w) > 0:
    Ok, pi = _1_PSA (T, w)
    if Ok : Niz_Izv (G[2], pi, w)
    else  : print w, 'NIJE REČENICA JEZIKA!'
    w = Ucitaj_W()
else :
  print 'Ne postoji gramatika s danim imenom!'
```

```
Exp5-LL1.GRM

N = { E , A , T , B , F }
T = { + , * , ( , ) , a }
S = E
P:
  E -> TA
  A -> +TA | #
  T -> FB
  B -> *FB | #
  F -> (E) | a
```

	+	*	(	)	a	#
E			TA,1		TA,1	
A	+TA,2			#,3		#,3
T			FB,4		FB,4	
B	#,6	*FB,5		#,6		#,6
F			(E),7		a,8	
+	pop					
*		pop				
(			pop			
)				pop		
a					pop	
$						accept

```
Upiši ulazni niz: (a+a)*a
( (a+a)*a E$          [] )
  |- ( (a+a)*a TA$          [1] )
  |- ( (a+a)*a FBA$         [1, 4] )
  |- ( (a+a)*a (E)BA$       [1, 4, 7] )
  |- ( a+a)*a  E)BA$        [1, 4, 7] )
  |- ( a+a)*a  TA)BA$       [1, 4, 7, 1] )
  |- ( a+a)*a  FBA)BA$      [1, 4, 7, 1, 4] )
  |- ( a+a)*a  aBA)BA$      [1, 4, 7, 1, 4, 8] )
```

```
|- ( +a)*a   BA)BA$       [1, 4, 7, 1, 4, 8] )
|- ( +a)*a   A)BA$        [1, 4, 7, 1, 4, 8, 6] )
|- ( +a)*a   +TA)BA$      [1, 4, 7, 1, 4, 8, 6, 2] )
|- ( a)*a    TA)BA$       [1, 4, 7, 1, 4, 8, 6, 2] )
|- ( a)*a    FBA)BA$      [1, 4, 7, 1, 4, 8, 6, 2, 4] )
|- ( a)*a    aBA)BA$      [1, 4, 7, 1, 4, 8, 6, 2, 4, 8] )
|- ( )*a     BA)BA$       [1, 4, 7, 1, 4, 8, 6, 2, 4, 8] )
|- ( )*a     A)BA$        [1, 4, 7, 1, 4, 8, 6, 2, 4, 8, 6] )
|- ( )*a     )BA$         [1, 4, 7, 1, 4, 8, 6, 2, 4, 8, 6, 3] )
|- ( *a      BA$          [1, 4, 7, 1, 4, 8, 6, 2, 4, 8, 6, 3] )
|- ( *a      *FBA$        [1, 4, 7, 1, 4, 8, 6, 2, 4, 8, 6, 3, 5] )
|- ( a       FBA$         [1, 4, 7, 1, 4, 8, 6, 2, 4, 8, 6, 3, 5] )
|- ( a       aBA$         [1, 4, 7, 1, 4, 8, 6, 2, 4, 8, 6, 3, 5, 8] )
|- ( #       BA$          [1, 4, 7, 1, 4, 8, 6, 2, 4, 8, 6, 3, 5, 8] )
|- ( #       A$           [1, 4, 7, 1, 4, 8, 6, 2, 4, 8, 6, 3, 5, 8, 6] )
|- ( #       $            [1, 4, 7, 1, 4, 8, 6, 2, 4, 8, 6, 3, 5, 8, 6, 3] )

(a+a)*a  je rečenica jezika! Može se dobiti nizom izvođenja:

Pi = [1, 4, 7, 1, 4, 8, 6, 2, 4, 8, 6, 3, 5, 8, 6, 3]

E
  => TA
  => FBA
  => (E)BA
  => (TA)BA
  => (FBA)BA
  => (aBA)BA
  => (aA)BA
  => (a+TA)BA
  => (a+FBA)BA
  => (a+aBA)BA
  => (a+aA)BA
  => (a+a)BA
  => (a+a)*FBA
  => (a+a)*aBA
  => (a+a)*aA
  => (a+a)*a
```

REKURZIVNI SPUST

S obzirom na to da je rekurzivni spust postupak sintaksne analize koji je ovisan o ulaznoj *LL*(1) gramatici i njezinim produkcijama, dajemo njegovu implementaciju u sintaksnoj analizi jezika generiranog gramatikom prikazanoj u BNF-u:

```
<izraz>    → <term> | <term> <op1> <izraz>
<term>     → <faktor> | <faktor> <op2> <term>
<faktor>   → <operand> | ( <izraz> )
<op1>      → + | -
<op2>      → * | /
<operand>  → <broj> | <broj> . <broj>
<broj>     → <znamenka> | <znamenka> <broj>
<znamenka> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Lako se možemo uvjeriti da je to jezik realnih izraza s četiri operacije, operandima cijelim ili realnim brojevima i zagradama. Slijedi program i primjer analize ulaznog niza:

REKSPUST.py

```
# Sintaksna analiza realnih izraza rekurzivnim spustom

from gramatika import *
from fun       import *

global w, Err

def izraz ():

  global w, Sym, Err, RF

  def ucitaj ():
    global Sym, w
    Sym = w[0];  w = w[1:]

  def neprazan (): return len(w) > 0

  def term ():
    global w, Sym, Err, RF

    def operator ():
      global w, Sym, Err, RF
      def broj ():
        Op = Sym;  Err = False
        while brojka (Sym) and neprazan():
          ucitaj ()
          if brojka (Sym): Op += Sym
        if neprazan():
          if Sym == '.' and neprazan():
            Op += Sym
            ucitaj ()
            if brojka (Sym):
              Op += Sym
              while brojka(Sym) and neprazan():
                ucitaj ()
                if brojka(Sym): Op += Sym
            else:
              Err = True
        return Op

      RF = RF.replace ('<faktor>', '<operand>');  print RF
      Op = broj ()
      if not Err: RF = RF.replace ('<operand>', Op);  print RF
      return

    def faktor ():
      global w, Sym, Err, RF
      RF = RF.replace ('<term>', '<faktor>');  print RF
      Err = False
      if brojka (Sym):
        operator()
      elif Sym == '(':
        if neprazan():
          RF = RF.replace ('<faktor>', '(<izraz>')
          izraz();  Err = Sym != ')'
          if not Err: RF += ')';  print RF
```

```
        if neprazan ():
            ucitaj ()
            Err = Sym not in '+-*/'
        else:
          Err = True
      else:
        Err = True
      return

    RF = RF.replace ('<izraz>', '<term>');  print RF
    faktor ()
    while not Err and Sym in '*/':
      RF += Sym +'<term>';  print RF
      if neprazan ():
        ucitaj ();  faktor ()
      else:
        Err = True
    return

  print RF
  if neprazan():
    ucitaj ()
    term ()
    while not Err and Sym in '+-':
      RF += Sym +'<izraz>';  print RF
      if neprazan(): ucitaj (); term ()
      else         : Err = True

  return

w = Ucitaj_W()
while len(w) > 0:
  n = len(w);  RF = '<izraz>'
  izraz ()
  if not Err and n == len(RF): print 'Niz JE u jeziku!'
  else                       : print 'Niz NIJE u jeziku!'
  print
  w = Ucitaj_W()

Upiši ulazni niz: 2*6.66*3.14
<izraz>
<term>
<faktor>
<operand>
2
2*<term>
2*<faktor>
2*<operand>
2*6.66
2*6.66*<term>
2*6.66*<faktor>
2*6.66*<operand>
2*6.66*3.14
Niz JE u jeziku!
```

```
Upiši ulazni niz: (1+2)*(3+4)
<izraz>
<term>
<faktor>
(<izraz>
(<term>
(<faktor>
(<operand>
(1
(1+<izraz>
(1+<term>
(1+<faktor>
(1+<operand>
(1+2
(1+2)
(1+2)*<term>
(1+2)*<faktor>
(1+2)*(<izraz>
(1+2)*(<term>
(1+2)*(<faktor>
(1+2)*(<operand>
(1+2)*(3
(1+2)*(3+<izraz>
(1+2)*(3+<term>
(1+2)*(3+<faktor>
(1+2)*(3+<operand>
(1+2)*(3+4
(1+2)*(3+4)
Niz JE u jeziku!

Upiši ulazni niz: (10+20))
<izraz>
<term>
<faktor>
(<izraz>
(<term>
(<faktor>
(<operand>
(10
(10+<izraz>
(10+<term>
(10+<faktor>
(10+<operand>
(10+20
(10+20)
Niz NIJE u jeziku!
```

Pitanja i zadaci

1) Pokažite da gramatika s produkcijama

$$S \rightarrow aAaB \mid bAbB$$
$$A \rightarrow a \mid ab$$
$$B \rightarrow aB \mid a$$

jest LL(3) ali nije LL(2).

2) Napišite algoritam za izračunavanje FOLLOW$_k$(A) *za neterminal* A.

3) Pokažite da je gramatika $\mathcal{G}$ *s produkcijama*

$$S \rightarrow aaSbb \mid a \mid \varepsilon$$

tipa LL(2). *Nađite ekvivalentnu gramatiku gramatici* $\mathcal{G}$ *koja će biti tipa* LL(1).

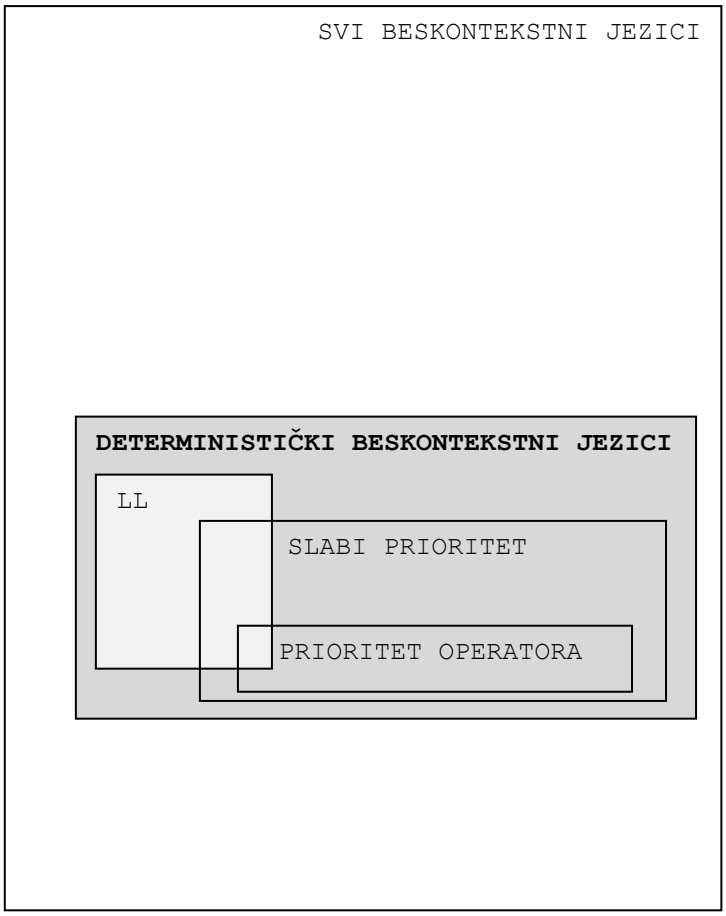
4) Pragram 1-predikatne analize dan je u prilogu 5. Proučite ga unosom ulaznih gramatika iz primjera 6.2 i 6.6.

5) Definirajte tablicu akcija i skokova za gramatiku $\mathcal{G}$ *s produkcijama*

$$S \rightarrow SaSb \mid \varepsilon$$

Potom provjerite je li niz aabb *u jeziku* $\mathcal{L}(\mathcal{G})$.

7.
LR(k) JEZICI I SINTAKSNA ANALIZA



U ovom će poglavlju biti riječi o najširoj klasi beskontekstnih jezika za koje je moguće konstruirati determinističke uzlazne postupke sintaksne analize: jezici (gramatike) tipa LR(k). Posebno ćemo opisati podklase gramatika LR(k), a to su gramatike tipa LR(0) i LR(1), te gramatike sa slabim i jakim prioritetom, operatorske gramatike i gramatike s prioritetom operatora.

7.1 JEZICI TIPA *LR*(*k*)

Za jezike tipa $LL(k)$ moguće je, dakle, ustrojiti jednoprolazne silazne postupke sintaksne analize, dok se ulazni niz pretražuje slijeva nadesno. Postoji analogna klasa jezika za koju se mogu ustrojiti jednoprolazni postupci SA generirajući pritom stablo izvođenja odozdo. Gramatike koje generiraju jezike s takvim svojstvom jesu tipa $LR(k)$, $k \geq 0$. Ovdje "L" označuje da se ulazni niz pretražuje slijeva ("left"), a "R" da se pritom izvode desne ("right") rečenične forme, koristeći k tekućih simbola ulaznog niza za odluku koja će produkcija biti upotrijebljena.

Gramatike tipa *LR*(0)

Najprije ćemo definirati gramatike tipa LR(0), podklasu LR gramatika, u kojoj LR(0) ima značenje "pretraživanje ulaznog niza slijeva izvodeći desnu rečeničnu formu, koristeći pritom 0 znakova ulaznog niza od tekuće pozicije u ulaznom nizu". Pokazuje se da takve gramatike generiraju jezike koji imaju svojstvo prefiksa (jezik $\mathcal{L}$ ima svojstvo prefiksa ako kad god je w u $\mathcal{L}$, nijedan svojstveni prefiks od w nije u $\mathcal{L}$). Primijetiti da svojstvo prefiksa nije strog uvjet, jer svaki beskontekstni jezik može imati svojstvo prefiksa ako se uvede jedan znak kao oznaka (marker) kraja svih rečenica (tj. proširi se gramatika uvođenjem produkcije $S' \to S@$, gdje je "@" oznaka kraja).

♣ Primjer 7.1

Gramatika s produkcijama:

S → SA| A A → (S)| ()

generira jezik "uparenih zagrada", {(),(()),()(),((())),(()()), ...} i nije tipa $LR(0)$, jer, na primjer, za rečenicu $w=x^n$, x=(), n>1, vrijedi da su rečenice x^k, k=1, 2, ..., n-1, svojstveni prefiksi od w. Ako gramatiku proširimo uvođenjem novog početnog simbola S' i produkcije S'→S@, dobili bismo gramatiku tipa LR(0).

♦ Stavka

Stavka beskontekstne gramatike $G=(\mathcal{N},\mathcal{T},\mathcal{P},S)$ jest produkcija koja sadrži znak "·", "·"$\notin(\mathcal{N}\cup\mathcal{T})$, bilo gdje u svojim alternativama, uključujući početak i kraj. Ako gramatika sadrži ε-produkciju, A→ε, stavka je A→·.

♣ Primjer 7.2

Pogledajmo stavke gramatike Exp:

```
Exp.GRM

N = { E , T , F }
T = { + , * , ( , ) , a }
S = E
P:
  E -> T+E | T
  T -> F*T | F
  F -> (E) | a

Stavke gramatike:

  E -> ·T+E | T·+E | T+·E | T+E· | ·T | T·
  T -> ·F*T | F·*T | F*·T | F*T· | ·F | F·
  F -> ·(E) | (·E) | (E·) | (E)· | ·a | a·
```

♦ Držač i održivi prefiks

Držač (*handle*) desne rečenične forme γ, $\gamma=\delta\beta w$, beskontekstne gramatike $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ jest podniz β ako je:

$$\mathcal{S} \underset{rm}{\overset{*}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\beta w$$

Dakle, držač desne rečenične forme γ jest podniz β (alternativa za A) koji je bio unesen u posljednjem izravnom izvođenju zdesna. U tom je kontekstu važan položaj β unutar γ. Održivi (*viable*) prefiks desne rečenične forme γ jest bilo koji njezin prefiks koji se završava ne dalje desno od desnog kraja držača od γ.

♣ Primjer 7.3

Analizirajmo desnu rečeničnu formu gramatike iz primjera 7.1:

$$S' \Rightarrow S@ \Rightarrow SA@ \Rightarrow S(S)@$$

Dakle, S(S)@ je desna rečenična forma. (S) je njezin držač. Održivi prefiksi su ε, S, S(, S(S i S(S).

♦ Valjana stavka

Reći ćemo da je $A\rightarrow\alpha\cdot\beta$ valjana stavka za održivi prefiks γ ako postoji krajnje desno izvođenje:

$$\mathcal{S} \underset{rm}{\overset{*}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$$

i $\delta\alpha=\gamma$. Poznavanje stavki koje su valjane za dani održivi prefiks može nam pomoći pri nalaženju krajnjeg izvođenja zdesna. Reći ćemo da je neka stavka potpuna ako je točka njezin posljednji znak. Ako je $A\rightarrow\alpha\cdot$ potpuna stavka valjana za γ, tada se pojavljuje tako da je $A\rightarrow\alpha$ moglo biti upotrijebljeno u posljednjem koraku i niz je $\delta A w$ bio prethodna desna rečenična forma u izvođenju γw.

♣ Primjer 7.4

Razmotrimo gramatiku iz primjera 7.1 i desnu rečeničnu formu ()@. Budući da je:

$$S' \underset{rm}{\overset{*}{\Rightarrow}} A@ \underset{rm}{\Rightarrow} ()@$$

zaključujemo da je stavka A→()· valjana za održivi prefiks (). Također zaključujemo da je stavka A→(·) valjana za održivi prefiks (i da je A→·() valjana za održivi prefiks ε. S obzirom da je A→()· potpuna stavka, konačno zaključujemo da je A@ bila prethodna desna rečenična forma za ()@.

IZRAČUNAVANJE SKUPA VALJANIH STAVKI

Definicija *LR*(0) gramatika i postupak prihvaćanja jezika $\mathcal{L}(\mathcal{G})$ za *LR*(0) gramatiku $\mathcal{G}$ deterministčkim potisnim (stogovnim) automatom (DPDA) u potpunosti je ovisno o postojanju skupa valjanih stavaka za svaki održivi prefiks γ. Pokazuje se da je za bilo koju beskontekstnu gramatiku $\mathcal{G}$ skup održivih prefiksa regularan skup i da se može prihvatiti nedeterminističkim konačnim automatom (NFA) u kojem su stanja stavke za $\mathcal{G}$. Koristeći podskup konstrukcije tog automata dolazi se do potisnog automata u kojem stanja koja odgovaraju održivom prefiksu γ jesu skup valjanih stavki za γ.

♦ Skup valjanih stavki

Nedeterministički konačni automat (NFA) koji prepoznaje održive prefikse gramatike $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},S)$ definiran je kao $\mathcal{M}=(Q, \mathcal{N}\cup\mathcal{T}, \delta, q_0, \mathcal{F})$, gdje je Q skup stavki za $\mathcal{G}$ plus stanje q_0 koje nije stavka, a funkcija prijelaza δ definirana je kao:

1) $\delta(q_0, \varepsilon)$ = {S → ·α: S→α je produkcija}
2) δ(A→α·Bβ, ε) = {B → ·γ: B→γ je produkcija}
3) δ(A→α·Xβ, X) = {A → αX·β}

Pravilo (2) dopušta ekspanziju simbola B koji se nalazi u sredini, desno od točke, a pravilo (3) dopušta pomicanje točke preko bilo kojeg simbola gramatike X, ako je X idući ulazni simbol.

♣ Primjer 7.5

Neka je dana gramatika: S'→S@, S→SA|A, A→aSb|ab (primjer 7.1, samo radi preglednosti a stoji umjesto (, a b umjesto)). Funkcija prijelaza nedeterminističkog konačnog automata koji prepoznaje održive prefikse dana je sljedećom tablicom (prijelaza):

	ε	S	A	a	b	@	
→ q_0	q_1						
q_1	q_4,q_7	q_2					S' → ·S@
q_2						q_3	S' → S·@
⊗ q_3							S' → S@·
q_4	q_4,q_7	q_5					S → ·SA
q_5	q_9,q_{13}		q_6				S → S·A
⊗ q_6							S → SA·
q_7	q_9,q_{13}		q_8				S → ·A
⊗ q_8							S → A·
q_9				q_{10}			A → ·aSb
q_{10}	q_4,q_7	q_{11}					A → a·Sb
q_{11}					q_{12}		A → aS·b
⊗ q_{12}							A → aSb·
q_{13}				q_{14}			A → ·ab
q_{14}					q_{15}		A → a·b
⊗ q_{15}							A → ab·

Stanja q_1 do q_{15} oznake su stavki danih u posljednjem stupcu.

- **Propozicija 7.1**

  NFA $\mathcal{M}$ definiran u skupu valjanih stavki ima svojstvo da $\delta(q_0,\gamma)$ sadrži $A\to\alpha\cdot\beta$ ako i samo ako je $A\to\alpha\cdot\beta$ valjana stavka za γ.

DEFINICIJA GRAMATIKE TIPA *LR*(0)

Poslije svih ovih uvodnih razmatranja možemo definirati gramatiku tipa *LR*(0).

♦ Gramatika tipa *LR*(0)

Reći ćemo da je $\mathcal{G}$ gramatika tipa *LR*(0) ako su ispunjeni sljedeći uvjeti:

1) početni se simbol ne pojavljuje niti u jednoj alternativi (tj. na desnoj strani produkcija), i
2) za svaki održivi prefiks γ iz $\mathcal{G}$, kad god je $A\to\alpha\cdot$ potpuna valjana stavka za γ, tada ne postoji nijedna druga potpuna stavka niti bilo koja stavka s terminalom desno od točke koja bi bila valjana za γ.

Propozicija 7.1 daje postupak izračunavanja skupa valjanih stavki za bilo koji održivi prefiks, odnosno kako treba pretvoriti NFA čija će stanja biti stavke deterministočkog konačnog automata (DFA). U DFA put od početnog stanja označenog s γ vodi do stanja koje je skup valjanih stavki za γ. Dakle, treba izgraditi DFA i provjeriti svako stanje da bi se vidjelo jesu li prekršeni uvjeti *LR*(0).

♣ Primjer 7.6

Tablica prijelaza DFA izgrađenog iz NFA primjera 7.5, iz kojeg su izbačena "mrtva" stanja (prazan skup stavaka) i prijelazi u "mrtvo" stanje, dana je s:

	S	A	a	b	@
→ I_0	I_1	I_2	I_3		
I_1		I_5	I_3		I_4
⊗ I_2					
I_3	I_6	I_2	I_3	I_7	
⊗ I_4					
⊗ I_5					
I_6		I_5	I_3	I_8	
⊗ I_7					
⊗ I_8					

gdje su:

I_0	I_1	I_2	I_3	I_4	I_5	I_6
S'→ ·S@	S'→ S·@	S → A·	A → a·Sb	S'→ S@·	S'→ S@·	A → aS·b
S → ·SA	S → S·A		A → a·b			S → S·A
S → ·A	A → ·aSb		S → ·SA			A → ·aSb
A → ·aSb	A → ·ab		S → ·A			A → ·ab
A → ·ab			A → ·aSb			
			A → ·ab			

I_7	I_8
A → ab·	A → aSb·

Stanja koja imaju više od jedne stavke nemaju nijednu potpunu stavku i, dakako, početni simbol ne pojavljuje se niti u jednoj alternativi. Zbog toga je gramatika iz primjera 7.5 tipa *LR*(0).

7.2 *LR*(0) GRAMATIKE I STOGOVNI PREPOZNAVAČI

Gramatika tipa *LR*(0) jednoznačna je pa generira deterministički beskontekstni jezik i, obrnuto, svaki deterministički beskontekstni jezik sa svojstvom prefiksa ima gramatiku tipa *LR*(0). Osim toga, gramatike tipa *LR*(0) su beskontekstne, pa se za njih može konstruirati deterministički stogovni automat (DSA).

Prije nego opišemo postupak izgradnje DSA, promijenimo prvobitnu definiciju konfiguracije stogovnog automata (q,w,α^R) u $[q,\alpha,w]$ (umjesto okruglih sada su uglate zagrade, a sadržaj stoga, α, sada ima vrh na svom desnom kraju). Da bismo simulirali krajnje izvođenje zdesna u *LR*(0) gramatici ne samo da imamo održivi prefiks u stogu, već se iznad svakog simbola nalazi stanje DSA koje prepoznaje održive prefikse. Ako je održivi prefiks

$$X_1 \ \dots \ X_k$$

u stogu, tada je potpuni sadržaj stoga

$$s_0X_1s_1 \ \dots \ X_ks_k$$

gdje je

$$s_i = \delta(X_1 \dots X_i)$$

i δ je funkcija prijelaza DSA. Stanje na vrhu stoga, s_k, osigurava valjane stavke za $X_1 \dots X_k$.

Ako s_k sadrži $A \to \alpha\cdot$, tada je $A \to \alpha\cdot$ valjana stavka za $X_1 \dots X_k$. Dakle, α je sufiks od $X_1 \dots X_k$, na primjer $\alpha = X_{i+1} \dots X_k$ (primijetiti da α može biti ε, tada je $i=k$). Međutim, postoji neki w tako da je $X_1 \dots X_k w$ krajnja desna rečenična forma i postoji izvođenje:

$$S \underset{rm}{\overset{*}{\Rightarrow}} X_1 \dots X_i A w \underset{rm}{\Rightarrow} X_1 \dots X_k w$$

Prema tome, da bi se dobila desna rečenična forma koja je prethodila krajnjoj desnoj rečeničnoj formi $X_1 \dots X_k w$, α je u izvođenju zdesna reducirano u A, zamjenjujući $X_{i+1} \dots X_k$ na vrhu stoga s A. Ili, ako to prikažemo preko niza pomaka:

$$[q,s_0X_1 \dots s_{k-1}X_ks_k,w] \vdash^* [q,s_0X_1 \dots s_{i-1}X_is_iAs,w]$$

gdje je $s = \delta(s_i,A)$. Ako je gramatika tipa *LR*(0), s_k sadrži samo $A \to \alpha\cdot$, osim ako je $\alpha = \varepsilon$, kada s_k može sadržavati nekoliko nepotpunih stavki. Međutim, iz definicije *LR*(0), nijedna od tih stavki nema terminal desno od točke, ili je potpuna. Dakle, za bilo koji y tako da je $X_1 \dots X_k y$ desna rečenična forma, $X_1 \dots X_i A y$ mora biti prethodna desna rečenična forma, tako da je reduciranje α u A ispravno, bez obzira na tekući ulaz.

Razmotrimo sada slučaj kad s_k sadrži samo nepotpune stavke. Tada desna rečenična forma koja prethodi rečeničnoj formi $X_1 \dots X_k w$ nije mogla biti dobivena reduciranjem sufiksa od $X_1 \dots X_k$ u neku varijablu, inače bi postojala potpuna valjana stavka za $X_1 \dots X_k$. Tada mora postojati držač koji se završava desno od X_k u $X_1 \dots X_k w$, i to tako da je $X_1 \dots X_k$ održivi prefiks. Dakle, jedina moguća akcija DSA jest premjestiti sljedeći ulazni znak u stog, to jest

$$[q,s_0X_1 \dots s_{k-1}X_ks_k,ay] \vdash [q,s_0X_1 \dots s_{k-1}X_ks_kat,y]$$

gdje je $t=\delta(s_k,a)$. Ako je t neprazan skup stavki, $X_1...X_k a$ jest održivi prefiks. Ako je t prazan, može se dokazati da ne postoji prethodna desna rečenična forma od $X_1...X_k ay$, tako da ulazni niz znakova nije rečenica jezika kojeg gramatika generira i DSA "umire" umjesto da načini pomak.

♣ Primjer 7.7

Pogledajmo DFA kao u primjeru 7.6 u kojem su stanja označena s 0, 1, ..., 8 umjesto I_0, I_1, ..., I_8, redom, i neka je ulazni niz aababb. Tada će DSA s funkcijom prijelaza jednakom funkciji prijelaza DFA, načiniti niz pomaka:

	stog	preostali ulaz	opis
1)	0	aababb@	inicijalizacija
2)	0a3	ababb@	premještanje
3)	0a3a3	babb@	premještanje
4)	0a3a3b7	abb@	premještanje
5)	0a3A2	abb@	reduciranje s A→ab
6)	0a3S6	abb@	reduciranje s S→A
7)	0a3S6a3	bb@	premještanje
8)	0a3S6a3b7	b@	premještanje
9)	0a3S6A5	b@	reduciranje s A→ab
10)	0a3S6	b@	reduciranje s S→SA
11)	0a3S6b8	@	premještanje
12)	0A2	@	reduciranje s A→aSb
13)	0S1	@	reduciranje s S→A
14)	0S1@4	-	premještanje
15)	-	-	prihvaćanje

Na primjer, u redu (1) stanje 0 je na vrhu stoga. Ne postoji potpuna stavka u skupu I_0, pa se premješta tekući znak na vrh stoga. Prvi ulazni simbol je a i postoji prijelaz iz I_0 u I_3 označen s a. Sadržaj stoga u redu (2) je 0a3, itd. U redu (9), na primjer, stanje 5 je na vrhu stoga. I_5 sadrži potpunu stavku S→SA. Izbacuje se SA iz stoga, pa ostaje 0a3. Potom se dodaje S u stog. Postoji prijelaz iz I_3 u I_6 označen sa S, pa dopisujemo stanje 6, što rezultira sadržajem stoga 0a3S6 u redu (10) itd.

Gramatike tipa *LR*(*k*)

Neformalno, kaže se da je gramatika tipa LR(k) ako se u danom krajnjem izvođenju zdesna:

$$S = \alpha_0 \underset{rm}{\Rightarrow} \alpha_1 \underset{rm}{\Rightarrow} \ldots \underset{rm}{\Rightarrow} \alpha_m = z$$

u svakoj rečeničnoj formi α_i može izolirati neki $\beta\in(\mathcal{N}\cup\mathcal{T})^*$ i jednoznačno utvrditi, pretražujući α_i slijeva nadesno, ali najviše k znakova za β, kojem je neterminalu A niz β alternativa.

Pretpostavimo da je $\alpha_{i-1}=\alpha Aw$ i $\alpha_i=\alpha\beta w$, gdje je $\alpha\in(\mathcal{N}\cup\mathcal{T})^*$, $w\in\mathcal{T}^*$ i $A\to\beta$ produkcija u $\mathcal{G}$. Dalje, pretpostavimo da je $\alpha\beta=x_1x_2...x_r$, $X_i\in\mathcal{N}\cup\mathcal{T}$. Ako je gramatika $\mathcal{G}$ tipa *LR*(k), tada možemo biti sigurni da vrijedi sljedeće:

1) Znajući $X_1X_2...X_j$ i prvih k znakova od $X_{j+1}...X_r w$ sigurni smo da kraj od β nije dosegnut sve dok nije $j=r$.
2) Znajući $\alpha\beta$ i najviše k prvih znakova od w možemo uvijek odrediti β i neterminal A iz kojeg je β izvedeno.

3) Kada je $\alpha_{i-1}=\mathcal{S}$ može se sa sigurnošću dojaviti da je ulazni niz prihvaćen (u jeziku je $\mathcal{L}(\mathcal{G})$).

Primijetimo da prolazeći kroz niz α_m, α_{m-1},..., α_0 počinjemo promatranjem samo $\mathtt{FIRST}_k(\alpha_m)=\mathtt{FIRST}_k(w)$. U svakom koraku promatramo samo k ili manje početnih znakova.

♦ Proširena gramatika

Neka je dana beskontekstna gramatika $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$. Definira se proširena gramatika izvedena iz $\mathcal{G}$, gramatika $\mathcal{G}'$ dobivena proširenjem gramatike $\mathcal{G}$,

$$\mathcal{G}' = (\mathcal{N}\cup\{\mathcal{S}'\}, \mathcal{T}, \mathcal{P}\cup\{\mathcal{S}'\rightarrow\mathcal{S}\}, \mathcal{S}')$$

Dakle, gramatika $\mathcal{G}'$ dobivena je iz $\mathcal{G}$ dodavanjem novog početnog simbola, $\mathcal{S}'$, i produkcije $\mathcal{S}'\rightarrow\mathcal{S}$.

♦ Gramatika tipa *LR*(*k*)

Neka je dana beskontekstna gramatika $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ i njezina proširena gramatika, $\mathcal{G}' = (\mathcal{N}', \mathcal{T}, \mathcal{P}', \mathcal{S}')$. Kaže se da je gramatika $\mathcal{G}$ tipa $LR(\mathtt{k})$, $\mathtt{k}\geq 0$, ako tri uvjeta:

1) $\mathcal{S}' \underset{\mathcal{G}'\,rm}{*\Rightarrow} \alpha\mathtt{A}w \underset{\mathcal{G}'\,rm}{\Rightarrow} \alpha\beta w$

2) $\mathcal{S}' \underset{\mathcal{G}'\,rm}{*\Rightarrow} \gamma\mathtt{B}x \underset{\mathcal{G}'\,rm}{\Rightarrow} \alpha\beta y$

3) $\mathtt{FIRST}_k(w)=\mathtt{FIRST}_k(y)$

impliciraju $\alpha\mathtt{A}y=\gamma\mathtt{B}x$ (tj. $\alpha=\gamma$, $\mathtt{A}=\mathtt{B}$ i $y=x$).

Intuitivno, prethodna definicija kaže da su $\alpha\beta w$ i $\alpha\beta y$ desne rečenične forme proširene gramatike s $\mathtt{FIRST}_k(w)=\mathtt{FIRST}_k(y)$, i ako je $\mathtt{A}\rightarrow\beta$ posljednje upotrijebljena produkcija u izvođenju $\alpha\beta w$ u krajnjem izvođenju zdesna, tada $\mathtt{A}\rightarrow\beta$ mora također biti upotrijebljeno u reduciranju $\alpha\beta y$ na $\alpha\mathtt{A}y$ u parsanjui zdesna. Budući da $\mathtt{A}$ može izvesti β neovisno o w, $LR(k)$ uvjet kaže da postoji dostatna informacija u $\mathtt{FIRST}_k(w)$ na temelju koje se može zaključiti da je $\alpha\beta$ bilo izvedeno iz $\alpha\mathtt{A}$. Dakle, nikada ne može postojati dilema kako treba reducirati bilo koju desnu rečeničnu formu proširene gramatike. Osim toga, s $LR(k)$ gramatikom uvijek se zna može li se prihvatiti ulazni niz ili se nastavlja analiza. Ako se početni simbol ne pojavljuje na desnoj strani niti u jednoj produkciji, može se alternativno definirati gramatiku $LR(\mathtt{k})$ kao gramatiku $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},\mathcal{S})$ u kojoj tri uvjeta:

1) $\mathcal{S} \underset{\mathcal{G}'\,rm}{*\Rightarrow} \alpha\mathtt{A}w \underset{\mathcal{G}'\,rm}{\Rightarrow} \alpha\beta w$ 2) $\mathcal{S} \underset{\mathcal{G}'\,rm}{*\Rightarrow} \gamma\mathtt{B}x \underset{\mathcal{G}'\,rm}{\Rightarrow} \alpha\beta y$ 3) $\mathtt{FIRST}_k(w)=\mathtt{FIRST}_k(y)$

impliciraju $\alpha\mathtt{A}y=\gamma\mathtt{B}x$.

♣ Primjer 7.8

Pogledajmo gramatiku $\mathcal{G}$ s produkcijama

$$\mathtt{S} \rightarrow \mathtt{SaSb} \mid \varepsilon$$

Proširena gramatika je

$$\mathtt{S'} \rightarrow \mathtt{S} \quad \mathtt{S} \rightarrow \mathtt{SaSb} \mid \varepsilon$$

Iz tri uvjeta:

1) $S' \overset{*}{\underset{\mathcal{G}'rm}{\Rightarrow}} SaSaSbb \underset{\mathcal{G}'rm}{\Rightarrow} SaSabb$
2) $S' \overset{*}{\underset{\mathcal{G}'rm}{\Rightarrow}} SaSbab \underset{\mathcal{G}'rm}{\Rightarrow} SaSaSbbab$
3) $FIRST_1(b)=FIRST_1(b)=b$

slijedi da je SaSb=Sab, što ne implicira da je S'a=S.

♣ Primjer 7.9

Pogledajmo gramatiku $\mathcal{G}$ s produkcijama

$S \rightarrow Sa \mid a$

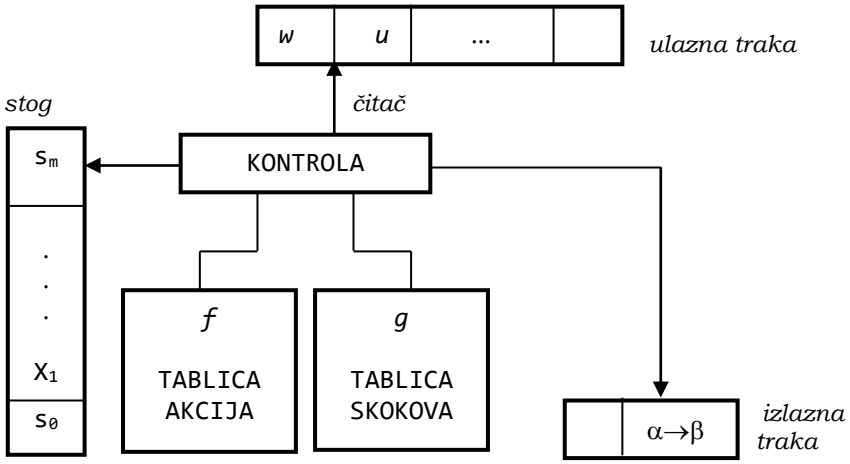
Ako zanemarimo činjenicu da se početni simbol pojavljuje na desnoj strani produkcije, tj. ako gledamo drugu alternativu, $\mathcal{G}$ bi mogla biti *LR*(0) gramatika. Međutim, primjenjujući definiciju, $\mathcal{G}$ nije *LR*(0), budući da uvjeti

1) $S' \overset{0}{\underset{\mathcal{G}'rm}{\Rightarrow}} S' \underset{\mathcal{G}'rm}{\Rightarrow} S$
2) $S' \underset{\mathcal{G}'rm}{\Rightarrow} S \underset{\mathcal{G}'rm}{\Rightarrow} Sa$
3) $FIRST_0(\varepsilon)=FIRST_0(a)=\varepsilon$

ne impliciraju da je S'a=S.

7.3 SINTAKSNA ANALIZA *LR*(1) JEZIKA

U praksi se često promatra klasa *LR*(1) jezika. U teoriji je formalnih jezika poznato da svi deterministički beskontekstni jezici imaju gramatiku tipa *LR*(1). Ta je klasa gramatika posebno važna u oblikovanju prevodilaca većine jezika za programiranje. Općenito se model SA jezika tipa *LR*(1) može predočiti kao što je prikazano na sljedećem crtežu:



Sl. 7.1 – *Model sintaksne analize LR(1) jezika.*

Vidimo da je to pretvornik sastavljen od ulazne trake, čitača, kontrole, stoga, tablice akcija, tablice skokova i izlazne trake.

Tablica akcija f i tablica skokova g, diktiraju ponašanje postupka sintaksne analize. Sadržaj stoga u bilo kojem trenutku jest niz:

$s_0X_1s_1X_2s_2 \ \dots \ X_ms_m$

gdje su s_i simboli *stanja*, a X_i simboli iz gramatike. Ukratko, postupak SA $LR(1)$ jezika ponaša se na sljedeći način. Najprije pretpostavimo da stog sadrži niz $s_0X_1s_1X_2s_2\dots X_ms_m$. Kontrola određuje s_m, stanje na vrhu potisne liste, i a, tekući ulazni znak. Potom ispituje $f(s_m,a)$, ulaz za s_m i a u tablici akcija. Taj ulaz može prouzročiti jednu od četiriju različitih vrsta akcija, ovisno o vrijednosti funkcije f:

1) $f(s_m,a) = \{$ shift $s\}$
   Premješta se ulazni simbol a na vrh potisne liste, potom stanje s. Nakon toga potisna lista sadrži:

   $s_0X_1s_1X_2s_2 \ \dots \ X_ms_mas$

2) $f(s_m,a) = \{$ reduciranje produkcijom $A\to Y_1Y_2\dots Y_r \}$
   Izuzima se $2r$ simbola iz potisne liste, pa je s_{m-r} sada na vrhu. Zatim se konzultira tablica skokova, tj. $g(s_{m-r},A)$. Ako je vrijednost te funkcije jednaka s, upisuje se As na vrh potisne liste. Produkcija $A\to Y_1Y_2\dots Y_r$ ispisuje se na izlaznu traku. Čitač se pomiče za jedno mjesto udesno. Poslije toga sadržaj potisne liste će biti:

   $s_0X_1s_1 \ \dots \ X_{m-r}s_{m-r}As$

3) $f(s_m,a) = \{$ accept $\}$
   Prekida se postupak i prihvaća ulazni niz kao sintaksno korektan.

4) Ako je $f(s_m,a)=\{$ error $\}$ ili ako se ne može dosegnuti nijedno od danih premještanja, dojavljuje se pogreška i prekida postupak.

Postupak sintaksne analize $LR(1)$ jezika započinje nekom početnom konfiguracijom u kojoj ulazna traka sadrži niz koji treba analizirati, sa znakom $ na svom kraju. Čitač je ispod prvog pravokutnika slijeva na ulaznoj traci. Potisna lista sadrži samo početno stanje s_0. Izlazna je traka prazna. Postupak SA potom obavlja niz premještanja, sve dok ne prihvati ulazni niz, ako je u jeziku, ili dok ne dojavi pogrešku.

♣ Primjer 7.10

Razmotrimo gramatiku G s produkcijama: (1) S → S(S), (2) S → ε. Tablice sintaksne analize i skokova su:

	(	)	$
0	red 2	err	red 2
1	sh 2	err	acc
2	red 2	red 2	err
3	sh 4	sh 5	err
4	red 2	red 2	err
5	red 1	err	red 1
6	sh 4	sh 7	err
7	red 1	red 1	err

	S
0	1
1	err
2	3
3	err
4	6
5	err
6	err
7	err

Stanje 0 je početno. Ovdje sh i stoji umjesto "premjesti stanje i u potisnu listu", red i označuje da treba reducirati produkciju numeriranu s i, acc stoji umjesto accept i err umjesto error. Pogledajmo kako će postupak sintaksne analize analizirati ulazni niz w=(()). Vrh stoga je na desnoj strani. Pozicija čitača označena je sa "↑". Postupak će obaviti premještanja:

	Stog	Ulazni niz	Izlaz	Opis
početak	0	(()) ↑		
1.	0S1	(())$ ↑	S→ε	Početno je stanje 0. Ulazni je znak (. Akcija je f[0,)]=red 2 pa treba upotrijebiti produkciju S→ε za reduciranje. Izbacuje se 2r znakova s vrha stoga (r=0, pa se ne izbacuje nijedan znak). Na vrhu stoga je stanje 0. g(0,S) je 1. Na vrh stoga premješta se najprije S pa stanje 1. Na izlaznu se traku upisuje produkcija kojom se reduciralo.
2.	0S1(2	())$ ↑		Na vrh se stoga premješta tekući znak (, potom stanje 2 (jer je f[1,)]=sh 2). Čitač se pomiče.
3.	0S1(2S3	())$ ↑	S→ε	Reducira se niz na vrhu stoga (kao u prvom koraku).
4.	0S1(2S3(4	))$ ↑		Najprije se na vrh stoga premješta znak (, potom stanje 4. Čitač se pomiče za jedno mjesto udesno.
5.	0S1(2S3(4S6	))$ ↑	S→ε	Opet se reducira prazan niz u S i unosi stanje 6 na vrh stoga.
6.	0S1(2S3(4S6)7	)$ ↑		Na vrh se stoga premješta tekući znak (, potom stanje 7. Čitač se pomiče za jedno mjesto udesno.
7.	0S1(2S3	)$ ↑	S→S(S)	Niz na vrhu stoga može se reducirati u S, koristeći prvu produkciju S→S(S). Izbacuje se 2r znakova s vrha stoga (r=4, pa će biti izbačeno 8 znakova). Na vrhu je stanje 2, pa je g(2,S)=3. Na vrh se stoga upisuje S pa stanje 3. Na izlaznu se traku ispisuje produkcija S→S(S).
8.	0S1(2S3)5	$ ↑		Na vrh se stoga premješta tekući znak), potom stanje 5. Čitač se pomiče za jedno mjesto udesno i sada je na oznaci kraja ulaznog niza, znaku $.
9.	0S1	$ ↑	S→S(S) accept	Opet se, kao u koraku 7, reducira niz na vrhu stoga koristeći prvu produkciju. Poslije toga na vrhu je stoga stanje 0, pa je vrijednost funkcije g za ulaz (0,S) jednaka 1. Dakle, na vrh se stoga upisuje S1. Na izlaznu se traku ispisuje S→S(S). Tekuće je stanje 1, tekući znak $, pa je f[1,$]=acc, tj. ulazni niz je prihvaćen.

Niz izvođenja je: S ⇒ S(S) ⇒ S(S(S)) ⇒ S(S()) ⇒ S(()) ⇒ (())

7.4 GRAMATIKE S RELACIJOM PRIORITETA

Postoji još nekoliko deterministčkih uzlaznih postupaka sintaksne analize koji rade nad podskupovima LR jezika. Posebno je zapažena sintaksna analiza jezika sa slabim prioritetom i s prioritetom operatora koje ćemo ovdje opisati.

♦ Relacija prioriteta

Neka je $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ svojstvena gramatika (tj. nema praznih produkcija niti petlji) i $ novi znak koji nije u $\mathcal{N}\cup\mathcal{T}$. Definiraju se tri relacije prioriteta $\lessdot$, $\doteq$ i $\gtrdot$ na $\mathcal{N}\cup\mathcal{T}\cup\{\$\}$ na sljedeći način:

1) $X \lessdot Y$ ako postoji produkcija $A\rightarrow\alpha XB\beta$ u $\mathcal{P}$ tako da $B^{+}\Rightarrow Y\gamma$ za neki γ.
2) $X \doteq Y$ ako postoji produkcija $A\rightarrow\alpha XY\beta$.
3) $X \gtrdot Y$, Y je terminal, ako postoji produkcija $A\rightarrow\alpha XZ\beta$ tako da $B^{*}\Rightarrow\gamma X$ i $Z^{*}\Rightarrow Y\delta$ za neke γ i δ.
4) $\$ \lessdot Y$ ako $S^{+}\Rightarrow Ya$ za neki a.
5) $X \gtrdot \$$ ako $S^{+}\Rightarrow\alpha X$ za neki a.

♦ Gramatika sa slabim prioritetom

Gramatika $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ jest gramatika sa slabim prioritetom ako:

1) $\mathcal{G}$ je svojstvena.
2) Definirana je najmanje jedna relacija prioriteta između svakog para simbola iz $\mathcal{N}\cup\mathcal{T}\cup\{\$\}$.
3) Ne postoje dvije produkcije koje imaju jednaku desnu stranu.

Za gramatike $\mathcal{G}$ sa slabim prioritetom može se definirati parser (prepoznavač) kao automat koji se sastoji od ulazne trake, stoga i matrice relacija prioriteta za $\mathcal{G}$. Ulazni niz $x=a_1a_2\ldots a_n$ koji se analizira nalazi se na ulaznoj traci i ima $ na svom kraju. Inicijalno se čitač nalazi na prvom znaku i stog sadrži $. Postupak će SA tada načiniti niz pomaka sve dok ne prihvati ulazni niz ili ne dojavi pogrešku.

Pomaci će biti određeni relacijom prioriteta između znaka na vrhu stoga i tekućeg znaka. Pretpostavimo da stog sadrži niz $X_1X_2\ldots X_m$, gdje je X_m na vrhu, te da je a_i tekući znak ($a_{n+1}=\$$). Idući pomak određen je vrijednošću matrice relacije prioriteta između X_m i a_i:

1) Ako je $X_m \lessdot a_i$ ili $X_m \doteq a_i$, premješta se a_i na vrh stoga.
2) Ako je $X_1X_2\ldots X_m=\$\mathcal{S}$ i $a_i=\$$, prekida se postupak SA i prihvaća ulazni niz.
3) Ako je $X_m \gtrdot a_i$, pretražuje se stog dok se ne pronađe simbol X_k tako da je $X_{k-1} \lessdot X_k \doteq X_{k+1} \doteq \ldots \doteq X_m$. Tada se pronalazi produkcija $A\rightarrow X_kX_{k+1}\ldots X_m$ i zamjenjuje $X_kX_{k+1}X_m$ na stogu s A. Iz definicije gramatike s jednostavnim prioritetom može postojati najviše jedna produkcija s takvim svojstvom. Ako skup produkcija $\mathcal{P}$ ne sadrži takvu produkciju ili ako za neki $j\leq m$, $X_j\doteq X_{j+1}\doteq\ldots\doteq X_m$ ne postoji relacija prioriteta između X_{j-1} i X_j, dojavljuje se pogreška i prekida analiza.
4) Ako ne postoji relacija prioriteta između X_m i a_i, dojavljuje se pogreška i prekida analiza.

Svaka gramatika sa slabim prioritetom istodobno je tipa $LR(1)$. No, klasa jezika generiranih gramatikama sa slabim prioritetom mali je podskup deterministčkih beskontekstnih jezika.

♦ Gramatika s jakim prioritetom

Svojstvena gramatika $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ jest gramatika s jakim prioritetom ako:

1) Operacija prioriteta $\dot{>}$ prema uniji $<$ i $\doteq$.
2) Ako su $A \to \alpha X \beta$ i $B \to \beta$ produkcije, ne postoji nijedna operacija prioriteta $X \mathrel{<\!\!\cdot} B$ niti $X \doteq B$ (tj. X se ne smije pojaviti ispred B ni u jednoj rečeničnoj formi).

Parser gramatike s jakim prioritetom operira slično parseru gramatika sa slabim prioritetom. Pretpostavimo da stog sadrži $X_1 \ldots X_m$ i da je a_i tekući ulazni simbol. Da bi načinio sljedeći pomak, parser izvodi relaciju prioriteta između X_m i a_i:

1) Ako je $X_m \mathrel{<\!\!\cdot} a_i$ ili $X_m \doteq a_i$, premješta se a_i na vrh stoga.
2) Ako je $X_1 \ldots X_m = \$\mathcal{S}$ i $a_i = \$$, prekida se postupak SA i prihvaća ulazni niz.
3) Ako je $X_m \mathrel{\dot{>}} a_i$, pretražuje se lista produkcija gramatike dok se ne pronađe najdulja desna strana jednaka sufiksu od $X_1 \ldots X_m$. Ako je produkcija $A \to X_k \ldots X_m$ s takvim svojstvom, parser zamjenjuje $X_1 \ldots X_m$ na vrhu s A.
4) Ako nijedan od prethodnih uvjeta ne stoji, dojavljuje se pogreška i prekida analiza.

♦ Operatorska gramatika

Operatorska gramatika jest gramatika bez praznih produkcija, s alternativama koje sadrže bar jedan terminal, a ne sadrže dva uzastopna neterminala.

♣ Primjer 7.11

Gramatika s produkcijama

$$S \to SOS \mid (S) \mid -S \mid x \qquad O \to + \mid - \mid * \mid /$$

nije operatorska gramatika, jer u alternativi SOS nije zadovoljen uvjet da ne smiju biti dva uzastopna neterminala (tu su čak tri). No, zamijenimo li O sa svim njegovim alternativama, dobit ćemo operatorsku gramatiku:

$$S \to S+S \mid S-S \mid S*S \mid S/S \mid (S) \mid -S \mid x$$

Za operatorsku gramatiku $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ definirane su relacije prioriteta operatora $<\!\!\cdot$, $\mathrel{\dot{>}}$ i $\doteq$ nad $\mathcal{T} \cup \{\$\}$ na sljedeći način:

1) $a \mathrel{<\!\!\cdot} b$ ako je $A \to \alpha a B \beta$ produkcija tako da $B^+ \Rightarrow \gamma b \delta$ za neki γ iz $\mathcal{N} \cup \{\varepsilon\}$.
2) $a \doteq b$ ako je $A \to \alpha a \beta b \gamma$ produkcija tako da je β iz $\mathcal{N} \cup \{\varepsilon\}$.
3) $a \mathrel{\dot{>}} b$ ako je $A \to \alpha B b \beta$ produkcija tako da $B^+ \Rightarrow \gamma a \delta$ za neki δ iz $\mathcal{N} \cup \{\varepsilon\}$.
4) $\$ \mathrel{<\!\!\cdot} b$ ako $\mathcal{S}^+ \Rightarrow \alpha b \beta$ za neki α iz $\mathcal{N} \cup \{\varepsilon\}$.
5) $a \mathrel{\dot{>}} \$$ ako $\mathcal{S}^+ \Rightarrow \alpha a \beta$ za neki β iz $\mathcal{N} \cup \{\varepsilon\}$.

♦ Gramatika s prioritetom operatora

Gramatika $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ jest gramatika s prioritetom operatora ako je operatorska i ako postoji najviše jedna relacija prioriteta operatora između svakog para simbola iz $\mathcal{T} \cup \{\$\}$.

♣ Primjer 7.12

Gramatika s produkcijama:

$$E \to E+T \mid T \qquad T \to T*F \mid F \qquad F \to (E) \mid a$$

jest gramatika s prioritetom operatora. U sljedećoj su tablici prikazane relacije prioriteta operatora između pojedinih parova simbola:

	$	(	+	*	a	)
$		<·	<·	<·	<·	
(		<·	<·	<·	<·	≅
+	·>	<·	·>	<·	<·	·>
*	·>	<·	·>	·>	<·	·>
a	·>		·>	·>		·>
)	·>		·>	·>		·>

Na primjer, iz produkcije $F \to (E)$ slijedi da je $(\cong)$, ili iz $E \to E+T$ i $T^+ \Rightarrow T*F$ i $T^+ \Rightarrow (E)$, slijedi da je $+ <\cdot *$ i $+ <\cdot ($, itd.

♦ Skeletna gramatika

Neka je $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ operatorska gramatika. Definira se skeletna gramatika

$\mathcal{G}_S=(\{\mathcal{S}\}, \mathcal{T}, \mathcal{P}', \mathcal{S})$

za $\mathcal{G}$ koja se sastoji od produkcija

$\mathcal{S} \to X_1 \ldots X_m$

tako da postoji produkcija $A \to Y_1 \ldots Y_m$ u $\mathcal{P}$ i, za $1 \le i \le m$, vrijedi:

(1) $X_i = Y_i$ ako je $Y_i \in \mathcal{T}$
(2) $X_i = \mathcal{S}$ ako je $Y_i \in \mathcal{N}$

i nije dopuštena produkcija $\mathcal{S} \to \mathcal{S}$ u $\mathcal{P}'$.

Dakle, skeletna gramatika $\mathcal{G}_S$ gramatike $\mathcal{G}$ s prioritetom operatora jest gramatika dobivena zamjenom svih neterminala u produkcijama početnim simbolom i, potom, izbacivanjem produkcija $\mathcal{S} \to \mathcal{S}$, ako postoje. Općenito će vrijediti uvjet da je $\mathcal{L}(\mathcal{G}) \subseteq \mathcal{L}(\mathcal{G}_S)$, tj. $\mathcal{L}(\mathcal{G}_S)$ može sadržavati rečenice koje nisu u $\mathcal{L}(\mathcal{G})$.

♣ Primjer 7.13

Skeletna gramatika $\mathcal{G}_S$ gramatike $\mathcal{G}$ iz primjera 6.18 jest:

$E \to E+E \mid E \quad E \to E*E \mid E \quad E \to (E) \mid a$

odnosno, poslije izbacivanja produkcija $E \to E$:

$E \to E+E \mid E*E \mid (E) \mid a$

Primijetiti da je $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}_S)$.

♥ Algoritam 7.1 *Parser gramatika (jezika) s prioritetom operatora*

Ulaz
Gramatika s prioritetom operatora, $\mathcal{G}=(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$.

Izlaz
Funkcije akcije i skokova, f i g, za $\mathcal{G}_S$.

Postupak
Neka je β jednako $\mathcal{S}$ ili ε.

(1) $f(a\beta,b)$ = shift ako je $a \lessdot b$ ili $a \doteq b$.
(2) $f(a\beta,b)$ = reduce ako je $a \gtrdot b$.
(3) $f(\$\mathcal{S},\$)$ = accept.
(4) $f(\alpha,w)$ = error, inače.
(5) $g(a\beta b\gamma,w)$ = i ako je

a) β jednako $\mathcal{S}$ ili ε
b) $a \lessdot b$
c) Vrijedi relacija $\doteq$ između prva dva terminala od γ, ako postoje, i
d) $\mathcal{S} \rightarrow \beta b \gamma$ je produkcija označena s i u G_S.

(6) $g(\alpha,w)$ = error, inače.

Parser gramatike s prioritetom operatora radi slično parseru gramatika sa slabim prioritetom, ali najčešće se realizira za skeletnu gramatiku. Skeletna gramatika može biti i višeznačna (a može biti i ekvivalentna osnovnoj gramatici, kao što je to bio slučaj u prethodnom primjeru), ali relacije prioriteta operatora uvijek će jednoznačno odrediti moguću analizu ulaznog niza. Štoviše, ustrojbom parsera skeletne gramatike uvijek se mogu koristiti originalne produkcije osnovne gramatike pri dojavi pogreške, ako to želimo. Ono što treba posebno istaknuti jest da je jezik $\mathcal{L}(S_S)$ najčešće nadskup jezika $\mathcal{L}(S)$, pa je postupak sintaksne analize uporabom skeletne gramatike općenitiji.

♣ Primjer 7.14

Parser gramatike jednostavnih aritmetičkih izraza, koristeći tablicu relacija prioriteta operatora iz primjera 7.12, analizirat će ulazni niz a+a*a kao što slijedi:

	stog	preostali ulaz	opis
1)	$	a+a*a$	premještanje a
2)	$a	+a*a$	reduciranje s E→a
3)	$E	+a*a$	premještanje +
4)	$E+	a*a$	premještanje a
5)	$E+a	*a$	reduciranje s E→a
6)	$E+E	*a$	premještanje *
7)	$E+E*	a$	premještanje a
8)	$E+E*a	$	reduciranje s E→a
9)	$E+E*E	$	reduciranje s E→E*E
10)	$E+E	$	reduciranje s E→E+E
11)	$E	$	prihvaćanje

P R O G R A M I

U ovom dijelu dajemo ustroj nekoliko algoritama koji se odnose na sintaksnu analizu *LR(0)* jezika i nekoliko primjera sintaksne analize.

Stavke.py *Stavke beskontekstne gramatike*

```
from gramatika import *

def Stavke (G):
  N, T, P, S = G;  P2 = []
  for X in P:
    Z = [X[0]]
    for Y in X[1:]:
      if Y <> '#' :
        Z.append (O +Y)
        for i in range (len(Y)):
          Z.append (Y[:i+1] +O +Y[i+1:])
      else:
        Z.append (O)
    P2.append (Z)

  return (N, T, P2, S)

def Ispisi_S (P):
  print 'Stavke gramatike:', NL
  for j in range (len(P)) :
    print ' ', P[j][0] + ' ->', P[j][1],
    for k in range (2, len(P[j])) : print '|', P[j][k],
    print
  print NL
"""
Grm, Ok, G = Ucitaj_G ('STAVKE GRAMATIKE:', '*.grm')
if Ok:
  Ispisi_G (Grm, G);  P2 = Stavke (G);  Ispisi_S (P2)
else :
  print ('Ne postoji gramatika s danim imenom!')
"""
```

NFA.py *Nedeterministički prepoznavač održivih prefiksa*

```
def NFA (G):
  N, T, P, S = G;  T.sort();  O = chr(183)
  Q   = [x for x in range (len(P)+1)]
  NT  = ['#'] +N +T;  NT.remove (S)
  q0  = Q[0]
  F   = [];  TP = [[[Q[1]]]]
  for j in range (1, len(NT)): TP[0].append ([])
  for i in range (1, len(Q)) :
    X =  [[]]
    for j in range (1, len(NT)): X.append ([])
    TP.append (X)
  for i in range (len(P)):
    X, Y = P[i]
    if Y[-1] == O:
      F.append (i+1)
    else:
      j = Y.find (O);  x = Y[j+1];  j = NT.index(x);  TP[i+1][j].append (i+2)
      if x in N:
        for q in range (len(P)):
          A, beta = P[q]
          if A == x and beta[0] == O: TP[i+1][0].append (q+1)
  return (Q, NT, TP, q0, F)
```

```
def Ispisi_NFA (M, P):
  print NL, 'N K A', NL
  Q, NT, TP, q0, F = M
  print '   ',
  for C in NT: print '%6s' % C,
  print
  for i in range (len(TP)):
    p = ' '
    if Q[i] in F: p ='x'
    print p +'%2d' %Q[i],
    for j in range (len(TP[i])):
      r = str (TP[i][j]);  r = r[1:-1]
      print '%6s' % r,
    if i > 0: print '  ' +P[i-1][0] +' -> ' +P[i-1][1],
    print
```

DFA.py *Deterministički prepoznavač održivih prefiksa*

```
importe Stavke, NFA

def DFA (M, P): # DFA iz NFA

  def Dodaj_It (T, p, s):
    Q_ = T[p][0]
    if Q_ != []:
      for p_ in Q_:
        It = P[p_-1]
        if It not in I[s]: I[s].append(It)
        n = T[p_][0]
        if n != [] and n != Q_: p = p_; Dodaj_It (T, p, s)

  Q, NT, T, q0, F = M;  O = chr(183)
  I = [[P[0]]]
  for i in range (1, len(T)):
    s = T[i][0]
    for j in s:
      if P[j-1] not in I[0]: I[0].append (P[j-1])

  TP = [];  L = [];  F = [];  q = 0;  i = 0
  while i < len (I):
    TP.append ([])
    for j in range (1, len(NT)): TP[i].append ([])
    if len (I[i]) == 1: F.append(i)
    else:
      for It in I[i]:
        X, Y = It;  k = P.index (It) +1
        if Y[-1] != O:
          j = Y.find (O);  x = Y[j+1];  j = NT.index(x)
          if T[k][j] != []:
            p = T[k][j][0];  Q = TP[i][j-1];  It = P[p-1]
            if Q == []:
              I.append ([It]);  q += 1;  s = q;  Q.append (s)
            else:
              s = Q[0]
              if It not in I[s]: I[s].append(It)
            Dodaj_It (T, p, s)
```

```
      R = TP[i]
      for r in range(len(R)):
        if R[r] != []:
          s = R[r][0]
          if I[s] in I and s != I.index(I[s]):
            p = I.index(I[s]);  TP[i][r] = [p]; I = I[:s] +I[s+1:];  q -= 1
            for o in range (r+1, len(R)):
              if R[o] != []:
                if s < R[o][0] : R[o] = [R[o][0] -1]
      i += 1
    NT = NT[1:]
    return (I, NT, TP, 0, F)

  def Ispisi_DFA (M):
    print NL, 'D F A', NL
    I, NT, TP, q0, F = M
    print '   ',
    for C in NT: print '%4s' % C,
    print NL, '-----' +'-'*5*len(NT)
    for i in range (len(TP)):
      p = ' I'
      if i == 0: p = '>I'
      if i in F: p ='xI'
      print p +'%2d' %i,
      for j in range (len(TP[i])):
        r = str (TP[i][j]); r = r[1:-1]
        if r != '': r = 'I' +r
        print '%4s' % r,
      print
    for i in range (len(I)):
      print NL, 'I'+str(i)
      for S in I[i]: x, y = S; print x +' -> ' +y

  Grm, Ok, G = Ucitaj_G ('STAVKE GRAMATIKE:', '*.grm')
  if Ok:
    Ispisi_G (Grm, G);  G2 = Stavke (G)
    N, T, P, S = G2;  P = Uredi_P (P);  G = (N, T, P, S)
    Mnfa = NFA (G);       Ispisi_NFA (Mnfa, P)
    Mdfa = DFA (Mnfa, P); Ispisi_DFA  (Mdfa)
  else:
    print ('Ne postoji gramatika s danim imenom!')
```

LR(0) SINTAKSNA ANALIZA

LR0.py *LR(0) sintaksna analiza*

```
from gramatika import *
from DFA import *

def DSA (M, w):

  def Ispis_ (i, stog, w, t):
     print '%2d' %i,
     for x in stog: print x,
     print ' '*(20-len(stog)*2), w, ' '*(20-len(w)), t
```

```
def Niz_I (Alt):
  SF = Alt[1][0]
  print SF
  for k in range (1, len(Alt)):
    i = SF.rfind(Alt[k][0])
    SF = SF[:i] +Alt[k][1] +SF[i+1:]
    print '  =>', SF
  print

Q, NT, TP, q0, F = M;  Err = False;  O = chr(183)
stog = [q0];  BC = 0;  t = 'inicijalizacija';  R = []
while not Err and stog != []:
  BC  += 1
  Ispis_ (BC, stog, w, t)
  if stog[-1] not in F:
    if w != '':
      Sym  = w[0]; q = stog [-1];
      if Sym in NT:
        p = NT.index(Sym);  w = w[1:];  n = TP[q][p]
        if n != []:
          q = n[0]
          t = 'premještanje'
          stog.append (Sym);  stog.append (q)
        else: Err = True
      else: Err = True
    else: Err = True
  else:
    q = stog[-1]
    for j in range (len (Q[q])):
      if Q[q][j][1][-1] == O: break
    A, beta = Q[q][j]; beta = beta[:-1]
    k = len(stog) -2*len(beta)
    x = ''
    for i in range (k, len(stog), 2): x += stog[i]
    if x == beta:
      stog = stog[:k] +[A]
      q    = stog[k-1]
      if A in NT:
        t = 'reduciranje s ' +A +' -> ' +beta
        p = NT.index(A); n = TP[q][p]
        stog.append(n[0])
      else:
         t = 'prihvaćanje  s ' +A +' -> ' +beta
         stog = []
      P = [A, beta]
      if R == []: R.append (P)
      else      : R = [P] +R
    else:
      Err = True

if not Err:
  Ispis_ (BC+1, stog, w, t)
  Niz_I (R)
else:
  print 'Niz nije rečenica jezika!'

return Err
```

```
   Grm, Ok, G = Ucitaj_G ('STAVKE GRAMATIKE:', '*.grm')
   if Ok:
     Ispisi_G (Grm, G)
     N, T, P, S = G;   P = [[chr(140), S+'@']] +P; S = chr(140); N = [S] +N
     G = (N, T, P, S); G2 = Stavke (G)
     N, T, P, S = G2;  P = Uredi_P (G2[2]);  G = (N, T, P, S)
     Mnfa = NFA (G);       # Ispisi_NFA (Mnfa, P)
     Mdfa = DFA (Mnfa, P); # Ispisi_DFA (Mdfa)
     w = Ucitaj_W();
     while len(w) > 0: w += '@'; Ok = DSA (Mdfa, w);  w = Ucitaj_W()
   else :
     print ('Ne postoji gramatika s danim imenom!')
```

```
Exp0.GRM

N = { E }
T = { + , * , ( , ) , a }
S = E
P:
  E -> E+E | E*E | (E) | a
```

```
D F A

       E    (    )    *    +    a    @
-----------------------------------------
>I 0   I1   I2                   I3
 I 1                 I6   I5         I4
 I 2   I7   I2                   I3
xI 3
xI 4
 I 5   I8   I2                   I3
 I 6   I9   I2                   I3
 I 7             I10   I6   I5
xI 8                   I6   I5
xI 9                   I6   I5
xI10

I0
Ś -> ·E@   E -> ·E+E   E -> ·E*E   E -> ·(E)   E -> ·a

I1
Ś -> E·@   E -> E·+E   E -> E·*E

I2
E -> (·E)   E -> ·E+E   E -> ·E*E   E -> ·(E)   E -> ·a

I3
E -> a·

I4
Ś -> E@·

I5
E -> E+·E   E -> ·E+E   E -> ·E*E   E -> ·(E)   E -> ·a

I6
E -> E*·E   E -> ·E+E   E -> ·E*E   E -> ·(E)   E -> ·a
```

```
I7
E -> (E·)   E -> E·+E   E -> E·*E

I8
E -> E+E·   E -> E·+E   E -> E·*E

I9
E -> E*E·   E -> E·+E   E -> E·*E

I10
E -> (E)·

Upiši ulazni niz: (a+a)*a
 1 0                    (a+a)*a@            inicijalizacija
 2 0 ( 2                a+a)*a@             premještanje
 3 0 ( 2 a 3            +a)*a@              premještanje
 4 0 ( 2 E 7            +a)*a@              reduciranje s E -> a
 5 0 ( 2 E 7 + 5        a)*a@               premještanje
 6 0 ( 2 E 7 + 5 a 3    )*a@                premještanje
 7 0 ( 2 E 7 + 5 E 8    )*a@                reduciranje s E -> a
 8 0 ( 2 E 7            )*a@                reduciranje s E -> E+E
 9 0 ( 2 E 7 ) 10       *a@                 premještanje
10 0 E 1                *a@                 reduciranje s E -> (E)
11 0 E 1 * 6            a@                  premještanje
12 0 E 1 * 6 a 3        @                   premještanje
13 0 E 1 * 6 E 9        @                   reduciranje s E -> a
14 0 E 1                @                   reduciranje s E -> E*E
15 0 E 1 @ 4                                premještanje
16                                          prihvaćanje , s Ś -> E@

E
  => E*E
  => E*a
  => (E)*a
  => (E+E)*a
  => (E+a)*a
  => (a+a)*a

   Zagrade.GRM

   N = { S }
   T = { ( , ) }
   S = S
   P:
     S -> S(S) | #

   D F A

         S    (    )    @
   -------------------------
   xI 0   I1
    I 1         I3        I2
   xI 2
   xI 3   I4
    I 4         I3   I5
   xI 5
```

```
I0
Ś -> ·S@   S -> ·S(S)   S -> ·

I1
Ś -> S·@   S -> S·(S)

I2
Ś -> S@·

I3
S -> S(·S)   S -> ·S(S)   S -> ·

I4
S -> S(S·)   S -> S·(S)

I5
S -> S(S)·


Upiši ulazni niz: (()()())
```

```
 1 0                                ()()())@        inicijalizacija
 2 0 S 1                            ()()())@        reduciranje s S ->
 3 0 S 1 ( 3                        )()())@         premještanje
 4 0 S 1 ( 3 S 4                    )()())@         reduciranje s S ->
 5 0 S 1 ( 3 S 4 ( 3                )())@           premještanje
 6 0 S 1 ( 3 S 4 ( 3 S 4            )())@           reduciranje s S ->
 7 0 S 1 ( 3 S 4 ( 3 S 4 ) 5        ()())@          premještanje
 8 0 S 1 ( 3 S 4                    ()())@          reduciranje s S -> S(S)
 9 0 S 1 ( 3 S 4 ( 3                )())@           premještanje
10 0 S 1 ( 3 S 4 ( 3 S 4            )())@           reduciranje s S ->
11 0 S 1 ( 3 S 4 ( 3 S 4 ) 5        ())@            premještanje
12 0 S 1 ( 3 S 4                    ())@            reduciranje s S -> S(S)
13 0 S 1 ( 3 S 4 ( 3                ))@             premještanje
14 0 S 1 ( 3 S 4 ( 3 S 4            ))@             reduciranje s S ->
15 0 S 1 ( 3 S 4 ( 3 S 4 ) 5        )@              premještanje
16 0 S 1 ( 3 S 4                    )@              reduciranje s S -> S(S)
17 0 S 1 ( 3 S 4 ) 5                @               premještanje
18 0 S 1                            @               reduciranje s S -> S(S)
19 0 S 1 @ 2                                        premještanje
20                                                  prihvaćanje s Ś -> S@
```

```
S
  => S(S)
  => S(S(S))
  => S(S())
  => S(S(S)())
  => S(S()())
  => S(S(S)()())
  => S(S()()())
  => S(()()())
  => (()()())
```

Pitanja i zadaci

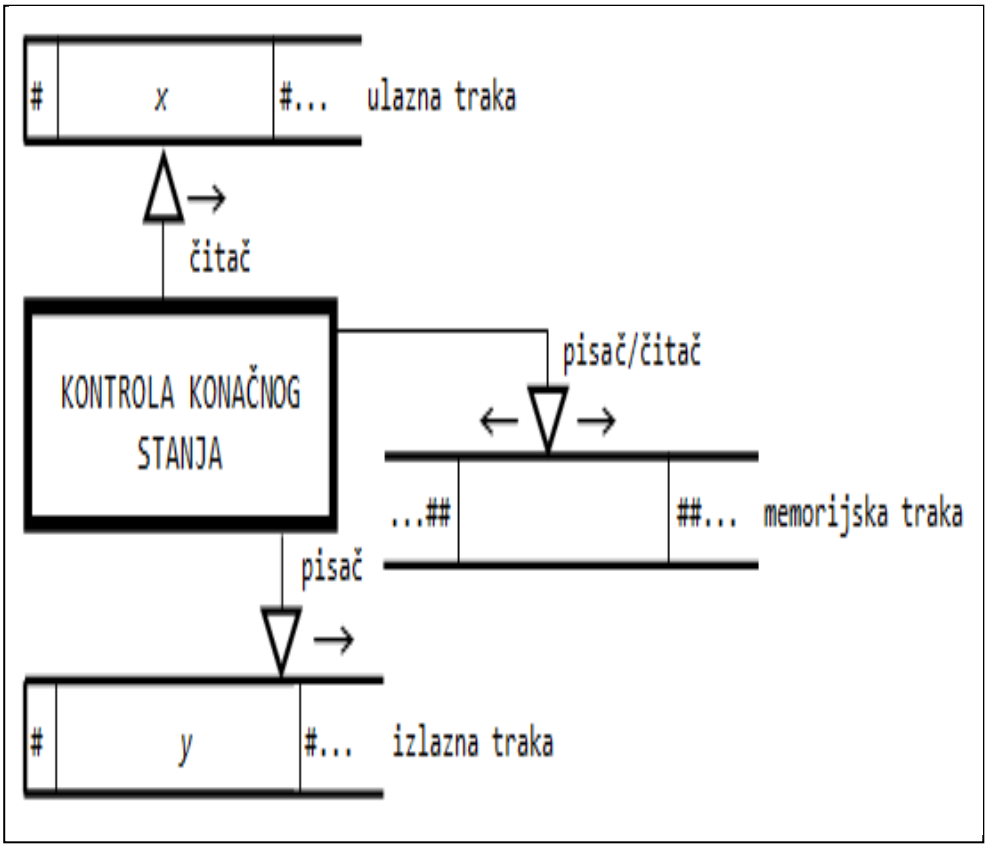
1) Definirajte tablicu akcija i skokova za gramatiku $\mathcal{G}$ *s produkcijama*

$S \rightarrow SaSb \mid \varepsilon$

Potom provjerite je li niz aabb *u jeziku* $\mathcal{L}(\mathcal{G})$.

2) Napišite program (parser) za sintaksnu analizu jezika s prioritetom operatora (algoritam 7.1)

8.
PREPOZNAVAČ KONTEKSTNIH JEZIKA



Prema Chomskyjevoj hijerarhiji gramatika i ekvivalentnih automata kontekstne gramatike su ekvivalentne linearno ograničenim automatima. U prvoj smo knjizi opisali linearno ograničeni automat u kojem je pomoćna memorija realizirana s dva stoga. Takav smo automat nazvali dvostruko-stogovni automat. Ovdje ćemo pokazati kako se dvostruko-stogovni automat koristi kao prepoznavač kontekstnih jezika.

Jezici bez ograničenja ili jezici tipa 0 najširi su klasa jezika. Mogu biti generirani gramatikama bez ograničenja i Turingovim stojevima, a sintaksna analiza moguća je samo uz pomoć Turingovog stroja – Turingovog prepoznavača.

8.1 DVOSTRUKO-STOGOVNI PREPOZNAVAČ

Vidjeli smo da nam je u slučaju prepoznavanja beskontekstnih jezika bio potreban automat koji ima pomoćnu memoriju sa strukturom stoga. Bio je to stogovni prepoznavač. Veličina memorije stoga rabljena u prepoznavanju rečenice w bila je linearno proporcionalna njezinoj duljini. Može se dokazati da takav automat ne može prepoznavati rečenice kontekstnog jezika. Za to je potrebno rabiti automate s dva stoga ili "linearno ograničeni" automat, čiju definiciju dajemo u nastavku.

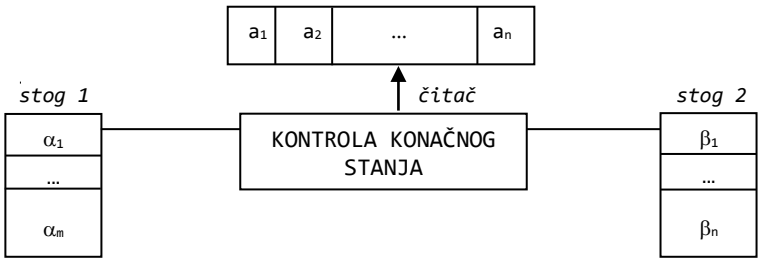
♦ Dvostruko-stogovni automat

Dvostruko-stogovni automat definiran je kao uređena sedmorka

$$\mathcal{P}t=(Q, \Sigma, \Gamma_1, \Gamma_2, \Delta, s, \mathcal{F})$$

gdje su:

Q	konačan skup stanja (kontrole konačnog stanja)
Σ	ulazni alfabet
Γ_1, Γ_2	alfabet prvog i drugog stoga
Δ	funkcija prijelaza, definirana kao $\Delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma_1 \times \Gamma_2 \rightarrow Q \times \Gamma_1^* \times \Gamma_2^*$
s	početno stanje, $s \in Q$
$\mathcal{F}$	skup konačnih stanja, $\mathcal{F} \subseteq Q$



Sl. 8.1 – *Dvostruko-stogovni prepoznavač.*

Vidimo da se ovaj automat razlikuje od stogovnog automata u definiciji funkcije prijelaza Δ koja koristi dva stoga kao pomoćnu memoriju. Kaže se da je $\mathcal{P}t$ linearno-ograničen jer je duljina oba stoga linearno proporcionalna duljini generiranog niza. Dvostruko-stogovni prepoznavač, koji ima čitač i ulaznu traku, prikazan je na sl. 8.1.

♦ Konfiguracija dvostruko-stogovnog prepoznavača

Konfiguracija dvostruko-stogovnog prepoznavača $\mathcal{P}t$ jest (q,w,α,β) iz $Q\times\Sigma^*\times\Gamma_1^*\times\Gamma_2^*$, gdje su:

q tekuće stanje
w ulazni niz znakova
α niz znakova koji predstavlja sadržaj prvog stoga; vrh je prvi znak niza
β niz znakova koji predstavlja sadržaj drugog stoga; vrh je prvi znak niza

Početna konfiguracija je $(s,w,\varepsilon,\varepsilon)$, a završna konfiguracija $(q,\varepsilon,\varepsilon,\varepsilon)$, $q\in\mathcal{F}$, $w\in\Sigma^*$.

♦ Pomak dvostruko-stogovnog prepoznavača

Pomak dvostruko-stogovnog prepoznavača $\mathcal{P}t$ jest relacija $\vdash_{\mathcal{P}t}$ (ili samo $\vdash$ ako se $\mathcal{P}t$ podrazumijeva):

$$(q, aw, \alpha, \beta) \vdash (q', w, \gamma_1\alpha, \gamma_2\beta)$$

ako $\delta(q, a, \gamma_1, \gamma_2)$ sadrži (q', γ_1, γ_2) za $q, q'\in Q$, $a\in\Sigma\cup\{\varepsilon\}$, $w\in\Sigma^*$, $\gamma_1\in\Gamma_1$, $\gamma_2\in\Gamma_2$. Kaže se da je niz w prihvatiljiv s $\mathcal{P}t$ ako

$$(s,w,\varepsilon,\varepsilon) \vdash^* (q,\varepsilon,\varepsilon,\varepsilon),\ q\in\mathcal{F}$$

Jezik definiran s $\mathcal{P}t$, označen s $\mathcal{L}(\mathcal{P}t)$, jest skup nizova w prihvatljivih s $\mathcal{P}t$. To je općenito kontekstni jezik:

$$\mathcal{L}(\mathcal{P}t)=\{w:\ w\in\Sigma^*\ \wedge\ (s,w,\varepsilon,\varepsilon) \vdash^* (q,\varepsilon,\varepsilon,\varepsilon),\ q\in\mathcal{F},\ w\in\Sigma^*\}$$

♣ Primjer 8.1

Dvostruko-stogovni automat gdje su:

$Q = \{q_0,q_1,f\}$ $\Sigma = \{a,b,c\}$ $\Gamma_1 = \{a\}$ $\Gamma_2 = \{b\}$ $s = q_0$ $\mathcal{F} = \{f\}$
$\Delta = \{ ((q_0,a,\varepsilon,\varepsilon),(q_0,a,\varepsilon)),((q_0,b,a,\varepsilon),(q_1,\varepsilon,b)),((q_1,b,a,\varepsilon),(q_1,\varepsilon,b)),((q_1,c,\varepsilon,b),(f,\varepsilon,\varepsilon)),$
$((f,c,\varepsilon,b),(f,\varepsilon,\varepsilon))\ \}$

prepoznaje kontekstni jezik $L_{abc}=\{a^n b^n c^n:\ n>0\}$. Na primjer, rečenica aaabbbccc bit će prepoznata nizom pomaka:

$(q_0, aaabbbccc, \varepsilon, \varepsilon) \vdash (q_0, aabbbccc, a, \varepsilon) \vdash (q_0, abbbccc, aa, \varepsilon) \vdash (q_0, bbbccc, aaa, \varepsilon)$
$\vdash (q_1, bbccc, aa, b) \vdash (q_1, bccc, a, bb) \vdash (q_1, ccc, \varepsilon, bbb)$
$\vdash (f, cc, \varepsilon, bb) \vdash (f, c, \varepsilon, b) \vdash (f, \varepsilon, \varepsilon, \varepsilon)$

8.2 DVOSTRUKO-STOGOVNI PREPOZNAVAČ S JEDNIM STANJEM

Rečenice kontekstnih jezika mogu biti generirane kontekstnim gramatikama ili dvostruko-stogovnim automatom (generatorom), pa očekujemo da će kontekstna gramatika zadanog kontekstnog jezika imati ekvivalentni dvostruko-stogovni (ili linearno-ograničeni) automat. Da bismo to pokazali, najprije definiramo dvostruko-stogovni automat s jednim stanjem.

♦ Dvostruko-stogovni automat s jednim stanjem

Ako je $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},S)$ kontekstna gramatika u Kurodinoj normalnoj formi, može se definirati dvostruko-stogovni automat s jednim stanjem kao uređena sedmorka

$$\mathcal{P}_q=(q,\Sigma,\Gamma_1,\Gamma_2,\Delta,z_1,z_2)$$

gdje su:

q	stanje
Σ	ulazni alfabet
Γ_1	alfabet prvog stoga, $\Gamma_1=\mathcal{N}\cup\mathcal{T}\cup\{z_2,z_\theta\}$
Γ_2	alfabet drugog stoga, $\Gamma_2=\mathcal{N}\cup\mathcal{T}\cup\{z_1\}$
Δ	funkcija prijelaza, definirana kao $\Delta: Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma_1\times\Gamma_2 \rightarrow Q\times\Gamma_1^*\times\Gamma_2^*$
z_1,z_2	početni simobli prvog i drugog stoga

Funkcija prijelaza definirana je kao

$\Delta(q,\varepsilon,z_1,z_2) = (q,z_1,Sz_\theta)$ $\quad$ $\Delta(q,\varepsilon,z_1,z_\theta) = (q,\varepsilon,\varepsilon)$

$\Delta(q,\varepsilon,z_1,S) = (q,\varepsilon,\varepsilon)$ ako je $S\rightarrow\varepsilon\in\mathcal{P}$

$\Delta(q,\varepsilon,A,B) = (q,C,D)$ ako je $AB\rightarrow CD\in\mathcal{P}$

$\Delta(q,\varepsilon,A,\varepsilon) = (q,\gamma,\varepsilon)$, $\quad$ $\Delta(q,\varepsilon,\varepsilon,A) = (q,\varepsilon,\gamma)$ ako je $A\rightarrow\gamma\in\mathcal{P}$

$\Delta(q,\varepsilon,A,\varepsilon) = (q,BC,\varepsilon)$, $\quad$ $\Delta(q,\varepsilon,\varepsilon,A) = (q,\varepsilon,BC)$ ako je $A\rightarrow BC\in\mathcal{P}$

$\Delta(q,a,a,\varepsilon) = (q,\varepsilon,\varepsilon)$, $\quad$ $\Delta(q,a,\varepsilon,a) = (q,\varepsilon,\varepsilon)$ ako je $A\rightarrow a\in\mathcal{P}$

$\Delta(q,\varepsilon,\varepsilon,Z) = (q,Z,\varepsilon)$, $\quad$ $\Delta(q,\varepsilon,Z,\varepsilon) = (q,\varepsilon,Z)$ ako je $Z\in\mathcal{N}\cup\mathcal{T}$

• Propozicija 8.1

Neka je $\mathcal{G}=(\mathcal{N},\mathcal{T},\mathcal{P},S)$ kontekstna gramatika u Kurodinoj normalnoj formi i $\mathcal{P}_q$ izvedeni dvostruko-stogovni automat s jednim stanjem tada je $\mathcal{L}(\mathcal{G})=\mathcal{L}(\mathcal{P}_q)$ pa kažemo da su $\mathcal{G}$ i $\mathcal{P}_q$ ekvivalentni.

Dakako, važno je upamtiti što propozicija 8.1 tvrdi. Dalje, valja primijetiti da za automat s praznom stogovnom listom ne trebamo imati nijedno završno stanje, jer je prazna potisna lista uvjet okončanja postupka generiranja rečenice jezika (ili prihvaćanja, ako je automat u ulozi prepoznavača).

♣ Primjer 8.2

Gramatika s produkcijama:

$S \rightarrow AD \mid ED$ $\quad$ $D \rightarrow BC$ $\quad$ $E \rightarrow AF$ $\quad$ $FB \rightarrow BF$ $\quad$ $FC \rightarrow XC$ $\quad$ $X \rightarrow YD$

$AY \rightarrow AE \mid AA$ $\quad$ $BY \rightarrow YB$ $\quad$ $A \rightarrow a$ $\quad$ $B \rightarrow b$ $\quad$ $C \rightarrow c$

jest kontekstna gramatika u KNF-u koja generira jezik L_{abc}. Na primjer, rečenica aabbcc može se dobiti nizom izvođenja:

$$S \Rightarrow ED \Rightarrow AFD \Rightarrow AFBC \Rightarrow ABFC \Rightarrow ABXC \Rightarrow ABYDC \Rightarrow AYBDC \Rightarrow AABDC$$
$$\Rightarrow aABDC \Rightarrow aaBDC \Rightarrow aaBBCC \Rightarrow aabBCC \Rightarrow aabbCC \Rightarrow aabbcC \Rightarrow \underline{aabbcc}$$

Ekvivalentni dvostruko-stogovni automat s jednim stanjem definiran je kao

$$\mathcal{P}q = (q, \{a,b,c,\}, \{a,b,c,A,B,C,D,E,F,S,X,Y,z_0,z_1,z_2\}, \{a,b,c,A,B,C,D,E,F,S,X,Y,z_1,z_2\}, \Delta, z_1, z_2)$$

gdje je funkcija prijelaza definirana kao

$$\Delta(q,\varepsilon,z_1,z_2) = (q,z_1,Sz_0) \qquad \Delta(q,\varepsilon,z_1,z_0) = (q,\varepsilon,\varepsilon)$$
$$\Delta(q,\varepsilon,S,\varepsilon) = \{(q,AD,\varepsilon),(q,ED,\varepsilon)\} \qquad \Delta(q,\varepsilon,\varepsilon,S) = \{(q,\varepsilon,AD),(q,\varepsilon,ED)\}$$
$$\Delta(q,\varepsilon,D,\varepsilon) = (q,BC,\varepsilon) \qquad \Delta(q,\varepsilon,\varepsilon,D) = (q,\varepsilon,BC)$$
$$\Delta(q,\varepsilon,E,\varepsilon) = (q,AF,\varepsilon) \qquad \Delta(q,\varepsilon,\varepsilon,E) = (q,\varepsilon,AF)$$
$$\Delta(q,\varepsilon,X,\varepsilon) = (q,YD,\varepsilon) \qquad \Delta(q,\varepsilon,\varepsilon,X) = (q,\varepsilon,YD)$$
$$\Delta(q,\varepsilon,A,Y) = \{(q,A,E),(q,A,A)\} \qquad \Delta(q,\varepsilon,B,Y) = (q,Y,B)$$
$$\Delta(q,\varepsilon,F,B) = (q,B,F) \qquad \Delta(q,\varepsilon,F,C) = (q,X,C) \qquad \Delta(q,\varepsilon,B,Y) = (q,Y,B)$$
$$\Delta(q,\varepsilon,A,\varepsilon) = (q,a,\varepsilon) \qquad \Delta(q,\varepsilon,B,\varepsilon) = (q,b,\varepsilon) \qquad \Delta(q,\varepsilon,C,\varepsilon) = (q,c,\varepsilon)$$
$$\Delta(q,a,a,\varepsilon) = (q,\varepsilon,\varepsilon) \qquad \Delta(q,a,\varepsilon,a) = (q,\varepsilon,\varepsilon)$$
$$\Delta(q,b,b,\varepsilon) = (q,\varepsilon,\varepsilon) \qquad \Delta(q,b,\varepsilon,b) = (q,\varepsilon,\varepsilon)$$
$$\Delta(q,c,c,\varepsilon) = (q,\varepsilon,\varepsilon) \qquad \Delta(q,c,\varepsilon,c) = (q,\varepsilon,\varepsilon)$$
$$\Delta(q,\varepsilon,\varepsilon,Z) = (q,Z,\varepsilon) \qquad \Delta(q,\varepsilon,Z,\varepsilon) = (q,\varepsilon,Z) \qquad Z \in \{a,b,c,A,B,C,D,E,F,X,Y\}$$

Na primjer, niz aabbcc bit će prihvaćen kao rečenica jezika L_{abc} nizom pomaka:

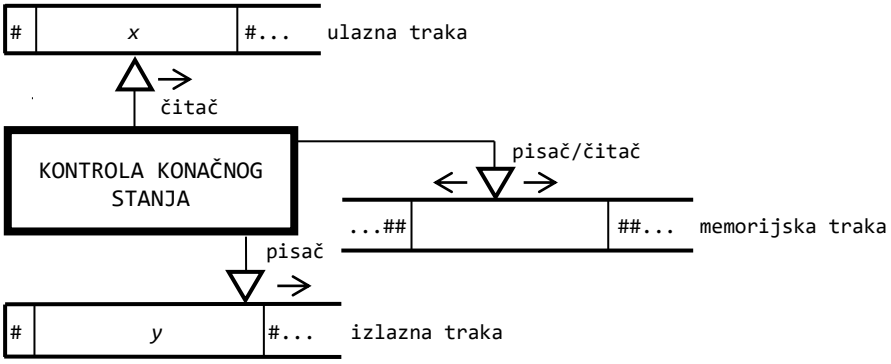
$$(q,aabbcc,z_1,z_2)$$
$$\vdash (q, aabbcc, z_1, Sz_0) \vdash (q, aabbcc, z_1, EDz_0) \vdash (q, aabbcc, z_1, AFDz_0)$$
$$\vdash (q, aabbcc, Az_1, FDz_0) \vdash (q, aabbcc, FAz_1, Dz_0) \vdash (q, aabbcc, FAz_1, BCz_0)$$
$$\vdash (q, aabbcc, BAz_1, FCz_0) \vdash (q, aabbcc, FBAz_1, Cz_0) \vdash (q, aabbcc, XBAz_1, Cz_0)$$
$$\vdash (q, aabbcc, BAz_1, XCz_0) \vdash (q, aabbcc, BAz_1, YDCz_0) \vdash (q, aabbcc, YAz_1, BDCz_0)$$
$$\vdash (q, aabbcc, Az_1, YBDCz_0) \vdash (q, aabbcc, Az_1, ABDCz_0) \vdash (q, aabbcc, az_1, ABDCz_0)$$
$$\vdash (q, abbcc, z_1, ABDCz_0) \vdash (q, abbcc, z_1, aBDCz_0) \vdash (q, bbcc, z_1, BDCz_0)$$
$$\vdash (q, bbcc, z_1, bDCz_0) \vdash (q, bcc, z_1, DCz_0) \vdash (q, bcc, z_1, BCCz_0)$$
$$\vdash (q, bcc, z_1, bCCz_0) \vdash (q, cc, z_1, CCz_0) \vdash (q, cc, z_1, cCz_0)$$
$$\vdash (q, c, z_1, Cz_0) \vdash (q, c, z_1, cz_0) \vdash (q, \varepsilon, z_1, z_0)$$
$$\vdash (q, \varepsilon, \varepsilon, \varepsilon)$$

8.3 TURINGOV STROJ

Koncept Turingovog stroja ("Turing machine") jedan je od najvažnijih matematičkih koncepata razvijen tridesetih godina dvadesetog stoljeća (Alan Turing, 1936.). Važno je napomenuti da su Turing u Engleskoj i istovremeno, ali ne u tako očitom obliku, E. Post u SAD, formulirali pojam apstraktnog računskog stroja i postavili osnovne načelne karakteristike nekoliko godina prije pojave prvog prototipa računala s automatskim upravljanjem.

Turingov stroj nije stvarno sagrađen. Danas se koristi kao osnovni model za razrješavanje biti pojma "izračunljivosti računalnih algoritama", "teorije složenosti" itd, a za nas je od posebnog značenja njegova primjena u teoriji i praksi formalnih jezika.

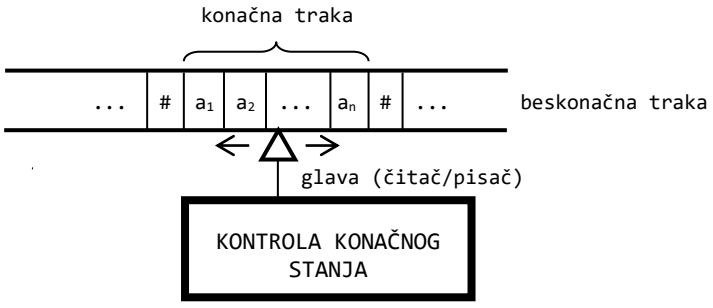
Postoji više inačica Turingovog stroja, od onih koji imaju samo jednu traku, pa do onih s neograničenim brojem traka. U našim primjenama općenita shema Turingovog stroja prikazana je na sl. 8.2.



Sl. 8.2 – *Turingov stroj*.

Turingov prepoznavač

Prema hijerarhijskoj strukturi jezika i automata Turingov stroj jest automat koji generira i prihvaća jezike tipa 0 ili jezike bez ograničenja. Turingov stroj u svojstvu generatora zvali smo Turingov generator, a u svojstvu prepoznavača jezika bez ograničenja – Turingov prepoznavač, sl. 8.3.



Sl. 8.3 – *Turingov prepoznavač*.

Dakle, Turingov prepoznavač ima samo jednu traku koja je istodobno ulazna, izlazna i memorijska. Kontrola konačnog stanja premješta glavu (čitač/pisač) od stanja do stanja, lijevo ili desno u svakom koraku, te čita i upisuje znakove u ćelije trake. Inicijalno je beskonačna traka prazna (sadrži beskonačno ćelija praznih znakova ili blankova). Glava se nalazi na proizvoljnoj poziciji. Beskonačna će traka na kraju u jednom svom konačnom dijelu sadržavati znakove alfabeta (rečenicu) jezika kojeg Turingov prepoznavač prihvaća. Poslije ovog neformalnog opisa Turingovog generatora evo i njegove formalne definicije.

♦ Turingov prepoznavač

Turingov prepoznavač jest uređena šestorka, $\mathcal{T}_R=(Q, \Sigma, \Gamma, \delta, q_0, F)$, gdje su:

- Q konačan skup stanja (kontrole konačnog stanja)
- Σ ulazni alfabet
- Γ konačni skup alfabeta znakova ulazno-izlazne trake, koji sadrži ulazni alfabet, posebne znakove koji nisu u ulaznom alfabetu (npr. velika slova) i posebni simbol # nazvan blank (razmak)
- δ funkcija prijelaza, definirana kao

  $$\delta:\ Q\times\Sigma\cup\{\#\} \rightarrow Q\times\{\Gamma\setminus\{\#\}\}\times\{-1,0,1\}$$

  gdje je -1 pomak glave ulijevo za jedno mjesto, 0 mirovanje glave i 1 pomak glave udesno za jedno mjesto
- q_0 početno stanje, $q_0\in Q$
- F skup završnih stanja, $F\subseteq Q$

♦ Konfiguracija i pomak Turingova prepoznavača

Konfiguracija Turingova prepoznavača $\mathcal{T}_R=(Q, \Sigma, \Gamma, \delta, q_0, F)$ jest $(\alpha,q,x\beta)$ iz $\Gamma^*\times Q\times\Gamma^*$, gdje su:

- α niz znakova konačne trake lijevo od pozicije glave, $\alpha\in\Gamma^*$
- q tekuće stanje, $q\in Q$
- x tekući znak (na poziciji glave), $x\in\Gamma$
- β niz znakova desno od pozicije glave, $\beta\in\Gamma^*$

Početna konfiguracija je (ε,q_0,w), a završna konfiguracija (w,q,ε), $q\in F$, $w\in\Sigma^*$.
Pomak Turingova prepoznavača $\mathcal{T}_R=(Q, \Sigma, \Gamma, \delta, q_0, F)$ jest relacija $\vdash_{\mathcal{T}_R}$ (ili samo $\vdash$ ako se $\mathcal{T}_R$ podrazumijeva) definirana kao što slijedi:

$(\alpha, q_1, a\beta) \vdash (\alpha, q_2, b\beta)$ ako je $\delta(q_1, a) = (q_2, b, 0)$
$(\alpha, q_1, a\beta) \vdash (\alpha b, q_2, \beta)$ ako je $\delta(q_1, a) = (q_2, b, 1)$
$(\alpha, q_1, \varepsilon) \vdash (\alpha, q_2, b)$ ako je $\delta(q_1, \#) = (q_2, b, 0)$
$(\alpha, q_1, \varepsilon) \vdash (\alpha b, q_2, \varepsilon)$ ako je $\delta(q_1, \#) = (q_2, b, 1)$
$(\alpha a, q_1, b\beta) \vdash (\alpha, q_2, acb)$ ako je $\delta(q_1, b) = (q_2, c, -1)$
$(\varepsilon, q_1, a\beta) \vdash (\varepsilon, q_2, \#b\beta)$ ako je $\delta(q_1, a) = (q_2, b, -1)$
$(\alpha a, q_1, \varepsilon) \vdash (\alpha, q_2, ab)$
$(\varepsilon, q_1, \varepsilon) \vdash (\varepsilon, q_2, \#b)$
} ako je $\delta(q_1, \#) = (q_2, b, -1)$

gdje su $q_1,q_2\in Q$, $a,b\in\Gamma$. Jezik definiran Turingovim prepoznavačem $\mathcal{T}_R$ definiran je kao

$$\mathcal{L}(\mathcal{T}_R) = \{w:\ (\varepsilon, q_0, w) \vdash^*\ (w, q, \varepsilon),\ w\in\Sigma^*,\ q\in F\}$$

Funkcija (tablica) prijelaza Turingova prepoznavača jezika L razlikuje se od funkcije prijelaza Turingova generatora tog istog jezika. U primjerima koji slijede pokazat ćemo kako se jednostavno mogu definirati Turingovi prepoznavači, najprije poznatog nam kontekstnog jezika L_{abc}, potom triju jezika bez ograničenja.

♣ Primjer 8.3

Jezik L_{abc} može biti definiran Turingovim prepoznavačem (**Labc.TR**) u kojem je tablica prijelaza dana sa:

	#	A	B	C	a	b	c
→0					(1,A, 1)		
1			(1,B, 1)		(1,a, 1)	(2,B, 1)	
2				(2,C, 1)		(2,b, 1)	(3,C,-1)
3	(6,ε, 1)	(3,A,-1)	(3,B,-1)	(3,C,-1)	(4,a,-1)	(3,b,-1)	
4		(5,A, 1)			(4,a,-1)		
5					(1,A, 1)		
6	(7,ε, 0)	(6,a, 1)	(6,b, 1)	(6,c, 1)			
⊗7							

Ideja je sljedeća:

1) Ulazni niz mora početi s a. Znak a se pretvori u A, prelazi se u stanje 1 i glava se pomiče udesno:

   $(\varepsilon,0,aw)\vdash(A,1,w)$

2) Pretražuje se ostatak niza sve do pozicije znaka b koji se pretvara u B, prelazi se u stanje 2 i nastavlja pretraživanje sve do prve pozicije znaka c:

   $(A,1,w) \vdash^* (A\alpha,1,b\beta) \vdash (A\alpha B,2,\beta) \vdash^* (A\alpha B\beta_1,2,c\beta_2)$

3) Prelaskom u stanje 2, pa u 3 i ponovo u 2: $\vdash(A^{n-1},2,A)\vdash(A^{n-1}a,3,\varepsilon)\vdash(A,2,aB)\vdash^*(\varepsilon,2,\#a^nB^n)$ na kraju je dosegnuta konfiguracija $(\varepsilon,2,\#a^nB^n)$, odnosno, niz A^n "pretvoren" je u a^n i dodan niz B^n.

4) Prelaskom u stanje 4 i konfiguraciju $(\varepsilon,4,a^nB^n)$ poslije n pomaka i konfiguracije $(a^n,4,B^n)$ prelazi se u konfiguraciju

   a) $(a^nb,5,B^{n-1}) \vdash^{n-1} (a^nbB^{n-1},5,\varepsilon) \vdash(a^nbB^{n-2},6,Bc) \vdash (a^nbB^{n-2},5,bc) \vdash^* (\varepsilon,6,\#a^nb^nc^n) \vdash (\varepsilon,7,\underline{a^nb^nc^n})$, ako je n>1

   b) $(ab,5,\varepsilon) \vdash (a,6,bc) \vdash^* (\varepsilon,6,\#abc) \vdash (\varepsilon,7,\underline{abc})$, ako je n=1

Na primjer, niz aabbcc (program **TR.py**) bit će prihvaćen nizom pomaka:

```
('', 0, 'aabbcc')
   |-- ('A', 1, 'abbcc')    |-- ('Aa', 1, 'bbcc')    |-- ('AaB', 2, 'bcc')
   |-- ('AaBb', 2, 'cc')    |-- ('AaB', 3, 'bCc')    |-- ('Aa', 3, 'BbCc')
   |-- ('A', 3, 'aBbCc')    |-- ('', 4, 'AaBbCc')    |-- ('A', 5, 'aBbCc')
   |-- ('AA', 1, 'BbCc')    |-- ('AAB', 1, 'bCc')    |-- ('AABB', 2, 'Cc')
   |-- ('AABBC', 2, 'c')    |-- ('AABB', 3, 'CC')    |-- ('AAB', 3, 'BCC')
   |-- ('AA', 3, 'BBCC')    |-- ('A', 3, 'ABBCC')    |-- ('', 3, 'AABBCC')
   |-- ('', 3, '#AABBCC')   |-- ('', 6, 'AABBCC')    |-- ('a', 6, 'ABBCC')
   |-- ('aa', 6, 'BBCC')    |-- ('aab', 6, 'BCC')    |-- ('aabb', 6, 'CC')
   |-- ('aabbc', 6, 'C')    |-- ('aabbcc', 6, '#')   |-- ('aabbcc', 7, '')
   |-- accept
```

Dok niz aabbc neće biti prihvaćen:

```
('', 0, 'aabbc')
   |-- ('A', 1, 'abbc')     |-- ('Aa', 1, 'bbc')     |-- ('AaB', 2, 'bc')
   |-- ('AaBb', 2, 'c')     |-- ('AaB', 3, 'bC')     |-- ('Aa', 3, 'BbC')
   |-- ('A', 3, 'aBbC')     |-- ('', 4, 'AaBbC')     |-- ('A', 5, 'aBbC')
   |-- ('AA', 1, 'BbC')     |-- ('AAB', 1, 'bC')     |-- ('AABB', 2, 'C')
   |-- ('AABBC', 2, '#')    |-- error
```

Prije ovoga primjera rekli smo kako se "jednostavno" mogu definirati Turingovi prepoznavači jezika bez ograničenja. Mali ispravak: "gotovo jednostavno" jer baš kad

pomislimo kako smo jednostavno izveli tablicu prijelaza, kao u primjeru 8.3, testirajući uz pomoć programa **TR.py** neke ulazne nizove, kao na primjer aBbCc:

```
('', 0, 'aBbCc')
  |-- ('A', 1, 'BbCc')   |-- ('AB', 1, 'bCc')   |-- ('ABB', 2, 'Cc')
  |-- ('ABBC', 2, 'c')   |-- ('ABB', 3, 'CC')   |-- ('AB', 3, 'BCC')
  |-- ('A', 3, 'BBCC')   |-- ('', 3, 'ABBCC')   |-- ('', 3, '#ABBCC')
  |-- ('', 6, 'ABBCC')   |-- ('a', 6, 'BBCC')   |-- ('ab', 6, 'BCC')
  |-- ('abb', 6, 'CC')   |-- ('abbc', 6, 'C')   |-- ('abbcc', 6, '#')
  |-- ('abbcc', 7, '')   |-- accept
```

vidimo da je ulazni niz aBbCc prihvaćen, iako ne može biti rečenica danog jezika!

Problem se može riješiti tako da se dodaju još tri stanja (**Labc1.TR**) i odgovarajuće funkcije prijelaza koje će "učitati" niz znakova i provjeriti jesu li svi znakovi iz alfabeta:

	#	A	B	C	a	b	c
→0					(8,a, 1)		
1			(1,B, 1)		(1,a, 1)	(2,B, 1)	
2				(2,C, 1)		(2,b, 1)	(3,C,-1)
3	(6,ε, 1)	(3,A,-1)	(3,B,-1)	(3,C,-1)	(4,a,-1)	(3,b,-1)	
4		(5,A, 1)			(4,a,-1)		
5					(1,A, 1)		
6	(7,ε, 0)	(6,a, 1)	(6,b, 1)	(6,c, 1)			
⊗7							
8	(9,ε,-1)				(8,a, 1)	(8,b, 1)	(8,c, 1)
9	(10,ε, 1)				(9,a,-1)	(9,b,-1)	(9,c,-1)
10					(1,A, 1)		

Ili, proširenjem i modificiranjem prve tablice (**Labc2.TR**):

	#	A	B	C	a	b	c
→0					(8,A, 1)		
1			(1,B, 1)		(1,a, 1)	(2,B, 1)	
2				(2,C, 1)		(2,b, 1)	(3,C,-1)
3	(6,ε, 1)	(3,A,-1)	(3,B,-1)	(3,C,-1)	(4,a,-1)	(3,b,-1)	
4		(5,A, 1)			(4,a,-1)		
5					(1,A, 1)		
6	(7,ε, 0)	(6,a, 1)	(6,b, 1)	(6,c, 1)			
⊗7							
8					(8,a, 1)	(9,B, 1)	
9						(9,b, 1)	(3,C,-1)

♣ Primjer 8.4

Jezik {*ww*: *w*∈{a,b,c}⁺} može biti prepoznat Turingovim prepoznavačem (**ww.TR**):

$$\mathcal{T}r = (Q, \Sigma, \Gamma, \delta, q_0, F)$$

gdje su:

Q = {0,1,2,...,17,18}
Σ = {a,b,c}
Γ = {#} ∪ Σ ∪ {A,B,C,$}

$q_0 = 0$
$\mathcal{F} = \{18\}$

i tablica prijelaza δ dana je sa:

	#	a	b	c	A	B	C	$
→0		(1,A, 1)	(1,B, 1)	(1,C, 1)				
1	(2,$,-1)	(1,a, 1)	(1,b, 1)	(1,c, 1)				(2,$,-1)
2		(3,$, 1)	(4,$, 1)	(5,$, 1)				
3								(6,a,-1)
4								(6,b,-1)
5								(6,c,-1)
6	(8,ε, 1)	(7,a,-1)	(7,b,-1)	(7,c,-1)	(6,A,-1)	(6,B,-1)	(6,B,-1)	(6,$,-1)
7		(7,a,-1)	(7,b,-1)	(7,c,-1)	(0,A, 1)	(0,B, 1)	(0,B, 1)	
8		(8,a, 1)	(8,b, 1)	(8,c, 1)	(9,a, 1)	(10,b, 1)	(11,c, 1)	
9		(9,a, 1)	(9,b, 1)	(9,c, 1)	(9,A, 1)	(9,B, 1)	(9,C, 1)	(12,$, 1)
10		(10,a, 1)	(10,b, 1)	(10,c, 1)	(10,A, 1)	(10,B, 1)	(10,C, 1)	(16,$, 1)
11		(11,a, 1)	(11,b, 1)	(11,c, 1)	(11,A, 1)	(11,B, 1)	(11,C, 1)	(17,$, 1)
12		(12,$,-1)						(13,a,-1)
13	(15,ε, 1)	(13,a,-1)	(13,b,-1)	(13,c,-1)	(14,A,-1)	(14,B,-1)	(14,C,-1)	
14		(8,a, 1)	(8,b, 1)	(8,c, 1)	(14,A,-1)	(14,B,-1)	(14,C,-1)	
15		(15,a, 1)	(15,b, 1)	(15,c, 1)				(18,#, 0)
16			(16,$,-1)					(13,b,-1)
17				(17,$,-1)				(13,c,-1)
⊗18								

Ako je $a_1a_2...a_n$ ulazni niz, prvo se načini niz pomaka

$$(\varepsilon,0,a_1a_2...a_n) \vdash^* (\varepsilon,8,A_1...A_i\$a_{i+1}...a_n)$$

gdje je prvih i znakova (malih slova) a_i zamijenjeno s jednakim takvim velikim slovima, A_i. Time je ulazni niz podijeljen na dva dijela. Znak $ je graničnik. Primijetimo da stanje 8 neće biti dosegnuto, tj. bit će dojavljena pogreška, ako ulazni niz sadrži znak koji nije dio alfabeta jezika ili ako je duljina ulaznog niza neparna (jer je $|ww|=|w|+|w|=2|w|$). Ovo potonje dogodit će se pri pomicanju graničnika $ od kraja ulaznog niza ako znak ispred $ nije u Σ.

Dakle, ako je dosegnuta konfiguracija $(\varepsilon,8,A_1...A_i\$a_{i+1}...a_n)$ zamijenit će se prvi znak, A_1, ulazno-izlazne trake sa svojim originalom, a_1, i provjeriti je li jednak prvom znaku poslije $, a_{i+1}. Ako jest, graničnik se pomiče za jedno mjesto udesno, vraća se na početak ulaznog niza i postupak ponavlja za A_2, itd. Ako u bilo kojem koraku a_{i+j} nije jednak a_j, dojavljuje se pogreška i prekida daljnje pretraživanje. Ako je posljednji znak a_n bio jednak a_i, prelazi se u stanje 15, pomiče glava na kraj ulaznog niza, uz prethodno izbacivanje graničnika $, i dojavljuje da je ulazni niz prihvaćen.

Pogledajmo, na primjer, kako će biti prepoznata rečenica abcabc:

```
('', 0, 'abcabc')
   |- ('A', 1, 'bcabc')     |- ('Ab', 1, 'cabc')     |- ('Abc', 1, 'abc')
   |- ('Abca', 1, 'bc')     |- ('Abcab', 1, 'c')     |- ('Abcabc', 1, '#')
   |- ('Abcab', 2, 'c$')    |- ('Abcab$', 5, '$')    |- ('Abca', 6, 'b$c')
   |- ('Abc', 7, 'ab$c')    |- ('Ab', 7, 'cab$c')    |- ('A', 7, 'bcab$c')
   |- ('', 7, 'Abcab$c')    |- ('A', 0, 'bcab$c')    |- ('AB', 1, 'cab$c')
   |- ('ABc', 1, 'ab$c')    |- ('ABca', 1, 'b$c')    |- ('ABcab', 1, '$c')
   |- ('ABca', 2, 'b$c')    |- ('ABca$', 4, '$c')    |- ('ABc', 6, 'a$bc')
   |- ('AB', 7, 'ca$bc')    |- ('A', 7, 'Bca$bc')    |- ('AB', 0, 'ca$bc')
   |- ('ABC', 1, 'a$bc')    |- ('ABCa', 1, '$bc')    |- ('ABC', 2, 'a$bc')
   |- ('ABC$', 3, '$bc')    |- ('AB', 6, 'C$abc')    |- ('A', 6, 'BC$abc')
   |- ('', 6, 'ABC$abc')    |- ('', 6, '#ABC$abc')   |- ('', 8, 'ABC$abc')
   |- ('a', 9, 'BC$abc')    |- ('aB', 9, 'C$abc')    |- ('aBC', 9, '$abc')
```

```
   |- ('aBC$', 12, 'abc')  |- ('aBC', 12, '$$bc')  |- ('aB', 13, 'Ca$bc')
   |- ('a', 14, 'BCa$bc')  |- ('', 14, 'aBCa$bc')  |- ('a', 8, 'BCa$bc')
   |- ('ab', 10, 'Ca$bc')  |- ('abC', 10, 'a$bc')  |- ('abCa', 10, '$bc')
   |- ('abCa$', 18, 'bc')  |- ('abCa', 18, '$$c')  |- ('abC', 13, 'ab$c')
   |- ('ab', 13, 'Cab$c')  |- ('a', 14, 'bCab$c')  |- ('ab', 8, 'Cab$c')
   |- ('abc', 11, 'ab$c')  |- ('abca', 11, 'b$c')  |- ('abcab', 11, '$c')
   |- ('abcab$', 19, 'c')  |- ('abcab', 19, '$$')  |- ('abca', 13, 'bc$')
   |- ('abc', 13, 'abc$')  |- ('ab', 13, 'cabc$')  |- ('a', 13, 'bcabc$')
   |- ('', 13, 'abcabc$')  |- ('', 13, '#abcabc$') |- ('', 15, 'abcabc$')
   |- ('a', 15, 'bcabc$')  |- ('ab', 15, 'cabc$')  |- ('abc', 15, 'abc$')
   |- ('abca', 15, 'bc$')  |- ('abcab', 15, 'c$')  |- ('abcabc', 15, '$')
   |- ('abcabc$', 16, '#') |- ('abcabc', 17, '$')  |- ('abcabc', 20, '')
   |- accept
```

Dok, na primjer, ulazni niz abac neće biti prihvaćen kao rečenica jezika:

```
('', 0, 'abac')
   |- ('A', 1, 'bac')      |- ('Ab', 1, 'ac')      |- ('Aba', 1, 'c')
   |- ('Abac', 1, '#')     |- ('Aba', 2, 'c$')     |- ('Aba$', 5, '$')
   |- ('Ab', 6, 'a$c')     |- ('A', 7, 'ba$c')     |- ('', 7, 'Aba$c')
   |- ('A', 0, 'ba$c')     |- ('AB', 1, 'a$c')     |- ('ABa', 1, '$c')
   |- ('AB', 2, 'a$c')     |- ('AB$', 3, '$c')     |- ('A', 6, 'B$ac')
   |- ('', 6, 'AB$ac')     |- ('', 6, '#AB$ac')    |- ('', 8, 'AB$ac')
   |- ('a', 9, 'B$ac')     |- ('aB', 9, '$ac')     |- ('aB$', 12, 'ac')
   |- ('aB', 12, '$$c')    |- ('a', 13, 'Ba$c')    |- ('', 14, 'aBa$c')
   |- ('a', 8, 'Ba$c')     |- ('ab', 10, 'a$c')    |- ('aba', 10, '$c')
   |- ('aba$', 18, 'c')    |- error
```

♣ Primjer 8.5

Jezik $\{a^{2^n}: n>0\}$ može biti prepoznat Turingovim prepoznavačem (**a2n.TR**) u kojem je tablica prijelaza dana sa:

	#	a	A	$
→0		(1,A, 1)		
1		(2,a, 1)		
2	(11,ε,-1)	(3,a, 1)		
3	(4,$,-1)	(3,a, 1)		(4,$,-1)
4		(4,$, 1)		(5,a,-1)
5		(6,a,-1)	(8,A, 1)	(5,$,-1)
6		(6,a,-1)	(7,A, 1)	
7		(3,A, 1)		
8	(9,ε,-1)	(8,a, 1)		(8,a, 1)
9		(10,#,-1)		
10		(10,a,-1)	(0,A, 1)	
11	(12,ε, 1)	(11,a,-1)	(11,a,-1)	
12	(13,ε, 0)	(12,a, 1)		
⊗13				

Ako je n=1 ulazni niz $w=a^2$ bit će prihvaćen kao rečenica jezika:

```
('', 0, 'aa')
   |- ('A', 1, 'a')    |- ('Aa', 2, '#')  |- ('A', 11, 'a')  |- ('', 11, 'Aa')
   |- ('', 11, '#aa')  |- ('', 12, 'aa')  |- ('a', 12, 'a')  |- ('aa', 12, '#')
   |- ('aa', 13, '')   |- accept
```

Za n>1 ulazni niz $w=a^{2^n}$ možemo napisati kao

$$w = a^{2^n} = (a^{2^{n-1}})^2 = ((a^{2^{n-2}})^2)^2 = \ldots = (\ldots((a^2)^2)^2\ldots)^2$$

što znači da će poslije n-1 polovljenja ulaznog niza na kraju ostati niz a^2 koji jest u jeziku. To je bila temeljna ideja pri definiranju danog prepoznavača.

Prvo se ulazni niz prepolovi prevodeći prvu polovicu znakova a u A koristeći graničnik $. Ako polovljenje nije moguće, prekida se analiza i dojavljuje pogreška. Ako jest, izbacuje se graničnik i postupak ponavlja (od početnog stanja 0) na desnoj polovici niza itd. Na primjer, za ulazni niz aaaa imamo:

```
('', 0, 'aaaa')
   |- ('A', 1, 'aaa')    |- ('Aa', 2, 'aa')    |- ('Aaa', 3, 'a')
   |- ('Aaaa', 3, '#')   |- ('Aaa', 4, 'a$')   |- ('Aaa$', 4, '$')
   |- ('Aaa', 5, '$a')   |- ('Aa', 5, 'a$a')   |- ('A', 6, 'aa$a')
   |- ('', 6, 'Aaa$a')   |- ('A', 7, 'aa$a')   |- ('AA', 3, 'a$a')
   |- ('AAa', 3, '$a')   |- ('AA', 4, 'a$a')   |- ('AA$', 4, '$a')
   |- ('AA', 5, '$aa')   |- ('A', 5, 'A$aa')   |- ('AA', 8, '$aa')
   |- ('AAa', 8, 'aa')   |- ('AAaa', 8, 'a')   |- ('AAaaa', 8, '#')
   |- ('AAaa', 9, 'a')   |- ('AAa', 10, 'a')   |- ('AA', 10, 'aa')
   |- ('A', 10, 'Aaa')   |- ('AA', 0, 'aa')
```

Niz je prepolovljen (sadržaj ulazno-izlazne trake je AAaa) i prelazi se u početno stanje s ulaznim (pod)nizom aa, a to je prihvatljivi niz jezika. Preostaje da se znakovi A na ulazno-izlaznoj traci preslikaju u svoje originale i ulazno-izlazna glava premjesti na kraj ulaznog niza:

```
   |- ('AAA', 1, 'a')
   |- ('AAAa', 2, '#')   |- ('AAA', 11, 'a')   |- ('AA', 11, 'Aa')
   |- ('A', 11, 'Aaa')   |- ('', 11, 'Aaaa')   |- ('', 11, '#aaaa')
   |- ('', 12, 'aaaa')   |- ('a', 12, 'aaa')   |- ('aa', 12, 'aa')
   |- ('aaa', 12, 'a')   |- ('aaaa', 12, '#')  |- ('aaaa', 13, '')
   |- accept
```

Evo i primjera ulaznog niza a^5 koji ne može biti prihvaćen kao rečenica jezika:

```
('', 0, 'aaaaa')
   |- ('A', 1, 'aaaa')   |- ('Aa', 2, 'aaa')   |- ('Aaa', 3, 'aa')
   |- ('Aaaa', 3, 'a')   |- ('Aaaaa', 3, '#')  |- ('Aaaa', 4, 'a$')
   |- ('Aaaa$', 4, '$')  |- ('Aaaa', 5, '$a')  |- ('Aaa', 5, 'a$a')
   |- ('Aa', 6, 'aa$a')  |- ('A', 6, 'aaa$a')  |- ('', 6, 'Aaaa$a')
   |- ('A', 7, 'aaa$a')  |- ('AA', 3, 'aa$a')  |- ('AAa', 3, 'a$a')
   |- ('AAaa', 3, '$a')  |- ('AAa', 4, 'a$a')  |- ('AAa$', 4, '$a')
   |- ('AAa', 5, '$aa')  |- ('AA', 5, 'a$aa')  |- ('A', 6, 'Aa$aa')
   |- ('AA', 7, 'a$aa')  |- ('AAA', 3, '$aa')  |- ('AA', 4, 'A$aa')
   |- error
```

♣ Primjer 8.6

Jezik $\{a^{n^2}: n>0\}$ može biti prepoznat Turingovim prepoznavačem (**an2.TR**) u kojem je tablica prijelaza dana sa:

	#	a	A	B
→0		(1,a, 1)		
1	(13,ε, 0)	(2,B, 1)		
2		(3,A, 1)	(2,A, 1)	
3		(4,A, 1)		
4	(10,ε,-1)	(5,B,-1)		
5			(6,a, 1)	
6		(6,a, 1)		(7,B, 1)
7		(8,A,-1)	(7,A, 1)	

	#	a	A	B
8			(8,A,-1)	(9,B,-1)
9		(9,a,-1)	(6,a, 1)	(12,a, 1)
10			(10,a,-1)	(11,a, 1)
11	(13,ε, 0)	(11,a, 1)		
12		(12,a, 1)		(2,B, 1)
⊗13				

Kao što smo pokazali u prvoj knjizi [15], ako se promatraju dvije rečenice danog jezika, a^{n^2} i $a^{(n-1)^2}$, razlika njihovih duljina jednaka je

$$|a^{n^2}| - |a^{(n-1)^2}| = n^2 - (n-1)^2 = (n-n+1)(n+n-1) = 2n-1$$

pa vrijedi

$$a^{n^2} = a^{(n-1)^2}a^{2n-1}$$

Za n=1 imamo $a^1=a^0a^1=a$, a za n=2 je $a^4=a^1a^3$ itd. Općenito će rečenica a^{n^2} biti dobivena dopisujući nizove a^{2i-1} od i=1 do n, tj $a^{n^2}=a^1a^3a^5...a^{2n-1}$. Svaki od tih podnizova za dva je dulji od prethodnog, $a^3=a^1a^2$, $a^5=a^3a^2$, ..., $a^{2n-1} = a^{2n-3}a^2$. To je temeljna ideja za definiranje Turingovog prepoznavača opisanoga u nastavku:

1) Ako je n=1, w=a i bit će prihvaćen kao rečenica jezika:

```
('', 0, 'a')  |- ('a', 1, '#')  |- ('a', 13, '')  |- accept
```

2) Za n=2 ulazni niz je $w=a^4$ i imamo sljedeći niz pomaka:

```
('', 0, 'aaaa')
  |- ('a', 1, 'aaa')   |- ('aB', 2, 'aa')   |- ('aBA', 3, 'a')
  |- ('aBAA', 4, '#')
```

u kojem je poslije prihvaćanja prvog znaka prihvaćeno još $a^3=a^1a^2$ znakova. Niz je u jeziku. Prelazi se u stanje 10 i slijedi prijepis pomoćnih znakova A i B s a i prihvat:

```
|- ('aBA', 10, 'A')  |- ('aB', 10, 'Aa')  |- ('a', 10, 'Baa')
|- ('aa', 11, 'aa')  |- ('aaa', 11, 'a')  |- ('aaaa', 11, '#')
|- ('aaaa', 13, '')  |- accept
```

3) Za n>2 najprije bi iz stanja 4 bio niz pomaka

```
|- ('aBAA', 4, 'aaaaa...')  |- ('aBA', 5, 'ABaaaa...')
|- ('aBAa', 6, 'Baaaa...')  |- ('aBAaB', 7, 'aaaa...')
|- ('aBAa', 8, 'BAaaa...')  |- ('aBA', 9, 'aBAaaa...')
|- ('aB', 9, 'AaBAaaa...')  |- ('aBa', 6, 'aBAaaa...')
|- ('aBaa', 6, 'BAaaa...')  |- ('aBaaB', 7, 'Aaaa...')
|- ('aBaaBA', 7, 'aaa...')  |- ('aBaaB', 8, 'AAaa...')
|- ('aBaa', 8, 'BAAaa...')  |- ('aBa', 9, 'aBAAaa...')
|- ('aB', 9, 'aaBAAaa...')  |- ('a', 9, 'BaaBAAaa...')
|- ('aa', 12, 'aaBAAaa...') |- ('aaa', 12, 'aBAAaa...')
|- ('aaaa', 12, 'BAAaa...')
```

kojim je prihvaćeno a^3 znakova, potom treba biti prihvaćeno još a^2 znakova:

```
|- ('aaaaB', 2, 'AAaa...')  |- ('aaaaBA', 2, 'Aaa...')
|- ('aaaaBAA', 2, 'aa...')  |- ('aaaaBAAA', 3, 'a...')
|- ('aaaaBAAAA', 4, '...')
```

Prihvaćen je podniz a^5. Ako je sljedeći znak #, ulazni niz je u jeziku, prelazi se u stanje 10 i slijedi prijepis pomoćnih znakova A i B s a i prihvat. Ako je sljedeći znak a, prelazi se u stanje 5 itd.

Na primjer, ulazni niz a^9 bit će prihvaćen sljedećim nizom pomaka:

```
('', 0, 'aaaaaaaaa')
  |- ('a', 1, 'aaaaaaaa')   |- ('aB', 2, 'aaaaaaa')   |- ('aBA', 3, 'aaaaaa')
  |- ('aBAA', 4, 'aaaaa')   |- ('aBA', 5, 'ABaaaa')   |- ('aBAa', 6, 'Baaaa')


  |- ('aBAaB', 7, 'aaaa')   |- ('aBAa', 8, 'BAaaa')   |- ('aBA', 9, 'aBAaaa')
  |- ('aB', 9, 'AaBAaaa')   |- ('aBa', 6, 'aBAaaa')   |- ('aBaa', 6, 'BAaaa')
  |- ('aBaaB', 7, 'Aaaa')   |- ('aBaaBA', 7, 'aaa')   |- ('aBaaB', 8, 'AAaa')
  |- ('aBaa', 8, 'BAAaa')   |- ('aBa', 9, 'aBAAaa')   |- ('aB', 9, 'aaBAAaa')
  |- ('a', 9, 'BaaBAAaa')   |- ('aa', 12, 'aaBAAaa')  |- ('aaa', 12, 'aBAAaa')
  |- ('aaaa', 12, 'BAAaa')  |- ('aaaaB', 2, 'AAaa')   |- ('aaaaBA', 2, 'Aaa')
  |- ('aaaaBAA', 2, 'aa')   |- ('aaaaBAAA', 3, 'a')   |- ('aaaaBAAAA', 4, '#')
  |- ('aaaaBAAA', 10, 'A')  |- ('aaaaBAA', 10, 'Aa')  |- ('aaaaBA', 10, 'Aaa')
  |- ('aaaaB', 10, 'Aaaa')  |- ('aaaa', 10, 'Baaaa')  |- ('aaaaa', 11, 'aaaa')
  |- ('aaaaaa', 11, 'aaa')  |- ('aaaaaaa', 11, 'aa')  |- ('aaaaaaaa', 11, 'a')
  |- ('aaaaaaaaa', 11, '#') |- ('aaaaaaaaa', 13, '')  |- accept
```

Dok ulazni niz aaa ne može biti prihvaćen kao rečenica danog jezika:

```
('', 0, 'aaaaa')
  |- ('a', 1, 'aaaa')  |- ('aB', 2, 'aaa')  |- ('aBA', 3, 'aa')
  |- ('aBAA', 4, 'a')  |- ('aBA', 5, 'AB')  |- ('aBAa', 6, 'B')
  |- ('aBAaB', 7, '#') |- error
```

P R O G R A M I

U ovom dijelu dajemo programe dvostruko-stogovnog i Turingovog prepoznavača. Oba prepoznavača učitavaju tablice prijelaza, D, iz tekstualnih datoteka. Tablice su definirane kao liste n-torki u Pythonu (uz pretpostavku da su napisane bez pogreške).

DVOSTRUKO-STOGOVNI PREPOZNAVAČ

Sljedeći je program ustroj dvostruko-stogovnog prepoznavača. Poslije učitavanja definicije prepoznavača, sedmorke:

```
Pt = (Q, A, _1, _2, D, s, F)
```

gdje su:

- Q konačan skup stanja (kontrole konačnog stanja)
- A ulazni alfabet
- _1, _2 alfabet prvog i drugog stoga
- D funkcija prijelaza
- *s* početno stanje
- F skup konačnih stanja

i poslije učitavanja ulaznog niza, prepoznavač `Pt` će prihvatiti ulazni niz, ako je u jeziku, ili će ga odbaciti, ako nije u jeziku.

🖫 **DSP.py** *Dvostruko-stogovni prepoznavač*

```
# -*- coding: cp1250 -*-
# DVOSTRUKO-STOGOVNI PREPOZNAVAČ KONTEKSTNIH JEZIKA

import os
from   fun import *
from   easygui import *

NL = '\n'

def Ucitaj_W (): # Učitavanje ulaznog niza
  return komp (raw_input ('Upiši ulazni niz: '))

def Ispisi_Pt (Ime):
  print Ime
  print NL, 'Pt = (Q, A, _1, _2, D, s, F)', NL
  print 'Q  =', Q, ' A =', A,  ' _1 =', _1, ' _2 =', _2, ' s =', s, ' F =', F
  print NL, 'D:'
  for d in D:
    print ' ', d[0], '=', d[1]
  print

def Ucitaj_Pt (): # Ucitavanje tablice prijelaza
  Ok = True
  Ime = fileopenbox ('UČITAVANJE DVOSTRUKO-STOGOVNOG PREPOZNAVAČA (Pt)', None,
'*.Pt', '*')
  if os.path.exists(Ime):
    for line in open (Ime, 'r') :
      exec (line)
  else:
    print 'Ne postoji Pt s danim imenom!'
    Ok = False

  return Ime, Ok, (Q, A, _1, _2, D, s, F)

def Ispisi_C (y, C):
  print y, C

def Pt (x):
  global Q, A, _1, _2, D, s, F

  Ok = True;  End = False;  q = s;  alfa = beta = ''
  C = (q, x, alfa, beta);
  Ispisi_C ('', C)

  while len(x)>0 and Ok and not End:
    if len(x) > 0:
      X = x[0];  a = '';  b = ''
      if len(alfa) > 0: a = alfa[0]
      if len(beta) > 0: b = beta[0]
      C0 = (q, X, a, b);  Ok = False
```

```
      for d in D:
        d0, d1 = d;  q0, e, g1, g2 = d0
        if q == q0 and e == X and g1 in a and g2 in b:
          q, g1, g2 = d1
          x = x[1:]
          if g1 == '' and a != '': alfa = alfa[1:]
          if g1 != ''            : alfa = g1 +alfa
          if g2 == '' and b != '': beta = beta[1:]
          if g2 != ''            : beta = g2 +beta
          Ok = True
          break
      if Ok:
        C = (q, x, alfa, beta);
        Ispisi_C ('   |--', C)
        if q in F and alfa == '' and beta == '': End = True
        if End and x != '': Ok = False
  Ok = Ok and End
  return Ok


Ime, Ok, DSP = Ucitaj_Pt ();  Q, A, _1, _2, D, s, F = DSP;  Ispisi_Pt (Ime)

w = Ucitaj_W(); print
while len(w) > 0:
  Ok = Pt (w)
  if Ok: Ispisi_C ('   |--', 'accept')
  else : Ispisi_C ('   |--', 'error')
  print;  w = Ucitaj_W();  print
```

Pogledajmo kako program radi u prepoznavanju rečenica jezika Labc. Prvo definirajmo prepoznavač (tekstualna datoteka Labc.Pt) jezika Labc:

```
Q  = [0, 1, 2]
A  = ['a', 'b', 'c']
_1 = ['a']
_2 = ['b']
s  = 0
F  = [2]
D  = [ ( (0,'a','',''), (0,'a','') ), ( (0,'b','a',''), (1,'','b') ),
       ( (1,'b','a',''), (1,'','b') ),  ( (1,'c','','b'), (2,'','') ),
       ( (2,'c','','b'), (2,'','') ) ]

C:\Python27\FJP\Labc.Pt
Pt = (Q, A, _1, _2, D, s, F)

Q  = [0, 1, 2]  A = ['a', 'b', 'c']  _1 = ['a']  _2 = ['b']  s = 0  F = [2]

D:
  (0, 'a', '', '') = (0, 'a', '')
  (0, 'b', 'a', '') = (1, '', 'b')
  (1, 'b', 'a', '') = (1, '', 'b')
  (1, 'c', '', 'b') = (2, '', '')
  (2, 'c', '', 'b') = (2, '', '')

Upiši ulazni niz: abc
```

```
(0, 'abc', '', '')
  |-- (0, 'bc', 'a', '')
  |-- (1, 'c', '', 'b')
  |-- (2, '', '', '')
  |-- accept
```

TURINGOV PREPOZNAVAČ

Prije nego što damo program Turingova prepoznavača opišimo kako ćemo definirati njegove komponente.

Definiranje prepoznavača

Turingov prepoznavač definirat ćemo u tekstualnoj datoteci koja će predstavljati inicijalizaciju komponenti prepoznavača – varijabli u Pythonu. Imena su sljedeća:

- `Q` - lista stanja
- `A` - alfabet (lista znakova)
- `T` - alfabet stoga (`['#'] + A +` lista "neterminala")
- `q0` – početno stanje
- `F` - lista konačnih stanja
- `D` - funcija prijelaza definirana kao n-torka para n-torki:

```
( ( (q, a), (q', t,  k) ), ( (…), (…) ) ... )
```

Slijede primjeri definicija Turingova prepoznavača nekoliko kontekstnih jezika.

Jezik $\{a^n b^n c^n : n>0\}$

Labc.TR

```
Q  = range (8)
A  = ['a', 'b', 'c']
T  = ['#'] +A +['A', 'B', 'C']
q0 = 0
F  = [7]
D  = ( ( (0, 'a'), (1, 'A',  1) ),

       ( (1, 'B'), (1, 'B',  1) ),  ( (1, 'a'), (1, 'a',  1) ),
       ( (1, 'b'), (2, 'B',  1) ),

       ( (2, 'C'), (2, 'C',  1) ),  ( (2, 'b'), (2, 'b',  1) ),
       ( (2, 'c'), (3, 'C', -1) ),

       ( (3, '#'), (6,  '',  1) ),  ( (3, 'A'), (3, 'A', -1) ),
       ( (3, 'B'), (3, 'B', -1) ),  ( (3, 'C'), (3, 'C', -1) ),
       ( (3, 'a'), (4, 'a', -1) ),  ( (3, 'b'), (3, 'b', -1) ),

       ( (4, 'A'), (5, 'A',  1) ),  ( (4, 'a'), (4, 'a', -1) ),

       ( (5, 'a'), (1, 'A',  1) ),

       ( (6, '#'), (7,  '',  0) ),  ( (6, 'A'), (6, 'a',  1) ),
       ( (6, 'B'), (6, 'b',  1) ),  ( (6, 'C'), (6, 'c',  1) )  )
```

Jezik {*ww*: *w*∈{a,b,c}+}

ww.TR

```
n  = 18
Q  = range (n+1)
A  = ['a', 'b', 'c']
T  = ['#'] +A +['A', 'B', 'C'] +['$']
q0 = 0
F  = [18]
D  = ( ( (0, 'a'), (1, 'A',  1) ),  ( (0, 'b'), (1, 'B',  1) ),
       ( (0, 'c'), (1, 'C',  1) ),

       ( (1, '#'), (2, '$', -1) ),  ( (1, 'a'), (1, 'a',  1) ),
       ( (1, 'b'), (1, 'b',  1) ),  ( (1, 'c'), (1, 'c',  1) ),
       ( (1, '$'), (2, '$', -1) ),

       ( (2, 'a'), (3, '$',  1) ),  ( (2, 'b'), (4, '$',  1) ),
       ( (2, 'c'), (5, '$',  1) ),

       ( (3, '$'), (6, 'a', -2) ),

       ( (4, '$'), (6, 'b', -2) ),

       ( (5, '$'), (6, 'c', -2) ),

       ( (6, '#'), (8,  '',  1) ),  ( (6, 'a'), (7, 'a', -1) ),
       ( (6, 'b'), (7, 'b', -1) ),  ( (6, 'c'), (7, 'c', -1) ),
       ( (6, 'A'), (6, 'A', -1) ),  ( (6, 'B'), (6, 'B', -1) ),
       ( (6, 'C'), (6, 'C', -1) ),

       ( (7, 'a'), (7, 'a', -1) ),  ( (7, 'b'), (7, 'b', -1) ),
       ( (7, 'c'), (7, 'c', -1) ),  ( (7, 'A'), (0, 'A',  1) ),
       ( (7, 'B'), (0, 'B',  1) ),  ( (7, 'C'), (0, 'C',  1) ),

       ( (8, 'a'), (8, 'a',  1) ),  ( (8, 'b'), (8, 'b',  1) ),
       ( (8, 'c'), (8, 'c',  1) ),  ( (8, 'A'), (9, 'a',  1) ),
       ( (8, 'B'), (10,'b',  1) ),  ( (8, 'C'), (11,'c',  1) ),

       ( (9, 'a'), (9, 'a',  1) ),  ( (9, 'b'), (9, 'b',  1) ),
       ( (9, 'c'), (9, 'c',  1) ),  ( (9, 'A'), (9, 'A',  1) ),
       ( (9, 'B'), (9, 'B',  1) ),  ( (9, 'C'), (9, 'C',  1) ),
       ( (9, '$'), (12,'$',  1) ),

       ( (10,'a'), (10,'a',  1) ),  ( (10,'b'), (10,'b',  1) ),
       ( (10,'c'), (10,'c',  1) ),  ( (10,'A'), (10,'A',  1) ),
       ( (10,'B'), (10,'B',  1) ),  ( (10,'C'), (10,'C',  1) ),
       ( (10,'$'), (16,'$',  1) ),

       ( (11,'a'), (11,'a',  1) ),  ( (11,'b'), (11,'b',  1) ),
       ( (11,'c'), (11,'c',  1) ),  ( (11,'A'), (11,'A',  1) ),
       ( (11,'B'), (11,'B',  1) ),  ( (11,'C'), (11,'C',  1) ),
       ( (11,'$'), (17,'$',  1) ),

       ( (12,'a'), (12,'$', -1) ),  ( (12,'$'), (13,'a', -1) ),

       ( (13,'#'), (15, '',  1) ),  ( (13,'a'), (13,'a', -1) ),
       ( (13,'b'), (13,'b', -1) ),  ( (13,'c'), (13,'c', -1) ),
```

```
        ( (13,'A'), (14,'A', -1) ),  ( (13,'B'), (14,'B', -1) ),
        ( (13,'C'), (14,'C', -1) ),

        ( (14,'a'), ( 8,'a',  1) ),  ( (14,'b'), ( 8,'b',  1) ),
        ( (14,'c'), ( 8,'c',  1) ),  ( (14,'A'), (14,'A', -1) ),
        ( (14,'B'), (14,'B', -1) ),  ( (14,'C'), (14,'C', -1) ),

        ( (15,'a'), (15,'a',  1) ),  ( (15,'b'), (15,'b',  1) ),
        ( (15,'c'), (15,'c',  1) ),  ( (15,'$'), (18, '',  0) ),

        ( (16,'b'), (16,'$', -1) ),  ( (16,'$'), (13,'b', -1) ),

        ( (17,'c'), (17,'$', -1) ),  ( (17,'$'), (13,'c', -1) )  )
```

Jezik $\{a^{2^n}: n>0\}$

a2n.TR

```
n  = 13
Q  = range (n+1)
A  = ['a']
T  = ['#'] +A +['A'] +['$']
q0 = 0
F  = [13]
D  = ( ( (0, 'a'), (1, 'A',  1) ),  ( (1, 'a'), (2, 'a',  1) ),

       ( (2, '#'), (11, '', -1) ),  ( (2, 'a'), (3, 'a',  1) ),

       ( (3, '#'), (4, '$', -1) ),  ( (3, 'a'), (3, 'a',  1) ),
       ( (3, '$'), (4, '$', -1) ),

       ( (4, 'a'), (4, '$',  1) ),  ( (4, '$'), (5, 'a', -1) ),

       ( (5, 'a'), (6, 'a', -1) ),  ( (5, 'A'), (8, 'A',  1) ),
       ( (5, '$'), (5, '$', -1) ),

       ( (6, 'a'), (6, 'a', -1) ),  ( (6, 'A'), (7, 'A',  1) ),

       ( (7, 'a'), (3, 'A',  1) ),

       ( (8, '#'), (9,  '', -1) ),  ( (8, 'a'), (8, 'a',  1) ),
       ( (8, '$'), (8, 'a',  1) ),

       ( (9, 'a'), (10,'#', -1) ),

       ( (10,'a'), (10,'a', -1) ),  ( (10,'A'), (0, 'A',  1) ),

       ( (11,'#'), (12, '',  1) ),  ( (11,'a'), (11,'a', -1) ),
       ( (11,'A'), (11,'a', -1) ),

       ( (12,'#'), (13, '',  0) ),  ( (12,'a'), (12,'a',  1) )  )
```

Jezik $\{a^{n^2}: n>0\}$

an2.TR

```
n  = 13
Q  = range (n+1)
A  = ['a']
T  = ['#'] +A +['A', 'B']
q0 = 0
F  = [n]
D  = ( ( (0, 'a'), (1, 'a',  1) ),

       ( (1, '#'), (13, '',  0) ),  ( (1, 'a'), (2, 'B',  1) ),

       ( (2, 'a'), (3, 'A',  1) ),  ( (2, 'A'), (2, 'A',  1) ),

       ( (3, 'a'), (4, 'A',  1) ),

       ( (4, '#'), (10, '', -1) ),  ( (4, 'a'), (5, 'B', -1) ),

       ( (5, 'A'), (6, 'a',  1) ),

       ( (6, 'a'), (6, 'a',  1) ),  ( (6, 'B'), (7, 'B',  1) ),

       ( (7, 'a'), (8, 'A', -1) ),  ( (7, 'A'), (7, 'A',  1) ),

       ( (8, 'A'), (8, 'A', -1) ),  ( (8, 'B'), (9, 'B', -1) ),

       ( (9, 'a'), (9, 'a', -1) ),  ( (9, 'A'), (6, 'a',  1) ),
       ( (9, 'B'), (12,'a',  1) ),

       ( (10,'A'), (10,'a', -1) ),  ( (10,'B'), (11,'a',  1) ),

       ( (11,'#'), (13, '',  0) ),  ( (11,'a'), (11,'a',  1) ),

       ( (12,'a'), (12,'a',  1) ),  ( (12,'B'), ( 2,'B',  1) )  )
```

Evo i prepoznavača. Najprije se učitavaju komponente prepoznavača zadanog jezika i ulazni niz. Prepoznavač, `Tr`, će poslije konačnog broja promjena svojih konfiguracija (pomaka) prihvatiti ulazni niz, ako je u jeziku, ili dojaviti pogrešku, ako nije.

TR.py *Turingov prepoznavač*

```
# -*- coding: cp1250 -*-
# TURINGOV PREPOZNAVAČ

import os
from   fun import *
from   easygui import *

NL = '\n'

def Ucitaj_W (): # Učitavanje ulaznog niza
  return komp (raw_input ('Upiši ulazni niz: '))
```

```
def Ispisi_Tr (Ime):
  print Ime
  print NL, 'Tr = (Q, A, T, D, q0, F)', NL
  print 'Q  =', Q, NL, 'A  =', A, NL, 'T  =', T, NL, 'q0 =', q0,  NL,'F  =', F
  print NL, 'D:'
  for d in D:
    print ' ', d[0], '=', d[1]
  print

def Ucitaj_Tr (): # Ucitavanje tablice prijelaza
  Ok = True; x = ''; Tp = False
  Ime = fileopenbox ('TURINGOV PREPOZNAVAČ (Tg)', None, '*.TR', '*')
  if os.path.exists(Ime):
    for line in open (Ime, 'r') :
      if len (line) > 0 and line[0] == 'D': Tp = True
      if Tp: x += line[:-1]
      if not Tp: exec (line)
    exec (x)
  else:
    print 'Ne postoji Pt s danim imenom!'
    Ok = False

  return Ime, Ok, (Q, A, T, D, q0, F)

def Ispisi_C (y, C):
  print y, C

def Tr (w):
  global Q, A, T, q0, F, D

  Ok = True; End = False
  t = '#' +w +'#';  i = 1
  X = t[1];  alfa = '';  q = q0;  beta = t[i+1:-1]
  C = (alfa, q, X+beta)
  Ispisi_C ('', C)

  while Ok and not End:
    C0 = (q, X)
    Ok = False
    for d in D:
      d0, d1 = d
      if C0 == d0:
        q, x, k = d1
        if x != ''     : t = t[:i] +x +t[i+1:]
        if t[-1] <> '#': t += '#'
        i += k
        X    = t[i]
        alfa = t[:i];  beta = t[i+1:]
        beta = beta.replace ('#', '');  alfa = alfa.replace ('#', '')
        Ok = True
        break
    if Ok:
      End = q in F
      if End: X =''
      C = (alfa, q, X +beta);  Ispisi_C ('   |-', C)

  return Ok
```

```
Ime, Ok, TP = Ucitaj_Tr ()
Q, A, T, D, q0, F = TP
Ispisi_Tr (Ime)

w = Ucitaj_W();  print
while len(w) > 0:
  Ok = Tr (w)
  if Ok: Ispisi_C ('   |-', 'accept')
  else : Ispisi_C ('   |-', 'error')
  print
  w = Ucitaj_W(); print
```

Pitanja i zadaci

1) Definirajte Turingov prepoznavač jezika $\{ww: w \in \{a,b,c\}^+\}$ *koji neće sadržavati graničnik* `$` *u alfabetu ulazno-izlazne trake.*

2) Definirajte Turingov prepoznavač jezika $\{a^{2^n}: n>0\}$ *koji neće sadržavati graničnik* `$` *u alfabetu ulazno-izlazne trake.*

3) Definirajte Turingov prepoznavač jezika $\{w^n: w \in \{a,b\}^+, n>1\}$.

4) Definirajte Turingov prepoznavač jezika $\{a^n$: *n je prim broj*$\}$

5) Definirajte Turingov prepoznavač jezika $\{0^{F_n}: n \geq 0\}=\{0,00,000,00000,00000000,\ldots\}$ *gdje je* F_n *n-ti Fibonaccijev broj, definiran kao* $F_0=0$, $F_1=1$, $F_n=F_{n-2}+F_{n-1}$, $n \geq 2$.

6) Napišite program dvostruko-stogovnog prepoznavača s jednim stanjem.

9.
PREPOZNAVAČ JEZIKA SA SVOJSTVIMA

slobodo moja
dugo sam te čuvao
kao rijetki biser
slobodo moja
ti si mi pomogla
da odvežem konopce
i krenem bilo kamo
i krenem do kraja
putova sudbine
da uberem sanjajući
ružu vjetrova
na zraci mjesečevoj

slobodo moja
pred tvojim htijenjima
moja duša je pokorena
slobodo moja
sve sam ti dao
svoju posljednju košulju
i koliko sam patio
da mogu zadovoljiti
sve tvoje zahtjeve
mijenjao sam zemlje
gubio prijatelje
da zadobijem tvoje povjerenje

slobodo moja
znala si me odvići
i od najmanjih navika
slobodo moja
ti zbog koje sam zavolio
čak i samoću
ti zbog koje sam se smiješio
dok sam gledao kako završava
jedna lijepa avantura
ti koja si me štitila
dok sam se skrivao
da njegujem svoje rane

slobodo moja
ipak sam te napustio
jedne decembarske noći
napustio sam
puteve lutanja
koje smo zajedno slijedili
i nisam se čuvao
šaka i nogu sputanih
ostavio sam sve
i izdao sam te
zbog jednog zatvora ljubavi
i njegove lijepe tamničarke

slobodo moja
ma liberté

(georges moustaki/
nepoznati student, 1973)

U ovom je završnom poglavlju opisan jedan jednoprolazni postupak sintaksne analize – prepoznavač jezika sa svojstvima – primjenljiv na svim beskontekstnim i kontekstnim jezicima, te posebno pogodan za implementaciju na računalima. Postupak prepoznavanja utemeljen je na dvije tablice: tablici prijelaza i tablici akcija.

9.1 JEZICI SA SVOJSTVIMA

Jezici za programiranje nisu beskontekstni. Na primjer, ako napišemo naredbu za pridruživanje vrijednosti varijabli A u jeziku Pascal:

```
A := B + C
```

zadovoljena je osnovna sintaksna struktura pisanja te naredbe, ali moraju biti zadovoljena i sljedeća svojstva pridružena imenima A, B i C:

1) A je varijabla; B i C mogu biti varijable ili konstante
2) A može biti cjelobrojnog, realnog, nizovnog ili skupovnog tipa
3) Ako je A cjelobrojnog ili skupovnog tipa, istog tipa moraju biti i B i C
4) Ako je A realnog tipa, B i C mogu biti cjelobrojnog ili realnog tipa
5) Ako je A nizovnog tipa, B i C mogu biti znakovnog ili nizovnog tipa

Znak "+" je operacija koja ima značenje ovisno o tipu varijable. Predstavlja operaciju zbrajanja, ako je A realnog ili cjelobrojnog tipa, nastavljanje nizova ako je A nizovnog tipa ili uniju skupova ako je A skupovnog tipa.

Navedena svojstva nije moguće definirati gramatikom niti kojeg tipa. Ovdje ćemo pokazati kako se takav problem može nadići uvođenjem jezika sa svojstvima do kojih dolazimo proširenjem značenja kontrole konačnog stanja konačnog automata, Dovedan [10].

♦ Jezik sa svojstvima

Jezik sa svojstvima jest jezik čiji je generator zadan kao uređena osmorka:

$$\mathcal{J} = (Q, \Sigma, \mathcal{V}, \delta, q_0, \mathcal{F}, \alpha, \mathcal{M})$$

gdje su:

- Q konačan *skup stanja* (kontrole završnog stanja)
- Σ *alfabet*
- $\mathcal{V}$ *rječnik*, $\mathcal{V} \subseteq \Sigma^+$
- δ *funkcija prijelaza*, definirana kao $\delta: Q \times \mathcal{V} \cup \{@\} \rightarrow P(Q)$ gdje je $P(Q)$ particija od Q; @ je oznaka kraja ulaznog niza
- q_0 *početno stanje*, $q_0 \in Q$
- $\mathcal{F}$ *skup završnih stanja*, $\mathcal{F} \subseteq Q$
- α *skup akcija* pridruženih svakom paru (q_i, s_j), $q_i \in Q$, $s_j \in \mathcal{V}$, za koji je definirana funkcija prijelaza
- $\mathcal{M}$ *pomoćna memorija*

Dakle, kontrola završnog stanja generatora jezika sa svojstvima podijeljena je na dva dijela: funkciju prijelaza i skup akcija. Funkcijom prijelaza δ (primijetimo da je definirana nad rječnikom) zadaju se sve moguće promjene stanja. Akcija je, općenito, postupak koji, ovisno o tekućem stanju q_i i informacijama dobivenih od pomoćne memorije, reducira broj mogućih prijelaza iz stanja q_i, zadanih funkcijom δ, u trenutačno ostvarive prijelaze. I ne samo to, akcija može promijeniti vrijednosti funkcije prijelaza. Ako je prijelaz iz stanja q_i u stanje q_j prijelazom s_k uvijek ostvariv, smatrat ćemo da je takvom prijelazu pridružena prazna akcija. Generator regularnih jezika primjer je u kojem su svi prijelazi uvijek ostvarivi. Stoga generator regularnih jezika nema potrebe za pomoćnom memorijom.

U ustrojbi prepoznavača jezika sa svojstvima kontrolu konačnog stanja i dalje ćemo prikazivati dijagramom (tablicom) prijelaza, ali ćemo svakom prijelazu dodati kôd njemu pridružene akcije sa značenjem:

0 (ili izostavljeno)	- prazna akcija
1, 2, ...	- neprazna akcija

Iz definicije jezika sa svojstvima slijedi da će takvi jezici u pravilu imati neka kontekstna svojstva, pa se postavlja pitanje: Može li beskontekstna gramatika generirati jezik sa svojstvima? Isto pitanje vrijedi i za regularne gramatike. Odgovor je potvrdan! To će biti u onim slučajevima kad beskontekstna gramatika u svojim produkcijama sadrži ekplicitno napisane "prave" rekurzije, odnosno, ako se za gramatiku bilo kojeg tipa može napisati niz izvođenja:

$X \overset{+}{\Rightarrow} \alpha X \beta \qquad \alpha, \beta \in (\mathcal{N} \cup \mathcal{T})^*$

Tada pomoćna memorija prepoznavača takvog jezika ima strukturu stoga.

♣ Primjer 9.1

Neka je dana beskontekstna gramatika $\mathcal{G}$ s produkcijama:

$E \rightarrow E+E \mid E*E \mid (E) \mid a$

koja generira jezik jednostavnih aritmetičkih izraza koji moraju zadovoljavati kontekstno svojstvo: broj otvorenih zagrada jednak je broju zatvorenih zagrada. Ekvivalentni jezik sa svojstvima jest:

$\mathcal{J}(\mathcal{G}) = (Q, \Sigma, \mathcal{V} \cup \{@\}, \delta, q_0, \mathcal{F}, \alpha, \mathcal{M})$

gdje su:

$Q = \{1,2\}$
$\Sigma = \mathcal{V} = \{(,),+,*,a\}$
$q_0 = 1$
$\mathcal{F} = \{2\}$

Tablica prijelaza je:

	(	)	+	*	a	@
1	1				2	
2		2	1	1		1

gdje @ označuje kraj ulaznog niza. Tablica akcija je:

	(	)	+	*	a	@
1	1	0	0	0	0	0
2	0	2	0	0	0	3

Pomoćna memorija je B, brojač prijelaza sa "(". Akcije su:

```
1: B += 1
2: if B > 0: B -= 1
   else    : print 'Error!';  break
3: if B > 0: print 'Error!'
```

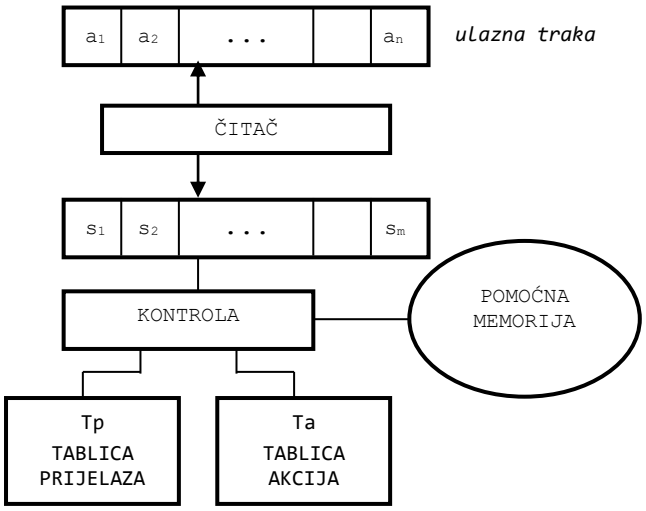
9.2 PREPOZNAVAČ JEZIKA SA SVOJSTVIMA

S obzirom na to da je jezik sa svojstvima definiran (generiran) posebnom vrstom automata u kojem je tablici prijelaza dodana tablica akcija, odnosno, svakom je prijelazu pridružena akcija (prazna ili neprazna), ako govorimo o problemu sintaksne analize jezika sa svojstvima, zapravo govorimo o prepoznavaču tih jezika.

Kod generatora jezika sa svojstvima akcijom je za svako stanje dijagrama (tablice) prijelaza bio određen skup ostvarivih prijelaza, što je u svakom trenutku bilo uvjetovano informacijama dobivenim iz pomoćne memorije.

U slučaju prepoznavača jezika sa svojstvima postavlja se pitanje: "Postoji li za simbol $s_j \in \mathcal{V}$ ostvariv prijelaz iz tekućeg stanja q_i u neko stanje q_k?". Prvi dio odgovora na ovo pitanje dat će nam funkcija (tablica) prijelaza. Ako je $q_k=\delta(q_i,s_j)$ definirano, prijelaz je moguć, a akcija pridružena paru (q_i,s_j) odredit će je li ostvariv.

Općenito se struktura prepoznavača jezika sa svojstvima može prikazati kao na sljedećem crtežu:



Sl. 9.1 - *Model prepoznavača jezika sa svojstvima.*

Pomoćna memorija

Zamislit ćemo da pomoćnu memoriju čine primitivne i strukturirane varijable svih tipova, kao što je to, na primjer, u Turbo Pascalu ili Pythonu. Inicijalno će te varijable sadržavati određene vrijednosti.

Čitač

Čitač je jednostavni pretvarač (program leksičke analize) koji prihvaća niz ulaznih znakova, kojima je na kraju dodan znak ¨@¨, i prevodi ih u niz simbola. Najčešće će rječnik biti jednak alfabetu, pa će skup simbola biti jednak skupu znakova alfabeta. Svakom simbolu pridužen je jedinstveni cjelobrojni kôd, od `1` do `k`, ako rječnik sadrži `k` simbola, te `k+1` za znak ¨@¨.

Sintaksna analiza

Ako je `C` kôd učitanog simbola, `S` tekuće stanje kontrole konačnog stanja i `Tp` tablica prijelaza, mogući prijelaz `Ss` zadan je s:

```
Ss := Tp [S][C]
```

a akcija kodom na mjestu `Ta[S][C]`, gdje je `Ta` tablica akcija.

Tada se kontrola završnog stanja prepoznavača jezika sa svojstvima (postupak sintaksne analize) može prikazati sljedećim dijelom programa:

```
S = 1;  Kraj = False;  Pogreska = False
while not Kraj and not Pogreska:
  Ucitaj_Sim; Ss = Tp [S-1][C]
  if C == 0 or Ss == 0:
    print '* Sintaksna pogreška'; Pogreska = True
  else:
    a = Ta [S-1][C]
    if   a == 0 :
      '* prazna akcija *'
      S = Ss
    elif a == 1:
      '* Akcija broj 1 *'
      _1 ()
    ...

    elif a == n:
      '* Akcija broj n *'
      _n ()
```

Najprije će u posebnoj proceduri biti inicijalizirane varijable programa, tablica prijelaza i tablica akcija. Procedura `Ucitaj_Sim` učitat će tekući simbol i vratiti njegov kôd, `C`. Ako je kôd različit od `0` i ako je definirano naredno stanje, izvršit će se odgovarajuća akcija. Akcija može biti prazna, tada se bezuvjetno prelazi u naredno stanje. Neprazna se akcija izvodi u posebnoj proceduri.

Postupak se sintaksne analize nastavlja sve do dosezanja kraja ulaznog niza i njegova prihvaćanja ili se prekida ako je učitan simbol koji nije u rječniku ili je pronađena sintaksna pogreška.

♣ Primjer 9.2

Prepoznavač jednostavnih aritmetičkih izraza može biti realiziran programom u Pythonu:

```
# -*- coding: cp1250 -*-
# PREPOZNAVAČ JEZIKA JEDNOSTAVNIH ARITMETIČKIH IZRAZA

from gramatika import *

def Init ():
  global Tp, Ta, Sym, S, B, Kraj, Pogreska, i

        #  (   )   +   *   a   @
  Tp = ( (  0, -1, -1, -1,  1, -1 ),    # 0
        ( -1,  1,  0,  0, -1,  0 ) )   # 1
  Ta = ( (  1,  0,  0,  0,  0,  0 ),
        (  0,  2,  0,  0,  0,  3 ) )

  Sym  ='()+*a@';  S = 0;  B = 0;  Kraj = False;  Pogreska = False;  i = 0

def Ucitaj_P (): # Unos niza
  R = komp (raw_input ('UČITAJ NIZ: '))
  R = R +"@"
  return R
def Leks_An (R, i): # Leksička analiza
  Z = R[i]; K = -1
  if Z in Sym : K = Sym.index (Z)
  i = i +1
  return K, i
def Sint_An (q, C): # Sintaksna analiza
  global Tp, Ta, Sym, S, B, Kraj, Pogreska, i
  Ss = Tp [q][C]; Er = Ss == -1
  if Er :
    print ('*** sint. pogreška ***')
    return True, Ss
  else :
    A = Ta[q][C]
    if   A == 0 :
      '* prazna akcija *'
    elif A == 1: B += 1
    elif A == 2:
      if B > 0 : B -= 1
      else     : return True, Ss
    elif A == 3:
      if B <> 0: return True, Ss
      else     : Kraj = True
    S = Ss
    return False, Ss
q = 0
w = Ucitaj_P ()
while w[0] != '@':
  Init ()
  while not Kraj and not Pogreska:
    C, i        = Leks_An (w, i)
    Pogreska, q = Sint_An (q, C)
  print 'Niz: ' +w[:-1],
  if not Pogreska : print 'jest u jeziku'
  else            : print 'NIJE u jeziku!'
  w = Ucitaj_P ()
```

U ovom je primjeru rječnik jednak alfabetu, pa se ulazni niz odmah učita do kraja i doda mu se simbol ¨@¨. Kodovi simbola rječnika određeni su inicijalizacijom varijable Sym, tj. kôd im je pozicija u tom nizu.

Dinamičke tablice prijelaza i akcija

Funkcija prijelaza bilo kojeg automata jest statična, nepromjenjiva. S obzirom na to da je akcija prema našoj definiciji općenito naredba ili niz naredaba u jeziku implementacije propoznavača jezika sa svojstvima, moguće je definirati takve akcije koje će mijenjati inicijalnu definiciju tablice prijelaza i/ili tablice akcija.

♣ Primjer 9.3

Tablica prijelaza iz primjera 9.2 može biti definirana kao

```
        #  (   )   +   *   a   @
Tp = [ [  0, -1, -1, -1,  1, -1 ],    # 0
       [ -1, -1,  0,  0, -1,  0 ] ]   # 1
```

gdje je inicijalno `Tp[1,1] = -1` (a bilo je jednako 1). Nailaskom na zagradu u ulaznom nizu akcija 1 je sada:

```
B += 1;  Tp[1][1] = 1
```

što znači da je poslije pojave otvorene zaagrade dopušteno pisanje zatvorene zagrade. Akcija 2 je sada

```
B -= 1
if B == 0: Ss = int(Tp [q][C]);  Tp[1][1] = -1
```

Dakle, dinamičkom tablicom prijelaza osigurali smo da će prijelaz iz stanja 1 sa zatvorenom zagradom biti definiran samo onda ako je prethodno postojala otvorena zagrada s kojom može biti sparena. Ako su otvorene zagrade "potrošene" jednakim brojem zatvorenih zagrada, ponovno je nedefiniran prijelaz `Tp[1][1]`.

P R I M J E N E

Primjene jezika sa svojstvima i njihovih prepoznavača posebno će doći do izražaja kad jezik sa svojstvima bude "pravi". Takvi su, prije svega, jezici za programiranje koje opisujemo u sljedećoj knjizi. Ovdje ćemo dati dva jednostavna primjera koji u dovoljnoj mjeri najavljuju mogućnosti primjene prepoznavača jezika sa svojstvima.

🖫 Meta_BNF.py

Kao što je pokazano u [15], primjeru 3.9, formalizam pisanja produkcija u BNF-u također je jezik. Ovdje dajemo realizaciju njegove sintaksne analize primjenom prepoznavača jezika sa svojstvima. Pravila pisanja (unosa) produkcija su sljedeća:

- neterminali su velika slova engleskog alfabeta
- terminali su mala slova engleskog alfabeta, brojke i ostali znakovi
- umjesto simbola → pišu se dva znaka -> bez razmaka
- alternative su odvojene znakom | (AltGr W)
- prazna produkcija je znak #

Na početku se unosi ime gramatike, prema pravilima pisanja imena datoteke u Windowsima. Na kraju se upiše @ kao oznaka kraja unosa. Evo jednoga primjera unosa gramatike:

```
E -> T + E | T
T -> T * F | F
F -> (E) | a
```

Slijedi program **Meta_BNF.py** koji koristi modul **gramatika.py**.

```
# PROGRAM Meta-BNF

from gramatika import *

     #  N   T   ->  |   #   @
Tp = (( 1, -1, -1, -1, -1,  0),    # Tablica prijelaza
      (-1, -1,  2, -1, -1, -1),
      ( 3,  3,  0, -1,  4, -1),
      ( 3,  3,  0,  2, -1,  0),
      (-1, -1,  0,  2, -1,  0) )

Ta = (( 1,  0,  0,  0,  0,  4),    # Tablica akcija
      ( 0,  0,  0,  0,  0,  0),
      ( 3,  2,  0,  0,  0,  0),
      ( 3,  2,  0,  0,  0,  5),
      ( 0,  0,  0,  0,  0,  5) )

def Ucitaj_P (): # Unos izraza
    R = raw_input ()
    R = R +"@"
    return R
def Leks_An (R, i): # Leksička analiza
    MetaSym = ['|', '#', '@'];
    while R[i] == ' ' : i = i+1
    Z = R[i]; K = -1
    if Z in MetaSym : K = MetaSym.index (Z)
    i = i +1
    if K != -1                                  : return K+3, i
    if ord(Z) in range (ord('A'), ord('Z')+1) : return 0, i
    if Z == '-' and R[i] == '>' : i = i+1;      return 2, i
    return 1, i
def Sint_An (q, NT, C): # Sintaksna analiza
    Ss = Tp [q][C]; Er = Ss == -1
    if Er :
      print ('*** sint. pogreška ***')
      return True, Ss
    else :
      NT = w[i-1]; A = Ta[q][C]

    return False, Ss

Grm, Ok, G = Ucitaj_G ('Meta-BNF', '*.grm')
if Ok : Ispisi_G (Grm, G)
N, T, P, S = G
print NL, NL, 'Upišite produkcije: '
```

```
q = 0;  x = Ucitaj_P ();  w = komp(x)
while w[0] != '@':
  i = 0;
  while i < len(w):
    C, i  = Leks_An (w, i)
    Er, q = Sint_An (q, w[i-1], C)
    if Er: break
  if not Er :
    w = w[0:-1]
    L = [w[0]]; y = ''
    for i in range (3, len(w)):
      if w[i] != '|' : y += w[i]
      else           : L.append (y); y = ''
    L.append (y)
    i = 0; k = 0;
    while i < len (P) and P[i][0][0] != L[0] : i += 1
    if i < len (P) : P[i] = L
    else           : P.append (L)
  w = Ucitaj_P ()
G = (N, T, P, S)
Ispisi_G (Grm, G)
Upamti_G (Grm, P)
```

Jezik L_{abc}

Prema definiciji jezika sa svojstvima poznati nam kontekstni jezik koji smo nazvali L_{abc}, a prisjetimo se da je to $\{a^n b^n c^n : n>0\}$, jezik je sa svojstvima. Napišimo njegov prepoznavač:

```
# PREPOZNAVAČ JEZIKA Labc

from gramatika import *

     #   a   b   c   @
Tp = ((  1, -1, -1, -1),    # Tablica prijelaza
      (  1,  2, -1, -1),
      ( -1,  2,  3, -1),
      ( -1, -1,  3,  0) )

Ta = ((  1,  0,  0,  0),    # Tablica akcija
      (  1,  2,  0,  0),
      (  0,  3,  4,  0),
      (  0,  0,  5,  6) )

def Ucitaj_P (): # Unos niza
  R = raw_input ('UČITAJ NIZ: ')
  R = R +"@"
  return R

def Leks_An (R, i): # Leksička analiza
  Sym = ['a', 'b', 'c', '@'];
  while R[i] == ' ' : i = i+1
  Z = R[i]; K = -1
  if Z in Sym : K = Sym.index (Z)
  i = i +1
  return K, i
```

```
def Sint_An (q, C): # Sintaksna analiza
  global a, b, c
  Ss = Tp [q][C]; Er = Ss == -1
  if Er :
    print ('*** sint. pogreška ***')
    return True, Ss
  else :
    A = Ta[q][C]
    if   A == 0 :
      '* prazna akcija *'
    elif A == 1: a += 1
    elif A == 2: b  = a -1
    elif A == 3:
      if b > 0: b -= 1
      else    : return True, Ss
    elif A == 4: c  = a -1
    elif A == 5:
      if c > 0: c -= 1
      else    : return True, Ss
    else:
      if b > 0 or c > 0: return True, Ss
  S = Ss
  return False, Ss

q = 0
x = Ucitaj_P ()
w = komp(x)
while w[0] != '@':
  i = 0; a = b = c = 0
  while i < len(w):
    C, i  = Leks_An (w, i)
    Er, q = Sint_An (q, C)
    if Er: break
  print 'Niz: ' +w[:-1],
  if not Er : print 'jest u jeziku'
  else      : print 'NIJE u jeziku!'

  w = Ucitaj_P ()
```

Evo primjera prepoznavača istog jezika uporabom dinamičke tablice prijelaza i akcija. (napisali smo samo promijenjene dijelove programa):

```
# PREPOZNAVAČ JEZIKA Labc S DINAMIČKOM TABLICOM PRIJELAZA I AKCIJA

from gramatika import *

def Tablice ():
  global Tp, Ta

       #  a   b   c   @
  Tp = [  0, -1, -1, -1 ]    # Tablica prijelaza
  Ta = [  1,  2,  4,  5 ]    # Tablica akcija

def Sint_An (C): # Sintaksna analiza
  global a, Tp, Ta
  Er = Tp [C] == -1
```

```
if Er :
    print ('*** sint. pogreška ***')
    return True
  else :
    A = Ta[C]
    if   A == 1: a += 1;  Tp[1] += 1
    elif A == 2:
      Tp[0] = -1;  Ta[C] = 3
      Tp[C] -= 1
      if Tp[1] == -1: Tp[2] = a -1
    elif A == 3:
      Tp[C] -= 1
      if Tp[1] == -1: Tp[2] = a -1
    elif A == 4:
      Tp[C] -= 1
      if Tp[2] == -1: Tp[3] = 0
    elif A == 5:
      Tp[C] -= 1
  return False

x = Ucitaj_P ();  w = komp(x)
while w[0] != '@':
  Tablice();  i = 0;  a = 0
  while i < len(w):
    C, i  = Leks_An (w, i)
    Er = Sint_An (C)
    if Er: break
  print 'Niz: ' +w[:-1],
  if not Er : print 'jest u jeziku'
  else      : print 'NIJE u jeziku!'
  w = Ucitaj_P ()
```

Pitanja i zadaci

1) Definirajte tablicu prijelaza i akcija gramatike s produkcijama:

S → AB:BC| B A → x| y B → ND C → ;S| ε
N → 1| 2| 3| 4| 5| 6| 7| 8| 9
D → N| DD| 0| ε

2) Napišite program sintaksne analize upravljan tablicom prijelaza i akcija za jezik definiran gramatikom s produkcijama:

L → -L| LOL| (L)| t| f
O → +| *| =| =>

3) Napišite program sintaksne analize upravljan dinamičkom tablicom prijelaza i akcija za jezik rimskih brojeva

R = {i,ii,iii,iv,...,c,..., cmxcix,m,mi,...,mmmcmxcix}

4) Napišite program sintaksne analize upravljan tablicom prijelaza i akcija za jezik regularnih izraza čija je beskontekstna gramatika definirana na str. 33.

Literatura

1. AHO, V. A.:
   Indexed grammars - an extension of context-free grammars, J. ACM 15, 647-671, 1968.

2. AHO, V. A.; ULLMAN, D. J.:
   The Theory of Parsing, Translation, and Compiling, vol. I: *Parsing*, Prentice-Hall, 1972.

3. AHO, V. A.; ULLMAN, D. J.:
   Principles of Compiler Design, Addison-Wesley Publishing Company, 1979.

4. AHO; SETHI; ULLMAN:
   Compilers: Principles, Techiques, and Tools, Addison-Wesley Publishing Company, 1986.

5. ANDERSON, A. J.:
   Automata Theory with Modern Applications, CAMBRIDGE UNIVERSITY PRESS, 2006.

6. BACKHOUSE, C. R.:
   Syntax of Programming Languages: Theory and Practice, Prentice-Hall, 1979.

7. BERRY, E. R.:
   Programming Language Translation, Ellis Horwood Limited, 1981.

8. CHRISWELL, I.:
   A Course in Formal Languages, Automata and Groups, Springer-Verlag, London, 2009.

9. DENNING, J. P.; DENNIS, B. J.; QUALITZ, E. J.:
   Machines, Languages, and Computation, Prentice-Hall, 1978.

10. DOVEDAN, Z.:
    Sintaktička analiza jezika sa svojstvima, Ljubljana, Informatica 82/3, 1983.

11. DOVEDAN, Z.:
    Jedan model sintaktičke analize jezika za programiranje, disertacija, Filozofski fakultet, Zagreb, 1992.

12. DOVEDAN, Z.:
    Pascal i programiranje (1), don, Zagreb, 1995.

13. DOVEDAN, Z.:
    FORMALNI JEZICI • sintaksna analiza, Zavod za informacijske studije, Filozofski fakultet, Zagreb, 2003.

14. DOVEDAN HAN, Z.:
    Pascal s tehnikama programiranja (1), VVG, Velika Gorica, 2011.

15. DOVEDAN HAN, Z.:
    FORMALNI JEZICI I PREVODIOCI • regularni izrazi, gramatike, automati, Element, Zagreb, 2012.

16. EIJCK, J. van:
    Sequentially Indexed Grammars, CWI and ILLC, Amsterdam, Uil-OTS, Utrecht, 2005.

17. FRIEDL, J. E. F.:
    Mastering Regular Expressions, O'Reilly, 2006.

18. GOOS, G.; HARTMANIS, J., editors:
Compiler Construction, An Advanced Course, Springer-Verlag, 1976.

19. GRUNE, D.:
Parsing Techniques – A Practical Guide, Ellis-Horwood, 1990.

20. HOPCROFT, E. J.; MOTWANI, R.; ULLMAN, D. J.:
Introduction to Automata Theory, Languages, and Computation, second edition, Addison-Wesley, 2001.

21. KALUŽNIN, A. L.:
Što je matematička logika, Zagreb, Školska knjiga 1975.

22. KARHUMÄKI, J.:
Automata and Formal Languages, Spring, 2005.

23. KUREPA, S.:
Uvod u matematiku, Zagreb, Tehnička knjiga, 1970.

24. LINZ, P.:
An Introduction to Formal Languages and Automata, third edition, Jones and Bartlett Publishers, 2001.

25. PAAKKI, J.:
Atribute Grammar Paradigms – A High-Level Methodology in Language Implementation, ACM Computing Surveys, Vol. 27, No. 2, June 1995.

26. SHALLIT, J.:
A Second Course in Formal Languages and Automata Theory, CAMBRIDGE UNIVERSITY PRESS, Cambridge, 2009.

27. SLONNEGER, K.; KURTZ, B. L:
Formal Syntax and Semantics of Programming Languages, Addison-Wesley Publishing Company, 1995.

28. TOMITA, M., editor:
Current Issues in Parsing Technology, Kluwer Academic Publishers, 1991.

29. WAITE, M. W.; GOOS, G.:
Compiler Construction, Springer-Verlag, 1984.

30. WIRTH, N.:
Algorithms + Data Structures = Programs, Prentice-Hall, 1976.

31. YEH, T. R., editor:
Applied Computation Theory: Analysis, Design, Modeling, Prentice-Hall, 1976.

PRILOZI

Ovdje je dana implementacija osnovnih algoritama transformiranja beskontekstnih gramatika opisanih u prvoj knjizi, uz dodatak algoritma za eliminiranje rekurzija slijeva, str. 55. To je program **ALGORITMI_FJ.py** *koji koristi dva modula,* **fun.py** *i* **gramatika.py**.

fun.py

```
# funkcije i globalne konstante formalnih jezika i automata

import string

A = string.ascii_uppercase
B = string.digits

def veliko_s (C): # veliko slovo?
    return A.find (C) <> -1

def brojka (C): # brojka?
    return B.find (C) <> -1

def komp (X): # izbacivanje razmaka iz niza znakova
    return X.replace (' ','')

def pos (Ch, Y): # pozicija znaka u nizu
    for i in range (len(Y)) :
      if Ch == Y[i] : return i
    return -1

def U (X, Y): # Unija skupuva X i Y
    if   len(X) == 0 : Z = Y
    elif len(Y) == 0 : Z = X
    else             : Z = X | Y
    return Z

def Str (X) :
    Y = ''
    for x in X : Y += x
    return Y
```

gramatika.py

```
# gramatika
# -*- coding: cp1250 -*-

import os
from fun     import *
from easygui import *
from Tkinter import *
```

```
NL = '\n'     # prelazak u novi red
S_ = chr(140) # startni simbol proširene gramatike, S' -> S
O  = chr (183)

def onclick(): pass

def Grm (P): # Definiranje gramatike (iz produkcija)
  N = T = N0 = ''
  for i in range (len(P)):
    C = P[i][0]
    N += C
    if C not in N0: N0 += C
    for j in range (1, len(P[i])):
      Beta = P[i][j]
      for k in range (len(Beta)):
        C = Beta[k]
        if veliko_s(C) and C not in N0 : N0 += C
        if C != '#' and not veliko_s(C) and C not in T:  T += C
  N = list(N); T = list(T); N0 = list(N0)
  return N, T

def Uredi_P (P): # Uređenje produkcija gramatike
  P_ = []
  for i in range (len(P)):
    for j in range (1, len(P[i])):
      P_ += [[P[i][0], P[i][j]]]
  return P_

def Print_skup (Ime, X):
  print Ime, '= {',
  print X[0],
  for x in X[1:]: print ',', x,
  print '}'

def Ispisi_G (Grm, G): # Ispis gramatike
  print Grm, NL
  Print_skup ('N', G[0])
  Print_skup ('T', G[1])
  print 'S =', G[3]
  i = 2
  print 'P:'
  for j in range (len(G[i])) :
    print ' ', G[i][j][0] + ' ->', G[i][j][1],
    for k in range (2, len(G[i][j])) : print '|', G[i][j][k],
    print
  print NL

def LSF (Pi, P): # Lijevo parsanje
  SF = P[0][0]
  print SF
  for i in range (len(Pi)):
    k = Pi[i] -1
    j = SF.find(P[k][0])
    SF = SF[:j] +P[k][1] +SF[j+1:]
    SF = SF.replace ('#', '')
    print '  =>', SF
  print
```

```
def RSF (Pi, P): # Desno parsanje
  SF = P[0][0];  print SF
  while Pi <> []:
    k = Pi.pop() -1;  j = SF.rfind(P[k][0])
    SF = SF[:j] +P[k][1] +SF[j+1:];  SF = SF.replace ('#','');  print '  =>', SF
  print

def Upamti_G (Grm, P): # Pamćenje gramatike
  R = ''
  for i in range (len(P)):
    R += P[i][0] + ' -> ' +P[i][1]
    for j in range (2, len(P[i])) : R += '| ' +P[i][j]
    R += '\n'
  open (Grm, 'w').write (R)

def Ucitaj_G (Poruka, Tip): # Učitavanje gramatike
  N=T=P=[]; S=''
  G = fileopenbox(Poruka, None, Tip, '*')
  if G == None: G = ''
  if os.path.exists(G):
    k = -1
    for line in open (G, 'r') :
      w = komp (line[0:-1]);  k += 1;
      if len(w) > 0:
        P.append ([w[0]]);  i = 1
        while w[i] != '>' : i += 1
        i += 1;  b = ''
        while i < len (w) :
          C = w[i]
          if C == '|' : P[k].append (b); b = ''
          else        : b += C
          i += 1
        P[k].append (b)
    Ok = True
    N, T = Grm (P)
    if G.rfind('\\') > 0: G = G[G.rfind('\\')+1:]
  else:
    Ok = False
  return G, Ok, (N, T, P, N[0])

def Ucitaj_W (): # Učitavanje ulaznog niza
  print
  return komp (raw_input ('Upiši ulazni niz: '))

def Skup_Ne (P, Ter): # Izracunavanje skupa Ne
  Ne = [];  End = False;  n = 0
  while not End:
    for i in range (len (P)):
      A = P[i][0]
      if A not in Ne:
        for j in range (1, len(P[i])):
          W = P[i][j]; C = W[0];  k = 0;  Ok = True
          while k < len (W) : Ok = Ok and C in Ter +['#'] +Ne;  k += 1
          if Ok and A not in Ne: Ne.append (A)
    End = n == len (Ne)
  n = len (Ne)
  return Ne
```

```
def Izbaci_e (G): # Alg. 6-4: Izbacivanje e-produkcija
  N, T, P, S = G;  Ne = ''; X = list(A)
  for x in N : X.remove(x)
  for i in range (len(P)):
    e = False; j = 1
    while not e and j < len(P[i]):
      e = P[i][j] == '#'
      j += 1
    if e : Ne += P[i][0]
  for i in range (len(P)):
    j = 1
    while j < len (P[i]):
      Alfa = Beta = P[i][j]; k = 0; Y = []
      for x in Ne :
        l = pos (x, Beta)
        while l != -1:
          Y.append (k+l); k += l+1; Beta = Beta [l+1:]; l = pos (x, Beta)
      if len(Y) > 0:
        m = len(Y); n = pow(2,m)
        for k in range (1,n):
          Z = bin (k); Z = Z[2:]; Z = '0'*(m-len(Z)) +Z
          Beta = Alfa; d = 0
          for l in range (len(Z)):
            if Z[l] == '1':
              p = Y[l]-d; Beta = Beta[:p] +Beta[p+1:]; d += 1
          j += 1
          P[i].insert (j, Beta)
      else:
        j += 1
    if '#' in P[i]: P[i].remove ('#')
  if S in Ne : P.insert (0, [X[0], S, '#']); S = X[0]
  return (N, T, P, S)

def Izbaci_JP (G) :
  N, T, P, S = G;  Ns = Str (N);  Ts = Str (T);

  "(1) Izgradnja skupa neterminala NA "
  X = []
  for A in N : X.append ([A])
  for i in range (len(P)-1,-1,-1):
    for j in range (1, len(P[i])):
      Beta = P[i][j]
      if Beta in N : X[i].append(Beta)
    for j in range (i, len(X)):
      if X[j][0] in X[i]:
        for k in range (1,len(X[j])):
          x = X[j][k]
          if x not in X[i]: X[i].append(x)
  "(2) zamjena jediničnih produkcija "
  for i in range (len(X)):
    for j in range (1, len(X[i])):
      x = X[i][j];  m = pos(x,Ns)
      if x in P[i]:
        k = P[i].index(x);  P[i].remove(x)
        for l in range (1,len(P[m])):
          Beta = P[m][l];  P[i].insert(k, Beta);  k+= 1
  return (N, T, P, S)
```

```
def Ima_JP (G):
  N, T, P, S = G
  for X in P:
    for Y in X[1:]:
      if Y in N: return True
  return False

def Ima_e (G):
  N, T, P, S = G
  for X in P:
    for Y in X[1:]:
      if Y == '#': return True
  return False

def Provjeri (Net0, Ter, Net):
  print ('N = ', Net0); print ('T = ', Ter)
  if len(Net0) < len(Net) :
    print ()
    print ('Nije definirana produkcija za: ')
    i = 0
    X = Net - Net0
    while i < len (X) :
      if x in X : print x,
      i += 1

"""
Grm, Ok, G = Ucitaj_G('Izaberi gramatiku', "*.grm")
if Ok : Ispisi_G (Grm, G); print
else  : print 'Ne postoji gramatika s danim imenom!'
"""
```

ALGORITMI_FJ.py *Algoritmi transformiranja gramatika*

```
# ALGORITMI TRANSFORMIRANJA GRAMATIKA
# -*- coding: cp1250 -*-

from gramatika import *
from fun       import *
import Tkinter

# 1 --- Izbacivanje jediničnih produkcija -------------------

def Skup_Ne (P, Ter): # Izracunavanje skupa Ne
  Ne = [];  End = False;  n = 0
  while not End:
    for i in range (len (P)):
      A = P[i][0]
      if A not in Ne:
        for j in range (1, len(P[i])):
          W = P[i][j]; C = W[0];  k = 0;  Ok = True
          while k < len (W) : Ok = Ok and C in Ter +['#'] +Ne;  k += 1
          if Ok and A not in Ne: Ne.append (A)
    End = n == len (Ne)
    n = len (Ne)
  return Ne
```

```
# 2 --- Izbacivanje nedokučivih simbola ---------------------

def Izbaci_P (Ne, G) :
  N, T, P, S = G;  i = 0
  while i < len(P) :
    if P[i][0] not in Ne :
      P.remove (P[i])
    else :
      j = 1
      while j < len (P[i]) :
        k = 0; Beta = P[i][j]; Ok = True
        while k < len (Beta) and Ok:
          Ok = Beta[k] in (Ne +T + ['#'])
          if not Ok : P[i].remove (Beta)
          k += 1
        j += 1
    i += 1
  return (Ne, T, P, S)


# 3 --- Izbacivanje neupotrebljivih simbola -----------------

def trans (G) :

  def presjek (X, Y): return [x for x in X if x in Y]

  N, T, P, S = G;  P2 = [];  V = [S];  End = False;  n = 0
  while not End:
    End = True
    for i in range (len (P)) :
      A = P[i][0]
      if A in V:
        for j in range (1, len(P[i])):
          W   = P[i][j]
          Vi  = [x for x in W if x != '#']
          for x in V:
            if x not in Vi: Vi.append(x)
          if len(Vi) != len(V): End = False
          V = Vi
      if not End: P2.append (P[i])
  N2 = presjek (N,V); T2 = presjek (T,V);  G = (N2, T2, P2, S)
  G = Izbaci_P (N2,G)
  return G


# 4 --- Izbacivanje e-produkcija ----------------------------

def Izbaci_e (G):
  N, T, P, S = G;  Ne = '';  X = list(A)
  for x in N : X.remove(x)

  for i in range (len(P)):
    e = False; j = 1
    while not e and j < len(P[i]):
      e = P[i][j] == '#'
      j += 1
    if e : Ne += P[i][0]
```

```
  for i in range (len(P)):
    j = 1
    while j < len (P[i]):
      Alfa = Beta = P[i][j]; k = 0; Y = []
      for x in Ne :
        l = pos (x, Beta)
        while l != -1:
          Y.append (k+l); k += l+1; Beta = Beta [l+1:]; l = pos (x, Beta)
      if len(Y) > 0:
        m = len(Y); n = pow(2,m)
        for k in range (1,n):
          Z = bin (k); Z = Z[2:]; Z = '0'*(m-len(Z)) +Z
          Beta = Alfa; d = 0
          for l in range (len(Z)):
            if Z[l] == '1':
               p = Y[l]-d; Beta = Beta[:p] +Beta[p+1:]; d += 1
          if P[i][0] <> Beta: j += 1;  P[i].insert (j, Beta)
      else:
        j += 1
    if '#' in P[i]: P[i].remove ('#')
    if ''  in P[i]: P[i].remove ('')

  if S in Ne : P.insert (0, [X[0], S, '#']); S = X[0]
  if S not in N: N.insert(0, S)

  return (N, T, P, S)


# 5 --- Izbacivanje jediničnih produkcija --------------------

def Izbaci_JP (G):
  N, T, P, S = G;  P2 = []
  for i in range (len(P)):
    P2.append ([])
    for x in P[i]: P2[i].append(x)

  "(1) Izgradnja skupa neterminala NA "
  X = [[A] for A in N]
  for i in range (len(P2)-1,-1,-1):
    for j in range (1, len(P2[i])):
      Beta = P2[i][j]
      if Beta in N : X[i].append(Beta)
    for j in range (i, len(X)):
      if X[j][0] in X[i]:
        for k in range (1,len(X[j])):
          x = X[j][k]
          if x not in X[i]: X[i].append(x)

  "(2) zamjena jediničnih produkcija "
  for i in range (len(X)):
    for j in range (1, len(X[i])):
      x = X[i][j];  m = N.index (x)
      if x in P2[i]:
        k = P2[i].index(x);  P2[i].remove(x)
        for l in range (1,len(P2[m])):
          Beta = P2[m][l];  P2[i].insert(k, Beta); k+= 1
  return (N, T, P2, S)
```

```
# 6 --- CNF (Chomskyjeva normalna forma) --------------------

def CNF (G):
  def zamijeni (x, y):
    k = n;
    while k < len(P):
      if P[k][1] == x: y = P[k][0]; return y
      k += 1
    y = Nf[0]; P.append([y, x]); Nf.remove(y); N.append(y)
    return y

  N, T, P, S = G;  Nf = list(A);  X = '';  Y = ''
  for x in N : Nf.remove(x)
  n  = len(N);  i  = 0
  while i < len(P):
    for j in range (1, len(P[i])):
      Beta = P[i][j]
      if len(Beta) == 1:
        if not Beta in T:
          print 'jedinična produkcija, ', P[i][0], '->', Beta
          return G
      elif len(Beta) == 2:
        x1 = Beta[0]; x2 = Beta[1]; y1, y2 = x1, x2
        if x1 not in N: y1 = zamijeni (x1, y1)
        if x2 not in N: y2 = zamijeni (x2, y2)
        P[i][j] = y1+y2
      else:
        x1 = Beta[0]; x2 = Beta[1:]; y1, y2 = x1, x2
        if x1 not in N: y1 = zamijeni (x1, y1)
        y2 = zamijeni (x2, y2);  P[i][j] = y1+y2
    i += 1
  return (N, T, P, S)


# 7 --- GNF (Greibachina normalna forma) --------------------

def GNF (G) :
  N, T, P, S = G;  Nf = list(A); X = ''; Y = ''
  for x in N : Nf.remove(x)
  n  = len(N);  i = n-1
  while i != -1:
    k = 1
    while k < len(P[i]):
      NT = P[i][k][0];  Y = []
      j = -1
      if NT in N: j = N.index(NT)
      if j > i:
        Alfa = P[i][k][1:]
        for m in range (1,len(P[j])):
          Beta = P[j][m]
          if Beta[0] in T:
            if m == 1 : P[i][k] = Beta+Alfa
            else      : P[i].insert(k, Beta+Alfa); k+= 1
      else :
        k += 1
    i = i -1
  X = ''; Y = ''; n = len(P)
```

```
  for i in range (n):
    for j in range (1, len(P[i])):
      Beta = P[i][j]
      for k in range (1,len(Beta)):
        NT = Beta[k]
        if NT in T :
          if pos(NT, Y) == -1:
            Y += NT; Z = Nf[0]; X += Z; Nf = Nf[1:]
            P.append ([Z, NT]);  N.append(Z)
          else:
            Z = X[pos(NT, Y)]
          Beta = Beta.replace(NT, Z)
      P[i][j] = Beta
  return (N, T, P, S)

# 8 --- Elimininranje rekurzija slijeva ----------------------

def Eliminiraj_R (G) :
  N, T, P, S = G; P2 = []; N2 = []
  for x in P: P2.append(x)
  for x in N: N2.append(x)
  Nf = list(A)
  for x in N : Nf.remove(x)
  n  = len(N);
  for i in range (len(P)):
    Ai = P[i][0];  X = P[i][1:]; Y = []; k = 0
    while k < len(X):
      Beta = X[k]; NT = Beta[0]
      if NT in T or NT in N and N.index(NT) > i : Y.append(Beta)
      k += 1
    for Beta in Y:
      if Beta in X: X.remove (Beta)
    if len(X) != 0:
      A2 = []; P2[i] = [Ai] +Y +X
      for j in range(0,i+1):
        Y = P2[i][1:]; P2[i] = P2[i][:1];  k = 0
        while k < len (Y):
          Beta = Y[k]; NT = Beta[0]
          if NT == N[j]:
            if j < i:
              Z = P2[j];  m = len(Z)
              for p in range (1, m): P2[i].append(Z[p] +Beta[1:])
            else :
              Nn = Nf[0];  N2.append(Nn);  Nf = Nf[1:]
              Y = Y[k:]; A2 = []
              while len (Y) > 0:
                Beta = Y[0]; NT = Beta[0]; Alfa = Beta[1:]
                if NT != Ai : P2[i].append(Beta)
                else        : A2 = A2 +[Alfa] +[Alfa +Nn]
                Y.remove (Beta)
              m = len (P2[i])
              for p in range (1, m): P2[i].append (P2[i][p] +Nn)
              P2.append([Nn] +A2)
          else:
            P2[i].append(Y[k])
          k += 1
  return (N2, T, P2, S)
```

```
Alg = ('1  Je li L(G) neprazan?',
       '2  Izbacivanje nedokučivih simbola',
       '3  Izbacivanje neupotrebljivih simbola',
       '4  Izbacivanje e-produkcija',
       '5  Izbacivanje jediničnih produkcija',
       '6  Konverzija u CNF (Chomskyjeva normalna forma)',
       '7  Konverzija u GNF (Greibachina normalna forma)',
       '8  Eliminiranje rekurzija slijeva' )


print ('IZABERITE ALGORITAM TRANSFORMIRANJA GRAMATIKE: ')
for i in range (len(Alg)):  print ' ', Alg[i]

i = input ('> ')
Alg2 = list (Alg)
if i <= len(Alg):
  Ime = Alg[i-1][2:]
  Grm, Ok, G = Ucitaj_G (Ime, '*.grm')

  if Ok :
    Ispisi_G (Grm, G);  N, T, P, S = G
    if   i == 1:
      "Je li L(G) neprazan?"
      Ne = Skup_Ne (P, T)
      print
      Print_skup ('Ne', Ne)
      print
      if S in Ne : print 'Jezik je neprazan'
      else       : print 'Jezik je prazan'

    elif i == 2:
      "Izbacivanje nedokučivih simbola"
      Ne = Skup_Ne (P, T)
      G  = Izbaci_P (Ne, G)
      Ispisi_G (Grm +"'", G)

    elif i == 3:
      "Izbacivanje neupotrebljivih simbola"
      Ne = Skup_Ne (P, T)
      G  = Izbaci_P (Ne, G)
      G2 = trans (G)
      print
      Ispisi_G (Grm +'"', G2)

    elif i == 4:
      "Izbacivanje e-produkcija"
      G2 = Izbaci_e(G)
      print NL, 'Ekvivalentna gramatika bez e-produkcija'
      Ispisi_G (Grm[:-3]+'###', G2)

    elif i == 5:
      "Izbacivanje jediničnih produkcija"
      G2 = Izbaci_JP (G)
      if G2 != G:
        print; Ispisi_G ('(bez jediničnih produkcija)', G2)
      else:
        print; print 'Gramatika ne sadrži jedinične produkcije!'
```

```
    elif i == 6:
      "Konverzija u CNF (Chomskyjeva normalna forma)"
      G2 = Izbaci_JP (G)
      if G2 != G:
        print
        print ('Gramatika sadrži jedinične produkcije!')
        Y = raw_input ('Eliminiram jedinične produkcije (D/N)? > ')
        if Y[0] in ['d','D']:
          G2 = CNF(G2)
          print
          Ispisi_G ('Ekvivalentna gramatika u CNF-u', G2)
      else:
        G2 = CNF(G)
        print
        Ispisi_G ('Ekvivalentna gramatika u CNF-u', G2)

    elif i == 7:
      "Konverzija u GNF (Greibachina normalna forma)"
      Ok = True
      for p in P:
        for q in p[1:]:
          if p[0] == q[0]: Ok = False
      if Ok:
        G2 = GNF(G)
        print()
        Ispisi_G ('Ekvivalentna gramatika u GNF-u', G2)
      else:
        print
        print 'Gramatika je rekurzivna slijeva!'

    elif i == 8:
      " Eliminiranje rekurzija slijeva"
      G2 = Eliminiraj_R (G)
      if G2 != G:
        print
        Ispisi_G ('Ekvivalentna gramatika bez rekurzija slijeva', G2)
      else:
        print ('Gramatika nije rekurzivna slijeva!')
  else :
    print 'Ne postoji gramatika s danim imenom!'
else:
  print 'Niste izabrali nijedan algoritam!'
```

Kazalo

Zagreb, prosinac 2012.