Earleyev parser

Lončar, Luka

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Luka Lončar

EARLEYEV PARSER

Diplomski rad

Voditelj rada:
doc. dr. sc. Vedran Čačić

Zagreb, rujan 2021.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Mentoru, doc. dr. sc. Vedranu Čačiću na velikom strpljenju, pomoći te uloženom vremenu pri izradi ovog diplomskog rada.
Roditeljima i bakama na bezuvjetnoj ljubavi i potpori tijekom školovanja, te prijateljima koji su me prebrojivo mnogo puta pitali kada ću završiti studij.

Sadržaj

Uvod

U programiranju se često pojavljuje potreba za parsiranjem nekog beskontekstnog jezika. U tom se slučaju obično koriste generatori parsera (kao što su *Parsec* ili *yacc*), ili se napiše neki parser specijaliziran za zadani problem. Cilj ovog diplomskog rada je implementacija Earleyevog algoritma, koji, po cijenu nekih komplikacija u implementaciji, pruža dobar kompromis teorijske čistoće i primjenjivosti na stvarne probleme prilikom parsiranja jezika zadanih beskontekstnim gramatikama.

U prvom poglavlju uvodimo definicije i pojmove iz teorije formalnih jezika koji su korisni za daljnju raspravu. Krenut ćemo s regularnim jezicima, zadanih pomoću regularnih izraza, s kojima se čitatelj sigurno već susreo. Zatim ćemo definirati beskontekstne gramatike koje ćemo koristiti kao ulazni parametar u algoritmu. Objasnit ćemo algoritam CYK koji se obično promatra u teorijskim razmatranjima i koji je dobar što se tiče složenosti u najgorem slučaju, ali se ne ponaša puno bolje na jezicima kakvi su u primjenama uobičajeni, a koji se obično mogu parsirati prilično brže.

U drugom poglavlju opisujemo implementaciju Earleyevog parsera, napisanu u programskom jeziku C++. Osnovna implementacija algoritma, kao i neka poboljšanja, napravljena je po uzoru na [6]. Kroz implementaciju ćemo objašnjavati sam Earleyev algoritam i navesti neke od dobro poznatih optimizacija koje unapređuju osnovni algoritam. Na primjeru nekoliko beskontekstnih gramatika pokazat ćemo proces parsiranja nekih riječi koje pripadaju jezicima koje te gramatike generiraju.

Poglavlje 1

Teorijska podloga

Prije nego što možemo početi raspravu o parsiranju, moramo uvesti neke osnovne pojmove teorije formalnih jezika. *Abeceda* je konačan neprazan skup koji označujemo sa Σ, a elemente abecede nazivamo *znakovima*. Znakove najčešće označujemo s α ili β, a konkretne znakove pišemo u fontu stalne širine[1]. *Riječ* nad Σ je niz konačno mnogo znakova iz Σ. Neka je w riječ nad abecedom Σ. S $|w|$ označujemo duljinu riječi w, tj. broj znakova koje w sadrži. Praznu riječ (riječ duljine 0) označujemo s ε, a svaku riječ duljine 1 poistovjećujemo s jedinim njenim znakom. Ako je w riječ duljine n, tada pišemo $w = \alpha_1\alpha_2\cdots\alpha_n$, pri čemu je $\alpha_i \in \Sigma$ za svaki $i \in \{1, 2, \ldots, n\}$. Na riječima imamo operaciju *konkatenacije*: za $u = \alpha_1 \cdots \alpha_n$ i $v = \beta_1 \cdots \beta_m$, pišemo $uv \coloneqq \alpha_1 \cdots \alpha_n \beta_1 \cdots \beta_m$.

Kao što se aritmetici bavimo proučavanjem brojeva i osnovnim računskim operacijama nad njima, tako se u teoriji formalnih jezika bavimo jezicima i operacijama nad njima. *Jezik* definiramo kao skup riječi nad istom abecedom. Svaki jezik koji sadrži samo jednu riječ poistovjećujemo s tom riječju: preciznije, jezik $\{w\}$ poistovjećujemo s riječju w. Specijalno, jezik koji sadrži samo praznu riječ označujemo s ε. Važno je napomenuti da se taj jezik razlikuje od *praznog jezika*, koji ne sadrži nijednu riječ i označuje se oznakom $\emptyset$.

1.1 Regularni jezici

Sada ćemo formalno definirati *regularne operacije* nad jezicima koje će nam služiti kasnije prilikom definiranja regularnih jezika i regularnih izraza (definicija je preuzeta iz [5]).

Definicija 1. Neka su A i B jezici. Definiramo regularne operacije:

- *unija*: $A \cup B \coloneqq \{x \mid x \in A \vee x \in B\}$;
- *konkatenacija*: $AB \coloneqq \{x\,y \mid x \in A \wedge y \in B\}$;

[1] Na primjer, konkretni znakovi a, b i 0 će u radu biti zapisani kao `a`, `b` odnosno `0`.

- *zvijezda*: $A^* := \{x_1 x_2 \cdots x_k \mid k \geq 0 \wedge (\forall i \leq k)(x_i \in A)\}$. ◁

Konkatenacija ima multiplikativnu notaciju i uobičajene oznake za potenciranje u skladu s njom. Tako, na primjer, za riječ w pišemo $w^2 := ww$, dok za jezik L pišemo $L^3 := LLL$. Zvijezdu još nazivamo *Kleenejevom zvijezdom* i u skladu s multiplikativnom notacijom konkatenacije za proizvoljan jezik L imamo: $L^* = \cup_{k\in\mathbb{N}_0} L^k$. Zgodno je radi nekih primjera spomenuti i postojanje operacije *Kleenejevog plusa* koji je za jezik L definiran pomoću: $L^+ := \cup_{k\in\mathbb{N}_+} L^k$.

Definicija 2. Kažemo da je R *regularni izraz* ako je R izraz jednog od sljedećih oblika:

1. α za neki $\alpha \in \Sigma$,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, pri čemu su R_1 i R_2 regularni izrazi,
5. $(R_1 R_2)$, pri čemu su R_1 i R_2 regularni izrazi,
6. R_1^*, pri čemu je R_1 regularan izraz. ◁

Primijetimo da u definiciji 2 regularne izraze R gledamo kao riječi nad proširenom abecedom $\Sigma \dot\cup \{(,), \cup, ^*, \varepsilon, \emptyset\}$. Zato su regularni izrazi R_1 i R_2 uvijek manji (kao riječi su strogo kraći) nego R i time smo izbjegli cirkularnost u definiciji. Obično, ako ne dolazi do dvosmislenosti, prilikom pisanja regularnih izraza ispuštamo nepotrebne zagrade, smatrajući da najveći prioritet ima zvijezda, zatim konkatenacija i na kraju unija (baš kao potenciranje, množenje i zbrajanje brojeva u aritmetici). Također, unija i konkatenacija su asocijativne operacije, pa možemo pisati uniju odnosno konkatenaciju triju ili više regularnih izraza bez zagrada.

Ako je R neki regularni izraz, $L(R)$ je jezik koji on opisuje. Kažemo da je jezik $L \subseteq \Sigma^*$ *RI-regularan* ako je opisan nekim regularnim izrazom. Klasu svih RI-regularnih jezika kraće označujemo s Reg_{RI}.

Regularni izrazi imaju veliku praktičnu primjenu u današnjem računarstvu. Na primjer, u primjenama koje obuhvaćaju obradu teksta, korisnici često imaju potrebu za traženjem nekog uzorka znakova u nekom tekstu. U tom slučaju, se regularni izrazi koriste zato što mogu opisati uzorke koje često želimo tražiti. No, nama je zanimljiviji značaj regularnih izraza pri izradi programskih prevodilaca (*compilers*) za programske jezike. Razlog tome je da *tokene*, elementarne objekte koji upravljaju sintaksnom analizom u programskim jezicima, možemo najčešće opisati pomoću regularnih izraza. Na primjer, nazivi varijabla, literali, separatori, ključne riječi itd. mogu biti opisani regularnim izrazima.

Primjer 3. Neka je $\Sigma = \{\mathtt{0}, 1\}$ abeceda. Evo jednog primjera regularnog izraza nad Σ:

$$R_1 := (\mathtt{0} \cup 1)\,\mathtt{0}^*.$$

On opisuje jezik sastavljen od riječi kojima je prvi znak $\mathtt{0}$ ili 1, nakon kojeg slijedi proizvoljno mnogo znakova $\mathtt{0}$. S obzirom na prioritete regularnih operacija imamo da su riječi $\mathtt{0}$, 1, $\mathtt{00}$ i $\mathtt{1000}$ članovi $L(R_1)$, dok riječ $\mathtt{0100}$ nije. Primijetimo da izostavljanjem zagrada dobijemo regularni izraz:

$$R_2 := \mathtt{0} \cup 1\,\mathtt{0}^*$$

koji opisuje drugi RI-regularni jezik, koji se sastoji od riječi $\mathtt{0}$ i svih riječi koje imaju znak 1 na prvom mjestu, a nakon njega proizvoljno mnogo znakova $\mathtt{0}$. Očito su riječi $\mathtt{0}$, 1, i $\mathtt{1000}$ također članovi jezika $L(R_2)$, dok je riječ $\mathtt{00}$ samo član $L(R_1)$, tj. vrijedi $\mathtt{00} \in L(R_1) \setminus L(R_2)$. ◁

Iako ih nećemo formalno definirati, dobro je znati da postoje idealizirani (matematički) strojevi koji prepoznaju regularne jezike. Na primjer, u [5], regularni jezici su definirani pomoću deterministčnih *konačnih automata (KA)*. Neformalno, automat se kreće kroz stanja prolazeći kroz riječ i ovisno o stanju u kojem se nalazi donosi odluku o prihvaćanju riječi. Kažemo da KA $\mathcal{M}$ *prepoznaje* jezik $L(\mathcal{M}) := \{w \in \Sigma^* \mid \mathcal{M} \text{ prihvaća } w\}$. Slično kao kod regularnih izraza, kažemo da je jezik *KA-regularan*, ako postoji neki KA koji ga prepoznaje, a s Reg_{KA} je označena klasa svih KA-regularnih jezika. Konačni automati se mogu poopćiti na *nedeterministične konačne automate (NKA)*. Kažemo da je jezik *NKA-regularan*, ako postoji neki NKA koji ga prepoznaje, i klasu takvih jezika označujemo s Reg_{NKA}. Pokaže se [5] da je jezik KA-regularan ako i samo ako je NKA-regularan te time jednakost klasa Reg_{KA} i Reg_{NKA}. Iako je klasa Reg_{RI} trivijalno zatvorena na uniju, konkatenaciju i zvijezdu prema definiciji 2, KA i NKA nam pomažu prilikom dokazivanja zatvorenosti te klase na neke druge skupovne operacije kao što je presjek. No, direktna pretvorba nekog regularnog izraza u KA nije nimalo jednostavna. Upravo je to motivacija za uvođenje NKA kao međukoraka: prvo, lako opisivom konstrukcijom dobijemo NKA iz nekog regularnog izraza, a onda partitivnom konstrukcijom iz tog NKA dobijemo KA. Cijeli postupak pretvorbe nekog regularnog izraza u KA je definiran i njegova korektnost dokazana u [7].

Uvedimo sada još jedan način definiranja formalnih jezika, preko *gramatika*. Možda nije intuitivno, ali najprije ćemo definirati općenitiji pojam gramatike od onog koji je potreban za generiranje regularnih jezika. Za razliku od strojeva koji postaju sve kompleksniji kako bi mogli prepoznati šire klase jezika, opće gramatike je vrlo jednostavno definirati, a uže klase jezika generiraju gramatike s raznim restrikcijama na pravila.

Definicija 4. *Gramatika* nad abecedom Σ je transformacijski sustav $\mathcal{G} = (V, \Sigma, \rightarrow, S)$, koji pored abecede Σ ima još:

- konačni skup V, disjunktan sa Σ, čije elemente zovemo *varijable*;
- istaknuti element $S \in V$ koji zovemo *početna varijabla*;
- konačnu binarnu relaciju $\rightarrow$ na Y^*, čije elemente zovemo *pravila*.

Elemente skupa $Y := \Sigma \dot{\cup} V$, dakle varijable i znakove, zovemo jednim imenom *simboli*. $\triangleleft$

Neka je $\mathcal{G}$ gramatika. Za riječi u i v nad Y, kažemo da u *daje* v, i pišemo $u \Rightarrow v$, ako postoji pravilo $s \rightarrow t$ u $\mathcal{G}$, te riječi a i b nad Y takve da je $u = asb$ i $v = atb$. *Izvod* u $\mathcal{G}$ je konačan niz riječi $u_0, u_1, \ldots, u_n$ nad Y, takav da je $u_0 = S$, te $u_{i-1} \Rightarrow u_i$ za sve $i \in \{1, 2, \ldots, n\}$. Zovemo ga *izvod za* w ako je $u_n = w$, i kratko pišemo $S \Rightarrow^* w$. Kažemo da $\mathcal{G}$ *izvodi* riječ w nad Σ ako postoji izvod za w u $\mathcal{G}$. Kažemo da $\mathcal{G}$ *generira* jezik $L(\mathcal{G}) := \{w \in \Sigma^* \mid \mathcal{G} \text{ izvodi } w\}$.

gramatike	**jezici**	**strojevi (automati)**
desnolinearne (DLG), lijevolinearne (LLG)	regularni (*Reg*)	konačni automati (KA, NKA)
beskontekstne (BKG), Chomskyjeve (ChNF)	beskontekstni (*BK*)	potisni automati (PA, JPA)
kontekstne (KG), monotone (MG)	kontekstni (*Kon*)	ograničeni automati (OA, JOA)
ne postoje	rekurzivni (*Rek*)	Turingovi odlučitelji (TO)
opće (OG)	rekurzivno prebrojivi (*RE*)	Turingovi strojevi (TP, NTP), Turingovi enumeratori (TE)

Tablica 1.1: Chomskyjeva hijerarhija formalnih gramatika, jezika koje generiraju i strojeva koji prepoznaju te jezike

Dodatno objašnjenje značenja skraćenica iz tablice 1.1 i pojmova koji stoje iza njih se može naći u [7].

Prvo definirajmo oblik gramatike potrebne za generiranje regularnih jezika tako što ćemo uvesti restrikcije na njena pravila.

Definicija 5. Neka je $\mathcal{G}$ gramatika. Kažemo da je $\mathcal{G}$ *desnolinearna (DLG)* ako je svako njeno pravilo *desnolinearno*, odnosno jednog od sljedeća dva oblika:

- $A \rightarrow \alpha B$,

- $A \to \varepsilon$,

gdje su A i B varijable, a α znak. ◁

Za jezik kažemo da je *DLG-regularan*, ako ga generira neka DLG. Klasu svih DLG-regularnih jezika označujemo s Reg_{DLG}.

Primjer 6. Uzmimo abecedu $\Sigma = \{\mathtt{0}, 1\}$, regularni izraz $(\mathtt{0} \cup 1)\,\mathtt{0}^*$ iz primjera 3 i konstruirajmo DLG $\mathcal{G}$ koja generira jezik koji taj regularni izraz opisuje. Neka je S početna varijabla. Regularni izraz $\mathtt{0} \cup 1$ opisuje jezik $\{\mathtt{0}, 1\}$, što znači da početna varijabla daje riječi $\mathtt{0}A$ ili $1A$, pri čemu je A nova varijabla. Nakon prvog znaka, dolazi proizvoljno mnogo znakova $\mathtt{0}$ što bi značilo da varijabla A daje riječ ε ili $\mathtt{0}A$. Time smo dobili sva pravila gramatike $\mathcal{G}$:

1. $S \to \mathtt{0}A$;
2. $S \to 1A$;
3. $A \to \mathtt{0}A$;
4. $A \to \varepsilon$;

pri čemu je $V = \{S, A\}$.

Često u praksi specificiramo gramatiku tako da samo zapišemo njezina pravila. Iz tih pravila se varijable mogu identificirati kao simboli koji se pojavljuju s lijeve strane pravila, dok su znakovi oni simboli koji se pojavljuju samo na desnoj strani pravila. Isto tako, početna varijabla se obično ne ističe, već se uzima da je početna ona varijabla koja se nalazi s lijeve strane prvog zapisanog pravila. Radi jednostavnosti grupiramo sva pravila koja s lijeve strane imaju istu varijablu te ih pišemo kao jedno pravilo. To pravilo s lijeve strane ima varijablu, a na desnoj strani novog pravila su mu nabrojeni izrazi koji se pojavljuju s desne strane svih grupiranih pravila, odvojeni znakom $|$. To predstavlja moguće supstitucije za varijablu s lijeve strane. Tako dobivamo kraći zapis gramatike $\mathcal{G}$:

$$S \to \mathtt{0}A \mid 1A$$
$$A \to \mathtt{0}A \mid \varepsilon$$

iz kojega možemo pročitati sve bitne informacije za specifikaciju gramatike. ◁

Vidjeli smo da postoji više načina definiranja klase regularnih jezika — preko regularnih izraza, konačnih automata ili pak preko desnolinearnih gramatika. No, u [5] je dokazana ekvivalentnost tih definicija, zbog čega je:

$$Reg_{\mathrm{KA}} = Reg_{\mathrm{NKA}} = Reg_{\mathrm{RI}} = Reg_{\mathrm{DLG}}.$$

Tu klasu jednostavno zovemo *klasa regularnih jezika* i označujemo je s Reg.

Primjer 7. Jezik koji se sastoji od riječi koje se sastoje od konačno mnogo (nula ili više) znakova a,

$$\{\mathsf{a}^n \mid n \in \mathbb{N}_0\},$$

očito je regularan jer je jednak $L(\mathsf{a}^*)$.

No, razmotrimo jezik:

$$L_= := \{\mathsf{a}^n\mathsf{b}^n \mid n \in \mathbb{N}_0\}.$$

Na prvi je pogled jezik $L_=$ sličan jeziku $L(\mathsf{a}^*)$ te bismo mogli pomisliti da je regularan. No, u [5] se pokaže da $L_=$ zapravo nije regularan korištenjem *leme o pumpanju*, koja predstavlja najčešće korišteni alat za dokazivanje neregularnosti pojedinih jezika. ◁

Iz primjera 7 vidimo da postoje vrlo jednostavni jezici koji ne spadaju u klasu *Reg*. Stoga, definirajmo *beskontekstne gramatike*, kojima ćemo se posebno baviti u ovom radu, kako bismo kasnije definirali klasu formalnih jezika širu od *Reg*.

1.2 Beskontekstni jezici

Regularni izrazi generiraju beskonačne jezike operacijom zvijezde, odnosno mogućnošću ponavljanja određenog obrasca proizvoljan broj puta. No, sada ćemo definirati gramatike i jezike koji dodaju mogućnost ugnježđivanja obrazaca, a to su *beskontekstne* gramatike i jezici. Kad kažemo ugnježđivanje, mislimo na situaciju kad se prilikom izvođenja riječi pojedina varijabla pojavljuje na lijevoj i na desnoj strani nekog pravila, pa tako instance korištenja određenog pravila mogu biti ugniježđene jedna u drugoj. Beskontekstne gramatike se obično koriste prilikom specifikacije i kompilacije programskih jezika, te se često koriste kao referenca prilikom učenja sintakse nekog programskog jezika.

Definicija 8. Neka je $\mathcal{G}$ gramatika. Kažemo da je $\mathcal{G}$ *beskontekstna (BKG)*, ako je svako njeno pravilo *beskontekstno*, odnosno oblika

$$A \to w$$

gdje je $A \in V$ varijabla, a $w \in Y^*$ konačni niz simbola. ◁

Drugim riječima, gramatika je beskontekstna ako za njena pravila vrijedi $(\to) \subseteq V \times Y^*$, tj. jednu varijablu uvijek zamjenjujemo s nula, jednim ili više simbola, ne mareći pritom za kontekst (znakove oko) te varijable. Jezik je *beskontekstan* ako ga generira neka BKG, a klasu svih beskontekstnih jezika označujemo s BK.

Primjer 9. Jedna generalizacija jezika $L_=$ iz primjera 7 jest *jezik dobro sparenih zagrada* koji je beskontekstan i koristan u parserima za aritmetičke izraze. Jezik $L_=$ možemo shvatiti kao jedan blok zagrada, kod kojeg umjesto znaka `(` koristimo znak a, a umjesto

znaka) koristimo znak b. Generalizacija jezika $L_=$ na jezik dobro sparenih zagrada nam omogućuje korištenje više vrsta zagrada i slaganje više blokova zagrada jedan do drugog. Pogledajmo sada primjer BKG koja generira jezik dobro sparenih zagrada nad abecedom $\Sigma = \{(,), [,]\}$. Neka je S početna i jedina varijabla. Pravila koja se pojavljuju u toj gramatici su:

1. $S \to SS$;
2. $S \to (S)$;
3. $S \to [S]$;
4. $S \to \varepsilon$.

BKG u ovom primjeru, prema konvenciji o kraćem zapisu gramatika iz primjera 6, kraće zapisujemo kao:

$$S \to SS \mid (S) \mid [S] \mid \varepsilon$$

i time smo pokazali da je jezik dobro sparenih zagrada beskontekstan. Ako u pravilu 2 zamijenimo znak) znakom b i znak (znakom a, te izostavimo pravila 1 i 3, dobijemo gramatiku

$$S \to \mathtt{a}S\mathtt{b} \mid \varepsilon$$

koja generira jezik $L_=$. ◁

Činjenica da se svaki regularni jezik može generirati pomoću DLG, povlači da je svaki regularni jezik beskontekstan (jer je očito svako desnolinearno pravilo beskontekstno). S druge strane smo vidjeli u primjeru 7 da jezik $L_=$ nije regularan, što povlači $Reg \subset BK$. Još jedna bitna razlika klase BK od klase Reg jest nezatvorenost na neke skupovne operacije.

Neka su $\mathcal{G}_i = (V_i, \Sigma, \to_i, S_i)$, $i \in \{1, 2\}$, BKG nad istom abecedom. Bez smanjenja općenitosti možemo pretpostaviti da vrijedi $V_1 \cap V_2 = \emptyset$. Neka je S varijabla koja se ne pojavljuje ni u V_1 ni u V_2, tj. vrijedi $S \notin V_1 \dot\cup V_2$. Sada možemo definirati nove BKG, dodavanjem beskontekstnih pravila, koje pokazuju zatvorenost BK na uniju, konkatenaciju i Kleenejeve operacije:

1. $\mathcal{G}_3 := (V_1 \dot\cup V_2 \dot\cup \{S\}, \Sigma, (\to_1) \cup (\to_2) \cup \{S \to S_1, S \to S_2\}, S)$;
2. $\mathcal{G}_4 := (V_1 \dot\cup V_2 \dot\cup \{S\}, \Sigma, (\to_1) \cup (\to_2) \cup \{S \to S_1S_2\}, S)$;
3. $\mathcal{G}_5 := (V_1 \dot\cup \{S\}, \Sigma, (\to_1) \cup \{S \to \varepsilon, S \to S_1S\}, S)$;
4. $\mathcal{G}_6 := (V_1 \dot\cup \{S\}, \Sigma, (\to_1) \cup \{S \to S_1, S \to S_1S\}, S)$.

Tako smo dobili nove BKG koje generiraju redom

$$L(\mathcal{G}_3) = L(\mathcal{G}_1) \cup L(\mathcal{G}_2),\ L(\mathcal{G}_4) = L(\mathcal{G}_1)L(\mathcal{G}_2),\ L(\mathcal{G}_5) = L(\mathcal{G}_1)^*,\ L(\mathcal{G}_6) = L(\mathcal{G}_1)^+,$$

što dokazuje zatvorenost skupa beskontekstnih jezika nad fiksnom abecedom Σ na spomenute operacije. No, u [5] je pokazano da zatvorenost na ostale skupovne operacije (presjek, komplement i skupovna razlika) ne vrijedi.

1.3 Parser

Objasnimo sada algoritme koji nam služe za utvrđivanje je li neka riječ izvedena iz određene BKG kako bismo kasnije mogli razgovarati o implementaciji jednog takvog algoritma. No, objasnimo prvo što rješavamo takvim algoritmima.

Kažemo da je *problem* familija pitanja (koja zovemo *instancama* problema), svako od kojih ima jednoznačan potvrdan ili niječan odgovor. Ulaz algoritma shvaćamo kao indeks kojim iz familije biramo konkretnu instancu problema koju rješavamo. Problem je *odlučiv* ako postoji algoritam koji prima ulaz te u konačno mnogo koraka daje izlaz *True* odnosno *False*, koji predstavlja točan odgovor na pitanje postavljeno u instanci problema indeksiranoj tim ulazom. Važan problem kojim se ovdje bavimo jest problem *prihvaćanja za* $\mathcal{G}$ (pri čemu je $\mathcal{G}$ neka BKG nad abecedom Σ), definiran kao familija $\{w \in^? L(\mathcal{G}) \mid w \in \Sigma^*\}$. U kontekstu upravo navedene definicije, rješenje problema bio bi algoritam koji daje odgovor na pitanje je li riječ, zadana kao njegov ulaz, izvedena iz BKG $\mathcal{G}$.

No, često bismo u primjenama htjeli znati više od samog odgovora na pitanje. Upravo to se događa prilikom kompilacije programa. Kompilacija jest proces pretvaranja odnosno prevođenja nekog programskog jezika u drugi, obično u strojni jezik, kako bi se mogao izvesti napisani izvorni kod. Prvi korak prilikom kompilacije nekog izvornog koda u strojni jezik zovemo *leksičkom analizom*. Leksički analizator je program koji pretvara izvorni kod u odgovarajuće tokene. Jednom, kad je validni sadržaj pojedinog tokena opisan regularnim izrazom, može se krenuti s leksičkom analizom. Leksički analizator kao ulaz prima neki niz znakova i pokušava ga raščlaniti na odgovarajuće tokene s obzirom na leksičku strukturu programskog jezika za koji je napisan. Leksički analizator se uobičajeno sparuje s *parserom*, kojem daje izlaz svog izvršavanja. Parser koristi tokene dobivene leksičkom analizom za provjeru odgovara li sintaksa napisanog izvornog koda sintaksi programskog jezika (opisanog beskontekstnom gramatikom nad abecedom koju čine tipovi tokena) za koji je taj izvorni kod napisan. Drugim riječima, parser izvršava *sintaksnu analizu*, koja provjerava je li izvorni kod sintaksno dobro napisan. On analizira tokene i raspoređuje ih u *stablo parsiranja* ili *apstraktno sintaksno stablo*, strukture koje se dalje koriste prilikom kompilacije programa. Parser samo provjerava sintaksnu ispravnost koda, dok kompajler koristi njegov izlaz kako bi provjerio i semantičku ispravnost koda. Neki kod može biti

sintaksno ispravan, no semantički ne mora imati smisla (na primjer, neka varijabla se koristi iako nikada nije bila definirana). Parser neće znati ima li kod smisla, već samo gleda strukturu koda (precizno, gleda samo tipove tokena, a ne njihove sadržaje), dok bi onda kompajler prolaskom kroz stablo parsiranja uvidio da izvorni kod nije valjan.

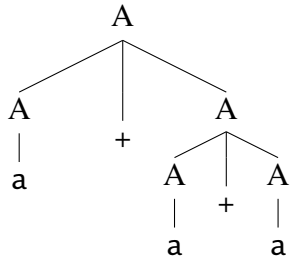
Mi želimo proučiti problem prihvaćanja za beskontekstne jezike — je li riječ u beskontekstnom jeziku koji generira BKG — te ako jest, konstruirati stablo parsiranja za tu riječ. Definirajmo stabla parsiranja kao u [7]. Kažemo da je *stablo* konačna skupina *čvorova* organiziranih u neku rekurzivnu strukturu. Preciznije, svako stablo je konačni niz $(k, s_1, s_2, \ldots, s_n), n \in \mathbb{N}_0$, gdje je k čvor zvan *korijen*, a $s_1, \ldots, s_n$ su (manja) stabla zvana *glavna podstabla*. Korijene glavnih podstabala zovemo *djecom* čvora k.

Stablo parsiranja iz BKG $\mathcal{G}$ je stablo čije čvorove označujemo simbolima gramatike $\mathcal{G}$, kojem je korijen stabla označen početnom varijablom, a svaki čvor je ili čvor bez djece (*list*) označen znakom iz Σ, ili čvor označen varijablom iz V čija djeca, čitana redom, predstavljaju desnu stranu jednog pravila za tu varijablu (pri čemu različiti čvorovi mogu biti označeni istim simbolom). *Stablo parsiranja za riječ w* je stablo parsiranja čiji listovi čitani redom čine riječ w. Riječ $w \in L(\mathcal{G})$ je *$\mathcal{G}$-višeznačna* ako postoji više stabala parsiranja iz $\mathcal{G}$ za w. BKG $\mathcal{G}$ je *višeznačna* ako postoji $\mathcal{G}$-višeznačna riječ iz $L(\mathcal{G})$.

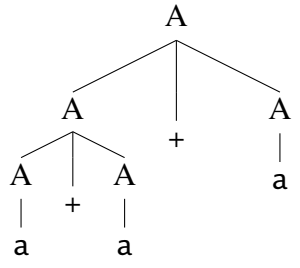
Neka je $\mathcal{G}$ neka BKG koja generira jezik L. Kažemo da je L *višeznačan* ako je svaka BKG koja generira L višeznačna. Primijetimo da višeznačnost $\mathcal{G}$ ne povlači višeznačnost od L, jer možda postoji jednoznačna BKG koja također generira L. Višeznačnost gramatike implicira da barem jedna riječ, element jezika koji ta gramatika generira, ima više stabla parsiranja. Takve gramatike obično nisu pogodne za opisivanje sintakse nekog programskog jezika — jer postojanje dva stabla parsiranja za neki izvorni kod često implicira dva različita načina izvršavanja napisanog programa i time dvije interpretacije izvornog koda, što pokušavamo izbjeći. Štoviše, u višeznačnoj gramatici, možemo imati eksponencijalno mnogo stabala parsiranja za neki ulaz.

Primjer 10. Neka je dana BKG s pravilom $A \to A{+}A \mid \mathtt{a}$. Riječ a+a+a ima sljedeća dva stabla parsiranja:

a)



b)



U oba primjera, čitajući listove slijeva nadesno, dobivamo istu riječ a+a+a — iako su stabla parsiranja različita. ◁

Primijetimo da kod utvrđivanja višeznačnosti gramatike nismo brojili izvode za neku riječ. Razlog tome je činjenica da stablo parsiranja možemo pretvoriti u izvod na više načina. Na primjer, dva izvoda se mogu razlikovati jedino u redoslijedu primjena pravila, a ne u njihovoj strukturi. Zato uvodimo određenu restrikciju na oblik izvoda, koja će nam omogućiti da vjerno opišemo jednoznačnost riječi.

Definicija 11. Neka je $\mathcal{G}$ gramatika. Za riječi u i v nad Y^*, kažemo da *u lijevo daje v* i pišemo $u \Rightarrow v$, ako postoji pravilo $s \to t$ u $\mathcal{G}$, te riječi a nad Σ^* i b nad Y^* takve da je $u = asb$ i $v = atb$. *Lijevi izvod za* $w \in \Sigma^*$ je izvod oblika $S \Rightarrow^* w$. ◁

Teorem 12. *Neka je $\mathcal{G}$ beskontekstna gramatika. Riječ $w \in L(\mathcal{G})$ je jednoznačna ako i samo ako postoji jedinstveni lijevi izvod iz $\mathcal{G}$ za w.*

Dokaz. Definirajmo rekurzivne funkcije koje pretvaraju lijevi izvod u stablo parsiranja i obrnuto.

Prvo opišimo rekurzivnu funkciju f koja prima stablo parsiranja i vraća lijevi izvod. Neka je riječ $w \in L(\mathcal{G})$ i neka je ulaz funkcije stablo parsiranja $(V, T_1, \ldots, T_n)$ za w. Ako je $n = 0$ vraćamo trivijalni izvod (simbol V „izvodi" samog sebe u 0 koraka). Inače, za svaki $i \in \{1, \ldots, n\}$, T_i je stablo parsiranja iz korijena stabla, nazovimo ga X_i, u neku riječ $w_i \in \Sigma^*$. Tad je $f(T_i)$ lijevi izvod $X_i \Rightarrow^* w_i$. Ako sad na početak svakog elementa od $f(T_i)$ dodamo $w_1 \cdots w_{i-1}$ i na kraj mu dodamo $X_{i+1} \cdots X_n$, dobit ćemo lijevi izvod:

$$w_1 \cdots w_{i-1} X_i X_{i+1} \cdots X_n \Rightarrow^* w_1 \cdots w_{i-1} w_i X_{i+1} \cdots X_n.$$

Ulančavanjem takvih izvoda i dodavanjem $V \to X_1 \cdots X_n$ na početak, dobijemo lijevi izvod $V \Rightarrow^* w_1 \cdots w_n$, koji vratimo iz f.

Napišimo sad rekurzivnu funkciju g koja prima lijevi izvod i generira stablo parsiranja. Neka je ulaz lijevi izvod za w oblika:

$$S \to X_1 X_2 X_3 \cdots X_n \Rightarrow^* w_1 X_2 X_3 \cdots X_n \Rightarrow^* w_1 w_2 X_3 \cdots X_n \Rightarrow^* \cdots \Rightarrow^* w_1 w_2 w_3 \cdots w_n.$$

Tada svaki X_i lijevo izvodi podriječ w_i, i svaki takav lijevi izvod D_i se može dobiti iz odgovarajućeg podniza ulaza tako da se u svakom njegovom članu s početka maknu w_j za $j < i$, te s kraja maknu X_k za $k > i$. Vratimo stablo $(S, g(D_1), \ldots, g(D_n))$. Rekurzija stane kad je $X_i = w_i \in \Sigma$ (izvod je trivijalan, u 0 koraka) i u tom slučaju vratimo list u kojem piše taj simbol w_i.

Lako se vidi da su f i g jedna drugoj inverzne funkcije, pa su obje bijekcije. To znači da je skup lijevih izvoda ekvipotentan skupu stabala parsiranja, pa ako jedan od tih skupova ima točno jedan element, onda i drugi skup ima točno jedan element — što dokazuje tvrdnju teorema. □

1.4 Algoritam CYK

Teorijsku značajnost zasigurno ima algoritam *CYK (Cocke–Younger–Kasami)*. To je jedan od prvih algoritama prepoznavanja i parsiranja. No, prije nego što prikažemo psuedokod algoritma, moramo definirati pojam *Chomskyjeve normalne forme (ChNF)* jer algoritam radi samo s onim BKG koje su u ChNF (moguće je proširiti algoritam tako da radi i s nekim gramatikama koje nisu u ChNF, ali time se nećemo baviti u ovom radu).

Definicija 13. Za gramatiku $\mathcal{G} = (V, \Sigma, \to, S)$ kažemo da je u *Chomskyjevoj normalnoj formi (ChNF)*, ako je svako njeno pravilo jednog od dva oblika:

- $A \to BC$ (*širenje*), gdje je $\{A, B, C\} \subseteq V$ i $S \notin \{B, C\}$;
- $A \to \alpha$ (*čitanje*), gdje su $A \in V$ i $\alpha \in \Sigma$. ◁

Očito su pravila širenja i čitanja beskontekstna, pa su sve gramatike u ChNF beskontekstne. Neka je $\mathcal{G} = (V, \Sigma, \to, S)$ BKG. Opišimo algoritam koji koristimo za pretvorbu gramatike $\mathcal{G}$ u gramatiku $\mathcal{G}'$ u ChNF za koju vrijedi $L(\mathcal{G}') = L(\mathcal{G}) \setminus \varepsilon$. Opisani algoritam je preuzet iz [7] i sastoji se od 5 faza:

START

Ako se početna varijabla S pojavljuje na desnoj strani nekog pravila, dodamo novu varijablu S', dodamo novo pravilo $S' \to S$, te proglasimo S' novom početnom varijablom.

TERM

Svakom znaku $\alpha \in \Sigma$ koji se pojavljuje na desnoj strani nekog pravila, ali ne sam, dodijelimo novu varijablu N_α, te u gramatiku dodamo pravilo čitanja $N_\alpha \to \alpha$. Zatim sve takve (desne, nesamostalne) pojave od α zamijenimo s N_α.

BIN

Za svako pravilo čija desna strana ima $n \geq 3$ simbola (nakon faze TERM oni moraju biti varijable), oblika $A \to V_1 V_2 \cdots V_n$, dodamo nove varijable $A_1, A_2, \ldots, A_{n-2}$, i to pravilo zamijenimo s $n - 1$ „ulančanih" pravila širenja

$$A \to V_1 A_1,\ A_1 \to V_2 A_2,\ A_2 \to V_3 A_3,\ \ldots,\ A_{n-3} \to V_{n-2} A_{n-2},\ A_{n-2} \to V_{n-1} V_n.$$

DEL

Prvo nađimo sve *nepozitivne varijable*:

$$np := \{A \in V \mid A \Rightarrow^* \varepsilon\}.$$

Opišimo ovaj postupak pseudokodom:

$np \coloneqq \emptyset$
dok np se promijenio u odnosu na prethodnu iteraciju **čini**
  za $(A, T_1 T_2 \cdots T_n) \in (\to)$ **čini**
    ako $\{T_1, T_2, \ldots, T_n\} \subseteq np$ **onda**
      $np \leftarrow np \cup \{A\}$

Sada svako pravilo u $\mathcal{G}$, na čijoj desnoj strani postoji $k > 0$ pojava varijabli iz np, zamijenimo s 2^k pravila: po jedno za svaki podskup skupa tih pojava, u kojem se svaka pojava iz tog podskupa briše. Kako smo prethodno proveli fazu BIN, jedino je moguće $k = 1$ ili $k = 2$, pa se time broj pravila poveća najviše s faktorom 4. Na kraju obrišemo sva pravila oblika $A \to \varepsilon$.

UNIT

Rješavamo se pravila oblika $A \to B$, koja na desnoj strani imaju samo jednu varijablu. Svako takvo pravilo maknemo, zapamtimo ga u nekom skupu, te za svako pravilo oblika $B \to u$ (gdje je $u \in Y^*$) dodamo pravilo $A \to u$, osim ako je ono već u skupu zapamćenih maknutih pravila.

Gramatika $\mathcal{G}'$ koju smo dobili pretvaranjem gramatike $\mathcal{G}$ je u ChNF. Primijetimo (može se dokazati) kako u ovom poretku izvršavanja, nijedna faza ne smeta prethodnima, tj. osigurava neko svojstvo ChNF ne kvareći već postignuta svojstva. Također se može vidjeti da sve faze čuvaju jezik generiran gramatikom, osim što faza DEL miče praznu riječ ε iz jezika (ako se nalazila u jeziku).

Opišimo sada algoritam CYK. Neka je $\mathcal{G} = (V, \Sigma, \to, S)$ gramatika u ChNF, te riječ $w = \alpha_0 \alpha_1 \cdots \alpha_{n-1} \in \Sigma^*$. Algoritam CYK parsira od dna prema vrhu koristeći dinamičko programiranje. To znači da algoritam CYK svodi problem „izvodi li S riječ w?" na manje probleme tipa: „izvodi li A podriječ $w_{i:j} \coloneqq \alpha_i \alpha_{i+1} \cdots \alpha_{j-1}$", skraćenog zapisa $A?w_{i:j}$, pri čemu je $A \in V$ i $0 \leq i < j \leq n \coloneqq |w|$. Takvih problema ima konačno mnogo, konkretno $\binom{n+1}{2} \cdot |V|$, i možemo ih rješavati redom, poredano rastuće po $|w_{i:j}| = j - i > 0$. Rješenja tih manjih problema zapisujemo u tablicu na mjestu (A, i, j). Radi jednostavnije implementacije, varijable poredamo u niz $S = V_0, V_1, \ldots, V_{r-1}$, pa tablicu prikazujemo kao trodimenzijsko polje N indeksirano brojevima koji počnu od 0, gdje intuitivnu poziciju (V_k, i, j) pišemo u programu kao $N[j - i - 1, i, k]$.

Prvi korak algoritma je rješavanje elementarnog problema $A?w_{i:i+1} = \alpha_i$. Gledamo postoji li u $\mathcal{G}$ pravilo čitanja $A \to \alpha_i$. U tablicu zapisujemo odgovor postoji li takvo pravilo.

Pretpostavimo da smo već riješili sve probleme oblika $V?w_{l:m}$ za $m - l < n$. Promotrimo problem $A?w_{i:j}$, gdje je $j - i = n$. Slučaj $n = 1$ smo riješili u prethodnom odlomku, pa pretpostavimo $n > 1$. Tada svako rješenje problema $A?w_{i:j}$ mora početi pravilom širenja $A \to BC$ za neke $B, C \in V$. Kako nijedna varijabla u $\mathcal{G}$ ne izvodi ε, znamo da riječi koje B

i C izvode moraju biti duljine strogo manje od n. To znači da postoji $k \in \{i+1, \ldots, j-1\}$ takav da $B \Rightarrow^* w_{i:k}$ i $C \Rightarrow^* w_{k:j}$. Primijetimo da smo probleme $B?w_{i:k}$ i $C?w_{k:j}$ već riješili (jer su $k-i$ i $j-k$ manji od n). Ako takvi izvodi postoje, zapišemo u tablicu trojku (B, C, k). Na kraju znamo da je riječ $w \in L(\mathcal{G})$ ako i samo ako problem $S?w_{0:n}$ ima rješenje, što jednostavno pogledamo u tablici.

Spremanjem svih tih informacija u tablicu omogućili smo konstrukciju čitavog stabla parsiranja za $w \in L(\mathcal{G})$. Opišimo algoritam rekonstrukcije stabla parsiranja. Pretpostavimo da vrijedi $w \in L(\mathcal{G})$, tj. na mjestu $(S, 0, n)$ tablice postoji bar jedna trojka (B, C, k). Promatramo u tablici mjesta $(B, 0, k)$ i (C, k, n) (ona moraju postojati zbog načina konstrukcije tablice) i rekurzivnim pozivom ih pretvorimo u stabla T_1 i T_2, te vratimo stablo (S, T_1, T_2). Kada dođemo do toga da u tablici gledamo rješenja od $A?w_{i:i+1}$, znamo da tamo piše *True*, i vratimo stablo $(A, (\alpha_i))$, kojem je korijen označen varijablom A i sadrži jedino list označen znakom α_i.

Algoritam 1 Algoritam CYK
Pseudokodom opisujemo samo rješenje problema prihvaćanja (odgovor *True* ili *False*), ne i rekonstrukciju stabla parsiranja kako je opisano u tekstu.

Neka je $\mathcal{G} = (\{V_0, V_1, \ldots, V_{r-1}\}, \Sigma, \to, V_0)$ gramatika u ChNF.
Neka je $w = \alpha_0\alpha_1 \cdots \alpha_{n-1}$ neprazna riječ nad Σ.
Inicijaliziraj tablicu N dimenzija $n \times n \times r$ čija polja sadrže *False*.

za $i = 0, \ldots, n-1$ **čini**
  za svako pravilo oblika $V_k \to \alpha_i$ **čini**
    $N[0, i, k] \leftarrow$*True*

za $l = 1, \ldots, n-1$ **čini**
  za $i = 0, \ldots, n-l-1$ **čini**
    za $j = 0, \ldots, l-1$ **čini**
      za svako pravilo oblika $V_a \to V_bV_c$ **čini**
        ako $N[j, i, b] =$ *True* **i** $N[l-j-1, i+j+1, c] =$ *True* **onda**
          $N[l, i, a] \leftarrow$ *True*

vrati $N[n-1, 0, 0]$.

Neformalno, kažemo da je vremenska složenost algoritma potrebno vrijeme za izvođenje algoritma u nekim osnovnim jedinicama (obično se koristi prosječno trajanje osnovnih instrukcija). Slično, kažemo da je prostorna složenost algoritma potrebna memorija za izvođenje algoritma. Često je teško izvesti točnu formulu za složenost nekog algoritma u najgorem slučaju jer bi trebalo riješiti sve instance problema kojeg algoritam rješava

te usporediti korišteno vrijeme i prostor tijekom rješavanja tih instanci. Iz tog razloga složenost algoritma izražavamo kao funkciju veličine ulaza problema kojeg treba riješiti (gledajući maksimum po svim ulazima iste veličine) i promatramo asimptotsko ponašanje te funkcije. Radi toga uvodimu *notaciju veliko O* (definicija je preuzeta iz [4]).

Definicija 14. Neka su $f, g : \mathbb{N} \to \mathbb{R}_+$ dvije funkcije. Kažemo da je funkcija g *asimptotska gornja međa* za funkciju f, ako postoji $c > 0$ i $n_0 \in \mathbb{N}$ tako da za svaki $n \geq n_0$ vrijedi $f(n) \leq c \cdot g(n)$. Oznaka: $f(n) = O(g(n))$. ◁

Pogledajmo algoritam CYK i odredimo njegovu vremensku i prostornu složenost za ulaznu riječ duljine n. Algoritam CYK ima tri ugniježđene ograničene petlje, tijelo svake od kojih se izvršava $O(n)$ puta. Doduše, imamo još jednu ugniježđenu petlju koja ide po pravilima gramatike. No, kako su broj pravila $|(\to)|$ i broj varijabli r konstantni (jer su određeni gramatikom), oni ne ovise o n, pa o njima ne ovise vremenska ni prostorna složenost algoritma (u smislu notacije veliko O, jer se radi o multiplikativnim konstantama). Zato imamo da je vremenska složenost algoritma CYK jednaka $O(n) \cdot O(n) \cdot O(n) \cdot O(|(\to)|)$, odnosno $O(n^3)$. Što se tiče memorije koja ovisi o n, algoritam koristi tablicu od $n \times n \times r$ elemenata, pa je prostorna složenost algoritma $O(r \cdot n^2) = O(n^2)$.

Earleyev parser

Pokazali smo da je algoritam CYK u najgorem slučaju vremenske složenosti $O(n^3)$ i prostorne složenosti $O(n^2)$. No, iako se CYK primjenjuje na gramatike koje su u ChNF, a sama pretvorba BKG u ChNF je također polinomne vremenske složenosti, on nije najpogodniji za sve BKG. Na primjer, ako želimo provjeriti je li prazna riječ ε izvediva, u tom slučaju ne pokrećemo algoritam, već prilikom pretvorbe $\mathcal{G}$ u ChNF moramo vidjeti je li početna varijabla nepozitivna i prema tome odrediti odgovor. Ali ipak, najveći nedostatak algoritma jest što dobiveno stablo parsiranja nije za zadanu BKG (osim ako ona nije već bila u ChNF), već za gramatiku u ChNF dobivenu iz te BKG. Pretvorba u ChNF, opisana na stranici 12, može dosta promijeniti početnu gramatiku uvođenjem novih pravila koja se ne pojavljuju u toj BKG. Zato je teško na kraju ustanoviti što u dobivenom stablu parsiranja predstavlja pravu informaciju o izvodu iz početne BKG, a što je posljedica njezinog pretvaranja u ChNF.

Zbog toga se u praksi koriste drugi algoritmi za parsiranje, a jedan od njih je *Earleyev parser*. Earleyev parser omogućuje parsiranje svih BKG, bez ikakvih restrikcija na (beskontekstna) pravila gramatike, u relativno brzom vremenu[2]. Earleyev parser konstruira listu *parcijalnih parsiranja* (rješenja manjih problema), slično kao kod CYK(algoritam 1),

[2]Ovdje koristimo frazu „relativno brzom vremenu" jer postoje specijalizirani parseri za specifične gramatike koji su znatno brži od Earleyevog parsera, no nisu primjenjivi na općenite BKG.

pomoću kojih se dolazi do krajnjeg rješenja — odbacujući pritom rješavanje manjih problema koji su nepotrebni za rješavanje prvobitnog problema. Izumio ga je Jay Earley, američki znanstvenik u područjima računarstva i psihologije, koji je 1968. godine objavio znanstveni rad u kojem je algoritam opisan. Objavio je još 12 znanstvenih radova u području računarstva, od kojih je Udruga za računske strojeve (*ACM*) upravo Earleyev parser proglasila jednim od 25 najboljih 25 znanstvenih radova zadnje četvrtine prošlog stoljeća [2]. Earley je kasnije doktorirao psihologiju i istražuje područje psihoterapije.

Poglavlje 2

Implementacija

Prije nego počinjemo govoriti o implementaciji moramo napomenuti par bitnih činjenica. Implementirani parser pretpostavlja da je leksička analiza već obavljena i da je ulaz niz tokena. Zato su ovdje svi tokeni predstavljeni jednim znakom pa je tako ulazni string duljine n shvaćen kao niz od n tokena, nad kojim se obavlja sintaksna analiza. U stvarnoj primjeni bi se svakako trebala provesti leksička analiza prije sintaksne.

Također, jednostavnosti radi, parser pri pokretanju programa obavlja sintaksnu analizu s obzirom na već unaprijed definiranu gramatiku. Za širu primjenu bi se trebalo implementirati čitanje iz dane datoteke, tako da se prilikom pokretanja programa naznači nad kojom gramatikom bi se trebala vršiti sintaksna analiza.

2.1 Podatkovna struktura

Gramatika

Definiramo klasu `Grammar`, čija instanca predstavlja BKG koju parser koristi prilikom parsiranja. Želimo da gramatika ima sve komponente iz definicije 4, te zato definiramo sljedeće varijable članice:

```
unordered_set<string> m_variables;
unordered_set<string> m_symbols;
string m_startVariable;
Rules m_rules;
```

pri čemu je `m_variables` skup varijabli, `m_symbols` skup znakova, `m_startVariable` početna varijabla i `m_rules` skup svih pravila te gramatike. Simbole gramatike prikazujemo pomoću stringova, a pravilo je par koji se sastoji od stringa (predstavlja varijablu s lijeve strane pravila) i vektora stringova (predstavlja simbole s desne strane pravila). Pra-

vila spremamo u strukturu tipa `multimap` koja koristi varijablu s lijeve strane pravila kao ključ za dohvaćanje objekta.

```
typedef multimap<string, vector<string>> Rules;
```

Uz funkcije `addRule`, `addVariable` i `addSymbol` koje služe za dodavanje sadržaja u gramatiku, koristimo još pomoćne funkcije `isStringVariable` i `isStringSymbol`[1], koje provjeravaju je li dani ulazni string varijabla odnosno znak.

```
bool Grammar::isStringVariable(string& string)
{
  return m_variables.find(string) != m_variables.end();
}

bool Grammar::isStringSymbol(string& string)
{
  return  m_symbols.find(string) != m_symbols.end();
}
```

Earleyeve stavke

Earleyeva stavka je glavni objekt koji se koristi prilikom parsiranja. Ona predstavlja neko parcijalno parsiranje. Slična je pravilu gramatike jer se na lijevoj strani stavke nalazi varijabla, a na desnoj niz simbola. No, na desnoj strani stavke se još nalaze točka i broj unutar zagrada. Točka nam daje informaciju o tome što je sve do sad parsirano — simboli koji se nalaze s lijeve strane točke su parsirani, a oni s desne strane točke još nisu. Kažemo da je Earleyeva stavka *završena*, ako je točka na kraju desne strane stavke. Broj unutar zagrada označuje na kojoj poziciji ulaznog stringa Earleyeva stavka započinje.

U kodu je definirana kao klasa `EarleyItem`. Ona sadrži sljedeće varijable članice:

```
int m_start = 0;
int m_parsedSymbols = 0;
string m_variable;
vector<string> m_symbols;
```

pri čemu `m_start` predstavlja početnu poziciju stavke, a `m_parsedSymbols` predstavlja poziciju točke. Dok, `m_variable`, odnosno `m_symbols`, predstavljaju varijablu s lijeve strane stavke i niz simbola s desne strane stavke redom (predstavljaju pravilo gramatike).

Klasa još sadrži pomoćne funkcije: `printItem`, koja ispisuje sadržaj stavke na konzoli, `operator==`, koji provjerava jesu li dvije stavke jednake (odnosno jesu li im sve varijable članice jednake), i funkciju `isCompleted` koja provjerava je li je stavka završena.

[1] U ovoj funkciji eng. `symbol` označuje znak i nije isto što i hrv. simbol, što može označavati i varijablu.

Skupovi stanja

Sljedeće strukture podataka su definirane u klasi `EarleyParser`, u kojoj se nalaze sve glavne funkcije algoritma. Ključne strukture podataka koje konstruiramo u prvoj fazi, a kasnije koristimo u drugoj fazi parsiranja su *skupovi stanja*. Skupovi stanja sadrže samo različite Earleyeve stavke. U svrhu implementacije Earleyevog algoritma, lakše ih je shvatiti kao *uređene* skupove (implementirane kao vektore) koje tijekom izvršavanja algoritma punimo novim Earleyevim stavkama.

Prilikom izvršavanja algoritma, čitajući po jedan znak ulaza, želimo saznati sve bitne informacije o pojedinom parcijalnom parsiranju, odbacujući sva nebitna parcijalna parsiranja. Zato Earleyev algoritam sadrži glavni niz, u kodu nazvan `m_state`. Niz `m_state` (također implementiran kao vektor) sadrži skupove stanja i on se na početku inicijalizira na $n + 1$ praznih skupova stanja, pri čemu je n duljina ulaznog stringa. `EarleyParser` sadrži sljedeće varijable članice:

```
vector<vector<EarleyItem>> m_state;
string m_input;
Grammar m_grammar;
```

pri čemu u `m_input` spremamo ulazni string, a u `m_grammar` BKG koju koristimo. Još imamo pomoćne funkcije `printState()` i `printStateSet(int i)` koje služe za ispis glavnog niza skupova stanja, te funkciju

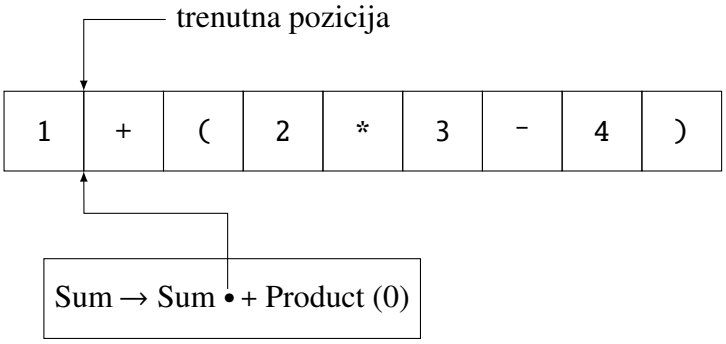
```
void EarleyParser::addEarleyItem(EarleyItem item, int stateSetIndex)
{
  auto& set = m_state[stateSetIndex];
  for (EarleyItem& setItem : set) {
    if (item == setItem) {
      return;
    }
  }
  set.emplace_back(item);
}
```

koju koristimo za ubacivanje Earleyevih stavki unutar naše strukture. Funkcija `addEarleyItem` prima kao argumente Earleyevu stavku `item` i indeks `stateSetIndex` skupa stanja u koji će se ta stavka pokušati ubaciti. U algoritmu se funkcija jedino poziva, ili s indeksom zadnjeg skupa stanja iz `m_state`, ili sa sljedećim indeksom nakon njega. Provjeravamo sadrži li skup sa zadanim indeksom stavku `item` i dodajemo je u skup ako ju ne sadrži (jer želimo osigurati jedinstvenost Earleyevih stavki unutar skupa stanja).

Primjer 15. Neka je gramatika $\mathcal{G}$ dana pravilima

$$\begin{aligned} \text{Sum} &\rightarrow \text{Sum} + \text{Product} \mid \text{Sum} - \text{Product} \mid \text{Product} \\ \text{Product} &\rightarrow \text{Product} * \text{Factor} \mid \text{Product} / \text{Factor} \mid \text{Factor} \\ \text{Factor} &\rightarrow \texttt{(}\ \text{Sum}\ \texttt{)} \mid \text{Number} \\ \text{Number} &\rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \end{aligned}$$

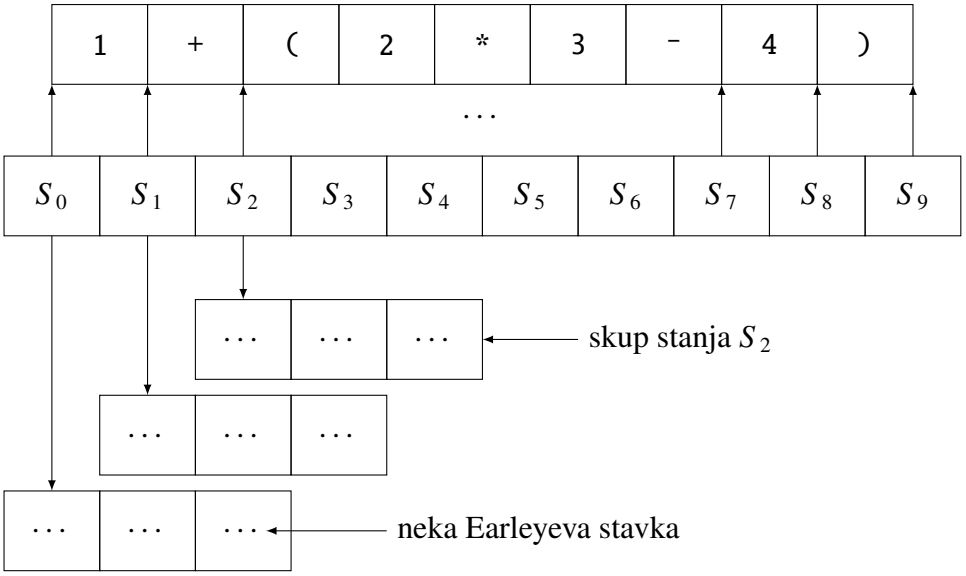
pri čemu je $V := \{\text{Sum}, \text{Product}, \text{Factor}, \text{Number}\}$ skup varijabli V, a $\Sigma := \{+, -, *, T/,), (, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ abeceda. Beskontekstna gramatika $\mathcal{G}$ generira beskontekstan jezik elementarne aritmetike jednoznamenkastih brojeva i koristit ćemo je u svim narednim primjerima kako bismo lakše opisali korake algoritma. Ona je također korištena za testiranje prilikom implementacije algoritma. Nadalje, u svim primjerima ćemo koristiti ulazni string $w_0 := \texttt{1+(2*3-4)}$ za koji ćemo na kraju konstruirati stablo parsiranja.



Pogledajmo primjer Earleyeve stavke Sum → Sum • + Product (0). Broj 0 između zagrada predstavlja početnu poziciju u ulaznom stringu gdje stavka započinje, tj. ova stavka započinje od početka ulaznog stringa. Podebljana točka nakon tokena Sum nam daje informaciju da smo negdje uspjeli završiti parcijalno parsiranje tog tokena — negdje između početka 0 i kraja Earleyeve stavke. Drugim riječima, uspjeli smo naći stavke koje će kasnije činiti stablo parsiranja za neku podriječ ulazne riječi između 0 i trenutne pozicije na ulaznom stringu. No, kao što vidimo, informacija trenutne pozicije u ulaznom stringu nije eksplicitno zapisana u stavci. Ona je implicitno zapisana u indeksu skupa stanja u kojem se ta stavka nalazi[2]. Tako znamo da se prikazana Earleyeva stavka sigurno nalazi u skupu stanja na indeksu 1.

Nakon što pročitamo neki token, algoritam prikuplja sve moguće informacije o parcijalnim parsiranjima koja imaju smisla, i tako puni skupove stanja. To znači da je glavni niz S konstruiran korak po korak, s jednim elementom za svaki pročitani token (plus početni skup stanja). Ovako skupovi stanja izgledaju nakon što smo pročitali dva tokena:

[2] Ako smo našli neku stavku unutar nekog skupa stanja, onda sigurno znamo njen indeks.



Strelice prema ulaznom stringu naznačuju poziciju svakog skupa stanja niza S. Ubacivanjem barem jedne stavke u skup stanja S_2 signaliziramo da smo pročitali drugi znak ulaznog stringa. ◁

2.2 Konstrukcija skupova stanja

Na početku izvršavanja izvornog koda, u funkciji `main`, tražimo unos ulazne riječi (koju parsiramo). Zatim instanciramo objekt klase `EarleyParser` pomoću njenog konstruktora

```
EarleyParser::EarleyParser(Grammar grammar, string input)
  : m_grammar(grammar), m_input(input)
{
  for (int i = 0; i <= m_input.size(); ++i) {
    m_state.emplace_back(vector<EarleyItem>());
  }
}
```

koji kao argumente prima gramatiku (s obzirom na koju parsiramo) i ulaznu riječ. Prilikom poziva konstruktora inicijalizira se glavni niz stanja tako što se postavlja $n + 1$ praznih vektora u `m_state`, gdje je n duljina ulazne riječi. Zatim pozivamo funkciju `parse`. U njoj se nalaze pozivi svih glavnih funkcija konstrukcije skupova stanja i konstrukcije stabla parsiranja, tj. glavni dio koda.

```
void EarleyParser::parse()
{
  findNullableVariables();
  buildItems();
  printState(m_state);
```

```
  finishBuilding();
  removeIncompleteItems();
  auto orderedState = orderStateByStart(m_state);
  printState(orderedState);
  auto tree = createTree(orderedState, m_input,
    0, m_grammar.getStartVariable(), nullptr);
  printTree(tree, "", true);
  cin.get();
}
```

Opišimo sad glavnu funkciju konstrukcije skupova stanja, `buildItems`. Ona se sastoji od dva dijela. Prvi dio (linije 3–8) predstavlja inicijalizaciju skupa stanja `m_state[0]`. Prolazimo kroz sva pravila gramatike te za svako pravilo koje s lijeve strane ima početnu varijablu stvaramo odgovarajuću Earleyevu stavku i ubacujemo je u početni skup stanja.

```
1  void EarleyParser::buildItems()
2  {
3    string startVariable = m_grammar.getStartVariable();
4    for (auto& rule : m_grammar.getRules()) {
5      if (rule.first == startVariable) {
6        addEarleyItem(EarleyItem(startVariable, rule.second, 0, 0), 0);
7      }
8    }
9
10   for (int i = 0; i < m_state.size(); ++i) {
11     auto& tempStateSet = m_state[i];
12     for (int j = 0; j < tempStateSet.size(); ++j) {
13       int dotIndex = tempStateSet[j].getParsedSymbols();
14       auto tempSymbols = tempStateSet[j].getSymbols();
15
16       if (tempStateSet[j].isCompleted()) {
17         complete(tempStateSet, i, j);
18       }
19       else {
20         string nextSymbol = tempSymbols[dotIndex];
21         if (m_grammar.isStringVariable(nextSymbol)) {
22           predict(tempStateSet, i, j, nextSymbol);
23         }
24         else if (m_grammar.isStringSymbol(nextSymbol)) {
25           scan(tempStateSet, i, j, nextSymbol, m_input);
26         }
27         else {
28           cerr << "Illegal rule!" << endl;
29           exit(1);
30         }
31       }
32     }
33   }
34 }
```

Glavna petlja

Nakon inicijalizacije podataka, možemo krenuti s glavnom petljom unutar funkcije `buildItems` (linije 10–33). Glavna petlja se sastoji od vanjske petlje, koja prolazi kroz glavni niz skupova stanja; i unutarnje petlje, koja prolazi kroz sve stavke danog skupa stanja.

Unutarnja petlja, prolazeći kroz skup stanja, dodaje nove stavke u trenutni ili sljedeći skup stanja. Ona se izvršava sve dok ne prođe kroz sve stavke danog skupa stanja. Uočimo da se broj stavki unutar skupa stanja može povećati i iz tog razloga su skupovi stanja promatrani kao dinamički nizovi.

Vanjska petlja prolazi kroz niz skupova stanja tako što prvo izvršava unutarnju petlju nad skupom stanja `m_state[0]`. Prolazeći kroz taj skup stanja, generiramo nove stavke. Kada završi unutarnja petlja nad `m_state[0]`, prelazimo na unutarnju petlju nad sljedećim skupom stanja `m_state[1]` itd., sve dok unutarnja petlja uspijeva dodati nove stavke u sljedeći skup stanja (jer smo došli do kraja ulaznog stringa ili smo zaključili da ulazni string „nema smisla").

Ovisno o stavci koju trenutno promatramo i poziciji njezine točke, dodavanje nove stavke možemo učiniti na sljedeća tri načina:

- *Predviđanje* — simbol s desne strane točke je varijabla iz `m_variables`, poziva se funkcija `predict`;
- *Skeniranje* — simbol s desne strane točke je znak iz `m_symbols`, poziva se funkcija `scan`;
- *Završetak* — točka je na kraju stavke, poziva se funkcija `complete`.

Svaka od tih triju funkcija kao argumente prima skup stanja `stateSet`, indeks glavne petlje `stateSetIndex` te indeks trenutno promatrane stavke `desiredStateSetIndex`. Predviđanje i skeniranje još primaju kao argument `symbol` — simbol s desne strane točke promatrane stavke, dok skeniranje još prima ulazni string.

Objasnimo proces dodavanja nove stavke (linije 12–32). Najprije u lokalne varijable spremamo podatke trenutno promatrane stavke (linije 13–14). Zatim provjeravamo je li stavka završena — točka se nalazi na kraju desne strane (linije 16–18). Ako je odgovor potvrdan, poziva se `complete`. Ako je odgovor niječan, dohvaćamo sljedeći simbol (linija 20) i provjeravamo je li on varijabla ili znak. Ako je varijabla, pozivamo funkciju `predict`, a u slučaju da je znak, pozivamo funkciju `scan`. Zadnji slučaj nam prijavljuje grešku prilikom izvođenja programa, koja znači da je nešto pošlo po krivu u inicijalizaciji stavke ili gramatike.

Predviđanje

Argument `symbol` je varijabla koja se nalazi s desne strane točke unutar promatrane stavke. Predviđanjem pokušavamo prebaciti točku preko varijable `symbol`, tako da dodamo nove stavke koje sadrže pravilo, kojem je ta varijabla s lijeve strane, kako bismo s vremenom došli do završetka trenutno promatrane stavke. Prolazimo kroz sva pravila gramatike te za svako pravilo koje sadrži `symbol` na lijevoj strani, pokušamo dodati novu stavku s istim pravilom u trenutni skup stanja, pri čemu ona ima točku na poziciji 0 i početnu poziciju jednaku indeksu skupa stanja. Kako smo prije spomenuli, funkcija `addEarleyItem` ubacuje samo već nepostojeće stavke u trenutni skup stanja, izbjegavajući tako moguću beskonačnu petlju predviđanja uvijek iste stavke.

```
1  void EarleyParser::predict(std::vector<EarleyItem>& stateSet,
2    int stateSetIndex, int desiredStateSetIndex, string& symbol)
3  {
4    for (auto& rule : m_grammar.getRules()) {
5      if (symbol == rule.first) {
6        addEarleyItem(EarleyItem(symbol, rule.second, stateSetIndex, 0),
7          stateSetIndex);
8      }
9    }
10 }
```

Skeniranje

Argument `symbol` je znak koji se nalazi s desne strane točke unutar promatrane stavke. Provjeravamo odgovara li `symbol` trenutno promatranom znaku `inputSymbol` (linije 10–12), gdje je `inputSymbol` znak ulaznog stringa koji se nalazi na poziciji jednakoj indeksu trenutnog skupa stanja (linija 9). Ako su oni jednaki, dodaje se stavka jednaka promatranoj s točkom pomaknutom udesno za jednu poziciju (linije 14–18).

```
1  void EarleyParser::scan(vector<EarleyItem>& stateSet, int stateSetIndex,
2    int desiredStateSetIndex, string& symbol, string& input)
3  {
4    if (stateSetIndex >= input.length()) {
5      return;
6    }
7    else {
8      string inputSymbol;
9      inputSymbol = input.at(stateSetIndex);
10     if (symbol != inputSymbol) {
11       return;
12     }
13
14     EarleyItem tempItem = stateSet[desiredStateSetIndex];
```

```
15      EarleyItem item = EarleyItem(tempItem.getVariable(),
16        tempItem.getSymbols(), tempItem.getStart(),
17        tempItem.getParsedSymbols() + 1);
18      addEarleyItem(item, stateSetIndex + 1);
19    }
20  }
```

Završetak

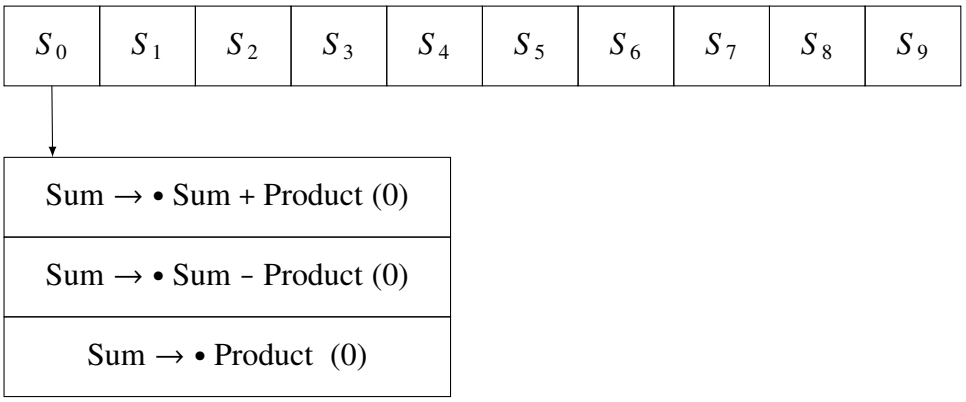
Funkcija `complete` se poziva ako promatrana stavka sadrži točku na kraju desne strane pravila, predstavljajući uspješno parcijalno parsiranje varijable s lijeve strane pravila. Earleyev algoritam garantira da to parcijalno parsiranje jest dio nekog „većeg" parsiranja, jer smo prošli kroz cijeli niz predviđanja, skeniranja i završetaka kako bismo došli do promatrane stavke. Došli smo i do svrhe postojanja početne pozicije u stavci — pomoću nje gledamo samo bitne skupove stanja, kako bismo našli stavku koja je pokrenula prvobitno predviđanje kojim smo došli do uspješnog parcijalnog parsiranja.

U kodu dohvaćamo željeni skup stanja `tempStateSet` i varijablu `desiredSymbol` pomoću početne pozicije promatrane stavke `tempItem` (linije 4–5). Zatim prolazimo kroz sve stavke koje se nalaze u `tempStateSet` i tražimo one koje imaju `desiredSymbol` s desne strane točke. Sve takve stavljamo u trenutni skup stanja (skup stanja s indeksom `stateSetIndex`) s točkom pomaknutom udesno za jednu poziciju (linije 16–20).

```
1  void EarleyParser::complete(vector<EarleyItem>& stateSet,
2    int stateSetIndex, int desiredStateSetIndex)
3  {
4    EarleyItem tempItem = stateSet[desiredStateSetIndex];
5    auto& tempStateSet = m_state[tempItem.getStart()];
6
7    string desiredSymbol = tempItem.getVariable();
8    for (auto& item : tempStateSet) {
9      auto& symbols = item.getSymbols();
10     int itemParsedSymbols = item.getParsedSymbols();
11     if (symbols.size() <= itemParsedSymbols) {
12       continue;
13     }
14     string checkingSymbol = symbols[itemParsedSymbols];
15
16     if (desiredSymbol == checkingSymbol) {
17       EarleyItem newEarleyItem = EarleyItem(item.getVariable(),
18        item.getSymbols(), item.getStart(), item.getParsedSymbols() + 1);
19       addEarleyItem(newEarleyItem, stateSetIndex);
20     }
21   }
22 }
```

Primjer 16. Prilikom inicijalizacije glavnog niza, stvara se skup stanja S_0 tako što u njega stavljamo Earleyeve stavke za svako pravilo koje ima početnu varijablu Sum s lijeve strane. Pa tako, nakon inicijalizacije, skup stanja S_0 sadrži stavke:

$$\text{Sum} \rightarrow \bullet\,\text{Sum+ Product (0)}$$
$$\text{Sum} \rightarrow \bullet\,\text{Sum- Product (0)}$$
$$\text{Sum} \rightarrow \bullet\,\text{Product (0)}$$



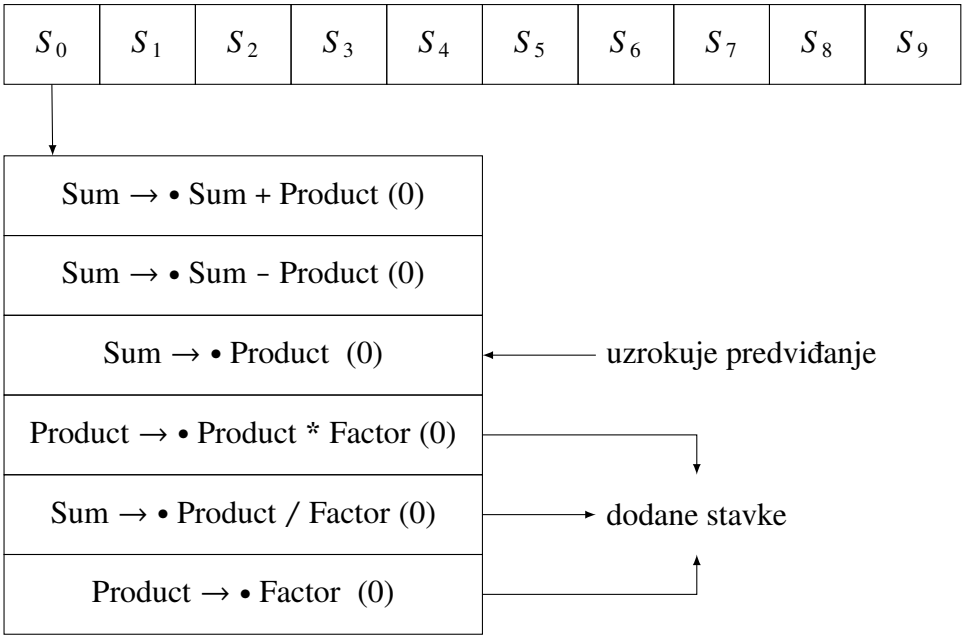
Pogledajmo što se događa prilikom jednog od predviđanja kojeg uzrokuju stavke u S_0. Za stavku:

$$\text{Sum} \rightarrow \bullet\,\text{Sum + Product (0)},$$

imamo varijablu `Sum` s desne strane točke. Stoga, pokušamo predvidjeti i ubaciti u S_0 nove Earleyeve stavke koje imaju `Sum` s lijeve strane pravila. No, kako su sve takve stavke već u S_0, ne činimo ništa. Isto se događa i sa sljedećom stavkom. Zatim gledamo zadnju stavku u početnom skupu stanja:

$$\text{Sum} \rightarrow \bullet\,\text{Product (0)},$$

i pokušamo ubaciti u S_0 sve stavke koje sadrže Product s lijeve strane pravila. Kako takve stavke ne postoje u S_0, dodajemo ih, pa je početni skup stanja oblika:



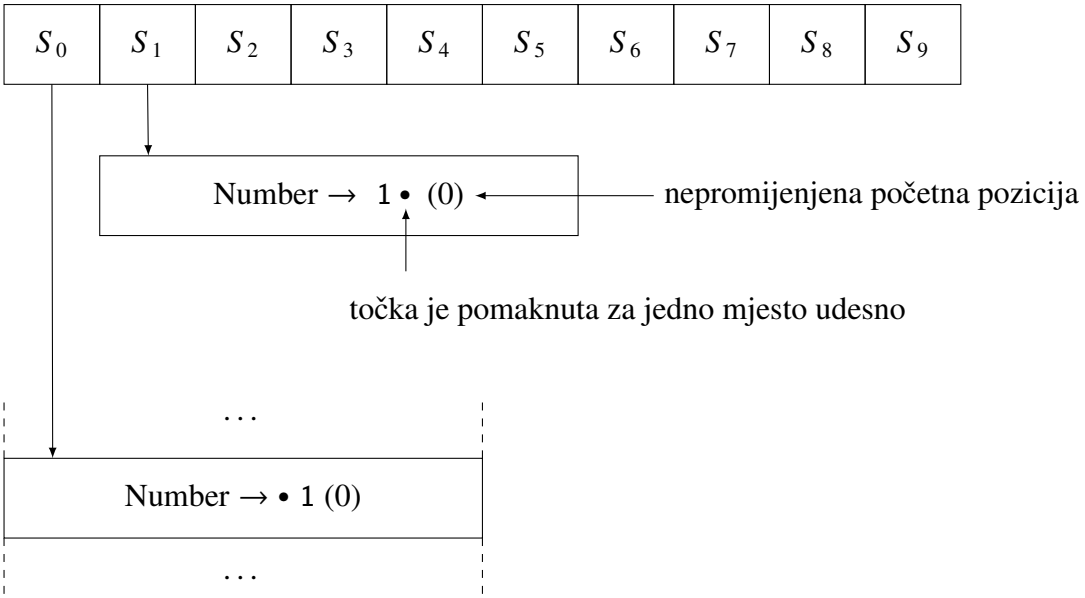
Nastavljamo sa prolaskom kroz početni skup stanja i predviđanjem stavki dok ne dođemo do stavke oblika:

$$\text{Factor} \rightarrow \bullet \text{ (Sum) } (0).$$

Kako se s desne strane točke nalazi znak (, skeniramo ga, tj. provjerimo nalazi li se on na početnoj poziciji ulazne riječi. Vidimo da ne odgovara prvom znaku, pa nastavljamo s prolaskom kroz skup stanja S_0. U jednom trenutku dolazimo do stavke:

$$\text{Number} \rightarrow \bullet \text{ 1 } (0),$$

koju također skeniramo. Kako se znak 1 nalazi na prvoj poziciji ulazne riječi, dodajemo novu stavku s istim pravilom u skup stanja S_1, s točkom pomaknutom za jednu poziciju udesno, te istom početnom pozicijom.



Nakon dovršenog prolaska kroz skup stanja S_0, započinjemo s prolaskom kroz skup stanja S_1 (jer on u sebi sadrži nove stavke koje smo dodali). U jednom ćemo trenutku doći do stavke:

$$\text{Number} \rightarrow 1 \bullet \ (0),$$

koja signalizira uspješno parcijalno parsiranje jer se točka nalazi na kraju desne strane pravila. Stavka nam pokazuje da od njene početne pozicije 0 i njene krajnje pozicije 1, podriječ (ulazne riječi) 1 je `Number`. Kako smo ranije rekli, algoritam garantira da to parcijalno parsiranje jest dio nekog „većeg" parsiranja. Na primjer, pretpostavimo da smo došli do nekog pravila oblika:

$$\text{Foo} \rightarrow \text{a b c} \bullet \ (i),$$

pri čemu su a, b i c simboli, a i je početak stavke. Postojanje te stavke znači da negdje također postoje stavke:

$$\text{Foo} \rightarrow \text{a b} \bullet \text{c} \ (i)$$
$$\text{Foo} \rightarrow \text{a} \bullet \text{b c} \ (i)$$
$$\text{Foo} \rightarrow \bullet \text{a b c} \ (i),$$

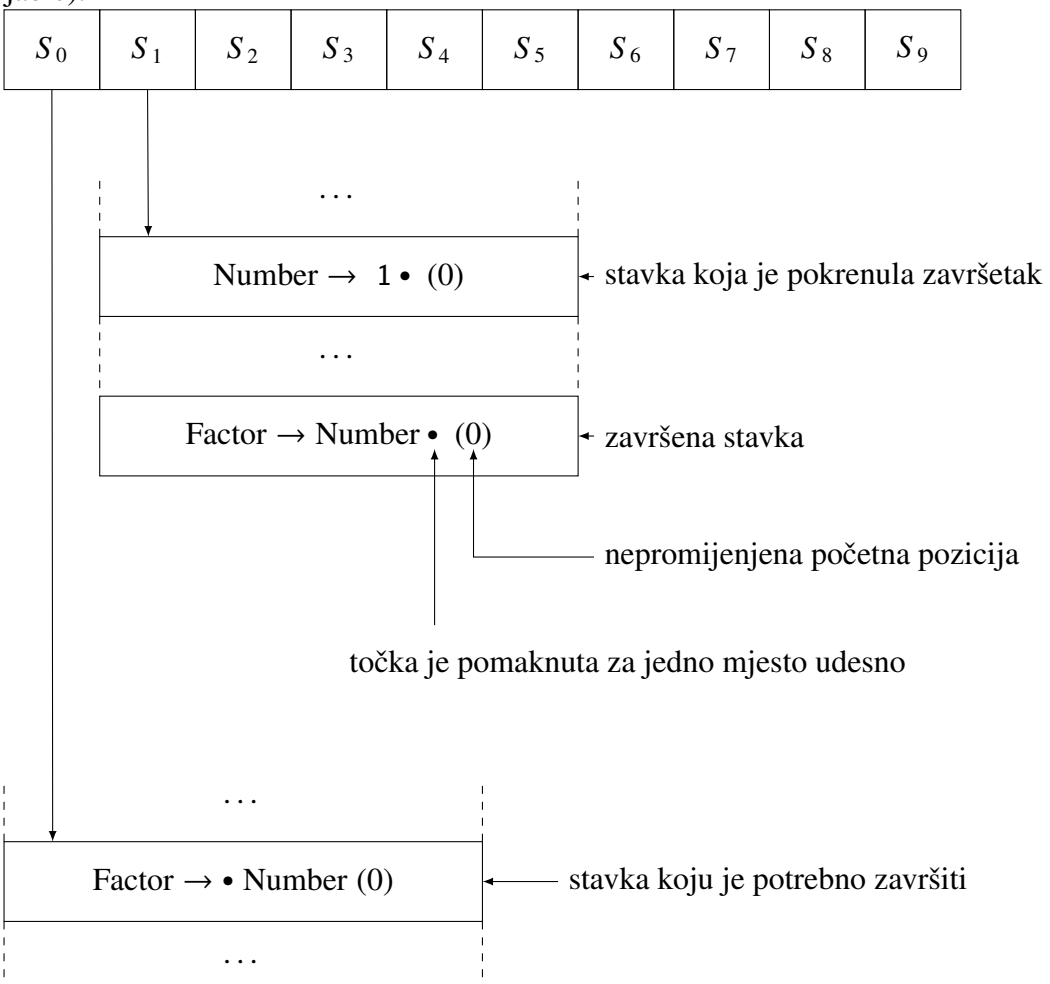
pri čemu se zadnja nalazi u S_i, skupu stanja u kojem predviđena. No, nas zapravo zanimaju stavke koje su „možda"[3] pokrenule izvorno predviđanje. Zanimaju nas stavke oblika:

$$\text{Bar} \rightarrow \text{a b} \bullet \text{Foo c d} \ (j),$$

[3]Koristimo izraz možda jer znamo da je samo jedna stavka pokrenula izvorno predviđanje, ali nas zanimaju sve stavke koje su ga mogle pokrenuti.

tj. stavke u S_i koje sadrže Foo na poziciji zdesna točki. One su „veće" parcijalno parsiranje kojeg je parsiranje od Foo dio. Početne pozicije stavki nam služe upravo kako bismo našli skupove stanja koji sadrže te stavke.

U našem primjeru, za stavku Number → 1 • (0) tražimo sve stavke u skupu stanja S_0 koje sadrže Number na desnoj strani pravila, na poziciji zdesna točki. Takva stavka jest oblika Factor → • Number (0). Sada pomičemo točku preko Number i dodajemo takvu stavku u skup stanja S_1 (postupak je sličan skeniranju, ali ovdje pomičemo točku preko varijable).



◁

Kraj glavne petlje

Kada prolazak kroz glavnu petlju završi, funkcijom `finishBuilding` provjeravamo je li konstrukcija skupova stanja uspješna. Prvi korak jest provjeriti broj skupova stanja — in-

deks zadnjeg skupa stanja mora biti jednak duljini ulazne riječi. Nakon toga prolazimo kroz zadnji skup stanja unutar vektora `m_state`, gledamo njegove Earleyeve stavke i tražimo stavku koja:

- sadrži točku na kraju desne strane pravila;
- njen početak `m_start` je jednak 0;
- varijabla s lijeve strane pravila je jednaka početnoj varijabli zadane gramatike.

Konstrukcija skupova stanja je uspješna ako je nađena takva stavka — inače se prijavljuje greška s odgovarajućom porukom.

Primjer 17. Glavna petlja će u jednom trenutku završiti za ulaznu riječ w_0 i tada moramo provjeriti jesu li skupovi stanja uspješno konstruirani, tj. je li parsiranje riječi w_0 uspješno. Da bi parsiranje bilo uspješno, zadnji skup stanja S_9 mora sadržavati završenu stavku koja ima početak u 0 i početnu varijablu `Sum` s lijeve strane pravila. Na kraju algoritma, skup stanja S_9 sadrži stavke:

$$\begin{aligned}
\text{Factor} &\to \texttt{(}\ \text{Sum}\ \texttt{)}\ \bullet \quad (2)\\
\text{Product} &\to \text{Factor}\ \bullet \quad (2)\\
\text{Sum} &\to \text{Sum} + \text{Product}\ \bullet \quad (0)\\
\text{Product} &\to \text{Product}\ \bullet\ \texttt{*}\ \text{Factor} \quad (2)\\
\text{Sum} &\to \text{Sum} + \bullet\ \text{Product} \quad (0).
\end{aligned}$$

Kao što vidimo, on sadrži upravo prije opisanu stavku Sum $\to$ Sum + Product • (0), iz čega zaključujemo da smo uspješno parsirali riječ w_0.

Do neuspjeha parsiranja može doći na dva načina:

1. cijeli ulaz je neuspješno parsiran — ulazna riječ sadrži nevaljan znak i parsiranje u jednom trenutku stane. Na primjer, algoritam bi za ulaz `1+#2` prošao kroz prva dva znaka i stao nakon trećeg;
2. algoritam je prošao kroz svaki znak ulazne riječi, ali ulazna riječ „nema smisla” — na primjer, neka je ulazna riječ `1+`. Iako ta riječ jest valjan početak pravila `Sum`, nedostaje drugi operand, stoga parsiranje nije uspjelo.

Kao što vidimo, neuspješno parsiranje ulazne riječi može sadržavati neka uspješna parcijalna parsiranja, iako na kraju zaključujemo da gramatika ne izvodi ulaznu riječ. ◁

Nepozitivne varijable

Iako je implementacija konstrukcije skupova stanja gotova, pogledajmo primjer gramatike koji nam i dalje zadaje probleme prilikom izvršavanja algoritma.

Primjer 18. Neka je gramatika $\mathcal{G}_\varepsilon := (\{A, B\}, \{\mathtt{c}\}, \to, A)$ dana s pravilima:

$$A \to \varepsilon \mid B$$
$$B \to A.$$

Očito je da $\mathcal{G}_\varepsilon$ izvodi samo praznu riječ ε. Štoviše, postoji beskonačno mnogo izvoda za ε:

- $A \Rightarrow \varepsilon$;
- $A \Rightarrow B \Rightarrow A \Rightarrow \varepsilon$;
- $A \Rightarrow B \Rightarrow A \Rightarrow B \Rightarrow A \Rightarrow \varepsilon$;
- $\cdots$

Trenutnom implementacijom prve faze parsera dobijemo samo jedan skup stanja koji sadrži sljedeće Earleyeve stavke:

$$A \to \bullet\ (0)$$
$$A \to \bullet\ B\ (0)$$
$$B \to \bullet\ A\ (0).$$ ◁

Primijetimo da je stavka $A \to \bullet\ (0)$ stavljena u skup stanja na prvo mjesto prilikom prvog koraka konstrukcije skupova stanja. Stoga, kada pokrećemo glavnu petlju, ona poziva završetak i traži stavku oblika $B \to \bullet\ A\ (0)$, da kasnije prebaci točku preko varijable A. No, u tom trenu se takva stavka ne nalazi u trenutnom skupu stanja, već će se ona naknadno dodati kad se pozove predviđanje sa stavkom $A \to \bullet\ B\ (0)$. Time dobivamo samo jedan skup stanja i kasnije samo jedno stablo parsiranja (iako ih ima beskonačno mnogo).

Upravo opisan problem se rješava dodavanjem *magičnog završetka predviđanja* u funkciju predviđanja za promatranu nepozitivnu varijablu (prisjetimo se definicije nepozitivnih varijabli u sklopu faze DEL na stranici 12). Ako prilikom predviđanja promatramo nepozitivnu varijablu, magičnim završetkom predviđanja dodajemo još jednu Earleyevu stavku u skup stanja pomicanjem točke preko te varijable. Tako je sad funkcija predviđanja sljedećeg oblika:

```
1 void EarleyParser::predict(vector<EarleyItem>& stateSet,
2   int stateSetIndex, int desiredStateSetIndex, string& symbol)
3 {
4   for (auto& rule : m_grammar.getRules()) {
```

```
5     if (symbol == rule.first) {
6       addEarleyItem(EarleyItem(symbol, rule.second, stateSetIndex, 0),
7         stateSetIndex);
8       if (isVariableNullable(symbol)) {
9         addEarleyItem(EarleyItem(symbol, rule.second, stateSetIndex,
10          stateSet[desiredStateSetIndex].getParsedSymbols() + 1),
11          stateSetIndex);
12      }
13    }
14  }
```

Pritom funkcija `isVariableNullable` provjerava nalazi li se varijabla u skupu svih nepozitivnih varijabli `m_nullableVariables`.

No, to znači da prije nego pokrenemo konstrukciju skupova stanja, moramo prvo pronaći sve nepozitivne varijable. Njih tražimo funkcijom `findNullableVariables` i spremamo ih u `m_nullableVariables`. Implementacija funkcije je napravljena prema pseudokodu iz [3] jer je manje vremenske složenosti, u ovisnosti o broju pravila iz zadane gramatike, nego implementacija iz [6].

```
1  void EarleyParser::findNullableVariables()
2  {
3    map<string, Rules> variableRulesRHS;
4    for (auto& variable : m_grammar.getVariables()) {
5      Rules rules = m_grammar.getRulesWithVariableInRHS(variable);
6      variableRulesRHS.emplace(variable, rules);
7    }
8
9    for (auto& rule : m_grammar.getEmptyRules()) {
10     m_nullableVariables.emplace(rule.first);
11   }
12
13   queue<string> workQueue;
14   for (auto variable : m_nullableVariables) {
15     workQueue.push(variable);
16   }
17
18   while (workQueue.size() > 0) {
19     string workVariable = workQueue.front();
20     workQueue.pop();
21
22     for (auto& rule : variableRulesRHS[workVariable]) {
23       if (isVariableNullable(rule.first)) {
24         continue;
25       }
26
27       bool isRuleNullable = true;
28       for (auto& symbol : rule.second) {
```

```
29          if (!isVariableNullable(symbol)) {
30            isRuleNullable = false;
31            break;
32          }
33        }
34        if (isRuleNullable) {
35          m_nullableVariables.emplace(rule.first);
36          workQueue.push(rule.first);
37        }
38      }
39    }
40  }
```

Najprije stvaramo preslikavanje koje svakoj varijabli pridružuje skup pravila u kojima se ta varijabla pojavljuje s desne strane (linije 3–7). U tu svrhu smo definirali funkciju `getRulesWithVariableInRHS` unutar klase `Grammar`, koja dohvaća sva takva pravila. Zatim dodajemo sve varijable koje se nalaze s lijeve strane nekog pravila oblika $A \to \varepsilon$ u skup `m_nullableVariables` (linije 9–11), pri čemu funkcija `getEmptyRules`, iz klase `Grammar`, dohvaća sva takva pravila.

Za glavni dio funkcije koristimo red `workQueue` kako bismo pratili za koje varijable moramo još izvršiti provjeru jesu li nepozitivne. Na početku stavimo sve ranije dobivene nepozitivne varijable u red i vrtimo petlju `while` sve dok red `workQueue` nije prazan. U petlji, za trenutnu varijablu `workVariable`, prolazimo kroz sva pravila u kojima se ona pojavljuje s desne strane. Ako je varijabla s lijeve strane pravila već označena kao nepozitivna, prelazimo na iduće pravilo (linije 22–25). Inače, prolazimo kroz sve simbole s desne strane trenutnog pravila (linije 27–33) i prelazimo na iduće pravilo ako se barem jedan od simbola ne nalazi u skupu nepozitivnih varijabli. Na kraju, ako je provjera uspjela, dodajemo trenutnu varijablu u red i u skup nepozitivnih varijabli.

2.3 Konstrukcija stabla parsiranja

Najprije ćemo reći nešto o ideji konstrukcije stabla parsiranja, a zatim ćemo objasniti njenu implementaciju. Iako je Earley prilikom stvaranja svog parsera našao metodu kako konstruirati stablo parsiranja usporedno s konstrukcijom skupova stanja uz pomoć pokazivača [1], ispostavilo se da je završni niz skupova stanja sasvim dovoljan da se rekonstruira stablo parsiranja. Štoviše, dovoljan je da se rekonstruiraju sva moguća stabla parsiranja za neki ulaz.

Neka je ulazna riječ w duljine n. Kao što smo ranije rekli, konstrukcija skupova stanja za riječ w završava ako imamo Earleyevu stavku oblika $S \to A_1 \cdots A_k \bullet$ (0) unutar skupa stanja koji se nalazi na indeksu n, pri čemu su A_i simboli (za svaki $i \in \{1, \ldots, k\}$) i S je početna varijabla gramatike. Točka na kraju stavke označuje da je stavka završena,

započinje na poziciji 0 i završava na poziciji n (nalazi se u n-tom skupu stanja). Kao što smo vidjeli u primjeru 16, postoji niz Earleyevih stavki kojim smo došli do završetka neke završene stavke. Tako u nekim skupovima stanja još postoje stavke sljedećih oblika:

$$\begin{aligned} S &\to A_1 A_2 \cdots A_{k-1} \bullet A_k \ (0) \\ S &\to A_1 A_2 \cdots \bullet A_{k-1} A_k \ (0) \\ S &\to A_1 \bullet A_2 \cdots A_{k-1} A_k \ (0) \\ S &\to \bullet A_1 A_2 \cdots A_{k-1} A_k \ (0). \end{aligned}$$

No, da bismo prebacili točku preko nekog simbola na desnoj strani Earleyeve stavke, moramo imati prvobitni oblik stavke, te jedno od sljedećeg: završenu drugu stavku koja sadrži taj simbol s lijeve strane pravila ili uspješno skeniranje (prebacivanje točke preko znaka). Primjerice, neka je A_i, $i \in \{1, \ldots, k\}$, simbol. Ako je A_i varijabla, u skupovima stanja se negdje nalazi završena Earleyeva stavka koja s lijeve strane pravila ima A_i. S druge strane, u slučaju da je A_1 znak, negdje postoji uspješno skeniranje kojim smo prebacili točku preko tog znaka. Tražimo upravo takve elemente, koji predstavljaju uspješna parcijalna parsiranja, kako bismo mogli konstruirati stablo parsiranja. Primijetimo da zapravo trebamo samo završene stavke, a ostale stavke možemo odbaciti.

Krećemo s kraja glavnog niza skupova stanja i prolazimo kroz skupove stanja kako bismo našli odgovarajuće stavke koje koristimo za konstrukciju stabla parsiranja. Gledamo simbol po simbol, i tražimo stavku koja predstavlja uspješno parcijalno parsiranje. Taj postupak ponavljamo za svaku nađenu stavku, sve dok ne dođemo do znakova koji predstavljaju listove stabala. Drugim riječima, izveli smo jedan oblik obilaska stabla parsiranja po dubini, pri čemu varijable s lijeve strane stavki i znakovi koji se pojavljuju s desne strane stavki označuju čvorove stabla. Takvim prolaskom kroz stavke dobijemo stablo parsiranja $(S, A_1, \ldots, A_k)$.

Stablo

Definiramo klasu `ParseTree`, čija svaka instanca predstavlja čvor stabla parsiranja. Klasa sadrži sljedeće varijable članice:

```
ParseTree* m_parent;
vector<ParseTree*> m_children;
string m_variable;
vector<string> m_symbols;
int m_start = 0;
int m_end = 0;
int m_length = 0;
string m_data;
```

gdje su `m_variable` i `m_symbols` varijable koje predstavljaju pravilo Earleyeve stavke čvora, dok su `m_start` i `m_end` početak odnosno kraj stavke. `ParseTree` još sadrži varijablu `m_data` — u koju spremamo oznaku čvora, varijablu `m_length` — koja služi prilikom konstrukcije cijelog stabla parsiranja (nešto o tome kasnije), i pokazivač `m_parent`, odnosno vektor pokazivača `m_children` — koji služe za obilazak stabla. Prilikom instanciranja stabla konstruktor kao argument prima, između ostalog, Earleyevu stavku iz koje se vade podaci kao što su pravilo, početak te kraj stavke. `ParseTree` također sadrži pomoćnu funkciju `isMatchingItem` koja kao argument prima Earleyevu stavku i broj `start`. Ona provjerava sadrži li stablo parsiranja isto pravilo, početak i kraj kao dana stavka.

```
bool ParseTree::isMatchingItem(EarleyItem item, int start)
{
  bool isSame = true;
  isSame &= m_variable == item.getVariable();
  isSame &= m_symbols == item.getSymbols();
  isSame &= m_start == start;
  isSame &= m_end == item.getStart();
  return isSame;
}
```

Primjer 19. Dobiveno stablo parsiranja riječi w_0 ispisujemo funkcijom `printTree` koja ga ispisuje na konzolu (funkcija je članica klase `Parser`). Stablo parsiranja ispisujemo tako što sve čvorove iste dubine ispisujemo vertikalno jedno ispod drugog, dok krajnje lijevo dijete nekog čvora ispisujemo prvo, zatim sljedeće dijete ispod njega, sve dok ne ispišemo svu djecu. Konkretno, ispis stabla parsiranja riječi w_0 izgleda ovako:

```
+- Sum
   +- Sum
   |  +- Product
   |     +- Factor
   |        +- Number
   |           +- 1
   +- +
   +- Product
      +- Factor
         +- (
         +- Sum
         |  +- Sum
         |  |  +- Product
         |  |     +- Product
         |  |     |  +- Factor
         |  |     |     +- Number
         |  |     |        +- 2
         |  |     +- *
         |  |     +- Factor
         |  |        +- Number
```

```
|  |           +- 3
|  +- -
|  +- Product
|     +- Factor
|        +- Number
|           +- 4
+- )
```

pri čemu znakovi listova čitani redom od vrha prema dnu daju upravo riječ w_0. ◁

Konstrukcija stabla parsiranja

Kao što smo vidjeli, potrebne su nam samo završene Earleyeve stavke stavke kako bismo konstruirali stablo parsiranja. Zato u glavnoj funkciji `parse` pozivamo pomoćnu funkciju `removeIncompleteItems` koja prolazi kroz skupove stanja i izbacuje nezavršene Earleyeve stavke.

```
void EarleyParser::removeIncompleteItems()
{
  for (auto& set : m_state) {
    auto it = set.begin();
    while (it != set.end()) {
      if (!(*it).isCompleted()) {
        it = set.erase(it);
      }
      else {
        ++it;
      }
    }
  }
}
```

Nakon nje, pozivamo još funkciju `orderStateByStart`. Trenutno je početak Earleyeve stavke eksplicitno zapisan u varijabli `m_start`, dok je kraj stavke implicitno određen indeksom skupa stanja u kojem se ta stavka nalazi. Funckijom `orderStateByStart` konstruiramo novi glavni niz, gdje početak i kraj stavke zamjenjujemo, tako da u varijabli `m_start` od sad držimo kraj stavke, a njen početak je implicitno određen indeksom novog skupa stanja u kojem se stavka nalazi.

```
vector<vector<EarleyItem>> EarleyParser::orderStateByStart
  (const vector<vector<EarleyItem>>& state)
{
  vector<vector<EarleyItem>> newState(state.size());
  for (int i = 0; i < state.size(); ++i) {
    for (auto item : state[i]) {
      EarleyItem newItem = EarleyItem(item.getVariable(),
        item.getSymbols(), i, item.getParsedSymbols());
```

```
      newState[item.getStart()].emplace_back(newItem);
    }
  }
  return newState;
}
```

Pogledajmo sada implementaciju glavne funkcije `createTree`. Funkcija se rekurzivno poziva sve dok ne dođemo do listova stabla, i vraća stablo parsiranja za neki ulazni string. Pozivi funkcije (prvi i svaki rekurzivni poziv) primaju dva argumenta koji se prilikom rekurzivnih poziva ne mijenjaju. To su:

- skup stanja `state` — iz kojeg smo prije prvog poziva maknuli nepotrebne Earleyeve stavke i zamijenili svim stavkama početak i kraj;
- ulazni string `input` — za koji konstruiramo stabla parsiranja.

Prilikom svakog poziva funkcije, varijable koje se mijenjaju su: broj `start` koji označuje početnu poziciju podriječi koju želimo parsirati, string `token` koji označuje varijablu gramatike za koju odabiremo pravila i pokazivač `parent` koji pokazuje na roditeljski čvor podstabla, čije ćemo dijete konstruirati.

```
1  ParseTree* EarleyParser::createTree(
2    const vector<vector<EarleyItem>>& state, const string& input,
3    int start, const string& token, ParseTree* parent)
4  {
5    EarleyItem chosenItem;
6    ParseTree* tree = nullptr;
7
8    for (int i = 0; i < state[start].size(); ++i) {
9      chosenItem = state[start][i];
10     if (chosenItem.getVariable() != token) {
11       continue;
12     }
13
14     int itemEnd = chosenItem.getStart();
15     if (parent != 0 && itemEnd > parent->m_end) {
16       continue;
17     }
18
19     bool found = false;
20     for (ParseTree* tree = parent; tree != 0; tree = tree->m_parent) {
21       if (tree->isMatchingItem(chosenItem, start)) {
22         found = true;
23         break;
24       }
25     }
26     if (found) {
```

```
27        continue;
28      }
29
30      tree = new ParseTree(parent, vector<ParseTree*>(), chosenItem,
31        start, itemEnd, 0, token);
32
33      int numOfSymbols = tree->m_symbols.size();
34      while (tree->m_children.size() < numOfSymbols) {
35        int index = tree->m_children.size();
36
37        string inputChar;
38        if (!input.empty()) {
39          inputChar += input[start + tree->m_length];
40        }
41
42        if (tree->m_symbols[index] == inputChar) {
43          int charLength = inputChar.empty() ? 0 : 1;
44          ParseTree* leafTree = new ParseTree(tree, vector<ParseTree*>(),
45            EarleyItem(), start + tree->m_length, start + tree->m_length
46            + charLength, charLength, inputChar);
47          tree->m_children.push_back(leafTree);
48          tree->m_length += charLength;
49          continue;
50        }
51
52        ParseTree* child = createTree(state, input,
53          start + tree->m_length, tree->m_symbols[index], tree);
54        if (child == nullptr) {
55          break;
56        }
57        else {
58          tree->m_length += child->m_length;
59          tree->m_children.push_back(child);
60        }
61      }
62    }
63    return tree;
64  }
```

Najprije ulazimo u glavnu petlju kojom prolazimo kroz sve Earleyeve stavke skupa stanja koji je određen ulaznim brojem `start` (prisjetimo se iz primjera 15 da indeks trenutno promatranog skupa stanja predstavlja broj dosad pročitanih znakova ulaznog stringa). Nakon što završimo s glavnom petljom vratimo stablo `tree`.

Na početku glavne petlje dohvaćamo i provjeravamo stavku po stavku, odgovaraju li kriterijima za konstrukciju stabla. Najprije spremimo stavku u `chosenItem` i provjerimo sadrži li stavka na lijevoj strani pravila varijablu `token` (linije 10–12). Nakon toga, provje-

ravamo je li kraj odabrane stavke `itemEnd` manji ili jednak kraju roditeljskog čvora (ako on postoji — u prvom pozivu funkcije se radi o korijenu stabla, koji nema roditelja). Ako jest, to znači da nećemo doći do valjanog stabla (linije 14–17). Iza toga postoji još jedna provjera o kojoj ćemo nešto reći kasnije.

Nakon provjere svih stavki, inicijaliziramo stablo `tree` koje sadrži informacije dobivene iz te stavke, s početkom `start` i oznakom `token` (linije 30–31). Zatim gledamo pravilo koje ono sadrži, spremamo broj simbola koji se nalaze s desne strane tog pravila u `numOfSymbols` i ulazimo u konačnu petlju. U njoj punimo vektor podstabala `m_children` dok broj djece ne postane jednak broju `numOfSymbols` (linije 33–61). To osigurava da je stablo parsiranja valjano, odnosno da smo prošli kroz sve simbole s desne strane pravila.

U petlji pristupamo svakom simbolu unutar desne strane pravila pomoću `index`, počevši od prvog simbola. Taj indeks ovisi o broju simbola broju djece promatranog stabla (linija 33). Zatim dohvatimo trenutni znak iz ulaza, `inputChar`, pomoću trenutne duljine stabla[4] i ulaznog parametra `start` (linije 37–40). Kako rekurzivno rješavamo podstabla, tako se duljina trenutno promatranog stabla povećava i time osiguravamo da ne čitamo isti znak više puta. Također, provjeravamo je li taj znak prazan kako ne bi došlo do pogreške prilikom dohvaćanja znaka (u slučaju gramatike koja izvodi praznu riječ).

Zatim provjeravamo je li `inputChar` jednak trenutno promatranom simbolu na poziciji `index` u vektoru `tree->m_symbols`. Ako jest, to znači da smo našli list koji je dijete stabla. Inicijaliziramo novo podstablo `leafTree` koje predstavlja taj list. Označimo ga s `inputChar` i dodamo ga u vektor podstabla trenutno promatranog stabla. Ako pročitani znak nije prazna riječ, povećamo promatranom stablu duljinu za jedan (linije 42–50).

Ako je trenutno promatrani simbol varijabla, rekurzivno pozovemo funkciju `createTree`, te izlaz spremimo u novo stablo `child` (linije 52–60). Rekurzivni poziv će nam vratiti podstablo parsiranja za tu varijablu. U slučaju da rekurzivni poziv nije vratio podstablo, odbacimo trenutno stablo jer trenutno parcijalno parsiranje nije valjano. Inače, dodamo stablu `tree` novo stablo `child` kao dijete te mu povećamo duljinu za duljinu djeteta.

Izlazak iz glavne petlje u liniji 63 i vraćanje stabla `tree`, predstavlja uspješnu konstrukciju podstabla trenutnog parsiranja, tj. znači da smo prošli kroz sve simbole desne strane pravila stabla. Primijetimo, ako je konstrukcija skupova stanja uspješno završila i pokrenuo se postupak konstrukcije stabla parsiranja, to garantira da postoji barem jedno valjano stablo parsiranja. Ako bi funkcija `createTree` vratila `nullptr`, to bi značilo da algoritam konstrukcije stabla nije ispravan.

[4] Ovdje s duljinom stabla mislimo na sadržaj članske varijable `m_length`.

Beskonačni broj stabala parsiranja

Na stranici 39 smo spomenuli provjeru koju radimo na početku glavne petlje algoritma, a to je provjera beskonačnosti broja stabala parsiranja za ulaznu riječ. Promotrimo gramatiku $\mathcal{G}_{A\varepsilon}$ danu pravilima:

$$A \to A$$
$$A \to \varepsilon.$$

$\mathcal{G}_{A\varepsilon}$ je višeznačna jer za praznu riječ (koju jedinu izvodi) postoji beskonačno mnogo stabala parsiranja. U ovom slučaju bi se glavna petlja u implementaciji algoritma rekurzivno pozivala sve dok ne bi napunila radnu memoriju računala i srušila program. U tu svrhu dodajemo provjeru u linijama 19–28. Polazimo od stabla `parent` i provjeravamo je li ono sadrži isto pravilo kao trenutno odabrana Earleyeva stavka `item` uz pomoć ranije opisane funkcije `isMatchingItem`. Ako je provjera neuspješna, nastavljamo provjeru roditeljem stabla `parent` i tako sve dok ne dođemo do korijena glavnog stabla. Ako nađemo stablo koje zadovoljava uvjete provjere, zaustavljamo provjeru i preskočimo trenutnu stavku jer bismo inače ušli u beskonačnu petlju.

Determinizam odabira stabla

Prema definiciji višeznačnih gramatika i riječi na stranici 10, za svaku višeznačnu riječ iz $L(\mathcal{G})$, postoji više stabala parsiranja iz gramatike $\mathcal{G}$. No, u implementaciji algoritma pronalazimo samo jedno stablo parsiranja za dani ulaz. Želimo osigurati da od svih stabala parsiranja (ako ih postoji više) uvijek odaberemo i ispišemo isto. Iz tog razloga smo implementirali skupove stanja kao vektore, za razliku od intuitivnije implementacije skupova stanja pomoću strukture `std::unordered_set`. Skup sam vodi računa o jedinstvenosti elemenata te ne bismo imali potrebu za implementacijom provjere jedinstvenosti stavki prilikom ubacivanja stavki u neki skup stanja. Ali zato vektori čuvaju uređenost stavki jer su skupovi stanja deterministično popunjeni prilikom njihove konstrukcije. Stoga, kad prolazimo kroz vektor i odabiremo stavke u glavnoj petlji konstrukcije stabla parsiranja, uvijek odaberemo istu sljedeću stavku.

Složenost algoritma

Pogledajmo sada vremensku i prostornu složenost implementacije algoritma u ovisnosti o duljini ulaznog stringa n. Prije faze konstrukcije skupova stanja imamo algoritam za pronalazak svih nepozitivnih varijabli. No njegova složenost ovisi samo o gramatici te broju njenih pravila i varijabli koji su konstantni (jer su određeni gramatikom). Dakle, taj dio algoritma ne ovisi o n.

U glavnoj petlji konstrukcije skupova stanja imamo petlju po elementima glavnog niza skupova stanja, koja se izvršava $O(n)$ puta. Unutar te petlje imamo ugniježđenu petlju po svim Earleyevim stavkama koja se izvršava također $O(n)$ puta (broj stavki u skupu stanja ovisi o ulaznom stringu, ali i o gramatici koja je konstantna). Nadalje, unutar te petlje se poziva funkcija `complete` koja u sebi sadrži još jednu petlju koja prolazi po nekom skupu stanja, dakle izvršava se također $O(n)$ puta. Zato imamo da je vremenska složenost konstrukcije skupova stanja jednaka $O(n) \cdot O(n) \cdot O(n)$, odnosno $O(n^3)$.

Prilikom konstrukcije stabla parsiranja imamo petlju po elementima skupova stanja koja se izvršava u vremenu $O(n)$. Ali, funkcija se rekurzivno poziva za svako podstablo. Kako znamo da imamo n listova (svaki list predstavlja znak iz ulaznog stringa), u najgorem slučaju se funkcija pozove n puta. Stoga imamo da je vremenska složenost konstrukcije stabla parsiranja $O(n^2)$. Sada imamo da je vremenska složenost cijelog algoritma $O(n^3) + O(n^2) = O(n^3)$.

S druge strane, za korištenu memoriju imamo glavni niz od $n + 1$ skupova stanja. Svaki pojedini skup stanja ima najviše $n \cdot p \cdot |(\rightarrow)|$ Earleyevih stavki, pri čemu p (broj mogućih pozicija točke na desnoj strani pravila) i broj pravila $|(\rightarrow)|$ ne ovise o n (jer su određeni gramatikom). Stoga je prostorna složenost algoritma $O(n + 1) \cdot O(n \cdot p \cdot |(\rightarrow)|) = O(n^2)$.

Bibliografija

[1] Jay Earley, *An efficient context-free parsing algorithm*, Commun. ACM **26** (1983), br. 1, 57–61, ISSN 0001-0782, https://doi.org/10.1145/357980.358005 (engleski).

[2] Jay Earley, *Jay's proffesional and biographical information*, https://personal-growth-programs.com/about/jays-professional-and-biographical-information/.

[3] Jeffrey Kegler, *Notes on loup's tutorials 2: Finding nullables*, https://github.com/jeffreykegler/kollos/blob/master/notes/misc/loup2.md.

[4] Saša Singer, *OAA predavanja - uvod u složenost*, http://degiorgi.math.hr/oaa/materijali/scans/pog_1.pdf.

[5] M. Sipser, *Introduction to the theory of computation*, Cengage Learning, 2012 (engleski).

[6] Loup Vaillant, *Earley parsing explained*, https://loup-vaillant.fr/tutorials/earley-parsing/.

[7] Vedran Čačić, *Interpretacija programa — predavanja*, https://web.math.pmf.unizg.hr/~veky/ip/IP%20slajdovi.pdf.

Sažetak

U početku rada smo definirali osnovne pojmove teorije formalnih jezika. Zatim smo definirali regularne izraze i uveli opću definiciju gramatika kako bismo mogli definirati klasu regularnih jezika. Zatim smo definirali beskontekstne gramatike i jezike koje one generiraju. Zatim smo uveli pojmove problema i parsiranja, te objasnili algoritam CYK koji ima teorijsku značajnost. U drugom poglavlju smo protumačili algoritam i implementaciju Earleyevog parsera. Prvo smo objasnili implementaciju strukture podataka. Zatim smo objasnili konstrukciju skupova stanja Earleyevih stavki. Naposljetku smo objasnili konstrukciju stabla parsiranja.

Summary

In the beginning of this thesis we defined basic terms of formal language theory. Then we defined regular expressions and introduced the universal definition of grammars so we could define the class of regular languages. Then we defined context-free grammars and languages they generate. Then we introduced concepts of problems and parsing, and explained the CYK algorithm which has theoretical significance. In the second chapter, we explained the algorithm and implementation of the Earley parser. First we explained the implementation of the used data structures. Then we explained the construction of state sets which consist of Earley items. In the end, we explained the construction of the parse tree.

Životopis

Rođen sam u Varaždinu 17. ožujka 1994. godine. Tamo sam pohađao II. osnovnu školu Varaždin te nakon završetka 2008. godine upisujem II. gimnaziju Varaždin. 2012. godine sam upisao preddiplomski sveučilišni studij Matematika i informatika na PMF-u Split. Nakon završenog preddiplomskog studija sam 2017. upisao diplomski sveučilišni studij Računarstvo i matematika na PMF-u Zagreb. Tijekom posljednje godine studija 2019. godine sam počeo raditi u tvrtki Nanobit d.o.o. na poziciji programera (*junior game developer*).

U slobodno vrijeme bavim se trčanjem i penjanjem, te razvojem videoigara i proučavanjem tehnologije *blockchain*.