Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Klemen Košir

Primerjava sintaksne analize vrste LLLR in LL(*)

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2019

Zahvala

Zahvaljujem se mentorju doc. dr. Boštjanu Slivniku za vse nasvete in vodenje pri izdelavi magistrskega dela. Hvala tudi moji družini in prijateljem za vso podporo, ki so mi jo nudili tekom celotnega študija.

Klemen Košir, 2019

Posvečeno babi Justi,
vse od malih nog.

Kazalo

Povzetek

Naslov: Primerjava sintaksne analize vrste LLLR in LL(*)

Sintaksna analiza se na področju računalništva uporablja v prevajalnikih za preverjanje veljavnosti prevajanega programa glede na izbrano gramatiko oziroma programski jezik. Novi in izboljšani pristopi ter algoritmi omogočajo hitrejšo sintaksno analizo ter dovoljujejo uporabo zahtevnejših razredov gramatik in programskih jezikov. V tem magistrskem delu smo implementirali metodo *LLLR*, ki združuje lastnosti metod *LL* in *LR*. Tako lahko sintaksno analizo izvedemo v linearnem času, v primeru konflikta v sintaksni tabeli pa izvajanje nadaljujemo z vgrajenim analizatorjem po metodi *LR*. Implementirano metodo smo najprej primerjali z metodo *LL(k)*. Čeprav sta metodi na gramatikah brez konfliktov v sintaksni tabeli enako hitri, na praktičnih primerih opazimo, da lahko metoda *LLLR* sintaksno analizo izvede ne glede na število konfliktov in rekurzivnost produkcij. Zatem smo metodo primerjali še z metodo *LL(*)*, ki konflikte razrešuje z determinističnimi končnimi avtomati. Metoda *LL(*)* je zaradi uporabe avtomatov v splošnem hitrejša, vendar ne podpira leve rekurzije. Za take gramatike je primernejša uporaba metode *LLLR*.

Ključne besede

sintaksna analiza, gramatika, jeziki, vgrajeni analizator

Abstract

Title: LLLR vs LL(*) parsing

Syntax analysis is a process used in compilers to determine the validity of a program source code according to the chosen programming language grammar. New and improved approaches and algorithms enable faster analysis and allow the use of more sophisticated grammars and programming languages. In this thesis, we implemented the *LLLR* method, which combines the features of the *LL* and *LR* methods. Thus, the syntax analysis can be performed in linear time, while conflicts in the parse table are resolved with the embedded parser using the *LR* method. We first compared the implemented method to the *LL(k)* method. Although the methods are equally fast when applied to grammars without conflicts in the parse table, practical examples have shown us that the *LLLR* method can perform the analysis regardless of the number of conflicts and recursiveness of productions. We then compared the method to *LL(*)*, which resolves conflicts with deterministic finite automata. The *LL(*)* method is generally faster due to the use of automata but does not support left recursion. For such grammars, the use of the *LLLR* method is more appropriate.

Keywords

syntax analysis, grammar, languages, embedded parser

Poglavje 1

Uvod

Sintaksna analiza se v računalništvu uporablja že vse od prvih programskih jezikov in je ključnega pomena v prevajalnikih. Z njo lahko določimo, ali je vhodna datoteka oziroma niz znakov veljaven glede na gramatiko, ki opisuje izbrani programski jezik ali strukturirani dokument. Napredek v zmogljivosti računalnikov je spremenil tudi pristope k raziskovanju metod za sintaksno analizo, saj lažje uporabimo algoritme, ki so bili pred nekaj leti časovno prezahtevni. To je prav tako vodilo do kompleksnejših programskih jezikov, ki programerjem olajšajo pisanje izvorne kode.

1.1 Pregled področja

Raziskovanje na področju kontekstno neodvisnih gramatik in sintaksne analize je potekalo že zelo zgodaj. Prvi članki na to temo so se pojavili v 60. letih prejšnjega stoletja. Metoda *LR*, ki jo je leta 1965 opisal Donald Knuth [1], se v enaki obliki uporablja še danes. Leto kasneje je Tadao Kasami prvi predstavil algoritem *Cocke-Younger-Kasami*, ki podpira širši razred gramatik kot metoda *LR* [2]. Na tem področju sta raziskovala tudi Lewis in Stearns [3], ki sta leta 1969 predstavila tudi metodo *LL(k)* [4].

Leta 1986 je izšla knjiga *Compilers: Principles, Techniques, and Tools* oziroma *Dragon Book* [5], ki še danes velja za enega izmed najpomembnejših

učbenikov za implementacijo prevajalnikov programskih jezikov. Do konca prejšnjega stoletja se je mnogo člankov osredotočalo na različne pristope ter implementacije algoritmov po metodah *LL* (od-zgodaj-navzdol) in *LR* (od-spodaj-navzgor), s poudarkom na učinkovitosti. Leta 1995 sta Schmeiser in Barnard predstavila pristop za izračun leve sledi izpeljave pri metodah od-spodaj-navzgor [6]. Ta dovoljuje uporabo metod *LR* za zmogljivejšo sintaksno analizo, leva sled izpeljave pa omogoča lažjo uporabo v prevajalnikih.

Tekom let je bilo objavljenih tudi nekaj člankov na temo močnejših sintaksnih analizatorjev (angl. *generalized parsers*) [7] in njihovih razširitev [8]. Ti se osredotočajo na sintaksno analizo dvoumnih in nedeterminističnih gramatik naravnih jezikov. Predstavljene so bile tudi metode, ki se poslužujejo analize v obeh smereh (angl. *ascent-descent*) [9].

Z vedno večjo zmogljivostjo računalnikov se je spremenil tudi pristop k raziskovanju področja. Gramatik programskih jezikov ni bilo več potrebno omejevati glede na obliko produkcij in drugih omejitev obstoječih metod sintaksne analize, kar je vodilo do kompleksnejših in zmogljivejših programskih jezikov, s tem pa se je začel tudi razvoj novih algoritmov za sintaksno analizo. Leta 2011 je bila predstavljena metoda *LL(*)* za uporabo v generatorju sintaksnih analizatorjev *ANTLR 3.3* [10]. Ta je implementirana z rekurzivnim spuščanjem. Glavna razlika in prednost pred drugimi algoritmi po metodi *LL* je ta, da metoda za analizo in razrešitev konfliktov namesto vnaprej določenega števila vhodnih simbolov (angl. *fixed window*) uporablja deterministične končne avtomate. Kljub temu da mora metoda v teoriji pregledati vse možne produkcije, se v praksi izkaže, da v povprečju potrebujemo le en ali dva vhodna simbola.

Leta 2003 je Boštjan Slivnik predstavil metodo *LLLR* [11, 12], ki za razreševanje konfliktov v sintaksni tabeli metode *LL* uporablja vgrajene stroje *LR* [13, 14]. Metoda tako uporablja enostavnost sintaksne analize po metodi *LL*, obenem pa podpira širok razred gramatik z metodo *LR*. Vgrajeni analizator stroje zgradi le za konfliktne vmesne simbole, ki pokrivajo zgolj manjši del sintaksnega drevesa, kar vodi do hitrejšega izvajanja analize. Poleg tega

se vgrajeni stroji uporabijo le za razrešitev konflikta, zatem pa analizo nadaljujemo s krovnim analizatorjem po metodi *LL*. Kljub uporabi metode iz skupine od-spodaj-navzgor metoda *LLLR* tvori levo sled izpeljave. Avtor je leta 2017 ovrednotil tudi uporabo drugih algoritmov znotraj posameznih analizatorjev [15].

Leta 2014 je bila predstavljena metoda *ALL(*)* (angl. *adaptive LL(*)*) za uporabo v analizatorju *ANTLR 4* [16]. Ta je osnovana na metodi *LL(*)*, vendar sled izpeljave gradi dinamično, zaradi česar ni potrebno vsakič pregledati vseh možnih produkcij. Poleg končnih avtomatov se metoda poslužuje tudi grafa skladov (angl. *graph-structured stack*), s katerim se lahko izognemo nepotrebnemu preračunavanju. Avtorji so metodo *ALL(*)* primerjali z metodama *GLL* in *GLR* ter ugotovili, da je sintaksna analiza na izvorni kodi knjižnice programskega jezika *Java* do 135-krat hitrejša.

1.2 Struktura dela

Namen tega magistrskega dela je predstaviti in implementirati sintaksno analizo po metodi *LLLR*, ki združuje lastnosti metod *LL* in *LR*. Podrobno bomo opisali kontekstno neodvisne gramatike in njihov zapis ter teorijo vseh treh metod. Opisali bomo funkcije in postopke za izračun vseh podatkovnih struktur, ki jih potrebujemo za izvajanje sintaksne analize, to pa prikazali tudi s primeri. Podrobno bomo predstavili tudi postopek izvajanja sintaksne analize za izbrane vhodne nize. Del poglavja bomo namenili razreševanju konfliktov v sintaksni tabeli metode *LL*.

Kasneje bomo predstavili arhitekturo implementiranega programa in uporabljene podatkovne strukture ter opisali obliko datotek za predstavitev gramatik. Opisali bomo tudi leksikalno analizo z regularnimi izrazi, s katerimi vhodni niz pretvorimo v simbole, na katerih izvajamo sintaksno analizo. Del opisa implementacije bo namenjen tudi gradnji strojev *LR*, ki so ključnega pomena v vgrajenem analizatorju implementirane metode. Glavni del poglavja bo namenjen podrobnemu opisu implementacije in izvajanja sintaksne

analize po metodi *LLLR*. Predstavili bomo tudi pristope za optimizacijo hitrosti izvajanja in porabe pomnilnika.

V zaključnem delu bomo ovrednotili metodo *LLLR* na tako teoretičnih kot tudi praktičnih primerih gramatik. Najprej bomo prikazali prednosti metode v primerjavi z metodo *LL(1)*, na kateri je osnovan krovni analizator. Zatem bomo ovrednotili primernost metode *LLLR* za različne oblike gramatik v primerjavi z metodo *LL(*)*, katere implementacija se uporablja v generatorju sintaksnih analizatorjev *ANTLR 3.3*. Ugotavljali bomo, kdaj je uporaba metode *LLLR* smiselna in kdaj je priporočljivo uporabiti druge metode sintaksne analize. Na kratko bomo predstavili tudi pristope za prilagoditev produkcij gramatike, s katerimi lahko izboljšamo hitrost izvajanja sintaksne analize po implementirani metodi.

Poglavje 2

Sintaksna analiza

Sintaksna analiza (angl. *syntax analysis, parsing*) se v računalništvu in lingvistiki uporablja za analizo poljubnega besedila. Analizator na osnovi podane gramatike za vhodno besedilo zgradi sintaksno drevo (angl. *parse tree*), ki predstavlja povezave med posameznimi besedami oziroma simboli. V kolikor je sintaksno drevo mogoče zgraditi v celoti, je vhodno besedilo veljavno po izbrani gramatiki.

V računalništvu se sintaksna analiza najpogosteje uporablja v prevajalnikih, kjer je potrebno ugotoviti, ali je prevajani program veljaven za programski jezik, ki ga prevajamo. Sintaksno drevo se nato uporabi v naslednjih fazah prevajalnika, najprej za semantično analizo (angl. *semantic analysis*).

2.1 Kontekstno neodvisne gramatike in jeziki

Sintaksna analiza, uporabljena v tem magistrskem delu, deluje na osnovi kontekstno neodvisnih gramatik (angl. *context-free grammars*), ki se uporabljajo za opis strukture programskih jezikov. Gramatika je formalno definirana s četverko $G = (N, T, P, S)$, ki je sestavljena iz:

- končne množice N, ki vsebuje spremenljivke oziroma vmesne simbole (angl. *nonterminal symbols*),

- končne množice T, ki vsebuje končne simbole (angl. *terminal symbols*), ki tvorijo opisani jezik,
- končne množice $P \subset N \times (N \cup T)^*$ z naborom produkcij (angl. *productions*), in
- začetnega simbola S iz množice N.

Vsaka produkcija je sestavljena iz glave oziroma vmesnega simbola ter niza končnih in vmesnih simbolov, ki predstavljajo telo produkcije. Posamezna produkcija ponazarja enega izmed načinov, na katerega lahko vmesni simbol nadomestimo z nizom simbolov, posledično pa tvorimo niz končnih simbolov, ki je veljaven po opisani gramatiki.

Primer opisanih komponent lahko prikažemo na gramatiki 2.1. V tem primeru množica N vsebuje le vmesni simbol E, ki je obenem tudi začetni simbol gramatike. Množica T vsebuje vse simbole, ki se ne pojavijo na levi strani produkcij, torej `+`, `*`, `(`, `)` in `id`. Primera produkcij v množici P sta $E \rightarrow E + E$ in $E \rightarrow id$.

Gramatika 2.1: Primer kontekstno neodvisne gramatike.

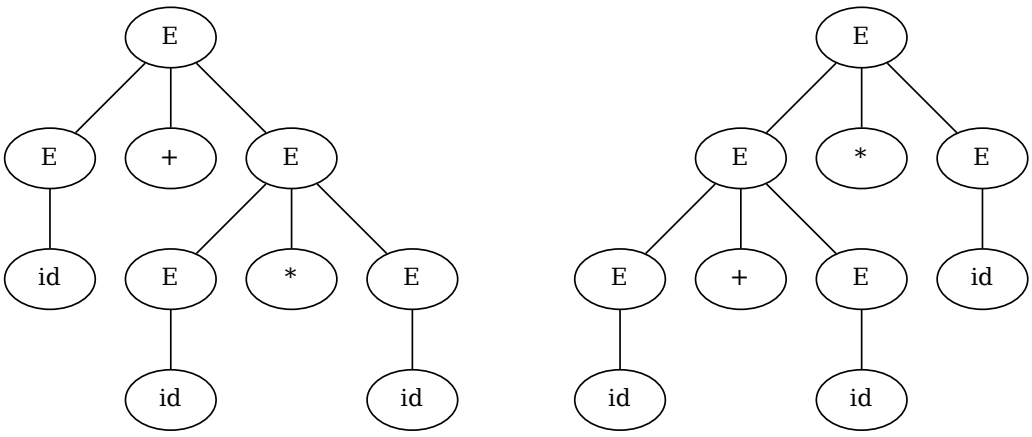
E → E `+` E
E → E `*` E
E → `(` E `)`
E → `id`

Za zapis kontekstno neodvisnih gramatik in algoritmov bomo uporabili naslednjo notacijo:

- Produkcija gramatike je zapisana z vmesnim simbolom, ki mu sledi puščica, nato pa zaporedje končnih in vmesnih simbolov. Če ima vmesni simbol več produkcij, so te lahko ločene z navpično črto (na primer $A \rightarrow a\ B\ b\ C$ in $A \rightarrow a + b \mid a\ *\ b$).
- Končni simboli so predstavljeni z malimi črkami in besedami, številkami ter posebnimi simboli (na primer a, `id`, `1`, `+` ...).

- Vmesni simboli so predstavljeni z veliki črkami in besedami (na primer S, A, EXPRESSION ...),
- Male grške črke predstavljajo zaporedje končnih in vmesnih simbolov, ki je lahko tudi prazno (na primer α, β, ω ...).
- Z grško črko ϵ določimo prazen niz simbolov, ki ne vsebuje nobenega vmesnega ali končnega simbola.
- Poseben simbol \$ predstavlja začetek in konec vhodnega niza.

Niz simbolov gramatike tvorimo tako, da začnemo z začetnim simbolom S, nato pa vmesne simbole postopoma zamenjujemo s simboli izbranih produkcij, dokler niz ne vsebuje le končnih simbolov. Obenem tvorimo tudi sled izpeljave, ki predstavlja korake, po katerih smo iz začetnega simbola prišli do končnega niza simbolov.



Slika 2.1: Drevesi izpeljav za niz `id + id * id` po gramatiki 2.1.

Če na vsakem koraku razvijemo prvi oziroma čisto levi vmesni simbol, tvorimo levo sled izpeljave (angl. *leftmost derivation*). V nasprotnem primeru, kjer vedno razvijemo zadnji oziroma čisto desni vmesni simbol, tvorimo desno sled izpeljave (angl. *rightmost derivation*). Primer leve sledi izpeljave

za vhodni niz `id + id * id` po gramatiki 2.1 je naslednji:

$$E \implies E + E \implies id + E \implies id + E * E$$
$$\implies id + id * E \implies id + id * id$$

Sled izpeljave lahko ponazorimo tudi z drevesom izpeljav (angl. *derivation tree*), kar prikazuje slika 2.1. Opazimo lahko, da sta drevesi leve in desne sledi izpeljave različni, kar pomeni, da je gramatika dvoumna, saj lahko do končnega niza simbolov pridemo na več kot en način. To pomeni, da gramatika 2.1, ki opisuje matematične izraze, ne zagotavlja vedno istega zaporedja izrazov, kar lahko vodi do različnih rezultatov. Za potrebe sintaksne analize želimo uporabljati le nedvoumne gramatike.

2.2 Metode sintaksne analize

Metode sintaksne analize lahko razdelimo v dve glavni skupini:

- metode *od-zgoraj-navzdol* (angl. *top-down parsing*), ki sintaksno drevo gradijo v smeri od korena do listov, in
- metode *od-spodaj-navzgor* (angl. *bottom-up parsing*), ki liste drevesa postopoma združujejo v večja drevesa v smeri navzgor, dokler vseh ne povežejo s korenskim vozliščem.

Sintaksna analiza po metodah skupine *od-zgoraj-navzdol* je ponavadi implementirana na enega izmed dveh načinov. Prvi pristop, ki ga lahko uporabimo v metodi *LL*, se poslužuje sintaksne tabele, s katero glede na trenutni vhodni simbol izberemo naslednjo produkcijo. Ko izračunamo sintaksno tabelo, lahko analizo izvedemo v linearnem času. Drugi pristop je implementiran z rekurzivnim spuščanjem (angl. *recursive descent parsing*). Implementacija take sintaksne analize je podobna sami gramatiki, saj so posamezni vmesni simboli implementirani s funkcijami, ki se med seboj rekurzivno kličejo. Funkcije so lahko implementirane iterativno z branjem posameznih

simbolov ali z ujemanjem vzorcev (angl. *pattern matching*). Primer takega pristopa je metoda *LL(*)*, ki jo bomo predstavili v zadnjem poglavju.

V preostanku tega poglavja si bomo ogledali teorijo in značilnosti metod sintaksne analize, uporabljenih v sklopu tega magistrskega dela, podrobnosti implementacije le-teh pa bomo spoznali v poglavju 3.

2.3 Metoda LL

Metoda *LL(k)* (angl. *left-to-right, leftmost derivation*) pripada skupini *od-zgoraj-navzdol* in velja za eno enostavnejših vrst sintaksne analize. Parameter k določa, koliko vhodnih simbolov je na voljo algoritmu, da lahko izbere naslednjo produkcijo. V sklopu tega magistrskega dela smo uporabili metodo *LL(1)*, ki med analizo za izbiro naslednje produkcije uporabi le en vhodni simbol. Metoda se izvede v linearnem času glede na dolžino vhodnega niza.

Gramatika 2.2: Primer gramatike za metodo *LL*.

$$
\begin{aligned}
\mathrm{E} &\rightarrow \mathrm{T}\ \mathrm{E}' \\
\mathrm{E}' &\rightarrow \texttt{+}\ \mathrm{T}\ \mathrm{E}' \mid \epsilon \\
\mathrm{T} &\rightarrow \mathrm{F}\ \mathrm{T}' \\
\mathrm{T}' &\rightarrow \texttt{*}\ \mathrm{F}\ \mathrm{T}' \mid \epsilon \\
\mathrm{F} &\rightarrow \texttt{(}\ \mathrm{E}\ \texttt{)} \mid \texttt{id}
\end{aligned}
$$

Za izvajanje sintaksne analize potrebujemo funkciji *FIRST* in *FOLLOW*. Funkcija *FIRST* je definirana kot $FIRST(A) = \{\ a \in T \mid A \Rightarrow^* a\beta\ \}$ in izračuna množico končnih simbolov, ki se pojavijo na prvem mestu nizov, ki jih lahko razvijemo iz simbola A. Množice lahko izračunamo z naslednjim postopkom:

1. če je simbol a končni simbol, potem množica *FIRST(a)* vsebuje le simbol a,
2. če je simbol A vmesni simbol in obstaja produkcija $A \rightarrow B$, v množico *FIRST(A)* dodamo vse simbole iz *FIRST(B)*,

3. če je simbol A vmesni simbol in obstaja produkcija $A \to B_1 B_2 \cdots B_n$, potem za vse definirane vrednosti i v množico *FIRST(A)* dodamo vse simbole iz *FIRST(B_i)*, dokler množica *FIRST(B_{i-1})* vsebuje simbol ϵ,

4. če obstaja produkcija $A \to \epsilon$, v množico *FIRST(A)* dodamo simbol ϵ,

5. postopek ponavljamo, dokler v nobeno množico ne moremo dodati novega simbola.

Funkcija *FOLLOW* je podobna funkciji *FIRST*, vendar vrne množico vseh končnih simbolov, ki v gramatiki neposredno sledijo izbranemu simbolu. Definiramo jo kot $FOLLOW(A) = \{\ b \in T \mid S\$ \Rightarrow^* \alpha A b \beta\$\ \}$, množice pa lahko izračunamo z naslednjim postopkom:

1. v množico *FOLLOW(S)* dodamo simbol \$,

2. za vsako produkcijo $A \to \alpha B \beta$ v množico *FOLLOW(B)* dodamo vse simbole iz *FIRST(β)*, če ta ne vsebuje simbola ϵ,

3. za vsako produkcijo $A \to \alpha B$ oziroma produkcijo $A \to \alpha B \beta$, kjer množica *FIRST(β)* vsebuje simbol ϵ, v množico *FOLLOW(B)* dodamo vse simbole iz *FOLLOW(A)*.

Tabela 2.1: Množici *FIRST* in *FOLLOW* za gramatiko 2.2.

simbol	FIRST	FOLLOW
E	`(`, `id`	`)`, \$
E′	`+`, ϵ	`)`, \$
T	`(`, `id`	`+`, `)`, \$
T′	`*`, ϵ	`+`, `)`, \$
F	`(`, `id`	`*`, `+`, `)`, \$

Primer množic *FIRST* in *FOLLOW* za gramatiko 2.2 prikazuje tabela 2.1. Preden lahko začnemo s sintaksno analizo, moramo izračunati še sintaksno

tabelo (angl. *parse table*). Ta za vsak vhodni simbol vsebuje produkcijo, po kateri lahko metoda razvije trenutni vmesni simbol. Sintaksna tabela na vmesnem simbolu A za vhodni simbol a vsebuje produkcijo $A \rightarrow \omega$, če množica *FIRST(ω)* vsebuje simbol a. V primeru, da množica vsebuje simbol ϵ, se mora simbol a nahajati v množici *FOLLOW(A)*. V sintaksno tabelo vključimo tudi simbol \$. Sintaksno tabelo za gramatiko 2.2 prikazuje tabela 2.2. Če lahko za nek vhodni simbol in vmesni simbol, ki je trenutno na skladu, izberemo več kot eno produkcijo, sintaksna tabela vsebuje konflikt. V takem primeru metoda za trenutno stanje ni sposobna izbrati naslednje produkcije in zato gramatike ne moremo uporabiti z metodo *LL*.

Tabela 2.2: Del sintaksne tabele za gramatiko 2.2.

simbol	`+`	`(`	`id`	\$
E		E → T E′	E → T E′	
E′	E′ → `+` T E′			E′ → ϵ
T		T → F T′	T → F T′	
T′	T′ → ϵ			T′ → ϵ
F		F → `(` E `)`	F → `id`	

Delovanje metode bomo predstavili na gramatiki 2.2. Metoda prične s skladom s simbolom \$ in začetnim simbolom E, nato pa glede na sintaksno tabelo in naslednjih k vhodnih simbolov izbere produkcijo, po kateri bo razvila simbol E. Metoda simbol na skladu nadomesti s telesom izbrane produkcije, nato pa začne ujemati vhodne simbole s končnimi simboli na skladu. Ko se na vrhu sklada ponovno pojavi vmesni simbol, metoda izbere naslednjo produkcijo. Ta postopek ponavljamo, dokler vhodni niz in sklad nista prazna. V primeru, da sintaksna tabela ne vsebuje produkcije za trenutni vhodni simbol oziroma se končni simbol ne ujema, vhodni niz ni veljaven po izbrani gramatiki. Primer izvajanja sintaksne analize za niz `3 + 5 * 7` prikazuje tabela 2.3.

Čeprav je izvajanje sintaksne analize po metodi *LL* enostavno, moramo

biti pazljivi, da pri definiranju gramatike ne uvedemo konfliktov. Na primer, nobena produkcija za nek vmesni simbol A se ne sme začeti z istim končnim simbolom a. Poleg tega moramo biti pazljivi, da produkcije niso rekurzivne (na primer $A \rightarrow A\ a$), kar med izvajanjem povzroči neskončne zanke.

Tabela 2.3: Izvajanje sintaksne analize po metodi *LL* za gramatiko 2.2.

sklad	vhod	produkcija
`$ E`	`3 + 5 * 7 $`	E → T E′
`$ E' T`	`3 + 5 * 7 $`	T → F T′
`$ E' T' F`	`3 + 5 * 7 $`	F → `id`
`$ E' T' id`	`3 + 5 * 7 $`	
`$ E' T'`	`+ 5 * 7 $`	T′ → ϵ
`$ E'`	`+ 5 * 7 $`	E′ → `+` T E′
`$ E' T +`	`+ 5 * 7 $`	
`$ E' T`	`5 * 7 $`	T → F T′
`$ E' T' F`	`5 * 7 $`	F → `id`
`$ E' T' id`	`5 * 7 $`	
`$ E' T'`	`* 7 $`	T′ → `*` F T′
`$ E' T' F *`	`* 7 $`	
`$ E' T' F`	`7 $`	F → `id`
`$ E' T' id`	`7 $`	
`$ E' T'`	`$`	T′ → ϵ
`$ E'`	`$`	E′ → ϵ
`$`	`$`	

2.4 Metoda LR

Metoda *LR(k)* (angl. *left-to-right, rightmost derivation in reverse*), ki jo imenujemo tudi metoda s pomiki in prevedbami (angl. *Shift-Reduce*), pripada skupini *od-spodaj-navzgor*. To pomeni, da sintaksno drevo gradimo z dna,

vmesne simbole znotraj produkcij pa razvijamo od desne proti levi.

Metoda *LR* se od metode *LL* razlikuje v tem, da je pred analizo potrebno zgraditi stroj *LR* (angl. *LR machine*), ki med izvajanjem za prehode med stanji uporablja vhodne in vmesne simbole, podobno kot pri regularnih izrazih (angl. *regular expressions*). Med gradnjo strojev uporabljamo tudi funkcijo *FIRST*.

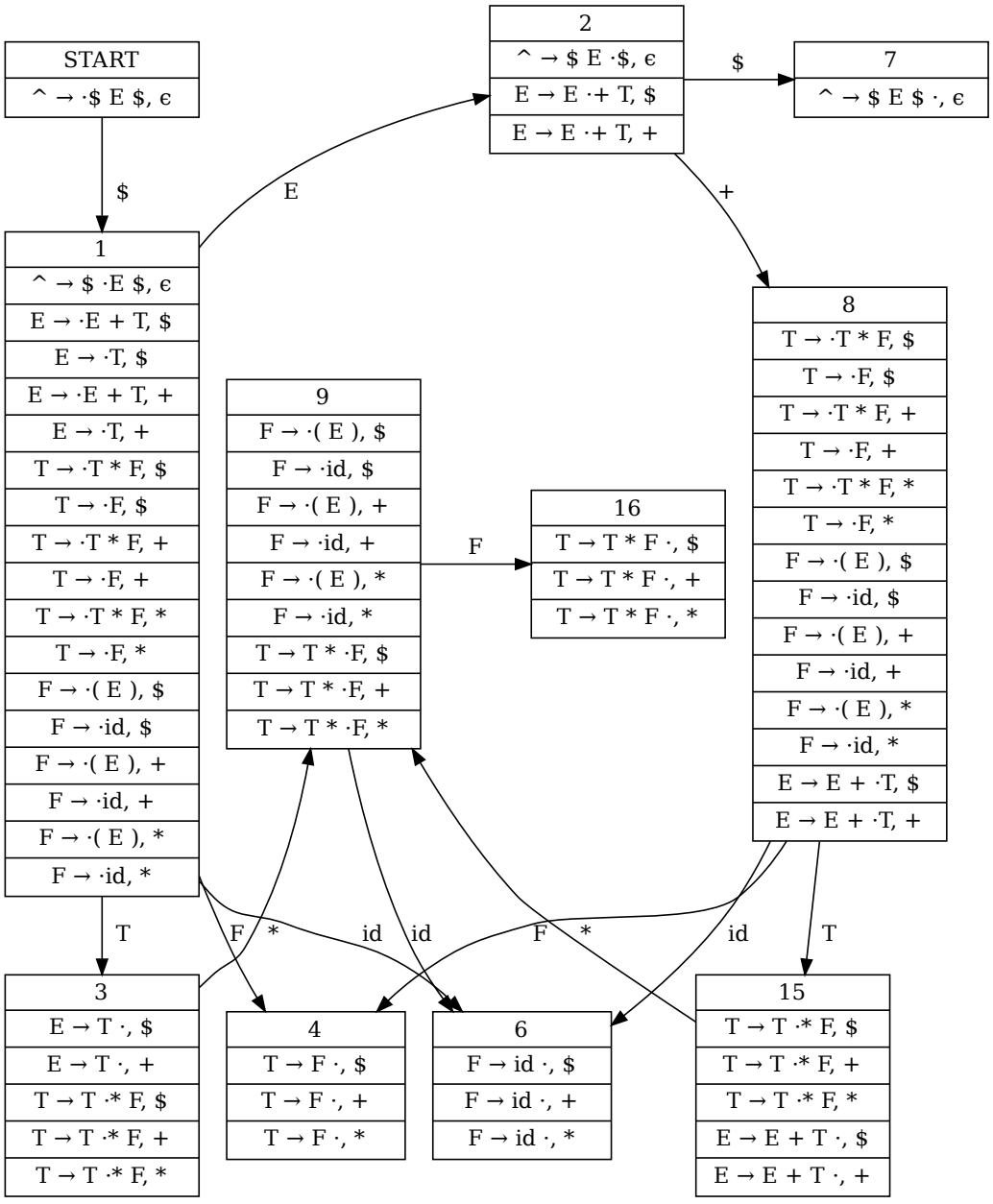
Gramatika 2.3: Primer gramatike za metodo *LR*.

$$
\begin{aligned}
\mathrm{E} &\rightarrow \mathrm{E}\ \texttt{+}\ \mathrm{T} \mid \mathrm{T} \\
\mathrm{T} &\rightarrow \mathrm{T}\ \texttt{*}\ \mathrm{F} \mid \mathrm{F} \\
\mathrm{F} &\rightarrow \texttt{(}\ \mathrm{E}\ \texttt{)} \mid \texttt{id}
\end{aligned}
$$

Vsako stanje v stroju *LR* sestoji iz opisov (angl. *items*), ki predstavljajo delno prebrano produkcijo gramatike, na primer $A \rightarrow a\ B \cdot c,\ d$. Ta opis predstavlja produkcijo vmesnega simbola A, v kateri smo že prebrali simbola a in B, v naslednjem koraku sledi simbol c, na vhodu pa mora simbolu c slediti simbol d (angl. *lookahead*). Posamezno stanje stroja je sestavljeno iz dveh delov — jedra (angl. *core*), ki vsebuje opise, ki smo jih izpeljali iz prejšnjih stanj, in ovojnice (angl. *closure*), ki vsebuje opise, izpeljane iz jedra. Z uporabo opisov se med sintaksno analizo odločimo, v katero stanje lahko glede na trenutno stanje in vhodni simbol premaknemo stroj.

Stroj *LR* zgradimo tako, da najprej določimo začetno stanje, ki vsebuje opis $S' \rightarrow \cdot\ \$\ S\ \$,\ \epsilon$. Nato za vsak simbol X, ki v opisu sledi piki, iz trenutnega stanja izpeljemo novo stanje. Ta v jedru vsebuje vse opise predhodnega stanja, katerim na piki sledi simbol X, piko pa prestavimo za en simbol naprej. Zatem za vsak vmesni simbol A, ki v opisih novega stanja sledi piki, v ovojnico dodamo vse produkcije tega simbola. Ko v ovojnico ne moremo dodati nobenega novega opisa, smo stanje v celoti izpeljali. Ta postopek ponavljamo, dokler ne izpeljemo vseh možnih stanj. Primer stroja *LR* za gramatiko 2.3 prikazuje slika 2.2.

Ko zaključimo z gradnjo stroja, lahko izračunamo tabeli, ki ju potrebujemo za sintaksno analizo. Tabela *ACTION* vsebuje akcije, ki jih lahko



Slika 2.2: Del stroja *LR(1)* za gramatiko 2.3.

izvedemo v posameznem stanju stroja glede na vhodni simbol. Tabela vsebuje tri različne akcije, ki morajo biti edinstvene za trenutno stanje in vhodni simbol:

1. *Shift(s)* oziroma *pomik*, ki na sklad prestavi vhodni končni simbol in novo stanje s,
2. *Reduce(r)* oziroma *prevedba*, s katero izberemo produkcijo r, s sklada pa odstranimo vse simbole, ki jih vsebuje telo produkcije,
3. *Accept* oziroma *sprejem*, s katero končamo sintaksno analizo.

Tabela 2.4: Del tabele *ACTION* za gramatiko 2.3.

stanje	`+`	`*`	`(`	`id`	ϵ
1			Shift(5)	Shift(6)	
2	Shift(8)				
3	Reduce(2)	Shift(9)			
4	Reduce(4)	Reduce(4)			
5			Shift(13)	Shift(14)	
6	Reduce(6)	Reduce(6)			
7					Accept
8			Shift(5)	Shift(6)	
13			Shift(13)	Shift(14)	

Akcije za pomik predstavljajo vsi prehodi med stanji stroja preko končnih simbolov. Za izračun preostalih akcij moramo pregledati vse opise v stroju. Na opisu lahko izvedemo akcijo za prevedbo, če piki v opisu ne sledi noben simbol — to pomeni, da smo produkcijo v opisu prebrali do konca. Podobno velja za akcijo za sprejem, vendar se ta lahko izvede le na začetnem opisu. Primer tabele *ACTION* prikazuje tabela 2.4.

Tabela *GOTO* vsebuje prehode med stanji, ki jih izvedemo, ko med sintaksno analizo izvedemo akcijo za prevedbo. Tabela sestoji iz vseh prehodov

Tabela 2.5: Del tabele *GOTO* za gramatiko 2.3.

stanje	S	E	T	F
1		2	3	4
2				
5		10	11	12
8			15	4
10				
13		20	11	12
17			21	12
19				22

med stanji stroja preko vmesnih simbolov. Primer tabele *GOTO* smo prikazali na tabeli 2.5.

Sintaksna analiza po metodi *LR* je veliko zahtevnejša od metod skupine *od-zgoraj-navzdol*. Izvajanje začnemo s skladom, ki vsebuje začetni simbol S v stanju 0. Na vsakem koraku sintaksne analize iz tabele *ACTION* glede na trenutno stanje stroja in vhodni simbol izberemo naslednjo akcijo. Če smo izbrali akcijo *Shift(s)*, se stroj preko vhodnega končnega simbola prestavi v stanje s, novo stanje in vhodni simbol pa dodamo na vrh sklada ter nadaljujemo z analizo. V primeru, da smo iz tabele izbrali akcijo *Reduce(r)*, se stroj nahaja v stanju, v katerem lahko izberemo produkcijo r. Zato z vrha sklada odstranimo telo produkcije r, iz tabele *GOTO* pa glede na trenutno stanje stroja in glavo produkcije r izberemo novo stanje stroja. Trenutnega simbola ne odstranimo z vhoda, saj ga moramo ponovno uporabiti na naslednjem koraku.

Zadnja akcija, ki jo lahko izberemo iz tabele *ACTION*, je *Accept*. Ta nakazuje, da smo stroj uspešno prestavili v zadnje stanje, v katerem lahko določimo, da je vhodni niz veljaven po izbrani gramatiki. Preden lahko zaključimo sintaksno analizo, moramo preveriti, da smo z vhoda prebrali vse simbole. V primeru, da tabela *ACTION* za trenutno stanje in vhodni

simbol ne vsebuje akcije, je vhodni niz neveljaven. Primer izvajanja sintaksne analize za vhod `3 + 5 * 7` po gramatiki 2.3 prikazuje tabela 2.6.

Tabela 2.6: Izvajanje sintaksne analize po metodi *LR* za gramatiko 2.3.

sklad	vhod	akcija	produkcija
S_0	`3 + 5 * 7 $`	Shift(5)	
S_0 id_5	`+ 5 * 7 $`	Reduce(6) $\rightarrow$ 3	F $\rightarrow$ `id`
S_0 F_3	`+ 5 * 7 $`	Reduce(4) $\rightarrow$ 2	T $\rightarrow$ F
S_0 T_2	`+ 5 * 7 $`	Reduce(2) $\rightarrow$ 1	E $\rightarrow$ T
S_0 E_1	`+ 5 * 7 $`	Shift(6)	
S_0 E_1 $+_6$	`5 * 7 $`	Shift(5)	
S_0 E_1 $+_6$ id_5	`* 7 $`	Reduce(6) $\rightarrow$ 3	F $\rightarrow$ `id`
S_0 E_1 $+_6$ F_3	`* 7 $`	Reduce(4) $\rightarrow$ 13	T $\rightarrow$ F
S_0 E_1 $+_6$ T_{13}	`* 7 $`	Shift(7)	
S_0 E_1 $+_6$ T_{13} $*_7$	`7 $`	Shift(5)	
S_0 E_1 $+_6$ T_{13} $*_7$ id_5	`$`	Reduce(6) $\rightarrow$ 14	F $\rightarrow$ `id`
S_0 E_1 $+_6$ T_{13} $*_7$ F_{14}	`$`	Reduce(3) $\rightarrow$ 13	T $\rightarrow$ T `*` F
S_0 E_1 $+_6$ T_{13}	`$`	Reduce(1) $\rightarrow$ 1	E $\rightarrow$ E `+` T
S_0 E_1	`$`	Accept	

Metoda *LR* je zmogljivejša od metode *LL*, saj podpira širši razred kontekstno neodvisnih jezikov, prav tako pa nima težav z levo rekurzijo. Kljub temu se uporablja redkeje, saj je gradnja stroja *LR* počasna in porabi več pomnilnika.

2.5 Metoda LLLR

Metoda *LLLR*, ki je glavni del tega magistrskega dela, je sestavljena iz metod *LL* in *LR*. Metoda v osnovi deluje enako kot *LL*, vendar dovoljuje konflikte v sintaksni tabeli. Ko naleti na konflikt, sintaksne analize ne prekine, ampak nadaljuje z vgrajenim analizatorjem po metodi *LR*. Ta delo nadaljuje, dokler

ne razreši konflikta. Metoda nato posodobi trenutni sklad sintaksne analize, izvajanje pa nadaljuje s krovnim analizatorjem po metodi *LL*. Zaradi tega so vgrajeni stroji *LR* manjši, saj se izvajajo le na delu gramatike, vendar moramo biti kljub temu pozorni pri pisanju gramatike, da lahko to lastnost izkoristimo.

Gramatika 2.4: Primer gramatike za metodo *LLLR*.

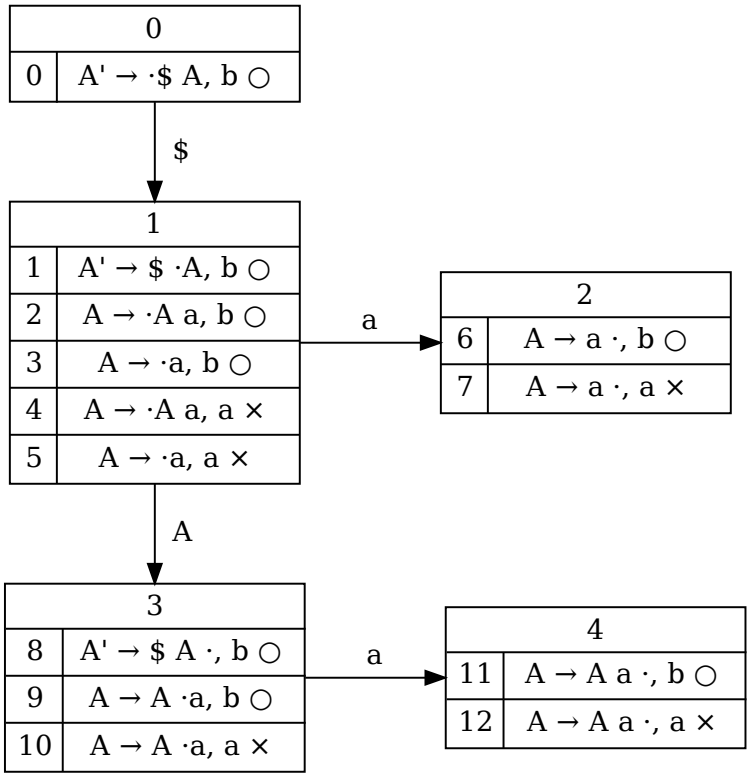
$$
\begin{aligned}
\mathrm{S} &\rightarrow \texttt{a}\ \mathrm{A}\ \texttt{b}\ \mathrm{B}\ \texttt{c} \\
\mathrm{A} &\rightarrow \mathrm{A}\ \texttt{a} \mid \texttt{a} \\
\mathrm{B} &\rightarrow \mathrm{A}\ \texttt{a}\ \texttt{b} \mid \texttt{b}
\end{aligned}
$$

Vgrajeni analizator v metodi *LLLR* uporablja stroj *LR*, vendar zahteva nekaj dodatnih izračunov. Poleg stanj ter prehodov med stanji in opisi je potrebno izračunati tudi enoličnost (angl. *uniqueness*) posameznih opisov. Opis je enoličen, če so vsi opisi, iz katerih smo razvili trenutnega, tudi enolični, prav tako pa morajo vsi predstavljati isto produkcijo gramatike. Poleg tega enolični opis ne sme razviti samega sebe. Enoličnost nam pove, ali lahko do izbranega opisa v stroju pridemo na natanko en način.

Enoličnost opisov znotraj posameznih stanj nato uporabimo za izračun tabele *LEFT*, s katero lahko določimo, kdaj vgrajeni analizator razreši konflikt v sintaksni tabeli, izvajanje pa lahko nadaljujemo po metodi *LL*. Tabelo *LEFT* izračunamo tako, da za vsak končni simbol ugotovimo, ali lahko v posameznem stanju izberemo natanko en enoličen opis.

Zadnja tabela, ki jo potrebujemo za izvajanje sintaksne analize po metodi *LLLR*, je tabela *BACKTRACK*. Ta vsebuje prehode med posameznimi enoličnimi opisi vse do začetnega opisa stroja. Uporablja se za popravljanje sklada sintaksne analize, ko končamo izvajanje z vgrajenim analizatorjem.

Preden lahko začnemo s sintaksno analizo vhodnega niza, je potrebno ustrezno prilagoditi gramatiko. Če začnemo izvajati vgrajeno sintaksno analizo takoj, ko naletimo na konflikt v sintaksni tabeli, obstaja verjetnost, da stroj za konfliktni simbol ni veljaven. Za primer lahko vzamemo gramatiko 2.4. V tej gramatiki bo v sintaksni tabeli zaradi leve rekurzije nastal



Slika 2.3: Stroj *LR(1)* za razrešitev konfliktnega simbola A.

konflikt na vmesnem simbolu A, zato moramo pojavitve simbola A v produkcijah simbolov A in B nadomestiti s strojem LR, ki analizo začne z opisom $A' \rightarrow \cdot\ A,\ a$, v produkciji simbola S pa z opisom $A' \rightarrow \cdot\ A,\ b$. Primer stroja za razrešitev konfliktnega simbola A prikazuje slika 2.3. Znaka ◯ in $\times$ ponazarjata enoličnost opisa.

Za razreševanje konfliktov potrebujemo tudi funkcijo *FSTFLW*, ki vrne množico simbolov, ki sledijo konfliktnemu vmesnemu simbolu. Če obstaja produkcija $S \rightarrow a\ A\ \omega$, kjer je simbol A konflikten, množica *FSTFLW(*A_S*)* vsebuje množico *FIRST(*ω*)*. Če ta vsebuje simbol ϵ, množico *FSTFLW(*A_S*)* nadomestimo s simboli iz množice *FOLLOW(S)*. Gramatika 2.4 v množici *FSTFLW(A)* vsebuje simbola a in b, odvisno od pojavitve simbola A.

V primeru opisa $A' \rightarrow A,\ a$ se pojavi težava, da stroja za simbol A' ni

mogoče zgraditi, saj analiza ne ločuje med simbolom a v produkciji $A \rightarrow A\ a$ in simbolom a, ki sledi začetnemu stanju. Posledično moramo v produkcijo dodati še sledilni simbol a. Tako produkcijo $B \rightarrow A\ a\ b$ spremenimo v produkcijo $B \rightarrow A'\ b$, kjer simbol A' predstavlja stroj z opisom $A' \rightarrow A\ a,\ b$, postopek pa ponavljamo, dokler niso vsi vgrajeni stroji veljavni.

Tabela 2.7: Tabela *LEFT* za simbol A v gramatiki 2.3.

stanje	simbol	opis
0	$	0
2	b	6
3	b	8
4	b	11

Primer tabel *LEFT* in *BACKTRACK* stroja *LR* za konfliktni simbol A v produkciji simbola S v gramatiki 2.4 prikazujeta tabeli 2.7 in 2.8. V tabeli *LEFT* lahko opazimo, da je končni simbol b ključen za razrešitev konflikta z vgrajenim analizatorjem.

Ko razrešimo vse konfliktne simbole v sintaksni tabeli in zgradimo stroje za vgrajeni analizator, lahko začnemo s sintaksno analizo po metodi *LLLR*. Krovni analizator deluje po metodi *LL*, zato tako struktura sklada kot tudi postopek izvajanja ostaneta enaka. Sintaksna analiza se razlikuje le za vmesne simbole, za katere smo pred začetkom izvajanja zgradili stroje *LR*.

Sintaksno analizo s krovnim analizatorjem metode *LLLR* izvajamo, dokler ne naletimo na konfliktni simbol, nato pa nadaljujemo z vgrajenim analizatorjem. Ta analizo začne s skladom, ki namesto simbola $ vsebuje konfliktni vmesni simbol v stanju 0, postopek izvajanja sintaksne analize pa poteka normalno po metodi *LR*. Ker metoda tvori desno sled izpeljave, moramo sled izpeljave ustrezno prilagoditi [6].

Kljub temu da lahko sintaksno analizo po metodi *LR* izvajamo, dokler ta ne izbere akcije za sprejem, želimo metodo *LLLR* večino časa izvajati s krovnim analizatorjem, zato moramo vgrajeni analizator ustaviti takoj, ko

Tabela 2.8: Tabela *BACKTRACK* za simbol A v gramatiki 2.3.

trenutno		prejšnje	
stanje	opis	stanje	opis
1	1	0	0
1	2	1	1
1	3	1	1
2	6	1	3
3	8	1	1
3	9	1	2
4	11	3	9

razrešimo konflikt. To storimo z uporabo tabel *LEFT* in *BACKTRACK*. Na vsakem koraku sintaksne analize po metodi *LR* zato najprej preverimo, ali tabela *LEFT* vsebuje enoličen opis za vhodni simbol v trenutnem stanju stroja. V tem primeru se nahajamo v stanju, do katerega lahko pridemo na en sam način, kar pomeni, da smo uspešno razrešili konflikt, izvajanje pa lahko nadaljujemo po metodi *LL*. Ker obstaja verjetnost, da smo izvajanje zaključili sredi opisa oziroma produkcije, moramo preostale simbole dodati na sklad krovnega analizatorja, kar storimo z uporabo tabele *BACKTRACK*. Najprej na sklad dodamo preostale simbole trenutnega opisa, nato pa se s tabelo vračamo po opisih vse do začetnega opisa stroja ter na sklad dodamo vse preostale simbole v različnih produkcijah gramatike. Pred tem je potrebno s sklada najprej odstraniti razrešeni konfliktni simbol.

Čeprav sintaksna analiza po metodi *LLLR* deluje na vseh gramatikah, je hitrost izvajanja odvisna od števila konfliktov v gramatiki opisanega jezika. Zgodi se lahko, da mnogo strojev ni veljavnih, zato je potrebno veliko časa nameniti razreševanju konfliktnih simbolov. Druga težava, s katero se lahko srečamo, je pojavitev konflikta na vrhu drevesa oziroma začetku gramatike. V takem primeru se večina sintaksne analize izvede z vgrajenim analizatorjem — to sicer ne vpliva na hitrost ali pravilnost izvajanja, vendar

je uporaba metode *LLLR* na taki gramatiki skoraj brez pomena. Posledično je pred začetkom izvajanja sintaksne analize priporočljivo gramatiko ročno prilagoditi, da lahko izkoristimo prednosti metode *LLLR*. Najbolj koristna sprememba je odstranitev rekurzivnih produkcij, vendar moramo biti pozorni, da ne spremenimo pomena in zaporedja matematičnih izrazov.

Tabela 2.9: Primer razreševanja leve rekurzije v gramatiki A.1.

definitions	→	definition
definitions	→	definitions ; definition
definitions	→	definition definitions_{opt}
definitions_{opt}	→	; definition definitions_{opt}
definitions_{opt}	→	ϵ

Primer prilagojene gramatike za hitrejše izvajanje sintaksne analize in manjše število konfliktnih simbolov prikazuje gramatika programskega jezika *Prev*, ki je opisana v dodatku A. Osnovna gramatika takoj na začetku definira vmesni simbol *definitions*, ki vsebuje levo rekurzijo. Ker metoda *LL* leve rekurzije ne podpira, bi morali skoraj celotno vhodno datoteko analizirati z vgrajenim analizatorjem. Zato smo v gramatiki odpravili levo rekurzijo v vseh produkcijah, ki ne vplivajo na obliko sintaksnega drevesa. Tako smo z manjšo spremembo močno zmanjšali obseg gramatike, ki jo analiziramo z uporabo metode *LR*. Enako smo storili za vmesna simbola *components* in *parameters*. Primer prilagoditve gramatike prikazuje tabela 2.9.

Poglavje 3

Implementacija

V tem poglavju bomo opisali vse podrobnosti implementacije sintaksne analize po metodi *LLLR*. To vključuje vse podatkovne strukture, implementacije metod, opisanih v poglavju 2, in optimizacije, s katerimi smo zagotovili čim hitrejše izvajanje in manjšo porabo pomnilnika. Prav tako bomo opisali težave oziroma omejitve, s katerimi smo se srečali, in navedli rešitve zanje.

Podrobno bomo opisali tudi obliko datotek za opis gramatik ter implementacijo leksikalne analize za pretvorbo vhodne datoteke v simbole z uporabo regularnih izrazov in omejitve le-teh. Poglavje bomo namenili tudi implementaciji gradnje strojev *LR*, ki predstavlja največji del časovne zahtevnosti izvajanja metode *LLLR*.

Izvorna koda implementacije je z licenco *MIT* objavljena na spletnem naslovu `https://github.com/KarboniteKream/syn`.

3.1 Programski jezik Rust

Program za sintaksno analizo smo v tem magistrskem delu implementirali z uporabo sistemskega programskega jezika *Rust*, ki ga je leta 2010 pod okriljem *Mozille* predstavil Graydon Hoare. Kljub temu da je bil programski jezik komaj predstavljen, je hitro postal popularen, saj je obljubljal hitro izvajanje in sintakso, podobno programskemu jeziku *C++*. Prav tako ima

edinstveno lastnost, da med prevajanjem izvorne kode zagotavlja varnost implementacije in dostopov do pomnilnika, kar je lahko velika težava pri ostalih nizkonivojskih jezikih.

Programski jezik *Rust* se je čez leta pogosto spreminjal, kar je bila ena večjih slabosti, vendar so avtorji leta 2015 z izidom različice 1.0 določili končno sintakso jezika. Leta 2018 je izšla druga različica programskega jezika, ki se je osredotočala na produktivnost razvijalcev. Ta je dodala nekaj programskih konstruktov iz drugih visokonivojskih programskih jezikov, prav tako pa je bil predstavljen nov sistem za preverjanje veljavnosti pomnilnika.

Ključna lastnost, ki *Rust* loči od drugih programskih jezikov, je koncept lastništva spremenljivk oziroma vrednosti v pomnilniku (angl. *ownership*), ki zagotavlja zanesljivo izvajanje kode brez potrebe po ročnem nadzoru nad uporabo dinamičnega pomnilnika ali sistemu za samodejno sproščanje pomnilnika (angl. *garbage collection*). Ta lastnost sicer močno vpliva na hitrost prevajanja, prav tako pa obstaja verjetnost, da prevajalnik zavrne pomnilniško veljavno izvorno kodo. Kljub manjšim pomanjkljivostim, ki so bile s časom razrešene, je uporabnost sistema lastništva prevladala.

V sistemu lastništva ima lahko vsaka vrednost v pomnilniku enega samega lastnika. V splošnem velja, da si neka spremenljivka lasti vrednost na nekem pomnilniškem naslovu. Vrednosti na tem naslovu ne moremo spreminjati, razen če pri definiciji spremenljivke uporabimo ključno besedo `mut`. Ostale spremenljivke lahko do istega pomnilniškega naslova dostopajo le, če si vrednost izposodijo (angl. *borrow*).

Izposoja je mogoča le pod naslednjimi pogoji:

- obstajati sme največ ena spremenljiva (angl. *mutable*) izposoja vrednosti,
- obstajati sme neomejeno število nespremenljivih (angl. *immutable*) izposoj vrednosti in
- v nekem trenutku ne sme obstajati tako spremenljiva kot tudi nespremenljiva izposoja vrednosti.

Čeprav je lastništvo najpomembnejša lastnost programskega jezika *Rust*, velja za najzahtevnejši del učenja tega programskega jezika, saj prevajalnik ne dopušča številnih pristopov, ki so veljavni v drugih programskih jezikih. Kljub zahtevnejšemu in počasnejšemu pisanju programske kode je razvoj programov preprostejši, saj večino napak, ki se ponavadi pojavijo med izvajanjem programa, odkrijemo že med samim prevajanjem.

3.2 Struktura programa

Strukturo in implementacijo sintaksne analize po metodah *LL*, *LR* in *LLLR* smo želeli ohraniti preprosto, s čimer smo zagotovili razširljivost in lažje vzdrževanje izvorne kode. Program smo razdelili na štiri samostojne module, ki si poleg podatkovnih struktur med seboj ne delijo kode:

- `grammar`, ki vsebuje simbole, produkcije ter branje in preverjanje veljavnosti gramatike,
- `automaton`, ki skrbi za gradnjo strojev *LR* in izračun podatkovnih tabel za sintaksno analizo,
- `lexer`, ki izvaja leksikalno analizo, s katero vhodno datoteko razdelimo na simbole, in
- `parser`, ki izvaja sintaksno analizo po metodi *LLLR* oziroma po metodah *LL* in *LR*.

Zaradi kompleksnosti stroja *LR* in velikega števila stanj smo morali paziti na velikost posameznih podatkovnih struktur, zato smo se namesto podvajanja podatkov raje sklicevali na deljene elemente. Na primer, podatkovna struktura za gramatiko vsebuje seznam vseh simbolov, posamezne produkcije pa se z zaporednimi številkami sklicujejo na elemente v tem seznamu. Tako smo se uspešno izognili podvajanju nizov znakov.

Največja podatkovna struktura v implementiranem programu predstavlja stroj za sintaksno analizo po metodi *LR*. Kljub temu da so opisi odvisni le od

stanja, v katerem se pojavijo, se v sklopu celotnega stroja začno hitro ponavljati. Kot velja v podatkovni strukturi za gramatiko, se tudi tu opisi znotraj stanj sklicujejo na elemente v globalnem seznamu. Tako sicer zmanjšamo porabo pomnilnika, vendar ta pristop poslabša lokalnost podatkov (angl. *data locality*) in zahteva dodatne dostope do pomnilnika.

Vsaka izmed pomembnejših podatkovnih struktur vsebuje nepredznačeno celo število `id`, ki je edinstveno za posamezno podatkovno vrsto. Vse podatkovne strukture hranimo znotraj vektorjev. Ker elementov nikoli ne odstranimo, polje `id` predstavlja tudi položaj elementa v vektorju.

Za podatkovne strukture, ki predstavljajo povezave med več elementi, kot je na primer sintaksna tabela, smo uporabiti zgoščevalne tabele (angl. *hash table*, *hash map*). Ker programski jezik *Rust* podpira *terice* (angl. *tuple*), smo se lahko izognili gnezdenju zgoščevalnih tabel.

Ker želimo zagotoviti determinističnost algoritma in sintaksnega drevesa, vse podatkovne strukture vedno uredimo. S tem zagotovimo, da imajo vse produkcije, stanja in opisi vedno enako zaporedno številko.

3.3 Gramatika

Za predstavitev kontekstno neodvisnih gramatik smo uporabili datoteke v obliki *TOML* (angl. *Tom's Obvious, Minimal Language*), ki združuje enostavnost oblike *INI* in zmogljivost oblike *JSON*. Tako lahko gramatiko hranimo v uporabniku prijazni obliki, hkrati pa vključimo vse potrebne metapodatke, ki jih za izvajanje potrebujeta leksikalna in sintaksna analiza. Datoteko za opis gramatike sestavljajo metapodatki, produkcije gramatike, regularni izrazi, s katerimi lahko vhodno datoteko razdelimo na posamezne simbole, in napotki za razreševanje konfliktov v metodi *LR*, kot lahko vidimo v izseku 3.1.

Metapodatki vključujejo ime in kratek opis gramatike ter začetni simbol. Če slednji ni definiran, sintaksna analiza prične s prvim simbolom v datoteki. Metapodatkom sledi odsek `[rules]`, v katerem so opisane vse pro-

dukcije gramatike v poljubnem zaporedju. Posamezni vmesni simbol lahko zapišemo v obliki `A = ["a B c", "d A"]`, kjer vsak niz simbolov predstavlja eno produkcijo vmesnega simbola. Produkcijo $A \to \epsilon$ lahko zapišemo kot `A = ""`.

Izsek 3.1: Primer gramatike 2.3 v obliki *TOML*.

```
1  name = "dragon"
2  description = "An example from the Dragon Book."
3  start_symbol = "E"
4
5  [rules]
6  E = [
7    "E + T",
8    "T",
9  ]
10
11 T = [
12   "T * F",
13   "F"
14 ]
15
16 F = [
17   "( E )",
18   "int",
19 ]
20
21 [tokens]
22 int = "[0-9]+"
23
24 [ignore]
25 whitespace = "[ \t\r\n]*"
26
27 [actions]
28 F = "shift"
```

Simboli za leksikalno analizo so opisani v odsekih `[tokens]` in `[ignore]`. Odsek `[tokens]` vsebuje regularne izraze za posamezne končne simbole, ki se pojavijo v odseku `[rules]`. Vse končne simbole, ki nimajo določenih

regularnih izrazov, leksikalna analiza ujema dobesedno. Odsek `[ignore]` vsebuje regularne izraze za simbole, ki niso del gramatike, kot so na primer belo besedilo in komentarji. Regularni izraz, ki se v odseku pojavi višje, ima prednost v leksikalni analizi. Poleg regularnih izrazov lahko uporabimo tudi seznam nizov znakov, ki simbole ujemajo dobesedno, tako kot končni simboli v odseku `[rules]`. Ta pristop lahko uporabimo za ključne besede za podatkovne vrste spremenljivk, kjer ne želimo uporabiti dodatnih produkcij.

Zadnji odsek v datoteki za opis gramatike je `[actions]`. Ta določa prednost akcij v primeru konflikta v tabeli *ACTION* za sintaksno analizo po metodi *LR*. Med analizo konflikti sicer niso dovoljeni, vendar obstajajo primeri, kjer se konfliktu brez prilagajanja gramatike ne želimo izogniti. Najbolj znan primer takega konflikta se imenuje *dangling else problem*, ki se pojavlja v programskih jezikih, kot sta *C* in *Java* [17]. Težava se pojavi, ko naletimo na zaporedje stavkov `if`, `if` in `else`, saj sintaksna analiza po metodi *LR* na osnovi običajnega zapisa strukture pogojnega stavka ni sposobna določiti, kateremu stavku `if` pripada stavek `else`. Večina programov za sintaksno analizo v tem primeru izbere akcijo za pomik. Dovoljeni vrednosti v tem odseku sta `shift` in `reduce`.

Pred začetkom sintaksne analize je potrebno preveriti tudi veljavnost gramatike in produkcij. V gramatiki mora obstajati izbrani začetni simbol, prav tako pa morajo biti vsi vmesni simboli dosegljivi iz le-tega. Poleg tega je potrebno za vsak vmesni simbol zagotoviti, da vsaj ena produkcija ni rekurzivna. Nekoliko zahtevnejše je preverjanje končnosti (angl. *realizability*), saj moramo zagotoviti, da lahko vsak vmesni simbol v celoti razvijemo. V prvem koraku lahko določimo končnost tistim vmesnim simbolom, za katere obstaja vsaj ena produkcija s samimi končnimi simboli. Nato je potrebno preveriti končnost vseh vmesnih simbolov, za katere obstaja vsaj ena produkcija s končnimi in vmesnimi simboli, za katere smo že določili končnost. Ta postopek ponavljamo, dokler ne določimo končnosti vseh vmesnih simbolov. V primeru, da vsaj ena od naštetih lastnosti ne velja, sintaksne analize po izbrani gramatiki ni mogoče izvesti.

Implementacija podatkovne strukture za gramatiko skrbi tudi za izračun množic *FIRST* in *FOLLOW* za posamezne simbole. Ker je izračun le-teh časovno zahteven, funkcije pa se kličejo zelo pogosto, množici izračunamo le enkrat, vsi nadaljnji klici funkcij pa vrnejo že izračunan rezultat. Zaradi omejitev programskega jezika *Rust* smo za hranjenje rezultatov uporabili strukturo `RefCell`, ki dovoljuje spreminjanje vrednosti v nespremenljivi strukturi `Grammar`. Izračun množic smo implementirali po postopku, opisanem v poglavju 2.3, vendar smo morali biti pazljivi pri izračunu množice *FIRST(A)* v produkcijah oblike $A \rightarrow \alpha\ A\ \beta$. Da se lahko izognemo neskončni zanki, moramo simbol A v telesu produkcije preskočiti, dokler ne pregledamo vseh drugih produkcij simbola A. Če po pregledu preostalih produkcij množica *FIRST(A)* še vedno vsebuje simbol ϵ, s produkcijo nadaljujemo po opisanem postopku, sicer vrnemo trenutni rezultat.

3.4 Leksikalna analiza

Leksikalna analiza (angl. *lexical analysis*) je postopek, ki razdeli vsebino vhodne datoteke na posamezne simbole oziroma žetone (angl. *tokens*), ki predstavljajo poljubne končne simbole gramatike. Vhodni simbol smo implementirali s strukturo `Token`. Ta poleg končnega simbola vsebuje tudi leksem (angl. *lexeme*), ki predstavlja niz znakov v vhodni datoteki, iz katerega smo ustvarili simbol. Leksikalna analiza je ponavadi v preprostih prevajalnikih implementirana z deterministčnim končnim avtomatom (angl. *deterministic finite automaton*) ali pa uporablja zunanje programe, kot sta *lex* in *flex*.

V sklopu tega magistrskega dela smo leksikalno analizo ročno implementirali s pomočjo regularnih izrazov. Kot smo opisali v poglavju 3.3, vsak končni simbol obravnavamo dobesedno (na primer za posebne znake in ključne besede programskega jezika) oziroma z določenim regularnim izrazom. Analizo smo želeli implementirati čim preprosteje, zato morajo regularni izrazi omogočati delno ujemanje. Tako smo lahko leksikalno analizo implementirali brez obravnavanja posebnih primerov in premikanja po vhodni datoteki. Na

primer, za iskanje nizov znakov lahko uporabimo izraz `/"[A-Za-z]*("|$)/`, ki zazna delno ujemanje na nizih `"` in `"abc` ter popolno ujemanju na nizu `"abc"`. Pri definiranju regularnih izrazov moramo biti pazljivi, da na koncu izraza vedno uporabimo zajemni podizraz (angl. *capturing group*), saj leksikalna analiza to lastnost potrebuje za delno ujemanje.

Algoritem 3.1 Psevdokoda leksikalne analize.

```
 1: tokens ← []
 2: text ← ""
 3: current_match ← None
 4: last_match ← None
 5:
 6: repeat
 7:     text ← text + next()
 8:     current_match ← match(text)
 9:     if not current_match and not last_match then
10:         return Error
11:     end if
12:     if not current_match then
13:         if current_match ≠ None then
14:             tokens ← tokens + last_match
15:         end if
16:         text ← text - last_match
17:         last_match ← None
18:     else if current_match.full_match then
19:         last_match ← current_match
20:     end if
21: until EOF
22:
23: if not last_match then
24:     return Error
25: end if
26:
27: return tokens
```

Leksikalna analiza podpira tudi delno ujemanje končnih simbolov brez uporabe regularnih izrazov. Zaradi enostavnejše implementacije program

podpira le ujemanje na posameznih vrsticah datoteke, zato nekateri konstrukti, kot so večvrstični komentarji, niso podprti.

Algoritem 3.1 opisuje implementacijo leksikalne analize, med katero vhodno datoteko beremo znak po znak, dokler se prebrani niz znakov ujema delno ali v celoti. Ko pridemo do stanja, v katerem se vhodni niz ne ujema z nobenim regularnim izrazom oziroma končnim simbolom, zadnje popolno ujemanje spremenimo v simbol, leksikalno analizo pa nadaljujemo od konca zadnjega popolnega ujemanja. Regularni izrazi, definirani v odseku `[ignore]`, predstavljajo simbol ϵ, ki ga opisani algoritem preskoči. Med iskanjem simbolov upoštevamo tudi vrstico in stolpec začetka ter konca le-teh, da lahko v primeru napake uporabniku sporočimo, kje v datoteki se ta nahaja.

Če imamo med leksikalno analizo vedno popolno ujemanje, je časovna zahtevnost izvajanja enaka $O(n * m)$, kjer n predstavlja dolžino vhodnega niza, m pa dolžino najdaljšega končnega simbola gramatike. V najslabšem primeru je časovna zahtevnost enaka $O(n^2 * m)$.

3.5 Gradnja strojev LR

Najzahtevnejši del izvajanja sintaksne analize po metodi *LLLR* je gradnja strojev *LR*, saj moramo sestaviti vsa možna stanja, katerih količina je odvisna od zahtevnosti izbrane gramatike.

Izračun vseh opisov v posameznem stanju načeloma ni zahteven, saj je potrebno zgolj prenesti opise iz prejšnjega stanja glede na simbol, po katerem izvajamo prehod. Zatem moramo v ovojnico dodati vse produkcije vmesnih simbolov, ki v opisih stanja neposredno sledijo piki. Kompleksnost se pojavi pri implementaciji, saj moramo ohranjati edinstvene zaporedne številke stanj in opisov za potrebe prehodov med elementi stroja *LR*. Vsakič, ko razvijemo nov opis, moramo preveriti, ali tak opis v stroju že obstaja, in ustrezno popraviti zaporedno številko ter vse prehode med opisi. Del zahtevnosti nastane tudi pri izračunu enoličnosti, saj v primeru, da opis razvije samega sebe, ta ni več enoličen. V tem slučaju je potrebno rekurzivno popraviti vse

že razvite opise v trenutnem stanju.

Med implementacijo stroja smo se morali odločiti, katero podatkovno strukturo želimo uporabiti za hranjenje elementov, kot so že razvita stanja in opisi. Potrebovali smo nekaj ključnih lastnosti:

1. vsak element znotraj podatkovne vrste potrebuje edinstveno zaporedno številko,
2. med primerjavo elementov te zaporedne številke ne želimo upoštevati,
3. obstoječi element želimo poiskati v konstantnem času,
4. do posameznega elementa želimo dostopati preko zaporedne številke in
5. za hranjenje želimo uporabiti zgolj eno podatkovno strukturo.

Prvi dve lastnosti smo zagotovili z lastno zgoščevalno funkcijo, ki ne upošteva polja `id`, kar omogoča primerjanje elementov z različnimi zaporednimi številkami. Pri definiranju zgoščevalne funkcije smo morali biti pazljivi na implementacijo ostalih primerjalnih funkcij (lastnosti `Eq`, `Ord` in `Hash`), katerih rezultati se morajo za podano podatkovno strukturo med seboj ujemati. Za hranjenje elementov smo uporabili podatkovno strukturo *IndexMap* oziroma *OrderMap*, ki ohranja zaporedje vstavljenih elementov, vendar pod pogojem, da elementov iz strukture nikoli ne odstranimo. Tako lahko do posameznih elementov dostopamo preko zgoščene vrednosti ali zaporedne številke vstavljenega elementa (polje `id`), s tem pa smo zagotovili preostale tri lastnosti.

Med razvijanjem novega stanja velja pravilo, da se posamezen opis lahko pojavi le enkrat, ne glede na enoličnost. Ker lahko v programskem jeziku *Rust* za poljubno strukturo definiramo zgolj eno zgoščevalno funkcijo, bi morali obstoječi opis v strukturi poiskati dvakrat, kar bi vplivalo tako na hitrost izvajanja kot tudi na kompleksnost programske kode. Zaradi te lastnosti smo v strukturo dodali podvojeni opis, ki se razlikuje le v polju `unique`. Ker obenem razvijamo zgolj eno stanje, število opisov v posameznem stanju pa je majhno, višja poraba pomnilnika ne predstavlja večjih težav.

Prav tako smo opazili, da se opisi in njihove lastnosti med razvijanjem stanja neprestano spreminjajo, nekatere pa tudi zavržemo, zato lahko začasno uporabimo poljubne zaporedne številke. Ko določimo končno stanje, ki ga stroj še ne vsebuje, vse zaporedne številke opisov ustrezno popravimo. Če stroj razviti opis že vsebuje, uporabimo obstoječo zaporedno številko, sicer ustvarimo novo. Ko stanje dodamo v stroj, moramo ustrezno prilagoditi vse prehode iz predhodnega stanja.

Zadnji korak pri gradnji stroja *LR* je izračun tabel za sintaksno analizo. Kot navajamo v poglavju 2.4, potrebujemo tabele *ACTION*, *GOTO*, *LEFT* in *BACKTRACK*, za izračun le-teh pa uporabimo vsa stanja in opise stroja ter prehode med njimi. Tabelo *ACTION* izračunamo tako, da za vsak prehod med stanji po končnem simbolu v tabelo dodamo akcijo za pomik. Nato dodamo še akcijo za prevedbo oziroma sprejem za vse opise, ki so do konca razvili podano produkcijo gramatike. Če za določeno stanje in simbol v tabeli *ACTION* že obstaja akcija, nastane konflikt *Shift/Reduce*, kar pomeni, da stroj *LR* za izbrano gramatiko ni veljaven. To lastnost uporabimo pri gradnji strojev za vgrajeni analizator, kar bomo podrobneje opisali v poglavju 3.6. Za primer tabele *ACTION* lahko uporabimo stroj na sliki 2.2. Iz stanja 2 obstaja prehod po simbolu `+` v stanje 8, zato lahko v tabelo dodamo akcijo *Shift(8)*. Stanje 6 vsebuje tri opise, kjer se pika nahaja na koncu, zato lahko za simbole \$, `+` in `*` v tabelo dodamo akcijo *Reduce(6)*, kjer število 6 predstavlja zaporedno številko produkcije. Enako lahko storimo v stanju 7, vendar opis predstavlja začetno produkcijo, zato v tabelo po simbolu ϵ dodamo akcijo *Accept*. Tabelo *GOTO* izračunamo podobno, vendar uporabimo prehode po vmesnih simbolih. Primer takega prehoda se nahaja med stanjema 1 in 2 po simbolu E.

Zatem moramo izračunati tabelo *LEFT*, s katero določimo, ali lahko v poljubnem stanju z vhodnim simbolom izberemo natanko en enoličen opis, posledično pa zaključimo izvajanje vgrajenega analizatorja. To storimo tako, da v posameznem stanju za vsak opis oblike $A \to \alpha \cdot \beta,\ c$ izračunamo množico *FIRST(βc)*. Enolični opis O in simbol a lahko dodamo v tabelo *LEFT* le

pod pogojem, da se simbol a pojavi zgolj v množici *FIRST(βc)* opisa O. Za primer lahko vzamemo stroj na sliki 2.3. Simbol `b` sledi le enoličnemu opisu 8 v stanju 3, zato opis in simbol lahko dodamo v tabelo *LEFT*. Ker simbol `a` sledi dvema opisoma v stanju, le-teh v tabelo ne smemo dodati.

Tabelo *BACKTRACK*, s katero posodobimo sklad sintaksne analize po metodi *LL*, izračunamo tako, da vanjo dodamo vse prehode med enoličnimi opisi stroja *LR*. Na sliki 2.3 lahko vidimo, da smo opis $A \rightarrow A\ a\ \cdot,\ b$ v stanju 4 s premikom po simbolu `a` razvili iz opisa $A \rightarrow A\ \cdot\ a,\ b$ v stanju 3, zato ta prehod lahko dodamo v tabelo *BACKTRACK*.

3.6 Sintaksna analiza

Ko zaključimo leksikalno analizo, lahko za vhodne simbole začnemo sintaksno analizo. Najprej je potrebno izračunati sintaksno tabelo za analizo po metodi *LL*, ki smo jo opisali v poglavju 2.3. V metodi pregledamo vse simbole v posameznih produkcijah gramatike in glede na množici *FIRST* in *FOLLOW* zgradimo sintaksno tabelo. V primeru, da v tabeli nastane konflikt, vrnemo vse konfliktne simbole.

Za podane konfliktne vmesne simbole je potrebno ustvariti nove produkcije, ki označujejo začetek analize po metodi *LR*. To storimo tako, da v gramatiki poiščemo vse pojavitve konfliktnega simbola A, ustvarimo produkcije $A'_i \rightarrow A_i$, nato pa zgradimo stroje *LR*, ki sintaksno analizo začnejo z opisom $A'_i \rightarrow \cdot A_i$, *FSTFLW($A_i$)*. V primeru, da podani stroj ni veljaven, v produkcijo dodamo simbol, ki neposredno sledi simbolu A_i, in poskusimo znova. Če smo v novo produkcijo dodali vse simbole iz obstoječe produkcije, stroj pa še vedno ni veljaven, produkcijo odstranimo ter glave obstoječe produkcije dodamo med konfliktne simbole. Ta postopek ponavljamo, dokler niso vsi vgrajeni stroji veljavni. Ker se nove produkcije med seboj razlikujejo le v zaporedju simbolov, lahko stroje uporabimo večkrat.

Ker je gradnja strojev za vse konfliktne simbole časovno zahtevna, smo se morali poslužiti nekaj bližnjic. Gramatika se med izvajanjem programa

ne spreminja, zato vemo, da izbranega vmesnega simbola po razrešitvi vseh konfliktov v naslednji iteraciji ni potrebno ponovno obravnavati.

V prvi implementaciji razreševanja konfliktov v sintaksni tabeli smo nove vmesne simbole vstavili neposredno v produkcije obstoječe gramatike. Pri tem smo zamenjave izvedli v obratnem vrstnem redu, s čimer smo se izognili preračunavanju položajev simbolov. Pri kompleksnejših gramatikah, kot je na primer gramatika programskega jezika *Java*, se je izkazalo, da spreminjanje gramatike vpliva na veljavnost že zgrajenih strojev, zato smo morali pristop delno prilagoditi. Končna implementacija namesto spreminjanja gramatike uporablja seznam zamenjav, katerega med analizo preverimo za vsak vmesni simbol v produkciji. Ker med razreševanjem konfliktov gramatike ne spreminjamo, obstoječi stroji ostanejo veljavni, zato jih pred začetkom sintaksne analize ni potrebno ponovno zgraditi.

Med implementacijo smo sklepali, da če obstaja produkcija $B \to \alpha A_i \beta$ ter sta vmesna simbola A in B oba konfliktna, stroja za simbol A_i ni potrebno zgraditi, saj analiza po metodi LL tega simbola ne bo nikoli dosegla. Kasneje se je izkazalo, da lahko vgrajeni analizator stroj ustavi sredi take produkcije, posledično pa mora krovni analizator nadaljevati s simbolom A_i.

Za sintaksno analizo po metodi LR ne potrebujemo celotnih strojev, zato lahko ohranimo le opise, tabele $ACTION$, $GOTO$, $LEFT$ in $BACKTRACK$ ter prihranimo na porabi pomnilnika. Ko razrešimo vse konflikte, ponovno izračunamo sintaksno tabelo ter pričnemo s sintaksno analizo.

3.6.1 Krovni analizator

Kot smo opisali v poglavju 2.5, sintaksno analizo po metodi $LLLR$ izvajamo v dveh korakih. Najprej začnemo z analizo s krovnim analizatorjem (angl. *backbone parser*) po metodi LL, ki uporablja eno samo sintaksno tabelo. Začnemo s skladom, ki vsebuje le simbol S', ki označuje začetek datoteke oziroma vhodnega niza. Na vsakem koraku sintaksne analize preberemo vhodni simbol, z vrha sklada vzamemo trenutni simbol ter preverimo, ali sintaksna tabela vsebuje vnos za podana simbola. To pomeni, da se nahajamo v sta-

nju, v katerem lahko izberemo naslednjo produkcijo gramatike, zato simbol na vrhu sklada nadomestimo s simboli v izbrani produkciji, to pa dodamo v sled izpeljave. Na sklad nikdar ne dodamo simbola ϵ. Ker se vhodni niz vedno začne s simbolom \$, bomo na prvem koraku vedno izbrali produkcijo $S' \rightarrow \$\ S\ \$$.

Algoritem 3.2 Psevdokoda krovnega analizatorja po metodi *LL*.

```
 1: rules ← []
 2: stack ← [S']
 3: input ← get_tokens()
 4:
 5: repeat
 6:     symbol ← stack.top()
 7:     token ← input.next()
 8:     if rule ∈ parse_table[symbol, token] then
 9:         stack ← stack - symbol
10:         stack ← stack + (rule.body - ϵ)
11:         rules ← rules + rule
12:     else if symbol ∈ conflicts then
13:         rules ← embedded_parser()
14:     else if symbol = token then
15:         stack ← stack - symbol
16:         input ← input - token
17:     else
18:         return Error
19:     end if
20: until stack = ϵ or input = ϵ
21:
22: if stack ≠ ϵ or input ≠ ϵ then
23:     return Error
24: end if
25:
26: return rules
```

Če sintaksna tabela ne vsebuje produkcije za podana simbola, moramo preveriti, ali za vmesni simbol na vrhu sklada obstaja stroj *LR*. V takem primeru sintaksno analizo nadaljujemo z vgrajenim analizatorjem, opisanem

v naslednjem poglavju. V nasprotnem primeru lahko predpostavimo, da sta simbola končna. Če sta simbola enaka, ju odstranimo z vhoda in sklada ter nadaljujemo v naslednji iteraciji, sicer sintaksno analizo prekinemo.

V kolikor po končani oziroma prekinjeni sintaksni analizi vhod vsebuje vsaj en simbol, vhodna datoteka ni veljavna po izbrani gramatiki in vsebuje sintaksno napako. Uporabniku zato sporočimo, kateri vhodni simbol je neveljaven in kje v datoteki se nahaja ter zaključimo izvajanje programa. Natančnejšo implementacijo krovnega analizatorja opisuje algoritem 3.2.

3.6.2 Vgrajeni analizator

Ko v sintaksni tabeli naletimo na vmesni simbol, za katerega smo na začetku sintaksne analize odkrili konflikt, lahko pričnemo z vgrajenim analizatorjem (angl. *embedded parser*) po prilagojeni metodi *LR*, ki jo izvajamo, dokler ne razrešimo konflikta. Sintaksna analiza z vgrajenim analizatorjem se od metode *LL* razlikuje v tem, da sklad poleg simbola vsebuje tudi stanje, v katerem se stroj trenutno nahaja. Za premikanje po stroju uporabimo tabeli *ACTION* in *GOTO*.

Pred začetkom izvajanja sintaksne analize na vhod dodamo simbol $. Tako poenostavimo implementacijo gradnje stroja in se izognemo obravnavanju posebnih primerov, saj lahko isto kodo uporabljamo tako za vgrajeni analizator kot tudi za samostojno analizo po metodi *LR*. Za sintaksno analizo poleg prej navedenih tabel potrebujemo tudi tabeli *LEFT* in *BACKTRACK*. Uporaba teh tabel je glavna razlika med vgrajenim analizatorjem in navadno implementacijo sintaksne analize po metodi *LR*.

Ko začnemo s sintaksno analizo, z vrha sklada vzamemo vmesni simbol in stanje stroja, preberemo naslednji vhodni simbol ter iz tabele *ACTION* izberemo ustrezno akcijo. Če je izbrana akcija enaka *Shift(s)*, z vhodnega niza odstranimo prebrani simbol, stroj premaknemo v stanje s ter izvajanje analize nadaljujemo z naslednjim vhodnim simbolom. V primeru akcije *Reduce(r)* lahko sintaksna analiza izbere produkcijo r. To pomeni, da moramo z vrha sklada odstraniti simbole, ki so del produkcije, pri čemer moramo biti

pozorni, da preskočimo simbol $, ki se uporablja le med gradnjo stroja. Nato glede na razviti vmesni simbol oziroma glavo produkcije gramatike s tabelo *GOTO* stroj premaknemo v naslednje stanje.

Algoritem 3.3 Psevdokoda vgrajenega analizatorja po metodi *LR*.

```
 1: rules ← []
 2: input ← $ + input
 3:
 4: repeat
 5:     state ← stack.top()
 6:     token ← input.next()
 7:     if item ← left_table[state, token] then
 8:         rules ← item.rule + rules
 9:         stack.restore()
10:         stack ← stack + get_remaining_symbols()
11:         return rules
12:     end if
13:     action ← action_table[state, token]
14:     if action = Shift(state) then
15:         stack ← stack + state
16:         input ← input - token
17:     else if action = Reduce(rule) then
18:         rules ← rule + rules
19:         stack ← stack - (rule.body - $)
20:         next_state ← goto_table[stack.top()]
21:         stack ← stack + next_state
22:     else if action = Accept(rule) then
23:         return rule + rules
24:     else
25:         return Error
26:     end if
27: until stack = ϵ or input = ϵ
28:
29: return rules
```

Tretja akcija, ki jo lahko izberemo iz tabele *ACTION*, je *Accept(r)*. Ta nakazuje, da smo izbrali produkcijo r in uspešno zaključili sintaksno ana-

lizo po metodi *LR*, izvajanje pa lahko nadaljujemo s krovnim analizatorjem. Pred tem moramo z vrha sklada odstraniti simbole izbrane produkcije. Preden lahko nadaljujemo z metodo *LL*, moramo desno sled izpeljave ustrezno spremeniti v levo. Če tabela *ACTION* za trenutno stanje stroja in vhodni simbol ne vsebuje akcije, vhodna datoteka ni veljavna glede na izbrano gramatiko. Neveljavna je tudi v primeru, če iz vhoda preberemo vse simbole, preden stroj izvede akcijo za sprejem. Natančnejšo implementacijo vgrajenega analizatorja opisuje algoritem 3.3.

Izvajanje sintaksne analize po metodi *LLLR* želimo v večini izvesti po metodi *LL*, metodo *LR* pa uporabiti le za razrešitev konfliktov. Zaradi tega moramo izvajanje vgrajenega analizatorja zaključiti takoj, ko uspešno izberemo produkcijo, po kateri lahko analizo nadaljujemo s krovnim analizatorjem. V nasprotnem primeru bi metodo *LR* izvajali vse do konca vhodnega niza. Da lahko dosežemo ta cilj, moramo uporabili tabeli *LEFT* in *BACKTRACK*. V vsakem koraku izvajanja vgrajenega analizatorja poskusimo v tabeli *LEFT* poiskati enoličen opis stroja za trenutno stanje in vhodni simbol. Ker lahko stroj takšen opis doseže na zgolj en način, nam prisotnost enoličnega opisa nakaže, da smo uspešno razrešili konflikt, posledično pa lahko izvajanje nadaljujemo po metodi *LL*. Če tak opis v trenutnem stanju ne obstaja, izvajanje nadaljujemo po metodi *LR*.

Preden lahko izvajanje nadaljujemo s krovnim analizatorjem, moramo ustrezno posodobiti sklad sintaksne analize. Tega moramo obnoviti na stanje pred začetkom izvajanja vgrajenega analizatorja, z vrha pa odstraniti razrešeni konfliktni vmesni simbol. Ker smo metodo *LR* ustavili predčasno, obstaja verjetnost, da smo izvajanje prekinili sredi produkcije gramatike. To pomeni, da nam je preostalo še nekaj simbolov, preden lahko produkcijo razvijemo do konca. Kot smo opisali v poglavjih 2.3 in 3.6.1, metoda *LL* izbira produkcije glede na vsebino sklada, zato moramo preostale simbole vgrajenega analizatorja dodati na sklad.

To lahko dosežemo z uporabo tabele *BACKTRACK*, ki vsebuje prehode med enoličnimi opisi stroja vse do začetnega opisa. V poglavju 3.5 nava-

Algoritem 3.4 Psevdokoda iskanja preostalih simbolov.

```
 1: state, item ← find_unique_item()
 2: tail ← item.rule[item.dot..]
 3: current_rule ← item.rule
 4:
 5: repeat
 6:     state, item ← backtrack_table[state, item]
 7:     if item.rule ≠ current_rule then
 8:         tail ← tail + item.rule[item.dot...]
 9:         current_rule ← item.rule
10:     end if
11: until item = None
12:
13: return tail
```

jamo, da vsak opis predstavlja delno oziroma popolnoma razvito produkcijo gramatike, zato lahko zlahka določimo preostale simbole. Najprej na seznam simbolov dodamo vse preostale simbole v produkciji prej izbranega enoličnega opisa, nato pa iz tabele *BACKTRACK* izberemo opis, iz katerega smo razvili trenutnega. Če ta predstavlja drugo produkcijo, na seznam dodamo preostale simbole le-te. Ta postopek ponavljamo, dokler se ne vrnemo do začetnega opisa, nato pa seznam simbolov dodamo na sklad metode *LL*. Implementacijo tega postopka natančneje prikazuje algoritem 3.4.

3.7 Optimizacija porabe pomnilnika

Prva različica implementacije sintaksne analize po metodi *LLLR* je namesto sklicev do posameznih elementov uporabljala celotne strukture. Slabosti tega pristopa sta povečana poraba pomnilnika ter zmanjšana hitrost izvajanja, saj je metoda ponekod zahtevala dodeljevanje večjih delov pomnilnika namesto ene same pomnilniške besede. Prav tako smo celotne strukture uporabljali v čakalnih vrstah za obdelavo podatkov, počasnejše pa je bilo tudi primerjanje podatkovnih struktur. Tako smo v prvi razvojni različici za stroj s 50000 stanji potrebovali približno deset gigabajtov pomnilnika.

Ko smo končali implementacijo gradnje strojev *LR*, smo vse produkcije gramatike, stanja in opise prestavili v samostojno podatkovno strukturo, posamezne elemente v vektorjih pa smo nadomestili z edinstvenimi zaporednimi številkami glede na podatkovno vrsto strukture. Tako smo zmanjšali porabo pomnilnika in izboljšali hitrost izvajanja, vendar je bilo ponekod potrebno delno spremeniti arhitekturo implementacije. V mnogih funkcijah smo potrebovali večje število polj določene podatkovne strukture, zaradi česar je bilo potrebno iz gramatike oziroma stroja z zaporedno številko pridobiti celotno strukturo. To je zaradi visoke verjetnosti, da podatki niso v predpomnilniku procesorja, vodilo do nekoliko zmanjšane hitrosti izvajanja.

Zaradi istega razloga na vseh podatkovnih strukturah ni bilo mogoče implementirati lastnosti `Display` z metodo `fmt()` brez argumentov, zaradi česar smo definirali in implementirali lastnost `AsString`. Ta definira funkcijo `string()`, ki sprejme dodatne argumente, s katerimi lahko izpišemo celotno podatkovno strukturo v uporabniku prijazni obliki.

Po končanih optimizacijah program za prej omenjeni stroj s 50000 stanji porabi približno en gigabajt pomnilnika.

Poglavje 4

Primerjava metod LLLR, LL(k) in LL(*)

V tem poglavju bomo predstavili ter podrobno opisali prednosti in slabosti sintaksne analize različnih gramatik z uporabo metod *LLLR*, *LL(k)* in *LL(*)*. Najprej si bomo ogledali nekaj teoretičnih primerov, na katerih lahko jasno vidimo vedenje metod, nato pa bomo ovrednotili izvajanje na praktičnih primerih z gramatikama za programska jezika *Prev* in *C*. Na gramatikah bomo izmerili hitrost izvajanja in porabe pomnilnika, obenem pa določili primernost metod sintaksne analize v različnih primerih.

4.1 Teoretični primeri

Teoretične primere bomo začeli z gramatiko 4.1, prilagojeno metodi *LL*, kar pomeni, da ne vsebuje leve rekurzije. Sintaksna tabela med množicama *FIRST* in *FOLLOW* ter znotraj množice *FIRST* ne vsebuje konfliktov. Oblika gramatike je tako podobna gramatiki 2.2.

Začeli bomo s sintaksno analizo po metodi *LL(k)*, za praktičnost primerov pa se bomo omejili na $k = 1$. Tako je izračun sintaksne tabele enak postopku, opisanem v poglavju 2.3. Če bi želeli povečati parameter k, bi morali izračunati množici *FIRST* in *FOLLOW* za vse možne vrednosti k,

Gramatika 4.1: Primer gramatike, prilagojene metodi *LL*.

$$
\begin{aligned}
\text{S} &\rightarrow \text{definitions} \\
\text{definitions} &\rightarrow \text{definition definitions}_{opt} \\
\text{definitions}_{opt} &\rightarrow \texttt{;}\ \text{definition definitions}_{opt} \\
\text{definitions}_{opt} &\rightarrow \epsilon \\
\text{definition} &\rightarrow \texttt{let identifier =}\ \text{expression} \\
\text{expression} &\rightarrow \text{multiplication expression}_{opt} \\
\text{expression}_{opt} &\rightarrow \texttt{+}\ \text{multiplication expression}_{opt} \\
\text{expression}_{opt} &\rightarrow \texttt{-}\ \text{multiplication expression}_{opt} \\
\text{expression}_{opt} &\rightarrow \epsilon \\
\text{multiplication} &\rightarrow \text{atom multiplication}_{opt} \\
\text{multiplication}_{opt} &\rightarrow \texttt{*}\ \text{atom multiplication}_{opt} \\
\text{multiplication}_{opt} &\rightarrow \texttt{/}\ \text{atom multiplication}_{opt} \\
\text{multiplication}_{opt} &\rightarrow \epsilon \\
\text{atom} &\rightarrow \texttt{constant} \\
\text{atom} &\rightarrow \texttt{identifier} \\
\text{atom} &\rightarrow \texttt{(}\ \text{expression}\ \texttt{)}
\end{aligned}
$$

kar bi močno vplivalo na hitrost izvajanja. Metoda *LL(1)* tako sintaksno tabelo za podano gramatiko izračuna brez težav, sintaksna analiza pa se vedno izvede v linearnem času, ne glede na veljavnost vhodne datoteke.

Tako kot metoda *LL(1)* tudi metoda *LLLR* s podano gramatiko nima težav, saj krovni analizator uporablja popolnoma enak algoritem. Posledično lahko trdimo, da sta metodi za gramatiko 4.1 enako primerni. Kljub temu menimo, da je zaradi široke uveljavljenosti in razpoložljivosti orodij v različnih programskih jezikih ter sistemih metoda *LL(1)* dostopnejša za gramatike, ki so metodi posebej prilagojene.

Metoda *LL(*)*, ki se uporablja v generatorju sintaksnih analizatorjev *ANTLR 3.3*, prav tako pripada skupini metod od-zgoraj-navzdol, vendar se od metode *LL(k)* razlikuje v tem, da lahko za izbiro naslednje produkcije uporabi neomejeno število vhodnih simbolov. To doseže tako, da v primeru konflikta

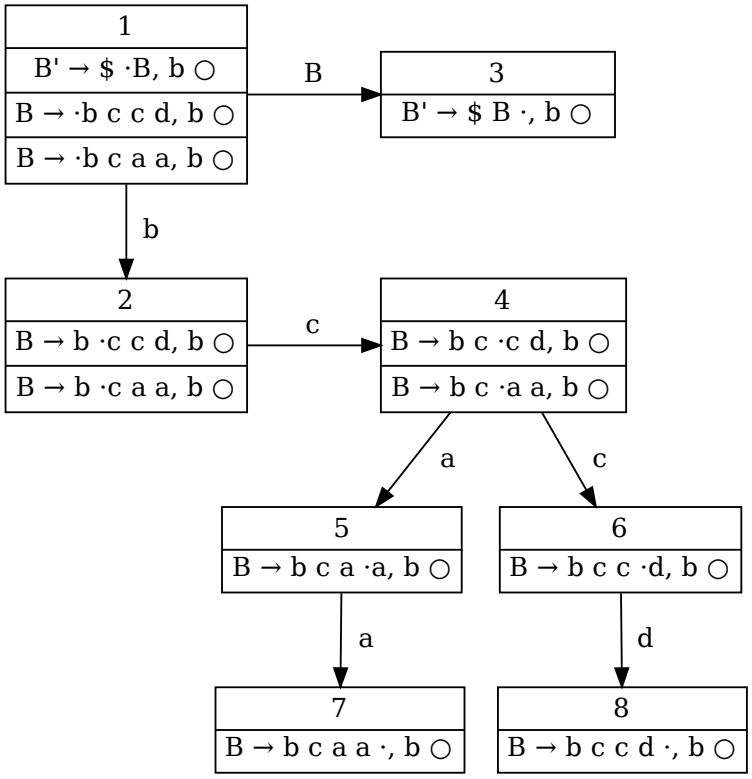
nadaljuje z drugačnim načinom analize, podobno kot metoda *LLLR*, kar si bomo ogledali na naslednjem teoretičnem primeru. Ker gramatika 4.1 ne vsebuje konfliktnih vmesnih simbolov in drugih posebnosti, tudi metoda *LL(*)* nima težav s sintaksno analizo. Na opisani gramatiki je razvidna prednost metode *LL(1)* in posledično metode *LLLR*, saj je glavni del metode *LL(*)* implementiran z rekurzivnim spuščanjem, kar vodi do nekoliko počasnejšega izvajanja.

Gramatika 4.2: Primer gramatike, veljavne le po metodi *LL(3)*.

S → `a a` A | `b b`
A → `a` A `b` | `c b` B `b`
B → `b c c d` | `b c a a`

Prva razlika med metodo *LL(1)* in metodo *LLLR* se pojavi pri gramatikah, ki jih lahko analiziramo le z metodami s parametrom $k > 1$, katerim pripada gramatika 4.2. Ta za vmesni simbol B potrebuje tri vhodne simbole, preden lahko metoda izbere ustrezno produkcijo, zato moramo sintaksno analizo izvesti po metodi *LL(3)*. Medtem bi metoda *LLLR* sintaksno analizo izvedla brez težav, saj konflikt v gramatiki razrešimo z vgrajenim analizatorjem. Kljub temu težko natančno določimo časovno zahtevnost obeh metod, saj je izračun sintaksne tabele oziroma gradnja stroja močno odvisna od izbrane gramatike. V kolikor bi želeli sintaksno analizo po metodi *LLLR* izvajati na širšem razredu gramatik, lahko algoritem krovnega analizatorja brez težav nadomestimo.

Izvajanje obeh metod na gramatiki 4.2 lahko prikažemo na vhodnem nizu `aacbbccdb`. Metodi začneta z vmesnim simbolom S. Metoda *LL* lahko glede na vhodni simbol `a` izbere produkcijo $S \rightarrow a\ a\ A$, njene simbole pa doda na sklad. Na vhodu se simbol `a` pojavi dvakrat, zato ponovitvi odstranimo tako z vhoda kot tudi s sklada. Metoda nato glede na simbol A na vrhu sklada ter vhodni simbol `c` iz sintaksne tabele izbere produkcijo $A \rightarrow c\ b\ B\ b$, v naslednjem koraku pa lahko vhodna simbola `c` in `b` ujemamo s simboli na vrhu sklada. Sedaj se na vhodu nahaja simbol `b`, na vrhu sklada pa vmesni

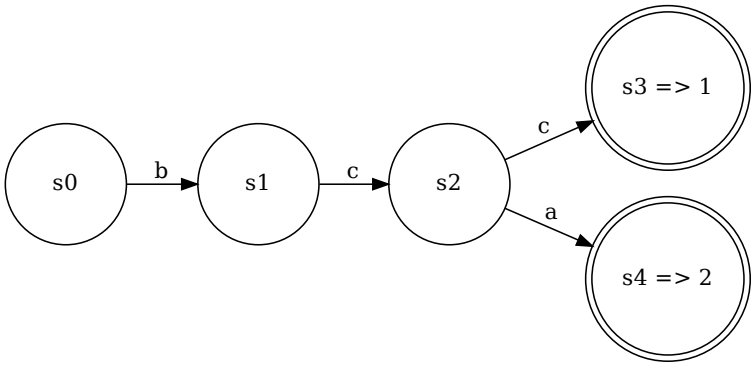


Slika 4.1: Stroj *LR* za vmesni simbol B v gramatiki 4.2.

simbol B. Ker metoda *LL(1)* med sintaksno analizo upošteva le naslednji simbol, ne more pravilno izbrati produkcije za simbol B, saj se obe začneta s simbolom `b`. Tako v sintaksni tabeli nastane konflikt, metoda *LL(1)* pa ne more uspešno določiti, ali je vhodni niz veljaven za podano gramatiko.

Metoda *LLLR* sintaksno analizo izvede po enakem postopku, pri čemer za vmesni simbol B uporabi vgrajeni analizator s strojem, prikazanem na sliki 4.1. Stroj začne v stanju 1, na vhodu pa se nahaja niz `bccdb`. Vgrajeni analizator v stanju 1 za vhodni simbol `b` iz tabele *LEFT* ne more izbrati enoličnega opisa, zato iz tabele *ACTION* izbere akcijo *Shift(2)* ter stroj prestavi v stanje 2. Postopek ponovimo za vhodni simbol `c`, stroj pa prestavimo v stanje 4. To stanje vsebuje dve produkciji, katerima na piki sledita različna simbola, zato lahko analizator za vhodni simbol `c` izbere enolični opis $B \rightarrow$

b c · c d, b. To pomeni, da smo uspešno razrešili konflikt, izvajanje pa lahko nadaljujemo s krovnim analizatorjem. Ker smo vgrajeni analizator prekinili sredi opisa (pika se nahaja sredi produkcije), moramo na sklad metode *LL* v obratnem vrstnem redu dodati simbola `c` in `d`. Krovni analizator nato nadaljuje izvajanje z vhodnim nizom `cdb`. Simbola `c` in `d` se ujemata s simboloma, ki smo ju pravkar dodali na vrh sklada, zadnji vhodni simbol pa se ujema s simbolom `b`, ki smo ga na sklad dodali med izbiro produkcije za vmesni simbol *A*. V naslednjem koraku sta tako sklad kot tudi vhod prazna, zato lahko metoda *LLLR* sintaksno analizo uspešno zaključi, saj je vhodni niz veljaven glede na gramatiko 4.2.



Slika 4.2: Končni avtomat za vmesni simbol *B* v gramatiki 4.2.

Na tej gramatiki je razvidna ključna prednost metode *LL(*)*. Ko metoda naleti na konflikt v gramatiki, izvajanje nadaljuje z regularnim izrazom oziroma deterministčnim končnim avtomatom za izbrani vmesni simbol. Primer takega avtomata prikazuje slika 4.2. Tako kot pri metodi *LLLR*, končni avtomat začne v stanju $s0$ z vhodnim nizom `bccdb`. Z vhodnim simbolom `b` se avtomat prestavi v stanje $s1$, s simbolom `c` pa v stanje $s2$. Naslednji vhodni simbol avtomat prestavi v stanje $s3 \Rightarrow 1$, s katerim lahko sprejmemo vhodni niz, za vmesni simbol *B* pa lahko pravilno izberemo produkcijo $B \rightarrow$ *b c c d*. Metoda *LL(*)* nato analizo vhodnega niza `db` nadaljuje z rekurzivnih spuščanjem. S tem teoretičnim primerom lahko metodo lažje primerjamo z metodo *LLLR*. Razlika v hitrosti izvajanja krovnega analizatorja in algo-

ritma za rekurzivno spuščanje je zanemarljiva, večja razlika pa se pojavi pri razreševanju konfliktov, saj je gradnja strojev *LR* v splošnem časovno zahtevnejša od gradnje končnih avtomatov za regularne izraze. Hitrejše je tudi izvajanje sintaksne analize, saj v gramatiki 4.2 ni potrebno izvajati vračanja za izbiro ustrezne produkcije. Poleg hitrosti izvajanja na tem primeru ni mogoče izpostaviti ključnih prednosti metod *LL(*)* in *LLLR*.

Gramatika 4.3: Gramatika 4.1, prilagojena metodi *LR(1)*.

S	→	definitions
definitions	→	definition
definitions	→	definitions `;` definition
definition	→	`let identifier =` expression
expression	→	multiplication
expression	→	expression `+` multiplication
expression	→	expression `-` multiplication
multiplication	→	atom
multiplication	→	multiplication `*` atom
multiplication	→	multiplication `/` atom
atom	→	`constant`
atom	→	`identifier`
atom	→	`(` expression `)`

Naslednji teoretični primer predstavlja gramatika 4.3, ki je osnovana na gramatiki 4.1, prilagojeni za metodo *LR(1)*. Nova gramatika tako vsebuje manjše število produkcij, saj metoda dovoljuje levo rekurzijo, obenem pa uporabniku močno olajša definiranje gramatik. V kolikor bi na konec poljubne produkcije v gramatiki iz prejšnjega primera želeli dodati izbirni simbol, bi morali ustvariti nov vmesni simbol z vsaj dvema produkcijama. Ker metoda *LR* dovoljuje levo rekurzijo, lahko v novi gramatiki enostavno dodamo novo produkcijo, ki razširja eno izmed obstoječih. Gramatika je tako bolj podobna vhodni datoteki, prav tako pa dovoljuje uporabo konstruktov, katere brez uporabe leve rekurzije težje definiramo.

Sintaksno analizo začnimo po metodi *LL(1)*. Težave se pojavijo že med izračunom sintaksne tabele, saj v njej nastane konflikt za vmesne simbole *definitions*, *expression* in *multiplication*. Posledično brez prilagoditve gramatike sintaksne analize po tej metodi ni mogoče izvesti, ne glede na vrednost parametra k. Poskusimo torej s sintaksno analizo po metodi *LLLR*. Tako kot pri metodi *LL* tudi tokrat med izračunom sintaksne tabele naletimo na konfliktne simbole, vendar lahko zgradimo stroje za vgrajeni analizator. Krovni analizator tako sintaksno analizo izvaja zgolj do vmesnega simbola *definitions*, zatem pa metoda nadaljuje z vgrajenim analizatorjem po metodi *LR*. Ker produkcije gramatike vsebujejo levo rekurzijo vse do dna drevesa, ki ga predstavlja vmesni simbol *atom*, vgrajenega analizatorja ni mogoče predčasno prekiniti. Ta primer ponazarja glavno prednost metode *LLLR* — sintaksno analizo lahko uspešno izvedemo ne glede na obliko gramatike. Tako uporabniku ni potrebno definirati zahtevnih oziroma posebej prilagojenih oblik produkcij. Če sintaksno analizo po metodi *LLLR* tokrat primerjamo z metodo *LR(1)*, lahko trdimo, da sta pristopa tako po sledi izpeljave kot tudi časovni zahtevnosti izvajanja skoraj enakovredna. Manjšo prednost ima metoda *LR(1)*, saj ni potrebno izračunati sintaksne tabele.

Ključno lastnost metode *LLLR* lahko ponazorimo, če v gramatiko 4.3 uvedemo nekaj produkcij iz gramatike 4.1. Kot omenjeno vgrajeni analizator začnemo blizu vrha drevesa gramatike. To pomeni, da je potrebno zgraditi stroj za obsežen del drevesa, posledično pa krovni analizator uporabimo za izbiro zgolj ene produkcije. V vmesnem simbolu *definitions* lahko odpravimo levo rekurzijo in s tem zmanjšamo velikost stroja. Tako konflikt nastane šele pri simbolu *expression*, vgrajeni analizator pa obravnava le tri namesto petih vmesnih simbolov.

Tukaj opazimo tudi prednost pred metodo *LR*, ki mora ne glede na obliko gramatike zgraditi stroj za celotno drevo, medtem ko mora metoda *LLLR* stroj zgraditi le za manjša poddrevesa. Hitrost izvajanja je sicer močno odvisna od končne oblike gramatike. Kljub temu da moramo v nekaterih primerih zgraditi manjše stroje, obstaja verjetnost, da je časovna zahtevnost zaradi

števila strojev in njihove morebitne neveljavnosti višja od časovne zahtevnosti za gradnjo enega samega stroja, ki pokriva celotno gramatiko. Zato je potrebno poiskati kompromis med hitrostjo izvajanja sintaksne analize in zahtevnostjo definiranja gramatike.

Na gramatiki 4.3 lahko opazimo ključno slabost metode *LL(*)*, saj tako kot metoda *LL(k)* tudi ta v produkcijah ne podpira leve rekurzije. Tako metode ne moremo uporabiti za opisano gramatiko brez prilagoditve produkcij, lahko pa uporabimo desno rekurzijo. Pri prepisovanju produkcij moramo sicer biti pazljivi, da ohranimo zaporedje matematičnih izrazov.

Izmed vseh opisanih metod ima metoda *LLLR* najvišjo porabo pomnilnika, saj mora poleg sintaksne tabele hraniti tudi večje število strojev. Metoda *LR* za sintaksno analizo potrebuje zgolj tabeli *ACTION* in *GOTO*, medtem ko vgrajeni analizator potrebuje še dve dodatni tabeli ter vse enolične opise v posameznih strojih.

4.2 Praktični primeri

Praktične primere bomo pričeli z gramatiko programskega jezika *Prev*, opisano v dodatku A. Tako kot večina gramatik za programske jezike tudi ta uporablja gramatiko, veljavno po metodi *LR*, zato bomo vrednotenje začeli z metodo *LLLR*. Sintaksno analizo bomo izvajali na vhodni datoteki `sieve.prev`, ki jo prikazuje izsek 4.1. Enako kot v predhodnih primerih bomo pozorni na število in velikost strojev, obenem pa bomo merili tudi dejansko hitrost izvajanja in porabo pomnilnika. Vse meritve v tem poglavju smo izvedli na procesorju *Intel Core i7-8550U* s frekvenco med 1.80 GHz in 4.00 GHz.

V začetni različici gramatike se v sintaksni tabeli krovnega analizatorja pojavi dvanajst konfliktnih simbolov. Prvi konflikt se pojavi takoj na vrhu drevesa z vmesnim simbolom *definitions*, zato vse produkcije razen prve izberemo z vgrajenim analizatorjem, kot smo prikazali z zadnjim teoretičnim primerom. Metoda za izračun strojev potrebuje približno šest sekund, medtem ko se sintaksna analiza izvede v nekaj deset milisekundah. Program za

izvajanje porabi 87 megabajtov pomnilnika.

Izsek 4.1: Vhodna datoteka `sieve.prev`.

```
1  typ arr_log: arr [500] logical;
2  var sieve: arr_log;
3  var i: integer;
4
5  fun main(argc: integer): integer = (
6    {sieve[0] = false},
7    {sieve[1] = false},
8    {for i = 2, 500, 1:
9      {sieve[i] = true}},
10   {for i = 2, sqrt(500) + 1, 1:
11     {if sieve[i] then (
12       {j = i * i},
13       {while j < 500: (
14         {sieve[j] = false},
15         {j = j + i})}
16       {where var j:integer})}},
17   {for i = 2, 500, 1:
18     {if sieve[i] then
19       printf('%d ', i)}},
20   printf('\n'),
21   0)
```

Kot smo opisali v poglavju 2.5 in prikazali na teoretičnih primerih, lahko z odstranitvijo leve rekurzije v produkcijah odpravimo nekaj konfliktov, vendar moramo biti pozorni na matematične izraze. V gramatiki A.1 lahko odpravimo konflikte za vmesne simbole *definitions*, *components*, *parameters* in *expressions*. Če ponovimo meritve z datoteko `sieve.prev`, lahko opazimo, da se je število konfliktnih simbolov zmanjšalo na osem, vendar sta se poslabšali tako hitrost izvajanja kot tudi poraba pomnilnika — sintaksna analiza sedaj potrebuje približno osem sekund, program pa porabi 100 megabajtov pomnilnika. Po vseh optimizacijah mora metoda *LLLR* za uspešno sintaksno analizo zgraditi štiri stroje. Pričakovali smo, da bo manjše število konfliktnih simbolov vodilo do hitrejšega izvajanja, vendar se izkaže, da se konfliktni vmesni simboli v produkcijah pojavijo na več mestih. Ker metoda

LLLR stroje gradi za posamezne pojavitve vmesnih simbolov, moramo za določene konfliktne simbole zgraditi večje število strojev.

Pri pregledu preostalih konfliktnih simbolov opazimo, da se vsi navezujejo na vmesni simbol *expression*. To pomeni, da smo z razrešitvijo konflikta na simbolu *expressions* uvedli večje število morebitnih novih strojev, zato lahko poskusimo povrniti levo-rekurzivno produkcijo vmesnega simbola *expressions*. Ko ponovno izvedemo meritve izvajanja, se v sintaksni tabeli pojavi devet konfliktnih simbolov, vendar se je hitrost izvajanja zmanjšala na približno štiri sekunde, saj je zaradi optimizacij potrebno zgraditi zgolj dva stroja. Obenem se je izboljšala tudi poraba pomnilnika — program tokrat zasede le 78 megabajtov pomnilnika.

Sintaksne analize po metodi *LL(1)* na podani gramatiki brez sprememb ne moremo izvesti, kar smo opazili že pri krovnem analizatorju metode *LLLR*. Obenem definicije gramatike brez večjih sprememb ne moremo prilagoditi. Zgoraj opisane spremembe sicer zlahka odpravijo levo rekurzijo, vendar spremenijo pomen matematičnih izrazov, kar vpliva na končno obliko sintaksnega drevesa. Če bi uporabnik želel ohraniti zaporedje izrazov, ki moral sintaksno drevo po končani analizi ročno prilagoditi oziroma popolnoma prepisati produkcije konfliktnih vmesnih simbolov.

Za izvajanje sintaksne analize po metodi *LL(*)* smo morali gramatiko prepisati v obliko, ki jo podpira orodje *ANTLR*, kar prikazuje izsek A.2. Kot smo navedli pri teoretičnih primerih, metoda *LL(*)* ne podpira leve rekurzije, zato moramo vse take produkcije prepisati v desno-rekurzivne. Obenem smo zmanjšali tudi število produkcij, saj za definicijo lahko uporabimo regularne izraze. Tako se znebimo tudi več primerov rekurzije, saj lahko za mnogo simbolov uporabimo eno samo produkcijo. Rekurzija tako ostane le v produkcijah simbolov *type* in *prefix_expression*. Preden lahko izvedemo sintaksno analizo, moramo prevesti gramatiko, saj orodje *ANTLR* same analize ne izvaja, vendar ustvari le izvorno kodo sintaksnega analizatorja, ki jo mora uporabnik nato ročno prevesti. Program je implementiran v programskem jeziku *Java*, zato ima tudi višjo porabo pomnilnika — za prevajanje grama-

tike A.2 porabi približno 100 megabajtov. Kljub temu postopek prevajanja traja le eno sekundo, sintaksna analiza pa se zaključi v nekaj milisekundah.

Izsek 4.2: Vhodna datoteka `main.c`.

```
1  int main(int argc, char **argv) {
2    int t1 = 0, t2 = 1, n;
3
4    printf("Enter the number of terms: ");
5    scanf("%d", &n);
6
7    printf("Fibonacci series: ");
8    for (int i = 1; i <= n; ++i) {
9      printf("%d ", t1);
10     int next = t1 + t2;
11     t1 = t2;
12     t2 = next;
13   }
14
15   return 0;
16 }
```

Drugi praktični primer smo izvedli na gramatiki programskega jezika *C* [18]. Gramatika je kompleksnejša od prejšnjega primera, zato je med dodatke nismo vključili. Ta temelji na definiciji, ki se uporablja v programu *yacc*. Analizo bomo izvajali na vhodni datoteki `main.c`, ki jo prikazuje izsek 4.2. Sintaksna analiza po metodi *LLLR* se na prvotni gramatiki izvaja 100 sekund, porabi pa 448 megabajtov pomnilnika. Sintaksna tabela vsebuje 51 konfliktnih vmesnih simbolov, metoda pa po optimizacijah zgradi štirinajst strojev. Sama sintaksna analiza tako kot v prejšnjih primerih potrebuje le nekaj deset milisekund.

Struktura gramatike programskega jezika *C* se od prvega praktičnega primera močno razlikuje, zato odpravljanje konfliktov vodi do drugačnih rezultatov. Levo rekurzijo lahko ročno odstranimo v produkcijah enajstih vmesnih simbolov, vendar se čas izvajanja programa podaljša, saj tokrat program potrebuje približno pet minut. Največji vpliv na hitrost izvajanja ima struktura gramatike, saj v tem primeru implementirane optimizacije ne

morejo zmanjšati števila strojev. Mnoge produkcije konfliktnih simbolov podane gramatike neposredno vsebujejo druge konfliktne simbole, zato z ročno odstranitvijo konfliktov povečamo število strojev. Na tem mestu lahko izpostavimo težavo z implementacijo metode *LLLR*, saj v določenih primerih obstaja verjetnost, da zgradimo več strojev, kot je potrebno. Nekatere gramatike lahko vsebujejo cikle v produkcijah, zaradi katerih je težko določiti, kateri vmesni simboli so dosegljivi s krovnim analizatorjem. Ta težava se pojavi tudi v gramatiki za programski jezik *Prev*, vendar zaradi manjšega števila produkcij nima znatnega vpliva.

Kljub temu da je gramatika programskega jezika *C* veliko večja od gramatike za *Prev*, metoda *LL(*)* sintaksno analizo izvede enako hitro. Vseeno pa je bilo potrebno izvorno gramatiko prepisati v celoti.

Gramatika programskega jezika *C* nam prikaže primer, kjer uporaba sintaksne analize po metodi *LLLR* nima prednosti pred drugimi metodami iz skupine *od-spodaj-navzgor* — sintaksna analiza se izvede pravilno, vendar traja dlje od metode *LR(1)*. Kljub temu lahko zaključimo, da se metoda *LL(1)* z metodama *LLLR* in *LL(*)* ne more primerjati, saj gramatike v taki obliki preprosto ni zmožna uporabiti za sintaksno analizo.

4.3 Meritve izvajanja

Za konec bomo ovrednotili še izvajanje metod *LLLR*, *LL(1)* in *LL(*)* na gramatiki 4.4. Ugotovili bomo, kakšen vpliv na hitrost izvajanja in porabo pomnilnika ima uporaba vgrajenega analizatorja ter strojev v metodi *LLLR*.

Gramatika 4.4: Gramatika za meritve izvajanja metode *LLLR*.

$$
\begin{aligned}
\mathrm{S} &\rightarrow \mathrm{A} \\
\mathrm{A} &\rightarrow \mathtt{a}\ \mathrm{B}\ \mathtt{a}\ \mathrm{A} \mid \mathtt{b} \\
\mathrm{B} &\rightarrow \mathtt{a}\ \mathtt{b}\ \mathrm{B} \mid \mathtt{a}\ \mathtt{a}
\end{aligned}
$$

Predstavljena gramatika je preprosta, vendar v sintaksni tabeli nastane konflikt za vmesni simbol B, zato mora metoda *LLLR* zgraditi stroja za

obe njegovi pojavitvi. Krovni analizator simbola B v produkciji $B \to a\ b\ B$ nikoli ne doseže neposredno, vendar se vgrajeni analizator ustavi takoj, ko v njej prebere simbol `b`, saj lahko na tem mestu enolično izbere produkcijo. Posledično razrešimo konflikt, simbol B iz produkcije pa dodamo na sklad krovnega analizatorja, ki simbol B začne razvijati z uporabo drugega stroja.

Izvajanje bomo primerjali z metodo *LL(1)* na osnovi gramatike 4.5, v kateri smo z dodatno produkcijo odpravili konflikt v gramatiki 4.4. Za izvajanje metode *LL(1)* je potrebno izračunati le množici *FIRST* in *FOLLOW* ter sintaksno tabelo, medtem ko moramo za metodo *LLLR* zgraditi še dva stroja, kar vpliva na hitrost izvajanja.

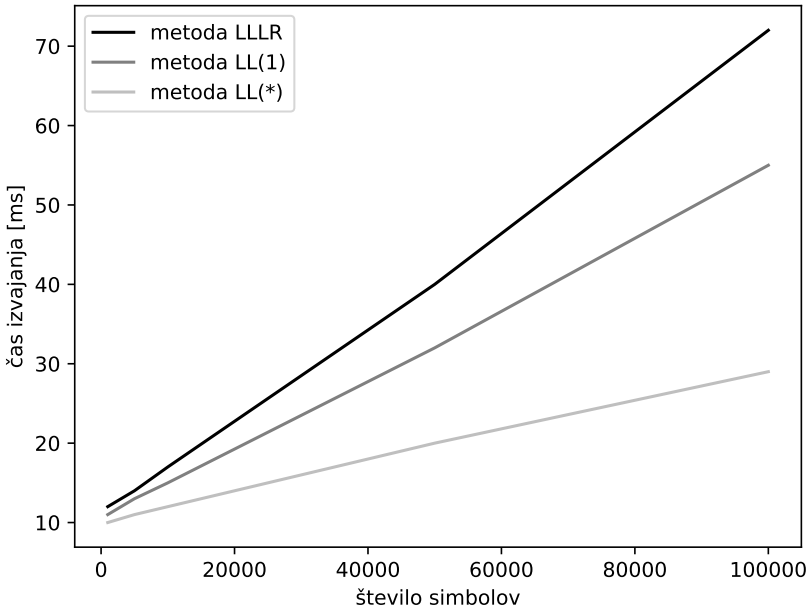
Gramatika 4.5: Gramatika za meritve izvajanja metode *LL(1)*.

$$
\begin{aligned}
\mathrm{S} &\to \mathrm{A} \\
\mathrm{A} &\to \texttt{a}\ \mathrm{B}\ \texttt{a}\ \mathrm{A} \mid \texttt{b} \\
\mathrm{B} &\to \texttt{a}\ \mathrm{B}' \\
\mathrm{B}' &\to \texttt{b}\ \mathrm{B} \mid \texttt{a}
\end{aligned}
$$

Sintaksno analizo smo izvedli na naboru datotek, ki vsebujejo med tisoč in milijon simbolov. Tako gramatiko kot tudi vhodne nize smo prilagodili z namenom, da metoda *LLLR* opravi čim več prehodov med krovnim in vgrajenim analizatorjem. V gramatiki želimo ponavljati produkcijo $A \to a\ B\ a\ A$, s katero povečamo dolžino vhodnega niza, za vmesni simbol B pa uporabimo vgrajeni analizator. Ne glede na produkcijo, ki jo metoda izbere, se po enem prebranem simbolu vrnemo v krovni analizator, postopek pa ponovimo takoj, ko se na skladu naslednjič pojavi konfliktni simbol B. Metoda *LL(1)* z gramatiko 4.5 nima težav, zato se izvede v linearnem času. Rezultate meritev prikazujeta sliki 4.3 in 4.4.

Meritve izvajanja metode *LL(1)* nam pokažejo, da za vhodni niz s tisoč simboli potrebujemo 11 milisekund, program pa za hranjenje vseh struktur porabi približno dva megabajta pomnilnika. Čas izvajanja vključuje tudi branje in preverjanje veljavnosti gramatike ter leksikalno analizo, ne vključuje pa izpisa sledi izpeljave. Analiza za vhodno datoteko z 10000 simboli po-

trebuje 16 milisekund, porabi pa štiri megabajte pomnilnika. Na preostalih datotekah tako čas izvajanja kot tudi poraba pomnilnika naraščata linearno.

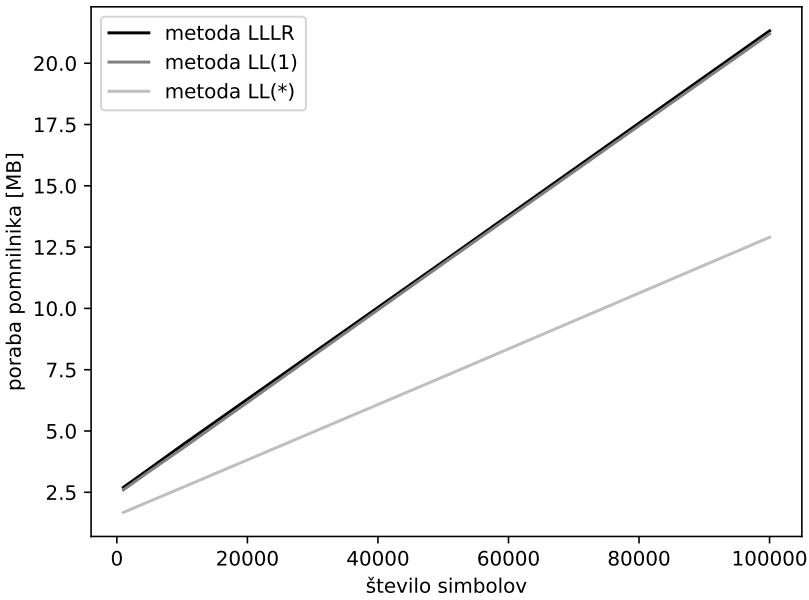


Slika 4.3: Hitrost izvajanja predstavljenih metod po gramatikah 4.4 (*LLLR* in *LL(*)*) ter 4.5 (*LL(1)*).

Metoda *LLLR* za sintaksno analizo datoteke s tisoč simboli po gramatiki 4.4 potrebuje 12 milisekund, izvede pa 293 prehodov na vgrajeni analizator. Metoda je na začetku počasnejša le za eno milisekundo, vendar se razlika z daljšimi nizi začne večati. Za analizo datoteke s 50000 simboli metoda *LLLR* potrebuje 40 milisekund in izvede 125126 prehodov, medtem ko metoda *LL(1)* potrebuje le 32 milisekund. Čas izvajanja kljub temu narašča linearno. Zaradi zanemarljive velikosti strojev poraba pomnilnika za vse vhodne nize ostane enaka. Glede na število prehodov lahko določimo, da vgrajeni analizator izvedemo za vsak tretji oziroma četrti vhodni simbol. Izpis programa med izvajanjem metode *LLLR* se nahaja v dodatku B.

Za meritve metode *LL(*)* smo gramatiko z orodjem *ANTLR* prevedli v sintaksni analizator v programskem jeziku *C++*. Metoda za vhodni niz s

tisoč simboli potrebuje 10 milisekund, porabi pa manj kot dva megabajta pomnilnika, kar je primerljivo z metodama *LLLR* in *LL(1)*. Razlika se začne večati z daljšimi nizi simbolov, saj metoda za datoteko s 100000 simboli potrebuje le 29 milisekund.



Slika 4.4: Poraba pomnilnika predstavljenih metod po gramatikah 4.4 (*LLLR* in *LL(*)*) ter 4.5 (*LL(1)*).

Pri metodi *LL(*)* je potrebno omeniti, da ima njena implementacija v orodju ANTLR težave s posebej dolgimi vhodnimi nizi. Ker je metoda implementirana z rekurzivnim spuščanjem, vsak vmesni simbol predstavlja rekurzivni klic funkcije. Posledično na vhodni datoteki s 500000 simboli prekoračimo velikost sklada (angl. *stack overflow*), metoda pa se ne more uspešno zaključiti. Čeprav se metoda *LL(*)* izvede hitreje kot metodi *LLLR* in *LL(1)*, moramo upoštevati dejstvo, da je bila njena implementacija čez leta zelo natančno optimizirana.

Poglavje 5

Sklepne ugotovitve

V sklopu tega magistrskega dela smo se seznanili z različnimi metodami in pristopi k sintaksni analizi. Ogledali smo si formalno definicijo gramatike in njenega zapisa ter postopek izračuna vseh množic in podatkovnih tabel, ki jih potrebujemo za izvajanje sintaksne analize po metodah *LL*, *LR* in *LLLR*. Sledil je podroben opis arhitekture programa in podatkovnih struktur, ki smo jih uporabili za učinkovito hranjenje podatkovnih tabel in strojev. Med implementacijo programa smo se srečali z različnimi težavami s pomnilnikom, preko katerih smo utemeljili izbiro strukture *IndexMap* za hranjenje stanj in opisov stroja ter opisali način za zmanjšanje porabe pomnilnika.

V zadnjem poglavju smo na različnih gramatikah primerjali izvajanje metod *LLLR*, *LR(k)* in *LL(*)*. Na teoretičnih primerih smo zaključili, da je metoda *LLLR* v splošnem zmogljivejša od metode *LL(k)*, z izjemo gramatik, ki so posebej prilagojene metodam *LL*. V takih primerih je uporabniku zaradi splošne razširjenosti metode *LL* na voljo mnogo več orodij za različne programske jezike. Metoda *LLLR* je primernejša za gramatike z zahtevnejšimi produkcijami, ki vsebujejo številne konflikte v sintaksni tabeli, katere brez večjih sprememb gramatike težko odpravimo. Uporabniki gramatike take oblike lažje definirajo, saj so produkcije podobne vhodnim datotekam, obenem pa izboljšajo razširljivost in prilagajanje v prihodnosti.

Metodo *LLLR* smo na istih primerih primerjali še z metodo *LL(*)*, ki je

zmogljivejša od metode *LL(k)*. Metodi sta si na prvi pogled zelo podobni, saj obe uporabljata vgrajene analizatorje za razrešitev konfliktov. Kljub temu se izkaže, da je metoda *LL(*)* v splošnem hitrejša od metode *LLLR*, saj lahko vgrajene končne avtomate zgradimo hitreje kot stroje *LR*. Ker metoda v gramatiki dovoljuje le desno rekurzijo, moramo biti pri definiranju gramatik posebej pazljivi. Metoda *LLLR* tako na tem področju prevlada, saj dovoljuje obe vrsti rekurzije. Vseeno pa je potrebno pozornost nameniti obliki gramatike, saj obstaja verjetnost, da implementacija metode *LLLR* zgradi več strojev, kot jih zares potrebujemo, kar lahko v določenih primerih močno vpliva tako na hitrost izvajanja kot tudi porabo pomnilnika.

Čeprav je metoda *LLLR* na praktičnih primerih počasnejša od metode *LL(*)*, za izvajanje ne potrebujemo drugih orodij. Zaključimo lahko, da je metoda *LL(*)* primernejša takrat, ko imamo že končno definirano gramatiko in jo lahko brez večjih težav po potrebi prilagodimo, medtem ko metodo *LLLR* raje uporabimo v primerih, kjer definicijo gramatike še vedno prilagajamo in želimo sintaksno analizo hitro preizkusiti. Ena izmed možnih izboljšav implementacije programa je, da uvedemo dodatne optimizacije, s katerimi lahko zmanjšamo število zgrajenih strojev. Največje težave so nam povzročale medsebojno odvisne produkcije, za katere smo težko določili, ali so dosegljive iz poljubnega simbola. Prav tako lahko izboljšamo hitrost izvajanja, če sintaksno tabelo in stroje za izbrano gramatiko ohranimo med različnimi izvajanji programa.

Dodatek A

Gramatika programskega jezika Prev

Izsek A.1 prikazuje gramatiko programskega jezika *Prev* v obliki *TOML*, ki se uporablja v implementiranem programu. Gramatika je osnovana na definiciji, primerni za analizator po metodi *LR*, vendar vsebuje nekaj prilagojenih produkcij, s katerimi smo izboljšali hitrost izvajanja sintaksne analize po metodi *LLLR*. Izvirne produkcije gramatike smo ohranili s komentarji.

Izsek A.1: Gramatika programskega jezika *Prev*.

```
1  name = "prev"
2  start_symbol = "source"
3
4  [rules]
5  source = "definitions"
6
7  # definitions = [
8  #   "definition",
9  #   "definitions ; definition",
10 # ]
11
12 definitions = "definition definitions_opt"
13 definitions_opt = [
14   "; definition definitions_opt",
15   "",
```

```
16 ]
17
18 definition = [
19   "type_definition",
20   "function_definition",
21   "variable_definition",
22 ]
23
24 type_definition = "typ identifier : type"
25 function_definition =
26   "fun identifier ( parameters ) : type = expression"
27 variable_definition = "var identifier : type"
28
29 type = [
30   "identifier",
31   "logical",
32   "integer",
33   "string",
34   "arr [ int_constant ] type",
35   "rec { components }",
36   "^ type",
37 ]
38
39 # components = [
40 #  "component",
41 #  "components , component",
42 # ]
43
44 components = "component components_opt"
45 components_opt = [
46   ", component components_opt",
47   "",
48 ]
49
50 component = "identifier : type"
51
52 # parameters = [
53 #  "parameter",
54 #  "parameters , parameter",
55 # ]
56
```

```
57 parameters = "parameter parameters_opt"
58 parameters_opt = [
59   ", parameter parameters_opt",
60   "",
61 ]
62
63 parameter = "identifier : type"
64
65 expression = [
66   "logical_or_expression",
67   "logical_or_expression { where definitions }",
68 ]
69
70 logical_or_expression = [
71   "logical_or_expression | logical_and_expression",
72   "logical_and_expression",
73 ]
74
75 logical_and_expression = [
76   "logical_and_expression & compare_expression",
77   "compare_expression",
78 ]
79
80 compare_expression = [
81   "additive_expression == additive_expression",
82   "additive_expression != additive_expression",
83   "additive_expression <= additive_expression",
84   "additive_expression >= additive_expression",
85   "additive_expression < additive_expression",
86   "additive_expression > additive_expression",
87   "additive_expression",
88 ]
89
90 additive_expression = [
91   "additive_expression + multiplicative_expression",
92   "additive_expression - multiplicative_expression",
93   "multiplicative_expression",
94 ]
95
96 multiplicative_expression = [
97   "multiplicative_expression * prefix_expression",
```

```
98    "multiplicative_expression / prefix_expression",
99    "multiplicative_expression % prefix_expression",
100   "prefix_expression",
101 ]
102
103 prefix_expression = [
104   "+ prefix_expression",
105   "- prefix_expression",
106   "^ prefix_expression",
107   "! prefix_expression",
108   "postfix_expression",
109 ]
110
111 postfix_expression = [
112   "postfix_expression ^",
113   "postfix_expression . identifier",
114   "postfix_expression [ expression ]",
115   "atom_expression",
116 ]
117
118 atom_expression = [
119   "log_constant",
120   "int_constant",
121   "str_constant",
122   "identifier",
123   "identifier ( expressions )",
124   "{ expression = expression }",
125   "{ if expression then expression }",
126   "{ if expression then expression else expression }",
127   "{ while expression : expression }",
128   "{ for identifier = expression , expression ,
129      expression : expression }",
130   "( expressions )",
131 ]
132
133 expressions = [
134  "expression",
135  "expressions , expression",
136 ]
137
138 # expressions = "expression expressions_opt"
```

```
139 # expressions_opt = [
140 #  ", expression expressions_opt",
141 #  "",
142 # ]
143
144 [tokens]
145 log_constant = ["true", "false"]
146
147 int_constant = "[0-9]+"
148 str_constant = "'(\\\\.|[^'])*('|$)"
149 identifier = "[A-Za-z_]+[A-Za-z0-9_]*"
150
151 [ignore]
152 whitespace = "[ \t\r\n]*"
153 comment = "#.*(\n|$)"
```

Gramatiko smo morali za izvajanje po metodi *LL(*)* prilagoditi, kar prikazuje izsek A.2. Levo rekurzijo, ki je metoda ne podpira, smo nadomestili z desno. Ostalih produkcij ni bilo potrebno spreminjati, prav tako pa smo lahko ohranili regularne izraze za leksikalno analizo.

Izsek A.2: Gramatika programskega jezika *Prev* za metodo *LL(*)*.

```
1  grammar Prev;
2
3  /* [rules] */
4  source: definitions ;
5  definitions: definition (';' definition)* ;
6
7  definition
8    : type_definition
9    | function_definition
10   | variable_definition ;
11
12 type_definition: 'typ' IDENTIFIER ':' type ;
13
14 function_definition
15   : 'fun' IDENTIFIER '(' parameters ')' ':'
16     type '=' expression ;
17
```

```
18 variable_definition: 'var' IDENTIFIER ':' type ;
19
20 type
21   : IDENTIFIER
22   | 'logical'
23   | 'integer'
24   | 'string'
25   | 'arr' '[' INT_CONSTANT ']' type
26   | 'rec' '{' components '}'
27   | '^' type ;
28
29 components: component (',' component)* ;
30 component: IDENTIFIER ':' type ;
31 parameters: parameter (',' parameter)* ;
32 parameter: IDENTIFIER ':' type ;
33
34 expression
35   : logical_or_expression ('{' 'where' definitions '}')* ;
36
37 logical_or_expression
38   : logical_and_expression ('|' logical_and_expression)* ;
39
40 logical_and_expression
41   : compare_expression ('&' compare_expression)* ;
42
43 compare_expression
44   : additive_expression
45     (('==' | '!=' | '<=' | '>=' | '<' | '>')
46      additive_expression)? ;
47
48 additive_expression
49   : multiplicative_expression
50     (('+' | '-') multiplicative_expression)* ;
51
52 multiplicative_expression
53   : prefix_expression
54     (('*' | '/' | '%') prefix_expression)* ;
55
56 prefix_expression
57   : ('+' | '-' | '^' | '!') prefix_expression
58   | postfix_expression ;
```

```
59
60 postfix_expression
61   : atom_expression
62     ('^' | '.' IDENTIFIER | '[' expression ']')* ;
63
64 atom_expression
65   : LOG_CONSTANT
66   | INT_CONSTANT
67   | STR_CONSTANT
68   | IDENTIFIER ('(' expressions ')')?
69   | '{' expression '=' expression '}'
70   | '{' 'if' expression 'then' expression
71     ('else' expression)? '}'
72   | '{' 'while' expression ':' expression '}'
73   | '{' 'for' IDENTIFIER '=' expression ',' expression
74     ',' expression ':' expression '}'
75   | '(' expressions ')' ;
76
77 expressions: expression (',' expression)* ;
78
79 /* [tokens] */
80 LOG_CONSTANT: 'true' | 'false' ;
81 INT_CONSTANT: '0'..'9'+ ;
82 STR_CONSTANT : '\'' (~('\'') | '\'\'')* '\'' ;
83
84 IDENTIFIER
85   : ('a'..'z' | 'A'..'Z' | '_')
86     ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')* ;
87
88 /* [ignore] */
89 WHITESPACE
90   : (' ' | '\t' | '\r' | '\n')
91     { $channel = HIDDEN; } ;
92
93 COMMENT
94   : '#' ~('\n' | '\r')* '\r'? '\n'
95     { $channel = HIDDEN; } ;
```

Dodatek B

Izpis programa med izvajanjem

Ta dodatek prikazuje izpis programa med izvajanjem sintaksne analize po metodi *LLLR* na osnovi gramatike 4.4, opisane v poglavju 4.3. Ker vhodni niz vsebuje vsaj tisoč simbolov, smo del izpisa preskočili.

Izsek B.1: Izpis programa za gramatiko 4.4.

```
1  GRAMMAR
2  bench
3  (0) ^ -> $ S $
4  (1) S -> A
5  (2) A -> 'a' B 'a' A
6  (3) A -> 'b'
7  (4) B -> 'a' 'b' B
8  (5) B -> 'a' 'a'
9
10 TOKENS
11 a @ 1:1 [5]
12 a @ 1:2 [5]
13 b @ 1:3 [7]
14 a @ 1:4 [5]
15 b @ 1:5 [7]
16 a @ 1:6 [5]
17 b @ 1:7 [7]
18 a @ 1:8 [5]
19 b @ 1:9 [7]
20 ...
```

```
21
22 AUTOMATON
23 States
24 (0) [(0) B' -> .$ B, 'a' o]
25 (1) [(0) B' -> $ .B, 'a' o;
26       (1) B -> .'a' 'b' B, 'a' o;
27       (2) B -> .'a' 'a', 'a' o]
28 (2) [(0) B -> 'a' .'b' B, 'a' o;
29       (1) B -> 'a' .'a', 'a' o]
30 (3) [(0) B' -> $ B ., 'a' o]
31 (4) [(0) B -> 'a' 'a' ., 'a' o]
32 (5) [(0) B -> .'a' 'b' B, 'a' o;
33       (1) B -> .'a' 'a', 'a' o;
34       (2) B -> 'a' 'b' .B, 'a' o]
35 (6) [(0) B -> 'a' 'b' B ., 'a' o]
36
37 State transitions
38 0, $ -> 1
39 1, 'a' -> 2
40 1, B -> 3
41 2, 'a' -> 4
42 2, 'b' -> 5
43 5, 'a' -> 2
44 5, B -> 6
45
46 Items
47 (0) B' -> .$ B, 'a' o
48 (1) B' -> $ .B, 'a' o
49 (2) B -> .'a' 'b' B, 'a' o
50 (3) B -> .'a' 'a', 'a' o
51 (4) B -> 'a' .'b' B, 'a' o
52 (5) B -> 'a' .'a', 'a' o
53 (6) B' -> $ B ., 'a' o
54 (7) B -> 'a' 'a' ., 'a' o
55 (8) B -> 'a' 'b' .B, 'a' o
56 (9) B -> 'a' 'b' B ., 'a' o
57
58 Item transitions
59 (0, 0), $ -> (1, 1)
60 (1, 1), e -> (1, 2)
61 (1, 1), e -> (1, 3)
```

```
(1, 1), B -> (3, 6)
(1, 2), 'a' -> (2, 4)
(1, 3), 'a' -> (2, 5)
(2, 4), 'b' -> (5, 8)
(2, 5), 'a' -> (4, 7)
(5, 2), 'a' -> (2, 4)
(5, 3), 'a' -> (2, 5)
(5, 8), e -> (5, 2)
(5, 8), e -> (5, 3)
(5, 8), B -> (6, 9)

AUTOMATON
States
(0) [(0) B' -> .$ B, 'a' o]
(1) [(0) B' -> $ .B, 'a' o;
     (1) B -> .'a' 'b' B, 'a' o;
     (2) B -> .'a' 'a', 'a' o]
(2) [(0) B -> 'a' .'b' B, 'a' o;
     (1) B -> 'a' .'a', 'a' o]
(3) [(0) B' -> $ B ., 'a' o]
(4) [(0) B -> 'a' 'a' ., 'a' o]
(5) [(0) B -> .'a' 'b' B, 'a' o;
     (1) B -> .'a' 'a', 'a' o;
     (2) B -> 'a' 'b' .B, 'a' o]
(6) [(0) B -> 'a' 'b' B ., 'a' o]

State transitions
0, $ -> 1
1, 'a' -> 2
1, B -> 3
2, 'a' -> 4
2, 'b' -> 5
5, 'a' -> 2
5, B -> 6

Items
(0) B' -> .$ B, 'a' o
(1) B' -> $ .B, 'a' o
(2) B -> .'a' 'b' B, 'a' o
(3) B -> .'a' 'a', 'a' o
(4) B -> 'a' .'b' B, 'a' o
```

```
(5) B -> 'a' .'a', 'a' o
(6) B' -> $ B ., 'a' o
(7) B -> 'a' 'a' ., 'a' o
(8) B -> 'a' 'b' .B, 'a' o
(9) B -> 'a' 'b' B ., 'a' o

Item transitions
(0, 0), $ -> (1, 1)
(1, 1), e -> (1, 2)
(1, 1), e -> (1, 3)
(1, 1), B -> (3, 6)
(1, 2), 'a' -> (2, 4)
(1, 3), 'a' -> (2, 5)
(2, 4), 'b' -> (5, 8)
(2, 5), 'a' -> (4, 7)
(5, 2), 'a' -> (2, 4)
(5, 3), 'a' -> (2, 5)
(5, 8), e -> (5, 2)
(5, 8), e -> (5, 3)
(5, 8), B -> (6, 9)

RULES
(1) S -> A
(2) A -> 'a' B 'a' A
(4) B -> 'a' 'b' B
(4) B -> 'a' 'b' B
(4) B -> 'a' 'b' B
(4) B -> 'a' 'b' B
(4) B -> 'a' 'b' B
(4) B -> 'a' 'b' B
...
(4) B -> 'a' 'b' B
(5) B -> 'a' 'a'
(2) A -> 'a' B 'a' A
(5) B -> 'a' 'a'
(2) A -> 'a' B 'a' A
...
```

Literatura

[1] D. E. Knuth, On the translation of languages from left to right, Information and Control 8 (6) (1965) 607–639.

[2] T. Kasami, An efficient recognition and syntax-analysis algorithm for context-free languages, Coordinated Science Laboratory Report no. R-257.

[3] P. M. Lewis II, R. E. Stearns, Syntax directed transduction, in: Proceedings of the 7th Annual Symposion on Switching and Automata Theory (SWAT'66), IEEE Computer Society Press, 1966, pp. 21–35.

[4] R. E. Stearns, P. Lewis, Property grammars and table machines, Information and Control 14 (6) (1969) 524–549.

[5] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd Edition, Pearson Education, 2007.

[6] J. P. Schmeiser, D. T. Barnard, Producing a top-down parse order with bottom-up parsing, Information Processing Letters 54 (6) (1995) 323–326.

[7] M. Tomita, LR parsers for natural languages, in: Proceedings of the 10th International Conference on Computational Linguistics and 22nd annual meeting on Association for Computational Linguistics, Association for Computational Linguistics, 1984, pp. 354–357.

[8] E. Scott, A. Johnstone, GLL parsing, Electronic Notes in Theoretical Computer Science 253 (7) (2010) 177–189.

[9] R. N. Horspool, Recursive ascent-descent parsers, in: Compiler Compilers, Third International Workshop CC '90, Schwerin, FRG, Vol. 477 of Lecture Notes in Computer Science, Springer-Verlag, 1990, pp. 1–10.

[10] T. Parr, K. Fischer, LL(*): The foundation of the ANTLR parser generator, ACM SIGPLAN Notices - PLDI'10 46 (6) (2011) 425–436.

[11] B. Slivnik, Kombinacija Knuthovega in Lewis-Stearnsovega sintaksnega analizatorja z minimalno uporabo Knuthove analize, Ph.D. thesis, Univerza v Ljubljani, Fakulteta za računalništvo in informatiko (2003).

[12] B. Slivnik, LLLR parsing: a combination of LL and LR parsing, in: Proceedings of the 5th Symposium on Languages, Applications and Technologies (SLATE'16), Dagstuhl Publishing, 2016, pp. 5:1–5:13.

[13] B. Slivnik, The embedded left LR parser, in: Proceedings of the Federated Conference on Computer Science and Information Systems, IEEE Computer Society Press, 2011, pp. 871–878.

[14] B. Slivnik, LL conflict resolution using the embedded left LR parser, Computer Science and Information Systems 9 (3) (2012) 1105–1124.

[15] B. Slivnik, On different LL and LR parsers used in LLLR parsing, Computer Languages, Systems & Structures 50 (2017) 108–126.

[16] T. Parr, S. Harwell, K. Fischer, Adaptive LL(*) parsing: The power of dynamic analysis, in: Proceedings of the 2014 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'14), ACM, ACM, 2014, pp. 579–598.

[17] J. C. Martin, Introduction to Languages and the Theory of Computation, 3rd Edition, McGraw-Hill, Inc., 2003.

[18] J. Degener, ANSI C Yacc grammar, `https://www.lysator.liu.se/c/ANSI-C-grammar-y.html` (1995).