VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA AUTOMATOCH S HLBOKÝMI ZÁSOBNÍKMI

PARSING BASED ON AUTOMATA WITH DEEP PUSHDOWNS

BAKALÁRSKA PRÁCA
BACHELOR'S THESIS

AUTOR PRÁCE **DOMINIKA PLOČICOVÁ**
AUTHOR

VEDÚCI PRÁCE **prof. RNDr. ALEXANDER MEDUNA, CSc.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Studentka: **Pločicová Dominika**
Program: Informační technologie
Název: **Syntaktická analýza založená na automatech s hlubokými zásobníky**
Parsing Based on Automata with Deep Pushdowns
Kategorie: Překladače
Zadání:

1. Dle pokynů vedoucího se seznamte detailně s automaty s hlubokými zásobníky a jejich vlastnostmi.
2. Dle pokynů vedoucího studujte vlastnosti těchto automatů.
3. Navrhněte metodu syntaktické analýzy, která je založena na automatech s hlubokými zásobníky.
4. Studujte užití metody syntaktické analýzy navržené v předchozích bodech. Zaměřte se na překladače programovacích jazyků. Po konzultaci s vedoucím zvolte vhodný programovací jazyk a sestrojte jeho syntaktický analyzátor, který provádí syntaktickou analýzu na základě této metody. Testujte výsledný syntaktický analyzátor.
5. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

- Meduna, A.: Automata and Languages, Springer, London, 2000

Pro udělení zápočtu za první semestr je požadováno:

- Splnění prvních 3 bodů zadání a části bodu 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Meduna Alexander, prof. RNDr., CSc.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 31. října 2019

Abstrakt

Jazyk je základným prostriedkom komunikácie. Formálne jazyky poskytujú základ pre komunikáciu človeka s počítačom. Cieľom tejto práce je prispieť práve do tejto oblasti a analyzovať možnosť spracovania kontextových prvkov formálnych jazykov. Práca sa venuje teoretickému popisu stavovej gramatiky a automatu s hlbokým zásobníkom, ktorý je modelom syntaktického analyzátora pre jazyky s kontextovými prvkami. Následne popisuje algoritmus, ktorý dokáže stavovú gramatiku na tento zásobníkový automat previesť. Práca ďalej opisuje syntaktickú analýzu, jej metódy a využitie vytvoreného automatu na jej vykonanie. Pre lepšie pochopenie sú v práci obsiahnuté aj príklady vytvorenia a činnosti automatu. Nakoniec je v práci popísaná implementácia programu, ktorý teoreticky popísané činnosti vykonáva. Funkčnosť programu bola overená na príkladoch, ktorých vstupy a výstupy sú tiež súčasťou textu práce.

Abstract

Language is a basic means of communication. Formal languages provide the basis for human-computer communication. The aim of this work is to contribute to this area and analyze the possibility of processing the context-sensitive elements of formal languages. The work includes the theoretical description of state grammar and deep pushdown automaton, which is a model of a parser for languages including context-sensitive elements. It describes an algorithm that can convert the state grammar to the said deep pushdown automaton. The work further describes parsing, its methods and the use of the created automaton in performing the syntax analysis. For a better understanding, the work also includes examples of the creation and operation of the automaton. Finally, the work describes the implementation of a program that performs theoretically described activities. The functionality of the program was verified on examples, whose inputs and outputs are also part of the text of the work.

Kľúčové slová

Automaty s hlbokým zásobníkom, syntaktická analýza, stavová gramatika

Keywords

Deep pushdown automata, parsing, state grammar

Citácia

PLOČICOVÁ, Dominika. *Syntaktická analýza založená na automatoch s hlbokými zásobníkmi*. Brno, 2020. Bakalárska práca. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedúci práce prof. RNDr. Alexander Meduna, CSc.

Syntaktická analýza založená na automatoch s hlbokými zásobníkmi

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracovala samostatne pod vedením pána Alexandra Medunu, prof. RNDr., CSc. Uviedla som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpala.

.....
Dominika Pločicová
27. mája 2020

Poďakovanie

Ďakujem za odborné, ochotné, stručné, rýchle a jasné vedenie pána profesora Medunu. Ďakujem najmä za veľmi rýchle reakcie, voľnú ruku a otvorenie možnosti v práci pokračovať.

Obsah

Kapitola 1

Úvod

Jazyk predstavuje základný komunikačný prostriedok. Prirodzený ľudský jazyk je dobrý na vyjadrovanie myšlienok a pocitov. Nie je však úplne priamy a obsahuje príliš veľa nejednoznačných vetných konštrukcií. Na jednoznačný a systémový popis problému, ktorý chceme riešiť pomocou stroja, teda nie je dostatočne vhodný.

Preto je dôležitou oblasťou informatiky teória formálnych jazykov. Teória formálnych jazykov pre reprezentáciu jazyka ponúka rôzne matematické modely, z ktorých najpoužívanejšie sú gramatiky a automaty. Gramatika umožňuje opísať štruktúru viet jazyka a automat umožňuje túto štruktúru identifikovať. Táto teória poskytuje aj algoritmy, ktoré umožňujú na základe špecifikácie syntaxe skonštruovať automat. Tieto algoritmy sa uplatňujú najmä v oblasti prekladačov programovacích jazykov. Na definíciu syntaxe programovacích jazykov sa úspešne využívajú gramatiky.

Na spracovanie bezkontextových jazykov existuje množstvo vhodných prostriedkov a šablón. Syntaktická analýza týchto jazykov je efektívne teoreticky vypracovaná a má známe metódy na spracovanie jazyka. Bezkontextové gramatiky sú najčastejšie využívané prostriedky na formálny popis programovacích jazykov.

Kontextové gramatiky sú pri formálnom popise programovacích jazykov využívané len zriedka najmä pre ich zložitosť. Napriek tomu obsahuje mnoho programovacích jazykov aj prvky kontextových jazykov. Tieto prvky sú spracovávané bez ohľadu na teóriu formálnych jazykov podľa toho, ako si ich spracuje programátor prekladača sám. Čiastočné riešenie tohto problému predstavuje stavová gramatika, ktorá je schopná formálne definovať aj jazyk, ktorý nie je čisto bezkontextový a obsahuje aj kontextové prvky. Tento jazyk následne môže byť spracovaný automatom s hlbokým zásobníkom ekvivalentným s danou stavovou gramatikou.

Teória formálnych jazykov predstavuje jedno z najprepracovanejších odvetví informačných technológií. Napriek tomu sa stále vyvíja a poskytuje ešte mnoho oblastí, ktoré by sa dali preskúmať, doplniť či overiť. Cieľom tejto práce je prispieť k praktickému overeniu možnosti vykonania syntaktickej analýzy založenej na automate s hlbokým zásobníkom.

Text práce je rozdelený do siedmich kapitol.

Druhá kapitola je teoretickým úvodom do problematiky. Zaoberá sa základnými reprezentáciami jazyka. Obsahuje všeobecný popis reprezentácie jazyka, Chomského hierarchické rozdelenie gramatík a následne popis stavovej gramatiky a jazyka, ktorý táto gramatika generuje. Stavová gramatika je prostriedkom, ktorý určuje syntax jazyka, ktorý bude automat spracovávať, a na základe ktorého bude automat vytvorený.

Tretia kapitola sa venuje automatu s hlbokým zásobníkom. Poskytuje definíciu tohoto automatu a popisuje jeho činnosti a jazyk, ktorý spracováva. Je základným prostriedkom

syntaktickej analýzy vykonávanej v tejto práci. Zastupuje však len funkciu rozpoznávača – určuje teda iba príslušnosť reťazca do jazyka bez vytvorenia iného výstupu.

Jadrom práce je štvrtá kapitola, ktorá sa zaoberá konverziou stavovej gramatiky na automat s hlbokým zásobníkom. Obsahuje hlavný algoritmus, pomocou ktorého sa táto konverzia dá vykonať, a tiež pomocné funkcie, ktoré dopĺňajú činnosť hlavného algoritmu.

Piata kapitola je venovaná syntaktickej analýze. Zaoberá sa všeobecným popisom syntaktickej analýzy a jej metód na základe bezkontextovej gramatiky a klasického zásobníkového automatu. Na to naväzuje podkapitola, ktorá sa zapodieva priamo syntaktickou analýzou založenou na automate s hlbokým zásobníkom a jeho operáciami.

Pre lepšie pochopenie uvedenej problematiky slúži šiesta kapitola, ktorá na dvoch príkladoch demonštruje celkový postup od gramatiky, cez konverziu až po samotné rozpoznávanie reťazca jazyka definovaného touto gramatikou.

Posledná, siedma, kapitola sa podrobne venuje samotnej implementácii jednotlivých algoritmov popísaných v predchádzajúcich kapitolách a tiež štruktúre programu. V tejto kapitole je uvedený aj postup na preklad a spustenie programu. Na naznačenie funkčnosti programu sú zobrazené príklady reálneho vstupu a výstupu prípadov z predchádzajúcej kapitoly.

V celej práci sa predpokladá, že čitateľ už má aspoň základné poznatky z oblasti teórie formálnych jazykov.

Kapitola 2

Reprezentácia jazyka

Jazyk môže byť reprezentovaný rôznymi spôsobmi [4]. Konečné jazyky môžu byť špecifikované aj jednoduchým vymenovaním slov. V praxi sa však častejšie stretávame s nekonečnými jazykmi, pre ktoré je takáto špecifikácia nemožná. Preto existujú aj iné prostriedky na špecifikáciu jazyka. Týmito prostriedkami sú gramatiky a automaty, ktoré predstavujú dva základné typy konečnej reprezentácie konečných aj nekonečných jazykov.

Podstatou každej gramatiky je konečná množina pravidiel, pomocou ktorých sa dá generovať daný jazyk, pričom jednotlivé slová jazyka sa vytvárajú postupným prepisovaním reťazcov podľa týchto pravidiel. Automat predstavuje algoritmus, ktorý rozhodne, či ľubovoľné slovo patrí alebo nepatrí do daného jazyka.

Poznáme niekoľko typov gramatík. Prvým z nich je všeobecná generatívna gramatika, gramatika typu 0. Ide o štvoricu

$$G = (N, T, P, S),$$

kde:

- N je abeceda neterminálnych symbolov
- T je abeceda terminálnych symbolov, pričom $N \cap T = \emptyset$
- $S \in N$ je počiatočný symbol
- $P \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$ je konečná množina pravidiel, pričom pravidlo $(x, y) \in P$ zapisujeme ako $x \to y$

Gramatika typu 0, pre ktorej každé pravidlo $v \to w \in P$ platí $|v| \leq |w|$[1], sa nazýva gramatika typu 1 alebo *kontextová gramatika*.

Gramatika typu 0, pre ktorej každé pravidlo $v \to w \in P$ platí $v \in N$, sa nazýva gramatika typu 2 alebo *bezkontextová gramatika*.

Gramatika typu 0, ktorej každé pravidlo má tvar $A \to xB$ alebo tvar $A \to x$, pričom $A, B \in N, x \in T^*$, sa nazýva gramatika typu 3 alebo *regulárna gramatika*.

Pre účely tejto práce je podstatná práve bezkontextová a kontextová gramatika.

2.1 Stavová gramatika

Stavová gramatika [5] predstavuje bezkontextovú gramatiku, ktorá je obohatená o stavový mechanizmus – jednotlivé kroky závisia aj na stave. Pri každom derivačnom kroku sa pre-

[1] s výnimkou $S \to \varepsilon$, ktoré implikuje, že sa S nevyskytuje na pravej strane žiadneho pravidla

pisuje najľavejší neterminálny symbol, ktorý sa môže prepísať na základe stavu. Stav sa pri prechode mení, čo môže ovplyvniť nasledujúci krok. V prípade, že sa všetky derivačné kroky prevedú v prvých n výskytoch neterminálnych symbolov, gramatiku môžeme považovať za n-limitovanú.

Definícia 2.1.1. Stavová gramatika je pätica

$$G = (V, W, T, P, S),$$

kde:

- V je abeceda
- W je konečná množina stavov
- $T \subset V$ je množina terminálnych symbolov
- $S \in V - T$ je počiatočný symbol
- $P \subseteq (W \times (V - T)) \times (W \times V^+)$ je konečná relácia, pričom namiesto zápisu $(q, A, p, v) \in P$ používame zápis $(q, A) \to (p, v) \in P$

Pre každé $z \in V^*$ definujeme

$$_G states(z) = \{q \in W | (q, A) \to (p, v) \in P, A \in alph(z)\},$$

pričom $alph(z)$ predstavuje množinu symbolov vyskytujúcich sa v z.

Ak $(q, A) \to (p, v) \in P, x, y \in V^*$ a $_G states(x) = \emptyset$, tak G môže spraviť derivačný krok z (q, xAy) do (p, xvy), čo zapíšeme ako

$$(q, xAy) \Rightarrow (p, xvy)[(q, A) \to (p, v)]$$

Ak je n kladné celé číslo vyhovujúce podmienke $\#_{V-T}(xA) \leq n$, hovoríme, že pravidlo $(q, xAy) \Rightarrow (p, xvy)[(q, A) \to (p, v)]$ je n-limitované, čo symbolicky zapíšeme ako

$$(q, xAy)_n \Rightarrow (p, xvy)[(q, A) \to (p, v)]$$

V prípade, že pri danom zápise nehrozí nedeterminizmus, zápis kroku môžeme skrátiť na tvar

$$(q, xAy) \Rightarrow (p, xvy)$$

$$(q, xAy)_n \Rightarrow (p, xvy)$$

Štandardným spôsobom rozširujeme $\Rightarrow$ na $\Rightarrow^m$ pre $m \geq 0$, pričom na základe $\Rightarrow^m$ definujeme $\Rightarrow^+$ a $\Rightarrow^*$.

Ak je n kladné celé číslo a $\nu, \omega \in W \times V^+$, na vyjadrenie toho, že každý derivačný krok v $\nu \Rightarrow^m \omega, \nu \Rightarrow^+ \omega, \nu \Rightarrow^* \omega$ je n-limitovaný, píšeme $\nu_n \Rightarrow^m \omega, \nu_n \Rightarrow^+ \omega, \nu_n \Rightarrow^* \omega$.

Pomocou `strings(`$\nu_n \Rightarrow^* \omega$`)` zapisujeme množinu všetkých reťazcov, ktoré sa vyskytujú v derivácii $\nu_n \Rightarrow^* \omega$.

2.1.1 Jazyk generovaný stavovou gramatikou

Jazyk generovaný gramatikou G, zapísaný ako L(G), je definovaný ako

$$L(G) = \{w \in T^* \mid (p, S) \Rightarrow^* (q, w), p, q \in W\}$$

Pre každé $n \geq 1$ je jazyk definovaný ako

$$L(G, n) = \{w \in T^* \mid (p, S)_n \Rightarrow^* (q, w), p, q \in W\},$$

pričom derivácia vo forme $(p, S)_n \Rightarrow^* (q, w)$, kde $p, q \in W$ a $w \in T^*$, reprezentuje úspešnú n-limitovanú generáciu w v G.

Stavová gramatika s limitom $n = 1$ generuje bezkontextové jazyky. Vyšší limit jej však umožňuje generovať aj podmnožinu kontextových jazykov.

Kapitola 3

Automat s hlbokým zásobníkom

Automat s hlbokým zásobníkom [5] funguje na podobnom princípe ako klasický zásobníkový automat. Rozdielnou vlastnosťou je to, že kým zásobníkový automat dokáže vykonávať operáciu expanzie len na vrchole zásobníka, automat s hlbokým zásobníkom je schopný expanziu vykonať aj hlbšie v zásobníku. To znamená, že môže expandovať neterminálny symbol až na n-tej alebo menšej pozícii v zásobníku pre každé $n \geq 1$. n predstavuje ako hĺbku automatu, tak aj limit stavovej gramatiky.

Podobne ako klasický zásobníkový automat, aj automat s hlbokým zásobníkom má konečnostavovú riadiacu jednotku a na vstupnej páske má zapísané vstupné slovo, ktoré číta pomocou čítacej hlavy. Je vybavený zásobníkovou pamäťou, ktorej vrchol či ktorýkoľvek symbol až do hĺbky n ovplyvňuje každý priamy prechod automatu.

Definícia 3.0.1. Automat s hlbokým zásobníkom je sedmica

$$_{deep}M = (Q, \Sigma, \Gamma, R, s, S, F),$$

kde:

- Q je konečná množina stavov
- Σ je vstupná abeceda
- Γ je zásobníková abeceda, pričom $\mathbb{N}, Q$ a Γ sú párovo disjunktné, $\Sigma \subseteq \Gamma$ a $\Gamma - \Sigma$ obsahuje špeciálny koncový symbol značený $\#$
- $R \subseteq (\mathbb{N} \times Q \times (\Gamma - (\Sigma \cup \{\#\})) \times Q \times (\Gamma - \{\#\})^+) \cup (\mathbb{N} \times Q \times \{\#\} \times Q \times (\Gamma - \{\#\})^* \{\#\})$ je konečná relácia, pričom namiesto zápisu $(m, q, A, p, v \in R)$ používame zápis $mqA \to pv \in R$ a $mqA \to pv$ nazývame pravidlom - R teda označujeme ako množinu pravidiel zásobníkového automatu
- $s \in Q$ je počiatočný stav
- $S \in \Gamma$ je počiatočný zásobníkový symbol
- $F \subseteq Q$ je konečná množina koncových stavov

Konfigurácia automatu M je trojica $Q \times T^* \times (\Gamma - \{\#\})^* \{\#\}$. χ predstavuje množinu všetkých konfigurácií M. Určíme si, že $x, y \in \chi$ sú 2 konfigurácie.

Automat vykoná operáciu *pop* v zásobníku z x do y, a to vtedy, keď $x = (q, au, az), y = (q, u, z)$, kde $a \in \Sigma, u \in \Sigma^*, z \in \Gamma^*$. Symbolicky túto operáciu zapíšeme ako

$$x_p \vdash y$$

Ak $x = (q, w, uAz), y = (p, w, uvz), mqA \to pv \in R$, kde $q, p \in Q, w \in \Sigma^*, A \in \Gamma, u, v, z \in \Gamma^*$ a počet symbolov reprezentujúcich neterminály v reťazci u je rovný $m - 1$, automat M vykoná operáciu expanzie z x do y, čo symbolicky zapíšeme ako

$$x_e \vdash y$$

Na vyjadrenie toho, že automat vykonáva operáciu expanzie $x_e \vdash y$ na základe pravidla $mqA \to pv$, využívame zápis

$$x_e \vdash y[mqA \to pv]$$

Hovoríme, že pravidlo $mqA \to pv$ je pravidlo hĺbky m. Vzhľadom na to nazývame aj operáciu expanzie $x_e \vdash y[mqA \to pv]$ expanziou hĺbky m. V prípade, že automat vykonáva $x_e \vdash y$ alebo $x_p \vdash y$, vykonáva prechod z x do y, pričom tento prechod zapisujeme ako

$$x \vdash y$$

Ak je $n \in \mathbb{N}$ minimálne kladné celé číslo také, že každé pravidlo automatu M je hĺbky n alebo menšej, hovoríme, že automat je hĺbky n, čo symbolicky zapíšeme ako ${}_nM$. Štandardným spôsobom rozširujeme ${}_p\vdash, {}_e\vdash, \vdash$ na ${}_p\vdash^m, {}_e\vdash^m, \vdash^m$ pre $m \geq 0$. Na základe tohto rozšírenia definujeme zápisy ${}_p\vdash^+, {}_p\vdash^*, {}_e\vdash^+, {}_e\vdash^*, \vdash^+, \vdash^*$.

3.1 Jazyk akceptovaný automatom s hlbokým zásobníkom

Jazyk $L({}_nM)$, ktorý je akceptovaný automatom s hlbokým zásobníkom ${}_nM$ hĺbky n, pričom $n \in \mathbb{N}$, je definovaný ako

$$L({}_nM) = \{w \in \Sigma^* \mid (s, w, S\#) \vdash^* (f, \varepsilon, \#) \text{ v } {}_nM \text{ s } f \in F\}$$

Jazyk $E({}_nM)$, ktorý je akceptovaný automatom ${}_nM$ pomocou prázdneho zásobníka, je definovaný ako

$$E({}_nM) = \{w \in \Sigma^* \mid (s, w, S\#) \vdash^* (q, \varepsilon, \#) \text{ v } {}_nM \text{ s } q \in Q\}$$

Pre každé $k \geq 1$, ${}_{deep}\mathbf{PDA}_k$ označuje jazyky definované automatom s hlbokým zásobníkom hĺbky i, kde $1 \leq i \leq k$. ${}^{empty}_{deep}\mathbf{PDA}_k$ označuje jazyky definované automatom s hlbokým zásobníkom hĺbky i akceptované pomocou prázdneho zásobníka, kde $1 \leq i \leq k$.

Automat s hlbokým zásobníkom hĺbky $m = 1$ je schopný akceptovať jazyky generované stavovou gramatikou s limitom $n = 1$, čo znamená, že akceptuje bezkontextové jazyky. Pre každé $m \geq 1$ je schopný akceptovať jazyky generované stavovou gramatikou s limitom $n \geq 1$, čo znamená, že akceptuje aj jazyky nachádzajúce sa medzi množinou bezkontextových a kontextových jazykov.

Kapitola 4

Konverzia gramatiky na zásobníkový automat

Veta 4.0.1. Pre každú stavovú gramatiku G a pre každé $n \geq 1$ existuje automat s hlbokým zásobníkom ${}_nM$ hĺbky n taký, že $L(G, n) = L({}_nM)$.

Vďaka tejto vlastnosti stavovej gramatiky sa teda budeme zaoberať generovaním automatu s hlbokým zásobníkom práve z nej.[5]

4.1 Algoritmus na konverziu gramatiky

Konverziu zo stavovej gramatiky na automat je možné previesť pomocou algoritmu založeného na dôkaze vyššie uvedenej vety v [5]. Algoritmus sa skladá zo 4 krokov, v ktorých využíva niekoľkých pomocných funkcií.

V kapitole je uvedený algoritmický popis jednotlivých krokov algoritmu aj pomocných funkcií, na základe ktorých bol následne implementovaný cieľový program.

Je dôležité poznamenať, že v nasledujúcom texte sú jednotlivé symboly reprezentované reťazcami znakov. Vo všetkých prípadoch vyhodnocovania dĺžky jednotlivých symbolov či reťazcov sa dĺžka počíta na počet symbolov, nie na počet znakov v reťazci.

4.1.1 Pomocné funkcie

Funkcia `f(x)`

Algoritmus 1: F

Input: N, x^*
Output: $x^* - T$

```
1: function F(x)
2:     nonterms = ε
3:     for each x ∈ x* do
4:         if x ∈ N then
5:             nonterms = nonterms + x
6:         else
7:             nonterms = nonterms + ε
8:         end if
9:     end for
10:    return nonterms
11: end function
```

Prvou pomocnou funkciou volanou v algoritme, konkrétne v argumente funkcie `prefix()`, je funkcia `f(x)`. Ide o funkciu, ktorá pre argument x v prípade, že ide o terminálny symbol, vráti ε, a v prípade, že ide o neterminálny symbol, vráti tento neterminál. Ak je argumentom reťazec, vyhodnotí sa ako zreťazenie výsledkov rekurzívneho volania pre jednotlivé symboly reťazca – výsledkom teda je, že odstráni všetky terminálne symboly a vracia reťazec zložený len z neterminálov alebo symbolu #. Vstupom je množina neterminálnych symbolov gramatiky N a reťazec symbolov x^*, výstupom reťazec x^* bez terminálnych symbolov.

Funkcia $_g$`states`

Algoritmus 2: GSTATES

Input: $G = (V, W, T, P, S), z$
Output: $Q = \{q \in W | (q, A) \to (p, v) \in P, A \in \text{ALPH(z)}\}$

```
1: function g_states(z)
2:     for each (q, A) → (p, v) ∈ P do
3:         if q ∈ alph(z) then
4:             Q = Q ∪ {q}
5:         end if
6:     end for
7:     return Q
8: end function
```

Druhá pomocná funkcia, $_g$`states(z)`, slúži na overenie niekoľkých podmienok a využíva sa v druhom a treťom kroku algoritmu. Je definovaná ako

$$_g states(z) = \{q \in W | (q, A) \to (p, v) \in P, A \in \texttt{alph(z)}\}$$

`alph(z)` v $_g$`states(z)` predstavuje množinu všetkých symbolov vyskytujúcich sa v reťazci z definovanú ako `alph(z)`$= \{a_1, a_2, ...a_n\}$.

Vstupom do funkcie je stavová gramatika G a reťazec symbolov z, výstupom je množina stavov Q. Stavy musia vyhovovať stanovenej podmienke $q \in W, (q, A) \to (p, v) \in P, A \in$ ALPH(z)

Funkcia `prefix(symbols, n)`

Algoritmus 3: PREFIX

Input: $symbols, n$
Output: $symbols[0...n]$

```
1: function PREFIX(symbols,n)
2:     if |symbols| ≤ n then
3:         return symbols
4:     else
5:         return symbols[0...n]
6:     end if
7: end function
```

Ďalšou pomocnou funkciou je `prefix(symbols, n)`. Vstupom je postupnosť symbolov *symbols* a dĺžka prefixu n, výstupom je postupnosť prvých n symbolov v reťazci *symbols*. Funkcia je volaná v 2. kroku algoritmu.

Funkcia `U`

Algoritmus 4: U

Input: N, n
Output: U

```
1: function U( )
2:     U := {ε}
3:     for each u ∈ U do
4:         if |u| + 1 ≤ n then
5:             for each s ∈ N do
6:                 U := U ∪ {u + s}
7:             end for
8:         end if
9:     end for
10: end function
```

Štvrtou pomocnou funkciou je funkcia na generovanie množiny reťazcov U. Ide o všetky kombinácie neterminálnych symbolov, ktoré počtom symbolov nepresahujú danú hĺbku automatu a sú jej nanajvýš rovné. Podmienka síce stanovuje, že ide o všetky kombinácie neobmedzenej dĺžky, pre naše potreby a v rámci šetrenia počítačových zdrojov je však plne postačujúci počet symbolov zodpovedajúci hĺbke automatu a dlhšie reťazce sú zbytočné. Vstupom je teda množina neterminálov N, ktorú si odvodíme rozdielom množiny abecedy z gramatiky a jej množiny terminálov, a hĺbka automatu n. Výstup pozostáva z množiny takto vygenerovaných kombinácií neterminálnych symbolov. Využitie množiny prichádza v 2. a 3. kroku algoritmu. Pred generovaním kombinácií sa v množine nachádza len prvok ε.

Funkcia `V`

Algoritmus 5: V

Input: N, n
Output: V

```
1:  function V_2( )
2:      V := {ε}
3:      for each v ∈ V do
4:          if |v| + 1 ≤ n then
5:              for each s ∈ N do
6:                  V := V ∪ {v + s}
7:              end for
8:          end if
9:      end for
10:     for each v ∈ V do
11:         if |v| + 1 ≤ n then
12:             V := V ∪ {v + #}
13:         end if
14:     end for
15: end function

16: function V_3( )
17:     V := {ε}
18:     for each v ∈ V do
19:         if |v| + 1 ≤ n then
20:             V := V ∪ {v + #}
21:         end if
22: end function
```

Predposledná pomocná funkcia, ktorá slúži na generáciu množiny reťazcov V, sa, podobne ako predchádzajúca funkcia, využíva v 2. a 3. kroku algoritmu, avšak v každom z nich sa správa rozdielne.

V druhom kroku algoritmu sa množina V generuje ako všetky možné kombinácie neterminálnych symbolov gramatiky, ktoré počtom symbolov spadajú do hranice určenej hĺbkou automatu, avšak na ich konci sa tiež môže nachádzať špeciálny symbol #, prípadne aj viac týchto symbolov. Počet symbolov by však stále nemal presiahnuť hĺbku automatu. Generovanie tejto množiny je znázornené funkciou V_2().

V treťom kroku sa množina V skladá už len z reťazcov pozostávajúcich zo špeciálnych symbolov #, pričom ich dĺžka zostáva ohraničená hĺbkou automatu. Generovanie množiny je popísané vo funkcii V_3().

V oboch prípadoch je v počiatočnom stave v množine len prvok ε. Vstupom je opäť množina neterminálnych symbolov N odvodená z definovanej gramatiky a hĺbka automatu pre prvý prípad množiny, pre druhý prípad postačuje len hĺbka automatu. Výstupom je množina V podľa toho, v ktorom kroku algoritmu sa práve využíva.

Funkcia `filter`

Algoritmus 6: FILTER

Input: P, R
Output: P_r

```
1: function FILTER( )
2:     P_r = ∅
3:     for each p ∈ P do
4:         if p ∈ R; p ∉ P_r then
5:             P_r = P_r + {p}
6:         end if
7:     end for
8:     return P_r
9: end function
```

Keďže algoritmus na konverziu generuje pomerne veľké množstvo pravidiel, ktoré sa zvyšuje spolu s hĺbkou automatu, počtom neterminálnych symbolov, stavov a podobne, je vhodné tieto pravidlá nakoniec zredukovať. Z množiny pravidiel teda odstránime všetky pravidlá, ktoré vychádzajú z nedosiahnuteľných stavov, a tiež všetky pravidlá, ktoré predstavujú duplikát niektorého z pravidiel.

Vstupom do funkcie je množina pravidiel automatu P, ktorá sa bude filtrovať, a tiež množina pravidiel vychádzajúcich z dosiahnuteľných stavov R. Postupne sa prechádza množina pravidiel P. Ak sa aktuálne pravidlo p vo výslednej množine pravidiel P_r ešte nenachádza a patrí do množiny pravidiel vychádzajúcich z dostupných stavov R, pridá sa do množiny Pr. Výstupom je množina P_r.

Tento krok je voliteľný a pri syntaktickej analýze nemá výrazný vplyv na výsledok, môže však skrátiť jej vykonanie a celkovo ju zjednodušiť z pohľadu procesorového času. Zredukované množstvo pravidiel tiež predstavuje prehľadnejšiu množinu pravidiel výsledného automatu.

4.1.2 Konverzia

Algoritmus na konverziu stavovej gramatiky na zásobníkový automat má 4 kroky. Vstupom do algoritmu je stavová gramatika ${}_sG$ a výstupom automat s hlbokým zásobníkom ${}_nM$. Pred

prvým krokom je potrebné automat inicializovať – nastaviť vstupnú abecedu, zásobníkovú abecedu, počiatočný stav, koncový stav a počiatočný symbol.

Algoritmus 7: KONVERZIA ${}_sG$ NA ${}_{deep}M$

Input: ${}_sG = (V, W, T, P, {}_gS)$
Output: ${}_nM = (Q, T, \{\#\} \cup V, R, s, {}_dS, \{\$\})$

```
1: function STEP1( )
2:     for each (p, S) → (q, x) ∈ P where p, q ∈ W; x ∈ V+ do
3:         if S = gS then
4:             R := R ∪ {1 sd S → ⟨p, S⟩S}
5:         end if
6:     end for
7: end function

8: function STEP2( )
9:     for each u ∈ N* where |u| ≤ n − 1 do
10:        for each v ∈ N*{#}* where |v| ≤ n − 1 do
11:            if |uv| + 1 = n then
12:                for each (p, A) → (q, x) ∈ P where p, q ∈ W; A ∈ N; x ∈ V*;
       p ∉ gstates(u) do
13:                    R := R ∪ {|uA| ⟨p, uAv⟩A → ⟨q,PREFIX(uf(x)v, n)⟩x}
14:                end for
15:            end if
16:        end for
17:    end for
18: end function

19: function STEP3( )
20:    for each u ∈ N* where |u| ≤ n − 1 do
21:        for each v ∈ {#}* where |v| ≤ n − 1 do
22:            if |uv| ≤ n − 1 then
23:                for each p ∈ W where p ∉ gstates(u) do
24:                    for each A ∈ N do
25:                        R := R ∪ {|uA| ⟨p, uv⟩A → ⟨p, uAv⟩A}
26:                        R := R ∪ {|uA| ⟨p, uv⟩# → ⟨p, uv#⟩#}
27:                    end for
28:                end for
29:            end if
30:        end for
31:    end for
32: end function

33: function STEP4( )
34:    for each q ∈ W do
       R := R ∪ {1⟨q, #^n⟩# → $#}
35:    end for
36: end function
```

V prvom kroku reprezentovanom funkciou STEP_1() spracovávame všetky pravidlá z gramatiky, v ktorých sa nachádza počiatočný stav gramatiky v časti pred prechodom. Odkontrolujeme, či je pravidlo korektné – skontrolujeme prítomnosť oboch stavov, aktuálneho aj nového, v množine stavov stavovej gramatiky, ako aj zloženie nového reťazca výhradne zo symbolov, resp. symbolu, nachádzajúcich sa v abecede gramatiky. V prípade, že je pravidlo správne, do množiny pravidiel automatu pridáme pravidlo $1s_dS \to \langle p, S\rangle S$.

Druhý krok reprezentovaný funkciou STEP_2() slúži na vygenerovanie ďalších pravidiel slúžiacich na prijímanie jazyka spracovávaného týmto automatom. Druhý krok využíva, narozdiel od prvého, všetky pravidlá gramatiky. Kontroluje sa prítomnosť oboch stavov v množine stavov v gramatike, prítomnosť aktuálneho symbolu v množine neterminálov, prítomnosť znaku/znakov nového reťazca v abecede gramatiky a neprítomnosť aktuálneho stavu v ${}_g states(u)$ popísanej vyššie. V tomto kroku využívame aj množiny U a V popísané v predchádzajúcej časti. Hľadáme všetky kombinácie reťazca z U a V také, pre ktoré platí, že počet symbolov je menší alebo rovný hĺbke automatu -1. V prípade, že teda platí $|uv|-1 \leq n$, pridávame pravidlá do automatu, a to tak, že vezmeme každé z pravidiel gramatiky a do pravidiel automatu pridáme pre každú vyhovujúcu kombináciu uv vygenerované pravidlo zapísané ako $|uA|\,\langle p, uAv\rangle A \to \langle q,\text{PREFIX}(u\text{f}(x)v, n)\rangle x$.

Nasledujúce dva kroky slúžia na vyprázdnenie zásobníka. V treťom kroku vo funkcii STEP_3() už nepozeráme na pravidlá gramatiky, ale na jej množinu stavov. Vo funkcii opäť využívame obe množiny U a V, pričom do úvahy berieme V generované pre tretí krok. Pre každý stav z množiny stavov gramatiky, ktorý nepatrí ${}_g states(u)$, a pre všetky vhodné kombinácie uv vygenerujeme pre každý neterminálny symbol až dve pravidlá: $|uA|\,\langle p, uv\rangle A \to \langle p, uAv\rangle A$ a $|uA|\,\langle p, uv\rangle \# \to \langle p, uv\#\rangle \#$.

Posledný, štvrtý krok, je najjednoduchší. Vezmeme všetky stavy z množiny stavov gramatiky a pre každý z nich pridáme pravidlo v tvare $1\langle q, \#^n\rangle \# \to \$\#$, pričom n zodpovedá hĺbke automatu.

Po štvrtom kroku, kedy už máme vygenerované všetky pravidlá, je len na nás, či si ich všetky ponecháme alebo ich prefiltrujeme a ponecháme si len tie, ktoré nie sú duplikátom iného pravidla a nevychádzajú z nedosiahnuteľného stavu.

Kapitola 5

Syntaktická analýza

Syntax popisuje spôsob, akým sú spájané slová pri vytváraní vety, či všeobecne spôsob, akým sú spájané jednotlivé časti do väčších celkov. Proces, v ktorom prebieha vyhodnotenie správnosti syntaxe, sa nazýva syntaktická analýza. Cieľom syntaktickej analýzy je teda vyhodnotenie, či je daná veta či iný celok syntakticky správne zapísaný, a tiež výsledná syntaktická štruktúra danej vety či celku. Z toho vyplýva, že je možné vykonať syntaktickú analýzu takmer akéhokoľvek celku. Z pohľadu práce je však najdôležitejšia syntaktická analýza zdrojového programu.

Syntaktická analýza [4] je jednou z fáz prekladu zdrojového programu (zo zdrojového jazyka do cieľového). Predchádza jej lexikálna analýza, ktorú vykonáva lexikálny analyzátor. Ten číta zdrojový program a prekladá ho na reťazec lexikálnych symbolov. Týmito symbolmi sú jednotlivé elementy zdrojového jazyka, napríklad identifikátory, kľúčové slová, a podobne. Výstup lexikálneho analyzátora je vstupom pre syntaktický analyzátor, parser, ktorý vykonáva syntaktickú analýzu.

Syntaktický analyzátor určuje správnosť a syntaktickú štruktúru zdrojového programu. Snaží sa zistiť, či je program syntakticky správne zapísaný – teda zisťuje, či reťazec symbolov vystupujúci z lexikálneho analyzátora patrí do zdrojového jazyka. Ak je reťazec symbolov zapísaný správne, syntaktický analyzátor reťazec patriaci do množiny bezkontextových jazykov spracuje a jeho výstupom je takzvaný derivačný strom alebo postupnosť čísel reprezentujúca poradie pravidiel, ktoré boli pri jednotlivých krokoch syntaktickej analýzy použité. Výstup syntaktického analyzátora, teda derivačný strom či postupnosť použitých pravidiel, je následne vstupom pre ďalšie fázy prekladača.

Derivačný strom predstavuje prostriedok pre zobrazenie štruktúry derivácie v bezkontextovej gramatike.

Definícia 5.0.1. Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Strom je derivačným stromom v G práve vtedy, keď:

1. každý uzol je ohodnotený symbolom z $N \cup T$
2. koreň je ohodnotený symbolom z S
3. ak má uzol najmenej jedného následovníka, tak je ohodnotený symbolom z N
4. ak $b_1, b_2, ..., b_n$ sú priami následovníci uzlu a ohodnoteným symbolom A a v poradí zľava doprava majú ohodnotenie $B_1, B_2, ..., B_n$, tak $A \to B_1B_2...B_n \in P$

Vetná forma, ktorá vznikne pri čítaní listov stromu zľava doprava, sa označuje ako výsledok.

5.1 Zásobníkový automat

Syntax programovacích jazykov sa najčastejšie definuje pomocou bezkontextových gramatík [4]. Bezkontextové gramatiky úzko súvisia so zásobníkovými automatmi. Pre každý bezkontextový jazyk totiž platí, že je bezkontextový práve vtedy, keď môže byť akceptovaný vhodným zásobníkovým automatom.

Zásobníkový automat tvorí teoretický model syntaktického analyzátoru pre bezkontextové jazyky. Vykonáva spomínanú konštrukciu derivačného stromu smerom zhora nadol. To znamená, že konštrukciu začína od koreňa a postupuje dolu, smerom ku vstupnému reťazcu. Hľadá ľavú deriváciu vstupnej vety. Vykonáva pri tom operáciu porovnania a expanzie.

Rozšírený zásobníkový automat tvorí ďalší teoretický model syntaktického analyzátoru pre bezkontextové jazyky. Na rozdiel od zásobníkového automatu vykonáva konštrukciu derivačného stromu v smere zdola nahor, pričom sa vykonávajú operácie posunu a redukcie. Konštrukciu teda začína od vstupného reťazca, ktorý sa snaží redukovať na počiatočný symbol. Hľadá v obrátenom poradí pravú deriváciu príslušnej vety.

Definícia 5.1.1. Deriváciu nazývame ľavou, resp. pravou, ak v každom jej kroku nahrádzame najľavejší, resp. najpravejší, neterminál zostávajúcej vetnej formy. Ak je teda

$$x_1 \Rightarrow x_2 \Rightarrow ... \Rightarrow x_n$$

ľavou, resp. pravou, deriváciou v gramatike $G = (N, T, P, S)$, tak sa pre $1 \leq i \leq n$ dá zapísať x_i ako $y_i A_i z_i$, resp. $z_i A_i y_i$, kde $y_i \in T^*, z_i \in (N \cup T)^*, A_i \rightarrow w_i \in P$ a $x_{i+1} = y_i w_i z_i$ (resp. $x_{i+1} = z_i w_i y_i$). Aj samotné S zodpovedá požadovanému vstupnému tvaru vetnej formy.

5.2 Metódy syntaktickej analýzy

Poznáme 2 prístupy k syntaktickej analýze [4]: syntaktickú analýzu metódou zhora nadol a syntaktickú analýzu metódou zdola nahor.

Definícia 5.2.1. Nech je $G = (N, T, P, S)$ bezkontextová gramatika, ktorá má n pravidiel očíslovaných 1, ..., n a nech $w \in L(G)$. Syntaktická analýza metódou zhora dolu je proces, ktorý vedie k nájdeniu postupnosti čísel pravidiel použitých pri ľavej derivácii vety w. Syntaktická analýza metódou zdola nahor je proces, ktorý vedie k nájdeniu obrátenej postupnosti čísel pravidiel použitých pri pravej derivácii vety w.

5.2.1 Syntaktická analýza zdola nahor

Syntaktická analýza zdola nahor [4] predsatvuje postup syntaktickej analýzy, kedy hľadáme obrátenú postupnosť pravidiel použitých pri pravej derivácii reťazca na vstupe. V prípade, že má byť výstupom syntaktického analyzátora derivačný strom, ide o postup zostavovania derivačného stromu, pri ktorom sa derivačný strom zostavuje zdola smerom nahor. To znamená, že sa najskôr identifikujú jednotlivé terminálne symboly, z ktorých je zložený vstupný reťazec. Následne sa hľadajú pravidlá, na základe ktorých sa vykonáva redukcia. Pri redukcii vznikajú neterminálne symboly a postpne sa analyzátor dostáva ku koreňu stromu.

Poznáme niekoľko rôznych prístupov k syntaktickej analýze smerom zdola nahor [1]. Najznámejším z deterministických prístupov je LR analyzátor, ktorý v prípade potreby umožňuje „pozrieť sa“ na k symbolov, čo sa značí ako LR(k). Využíva dve operácie: posun a redukciu.

Najvšeobecnejším prístupom je nedeterministická metóda používajúca rovnaké operácie. Na vstupe je reťazec, ktorý má byť spracovaný. Operácia posunu presúva symbol zo vstupu na vrchol zásobníka. Ak existuje pravidlo, na ktorého pravej strane sa nachádza kombinácia symbolov na zásobníku, vykoná sa operácia redukcie a tieto symboly sa prepíšu na neterminálny symbol na ľavej strane pravidla. Ak bolo použité nesprávne pravidlo, je potrebné vrátiť sa a hľadať iné použiteľné pravidlo.

Na konci činnosti analyzátora môžu nastať dva prípady: prijatie alebo chyba. Ak zostane v zásobníku len počiatočný symbol a vstup je vyprázdnený, ide o prijatie a reťazec teda patrí do jazyka. V ostatných prípadoch dochádza k chybe a reťazec do jazyka nepatrí.

5.2.2 Syntaktická analýza zhora nadol

Syntaktická analýza metódou zhora nadol [4] predstavuje postup syntaktickej analýzy, kedy hľadáme postupnosť pravidiel použitých pri ľavej derivácii. V prípade zostavovania derivačného stromu postupujeme od koreňa označeného počiatočným symbolom gramatiky smerom dolu k listom, zľava doprava.

Metóda na spracovanie reťazca používa dve operácie so zásobníkom: porovnanie (pop/chyba) a expanziu. Ak sa na vrchole zásobníka nachádza terminálny symbol, porovná sa s prvým symbolom na vstupe a v prípade, že sú rovnaké, sa vykoná operácia pop. Ak rovnaké nie sú, došlo k chybe. Ak sa na vrchole zásobníka nachádza neterminálny symbol, hľadá sa pravidlo, ktoré vychádza z tohto symbolu. Ak sa pravidlo nájde, vykoná sa expanzia, v opačnom prípade dochádza k chybe.

K tejto metóde syntaktickej analýzy existujú dva prístupy [1]. Prvým z nich, najčastejšie používaným, je použitie deterministickej analýzy. Príkladom deterministického analyzátora využívajúceho metódu zhora nadol je LL analyzátor, ktorý sa môže „pozrieť“ na prvých k symbolov na vstupe, čo značíme LL(k). Ak je jazyk dobre navrhnutý, je plne postačujúci analyzátor LL(1). Ďalším zdrojom dodatočných informácií pre rozhodovanie môžu byť symboly nachádzajúce sa aj hlbšie v zásobníku (v prípade klasického zásobníkového automatu sa však upravuje len vrchol zásobníka).

Druhým možným prístupom, avšak menej efektívnym a preto menej využívaným, je analýza s návratom alebo tzv. backtracking. V niektorých prípadoch pri syntaktickej analýze môže dôjsť k situácii, kedy nie je úplne jasné, ktoré z pravidiel sa má použiť, čiže ide o nedeterminizmus. V takom prípade je nutné zvoliť jedno z pravidiel, uložiť stav pred rozhodnutím, a následne vykonať operáciu expanzie podľa zvoleného pravidla. Ak sa ukáže, že zvolené pravidlo nebolo vyhovujúce, musíme sa vrátiť do stavu pred zvolením nesprávneho pravidla a skúsiť vykonať operáciu expanzie s náhradným pravidlom. Ak sú možnosti rozhodovania vnorené do seba, je potrebné sa vrátiť vždy k tomu poslednému, a následne postupovať smerom k starším rozhodovacím „križovatkám“. Tento prístup nie je veľmi vhodný pre syntaktickú analýzu programovacích jazykov, väčšina konštrukcií v programovacích jazykoch však umožňuje deterministickú analýzu bez návratu.

5.3 Syntaktická analýza založená na základe automatu s hlbokým zásobníkom

Syntaktická analýza založená na automate s hlbokým zásobníkom využíva metódy syntaktickej analýzy zhora nadol. Ide o klasický prístup k syntaktickej analýze reťazcov patriacich do bezkontextových jazykov, avšak s tým rozdielom, že je možné v zásobníku upravovať aj

symboly, ktoré sú hlbšie ako len na vrchole zásobníka a dokáže rozpoznať aj reťazce patriace do bezkontextových jazykov s kontextovými prvkami.

Pri svojej činnosti používa rovnaké operácie ako klasický zásobníkový automat. Popis týchto operácií pre automat s hlbokým zásobníkom sa nachádza nižšie.

Operácia POP

Algoritmus 8: POP

Input: $input, Z$
Output: $input[1::], Z[1::], true/false$

```
1: function POP( )
2:     if Z[0] = input[0] then
3:         input = input[1::]
4:         Z = Z[1::]
5:     return true
6:     end if
7:   return false
8: end function
```

Operácia **pop** funguje na rovnakom princípe ako pri syntaktickej analýze pomocou obyčajného zásobníkového automatu. Vstupom je vstupný reťazec *input* a reťazec na zásobníku Z. Výstupom v prípade, že na vrchole zásobníka aj na začiatku vstupného reťazca sú rovnaké symboly, je vstupný reťazec a reťazec na zásobníku bez prvého symbolu. Funkcia v tom prípade vracia $true$. V opačnom prípade funkcia len vráti hodnotu $false$.

Operácia EXPANZIA

Algoritmus 9: EXPANZIA

Input: P, Z, N
Output: Z

```
1: function EXPAND( )
2:   counter = 0
3:   for each z ∈ Z do
4:     if z ∈ N then
5:       counter = counter + 1
6:     end if
7:     if P.depth = counter then
8:       z = P.toString
9:       break
10:    end if
11:  end for
12: end function
```

Operácia **expanzie** funguje podobne ako pri klasickom zásobníkovom automate, no v prípade automatu s hlbokým zásobníkom k expanzii dochádza aj hlbšie v zásobníku. Hĺbka symbolu, ktorý má byť expandovaný, musí zodpovedať hĺbke určenej pravidlom. Vstupom je teda pravidlo P, zásobník Z a množina neterminálnych symbolov gramatiky N. Funkcia cyklom prechádza symboly na zásobníku a v prípade, že nájde neterminál v hĺbke určenej

pravidlom, vykoná operáciu expanzie a tento symbol zmení na reťazec určený pravidlom. Výstupom je usporiadaná množina symbolov na zásobníku.

P.depth v algoritme predstavuje hĺbku pravidla, *counter* predstavuje počítadlo, ktoré hovorí o tom, aká je hĺbka neterminálu a *P.toString* je reťazec, na ktorý sa má symbol v zásobníku zmeniť.

Vzhľadom na to, že na deterministickú analýzu je potrebná vhodná gramatika, prípadne spomínané ďalšie informácie, v práci je využitá syntaktická analýza pracujúca na základe návratu aj napriek zníženej efektívnosti a zvýšenej spotrebe procesorového času.

Funkcia NÁVRAT

Funkcia návratu predstavuje možnosť, ako zabezpečiť, aby sa analyzátor vrátil do bodu, z ktorého vie pokračovať výberom iného pravidla, ktoré by mohlo viesť k úspešnému výsledku. Vstupom je pole konfigurácií $config\_field[]$, v ktorých nastalo vetvenie. K vetveniu dochádza vtedy, keď má analyzátor na výber z viacerých pravidiel, ktoré by mohol použiť. Výstupom je konfigurácia $config$, z ktorej bude následne analyzátor vychádzať, prípadne $NULL$ hovoriaci o tom, že už neexistuje konfigurácia, do ktorej by sa dalo vrátiť.

Algoritmus 10: NÁVRAT

Input: $config\_field[]$
Output: $config/NULL$

```
1:  function COMEBACK( )
2:      if config_field.back.freerules = ∅ then
3:          config_field.pop(config_field.back)
4:      end if
5:      if config_field = ∅ then
6:          return NULL
7:      else
8:          config = config_field.back
9:          return config
10:     end if
11: end function
```

$config\_field.back$ predstavuje poslednú konfiguráciu pridanú z hlavného tela analýzy, $config\_field.back.freerules$ teda reprezentuje množinu použiteľných pravidiel tejto poslednej konfigurácie.

Syntaktická analýza

Samotná syntaktická analýza prebieha v syntaktickom analyzátore, ktorého vstupom je automat s hlbokým zásobníkom ${}_{deep}M$ a vstupný reťazec s. Na začiatku sa inicializuje konfigurácia a pole, ktoré bude obsahovať konfigurácie neskôr využívané na návrat. Následne syntaktická analýza prebieha v cykle, ktorý končí v momente, kedy sa automat dostane do koncového stavu. V cykle sa automat pozrie, čo je prvým symbolom na vstupe a prvým symbolom v zásobníku. Ak ide o terminálne symboly, volá sa operácia pop. Ak operácia pop neuspeje, teda na začiatku nie sú rovnaké terminály, musí sa vykonať návrat. V prípade, že je na zásobníku prvým symbolom neterminál, volá sa funkcia vykonávajúca operáciu expanzie. Ak neuspeje, opäť musí dôjsť k návratu. Ak funkcia návratu nie je schopná vrátiť návratovú konfiguráciu, teda zásobník konfiguácií je už prázdny, syntaktická analýza končí neúspechom

a môžeme prehlásiť, že reťazec na vstupe nepatrí do jazyka generovaného pôvodnou stavovou gramatikou. Ak však skončí úspechom, vstupný reťazec do jazyka generovaného gramatikou patrí.

Algoritmus 11: Syntaktická analýza

Input: ${}_{deep}M, s$
Output: bool hodnota hovoriaca o tom, či reťazec na vstupe patrí alebo nepatrí do jazyka generovaného danou stavovou gramatikou

```
1:  function SYNTAX_ANALYSIS( )
2:      init config
3:      config_field = ∅
4:      while deepM.state <> $ do
5:          if s[0] ∈ T and deepM.pda[0] ∈ T then
6:              if not pop() then
7:                  comeback()
8:              end if
9:          else
10:             if config.freerules.size > 1 then
11:                 config_field.push(config)
12:             end if
13:             if not expand() then
14:                 comeback()
15:             end if
16:         end if
17:     end while
18: end function
```

Výstupom syntaktickej analýzy založenej na automate s hlbokým zásobníkom je len rozhodnutie, či reťazec do daného jazyka patrí alebo nie. Možnosťou je aj finálna postupnosť pravidiel použitých na spracovanie reťazca, no v prípade, že by došlo k návratu, bolo by vždy potrebné postupnosť upraviť. Derivačný strom pre tento typ analýzy nie je možné zostaviť v jeho oficiálnej podobe a jeho definícia by si vyžadovala isté úpravy, ktoré už však nie sú náplňou tejto práce.

Kapitola 6

Príklad

Konverziu stavovej gramatiky na zásobníkový automat pomocou algoritmu a činnosť automatu si ukážeme na jednoduchom príklade. Vstupom je stavová gramatika, ktorá generuje jazyk $L = a^n b^n c^n | n > 0$.

Stavová gramatika je definovaná ako ${}_sG = (\{S, A, C, a, b, c\}, \{p, q, f\}, \{a, b, c\}, P, S)$. Množina pravidiel P obsahuje tieto pravidlá:

- $(p, S) \rightarrow (p, AC)$
- $(p, A) \rightarrow (q, aAb)$
- $(p, A) \rightarrow (f, ab)$
- $(q, C) \rightarrow (p, cC)$
- $(f, C) \rightarrow (f, c)$

Pred konverziou je potrebné si inicializovať automat. HĺBka automatu je daná limitom gramatiky. Inicializuje sa tiež počiatočný stav automatu, ktorý sa zároveň pridá do množiny stavov, a tiež koncový stav \$, ktorý sa tiež pridá do množiny stavov automatu. Vstupnou abecedou do automatu je množina terminálnych symbolov z gramatiky, zásobníkovou abecedou je množina všetkých symbolov doplnená o špeciálny symbol #.

Následne prichádza na rad činnosť algoritmu vykonávajúceho konverziu. Vykonajú sa jednotlivé kroky, čím sa vytvoria pravidlá automatu. Tie sa nakoniec vyfiltrujú, aby sa vo výsledku nenachádzali pravidlá vychádzajúce z nedostupných stavov a opakujúce sa pravidlá. Po filtrácii vznikne nasledujúca množina pravidiel:

Po prvom kroku:

- $1 < s > S \rightarrow < p, S > S$

Po druhom kroku:

- $1 < p, S > S \rightarrow < p, AC > AC$
- $1 < f, C > C \rightarrow < f, > c$
- $1 < p, AC > A \rightarrow < q, AC > aAb$
- $1 < p, AC > A \rightarrow < f, C > ab$
- $1 < f, C > C \rightarrow < f, \# > c$

- $2 < q, AC > C \rightarrow < p, AC > cC$

Po treťom kroku:

- $1 < f, > S \rightarrow < f, S > S$
- $1 < f, > A \rightarrow < f, A > A$
- $1 < f, > C \rightarrow < f, C > C$
- $1 < f, > \# \rightarrow < f, \# > \#$
- $1 < f, \# > S \rightarrow < f, S\# > S$
- $1 < f, \# > A \rightarrow < f, A\# > A$
- $1 < f, \# > C \rightarrow < f, C\# > C$
- $1 < f, \# > \# \rightarrow < f, \#\# > \#$

Po štvrtom kroku:

- $1 < f, \#\# > \# \rightarrow < \$ > \#$

Pomocou týchto pravidiel následne automat môže spracovať daný jazyk. Ukážeme si to na príklade spracovania reťazca *aaabbbccc*. Spočiatku sa na vstupe nachádza celý reťazec, automat je vo svojom počiatočnom stave a v zásobníku sa nachádza len počiatočný symbol S spolu so špeciálnym symbolom $\#$.

$(s, aaabbbccc, S\#)_e \vdash (< p, S >, aaabbbccc, S\#)[1 < s > S \rightarrow < p, S > S]$
$_e \vdash (< p, AC >, aaabbbccc, AC\#)[1 < p, S > S \rightarrow < p, AC > AC]$
$_e \vdash (< q, AC >, aaabbbccc, aAbC\#)[1 < p, AC > A \rightarrow < q, AC > aAb]$
$_p \vdash (< q, AC >, aabbbccc, AbC\#)$
$_e \vdash (< p, AC >, aabbbccc, AbcC\#)[2 < q, AC > C \rightarrow < p, AC > cC]$
$_e \vdash (< q, AC >, aabbbccc, aAbbcC\#)[1 < p, AC > A \rightarrow < q, AC > aAb]$
$_p \vdash (< q, AC >, abbbccc, AbbcC\#)$
$_e \vdash (< p, AC >, abbbccc, AbbccC\#)[2 < q, AC > C \rightarrow < p, AC > cC]$
$_e \vdash (< f, C >, abbbccc, abbbccC\#)[1 < p, AC > A \rightarrow < f, C > ab]$
$_p \vdash^6 (< f, C >, c, C\#)$
$_e \vdash (< f >, c, c\#)[1 < f, C > C \rightarrow < f, \# > c]$
$_p \vdash^6 (< f, \# >, , \#)$
$_e \vdash (< f, \#\# >, , \#)[1 < f, \# > \# \rightarrow < f, \#\# > \#]$
$_e \vdash (< \$ >, , \#)[1 < f, \#\# > \# \rightarrow < \$ > \#]$

Automat sa dostal do koncového stavu a na vstupe nezostal žiadny symbol, reťazec je teda správny a bol spracovaný týmto automatom.

Druhým príkladom môže byť napríklad spracovanie jazyka $L = (0^n 1^n)(0^m 1^m) | n > 0, m >= 0$. Vstupom je stavová gramatika definovaná ako ${}_sG = (\{S, A, 0, 1\}, \{p, q, f\}, \{0, 1\}, P, S)$. Množina pravidiel P obsahuje tieto pravidlá:

- $(p, S) \rightarrow (p, A)$
- $(p, S) \rightarrow (q, AA)$
- $(p, A) \rightarrow (p, 0A1)$

- $(p, A) \rightarrow (f, 01)$
- $(q, A) \rightarrow (p, A)$
- $(f, A) \rightarrow (p, A)$

Algoritmus na konverziu gramatiky nám vygeneruje tieto pravidlá:
Po prvom kroku:

- $1 < s > S \rightarrow < p, S > S$

Po druhom kroku:

- $1 < p, S > S \rightarrow < p, A > A$
- $1 < p, A > A \rightarrow < f, > 01$
- $1 < p, A > A \rightarrow < p, A > 0A1$
- $1 < p, S > S \rightarrow < q, AA > AA$
- $1 < f, A > A \rightarrow < p, A > A$
- $1 < p, AA > A \rightarrow < f, A > 01$
- $1 < p, AA > A \rightarrow < p, AA > 0A1$
- $1 < q, AA > A \rightarrow < p, AA > A$
- $1 < p, A\# > A \rightarrow < f, \# > 01$
- $1 < p, A\# > A \rightarrow < p, A\# > 0A1$
- $2 < p, AA > A \rightarrow < f, A > 01$
- $2 < p, AA > A \rightarrow < p, AA > 0A1$
- $1 < q, AA > A \rightarrow < p, AA > A$

Po treťom kroku:

- $1 < f, > S \rightarrow < f, S > S$
- $1 < f, > A \rightarrow < f, A > A$
- $1 < f, > \# \rightarrow < f, \# > \#$
- $1 < f, \# > S \rightarrow < f, S\# > S$
- $1 < f, \# > A \rightarrow < f, A\# > A$
- $1 < f, \# > \# \rightarrow < f, \#\# > \#$

Po štvrtom kroku:

- $1 < f, \#\# > \# \rightarrow < \$ > \#$

Následne si pomocou tohto vytvoreného automatu spracujeme reťazec 001101.
$(s, 001101, S\#)_e \vdash (< p, S >, 001101, S\#)[1 < s > S \rightarrow < p, S > S]$
$_e \vdash (< q, AA >, 001101, AA\#)[1 < p, S > S \rightarrow < q, AA > AA]$
$_e \vdash (< p, AA >, 001101, AA\#)[1 < q, AA > A \rightarrow < p, AA > A]$
$_e \vdash (< p, AA >, 001101, 0A1A\#)[1 < p, AA > A \rightarrow < p, AA > 0A1]$
$_p \vdash (< p, AA >, 01101, A1A\#)$
$_e \vdash (< f, A >, 01101, 011A\#)[1 < p, AA > A \rightarrow < f, A > 01]$
$_p \vdash^3 (< f, A >, 01, A\#)$
$_e \vdash (< p, A >, 01, A\#)[1 < f, A > A \rightarrow < p, A > A]$
$_e \vdash (< f, >, 01, 01\#)[1 < p, A > A \rightarrow < f, > 01]$
$_p \vdash^2 (< f, >, , \#)$
$_e \vdash (< f, \# >, , \#)[1 < f, > \# \rightarrow < f, \# > \#]$
$_e \vdash (< f, \#\# >, , \#)[1 < f, \# > \# \rightarrow < f, \#\# > \#]$
$_e \vdash (< \$ >, , \#)[1 < f, \#\# > \# \rightarrow < \$ > \#]$

Automat sa dostal do koncového stavu a na vstupe nie je žiadny symbol. Reťazec bol teda spracovaný a vyhodnotený za správny.

Gramatika uvedená v druhom príklade však nie je veľmi ideálna, pretože je veľmi nedeterministická a jej pravidlá nie sú zapísané najlepším spôsobom. V dôsledku horšie napísaných pravidiel gramatiky vznikajú v automate nedeterministické pravidlá, ktoré vychádzajú z rovnakého stavu, končia v rovnakých stavoch, a prepisujú ten istý neterminál v rozdielnej hĺbke. V konkrétnom príklade ide o pravidlá $1 < p, AA > A \rightarrow < f, A > 01$, $1 < p, AA > A \rightarrow < p, AA > 0A1$, $2 < p, AA > A \rightarrow < f, A > 01$ a $2 < p, AA > A \rightarrow < p, AA > 0A1$. Ak máme teda v zásobníku dva symboly „A“ a automat je v stave $< p, AA >$, môže dôjsť k zacykleniu automatu. Keďže ide o syntaktickú analýzu zhora nadol s návratom, automat bez ďalších obmedzení nemusí byť schopný zdetekovať zacyklenie a nesprávnosť reťazca. Vždy pri zvolení pravidla $2 < p, AA > A \rightarrow < p, AA > 0A1$ sa bude prepisovať druhý symbol „A“ a automat môže pokojne expandovať tento druhý symbol donekonečna.

Kapitola 7

Implementácia

Jednoduchý program na konverziu stavovej gramatiky je implementovaný v jazyku C++. Vstupom do programu je stavová gramatika. Program ju spracuje a výstupom je automat s hlbokým zásobníkom. V prípade, že je vstupom aj určitý reťazec, výstupom je navyše aj vyhodnotenie, či daný reťazec patrí do jazyka generovaného gramatikou.

7.1 Vstup

Na vykonanie konverzie je teda potrebná stavová gramatika. Stavová gramatika sa programu predáva pomocou argumentov v textovom súbore pri spúšťaní programu. Program ju prečíta a následne ju konvertuje na zásobníkový automat.

Stavová gramatika musí byť zapísaná v danom formáte, a to:

```
L:
limit
V:
terminál1,neterminál1,terminál2,terminál3,neterminál2...
W:
stav1,stav2,stav3,stav4...
T:
terminál1,terminál2,terminál3...
S:
počiatočný_stav
P:
pravidlo1
pravidlo2
pravidlo3
...
```

kde

- L predstavuje limit gramatiky
- V reprezentuje abecedu gramatiky
- W je množina stavov gramatiky
- S označuje počiatočný stav

- P predstavuje množinu pravidiel, pričom pravidlo musí byť zapísané vo formáte $(stav1, symbol1) -> (stav2, symbol2)$. Ak reťazec *symbol2* obsahuje viac symbolov, jednotlivé symboly musia byť oddelené znakom „.“. *symbol2* teda v tom prípade musí mať formát *symbol2a.symbol2b.symbol2c*.

Ďalším vstupom môže byť reťazec, o ktorom program bude rozhodovať, či do jazyka generovaného gramatikou patrí alebo nepatrí. Reťazec je programu podávaný 2 spôsobmi.

Prvým z nich je predanie pomocou vstupného súboru prostredníctvom argumentu programu. V textovom súbore je napísaný reťazec jazyka, program jazyk prečíta a následne ho automat spracuje. V druhom prípade program reťazec prečíta zo štandardného vstupu a následne ho predá automatu na spracovanie.

Formát jazyka je tiež pevne daný, a to následovne:

$$symbol1.symbol2.symbol3....symbolN$$

Jednotlivé symboly sú, tak ako aj v gramatike, oddelené znakom „.“.

Môj vstup

Príkladom vstupu môže byť napríklad niektorá z gramatík popísaných v predchádzajúcej kapitole. Na obrázku 7.1 je zobrazený výpis súboru gramatiky generujúcej jazyk $L = a^n b^n c^n | n > 0$. Na obrázku 7.2 je zasa výpis súboru gramatiky, ktorá generuje jazyk $L = (0^n 1^n)(0^m 1^m) | n > 0, m >= 0$. Výstup si ukážeme aj v prípade vstupu, ktorý spúšťa spracovanie reťazca. V prvom prípade pôjde o reťazec *a.a.a.b.b.b.c.c.c*, v druhom prípade o reťazec 0.1.0.0.0.0.0.1.1.1.1.1.

```
dominika@dp-pc:~/BP/BP$ cat bp/grammar1.txt
L:
2
V:
S,A,C,a,b,c
W:
p,q,f
T:
a,b,c
S:
S
P:
(p,S)->(p,A.C)
(p,A)->(q,a.A.b)
(p,A)->(f,a.b)
(q,C)->(p,c.C)
(f,C)->(f,c)
```

Obr. 7.1: Gramatika generujúca jazyk $L = a^n b^n c^n | n > 0$

```
dominika@dp-pc:~/BP/BP$ cat grammars/g_01.txt
L:
2
V:
S,A,B,0,1
W:
p,f,q
T:
0,1
S:
S
P:
(p,S)->(p,A)
(p,A)->(f,0.1)
(p,A)->(p,0.A.1)
(p,S)->(q,A.A)
(q,A)->(p,A)
(f,A)->(p,A)
```

Obr. 7.2: Gramatika generujúca jazyk $L = (0^n1^n)(0^m1^m)|n > 0, m >= 0$

7.2 Program

V kóde programu sú importované knižnice `iostream`, `fstream`, `vector`, `string.h` a `regex`. Program sa skladá len z 1 súboru pomenovaného `gta.cc` a prekladá sa pomocou `g++` a štandardom `c++11`. Preklad programu je definovaný v súbore `Makefile`. V priloženom archíve sa nachádza aj adresár s gramatikami `grammars` a adresár s reťazcami generovanými týmito gramatikami `strings`. Použitie týchto adresárov a súborov v nich opisuje súbor `README.md`. Súčasťou je aj doxygen dokumentácia.

7.2.1 Štruktúra programu

Na začiatku súboru sú importované knižnice. Nasleduje definícia všetkých pomocných funkcií, definície dvoch tried a štruktúry definujúce pravidlá gramatiky, pravidlá automatu s hlbokým zásobníkom a konfiguráciu kroku automatu.

Prvou triedou je trieda reprezentujúca stavovú gramatiku `class stateGrammar`. Trieda obsahuje niekoľko atribútov:

- `int limit` reprezentujúci limit gramatiky
- `vector<string> alphabet` predstavujúci abecedu gramatiky (terminálne aj neterminálne symboly)
- `vector<string> states` ako množinu stavov gramatiky
- `vector<string> terminals` reprezentujúci terminálne symboly gramatiky
- `vector<string> nonterminals` reprezentujúci neterminálne symboly gramatiky, pričom jednotlivé neterminály sa získavajú ako rozdiel množiny abecedy a množiny terminálnych symbolov
- `vector<struct gRule> rules` predstavujúci množinu pravidiel gramatiky
- `string startSymbol` reprezentujúci počiatočný symbol gramatiky

Štruktúra `struct gRule` predstavuje pravidlo gramatiky, pričom obsahuje 4 položky typu `string`:

- `fromState` je stav, z ktorého pravidlo vychádza
- `fromString` je reťazec (symbol), ktorý bude pravidlo prepisovať
- `toState` je stav, do ktorého sa po prechode gramatika dostane
- `toString` je reťazec (symbol), prípadne reťazce (symboly), na ktoré sa pôvodný reťazec (`fromString`) bude prepisovať

Trieda obsahuje 3 metódy: metódu na inicializáciu gramatiky `void get_grammar(string grammarFile)`, metódu na kontrolu správnosti naplnenia gramatiky `void check_if_full()` a metódu na výpis tejto gramatiky `void print_grammar()`.

Druhou triedou je trieda `class deepPDA` reprezentujúca automat s hlbokým zásobníkom, ktorá obsahuje atribúty:

- `int limit` určujúci hĺbku automatu
- `vector<string> states` predstavujúci množinu stavov automatu
- `vector<string> inAlphabet` predstavujúci vstupnú abecedu (terminálne symboly gramatiky)
- `vector<string> pdaAlphabet` predstavujúci zásobníkovú abecedu (celá abeceda gramatiky, počiatočný symbol automatu aj špeciálny symbol #)
- `string startState` reprezentujúci počiatočný stav
- `string startSymbol` reprezentujúci počiatočný symbol
- `vector<string> endStates` predstavujúci množinu koncových stavov
- `vector<struct dpdaRule> rules` predstavujúci množinu pravidiel automatu

Štruktúra `struct dpdaRule` predstavuje pravidlo gramatiky a obsahuje 5 položiek:

- `int depth` je hĺbka v zásobníku, v ktorej sa má daný neterminál prepisovať
- `string fromState` je stav, z ktorého pravidlo vychádza
- `string fromString` je reťazec (symbol), ktorý bude pravidlo expandovať (v hĺbke danej `int limit`)
- `string toState` je stav, do ktorého sa po konfigurácii automat dostane
- `string toString` je reťazec (symbol), prípadne reťazce (symboly), na ktoré pôvodný reťazec (`fromString`) expanduje

Trieda obsahuje 7 metód: metódu na filtráciu pravidiel `void delete_rules()`, metódu na výpis automatu `void print_dpda()` vo formáte popísanom nižšie a 5 metód slúžiacich pri spracovaní jazyka. Prvou z nich je hlavná metóda, `bool accept_or_reject(string inString, stateGrammar &grammar)`, slúžiaca priamo na spracovanie, ostatné sú jej pomocnými metódami. Týmito metódami sú `vector <struct dpdaRule> get_rules(string`

`stat)` na získavanie použiteľných pravidiel, `bool op_pop(vector <string> &ins, vector <string> &pds)` vykonávajúca operáciu pop, `bool op_expansion(vector <string> &pds, struct dpdaRule r, vector <string> nonterms)` vykonávajúca operáciu expanzie a `bool get_back_config(struct configuration &config, vector <struct configuration> &cs)`, ktorá zabezpečuje návrat k poslednej použiteľnej vetviacej sa konfigurácii. Štruktúra konfigurácie predstavuje krok pri spracovávaní jazyka automatom. Obsahuje 5 položiek:

- `string state` je aktuálny stav automatu v danom kroku
- `string inputString` je reťazec na vstupe automatu
- `string pushdownString` je reťazec v zásobníku automatu
- `size_t count` slúži na indexovanie toho, ktoré pravidlo sa bude používať v prípade, že má automat možnosť vybrať si z viacerých pravidiel na použitie
- `vector <struct dpdaRule> rulesToBeUsed` predstavuje množinu pravidiel, ktoré vychádzajú zo stavu `state` a teda môžu byť pri spracovaní jazyka v aktuálnom kroku použité. Ak je počet týchto pravidiel väčší ako 1, vieme, že sa program bude musieť rozvetviť a je potrebné si konfiguráciu uložiť pre prípad, že by bolo nutné sa k nej vrátiť a zvoliť iné pravidlo.

Jednotlivé triedy sú inicializované v hlavnom tele programu v `int main(int argc, char *argv[])`. Argumenty programu zadané pri spúšťaní sa spracujú a určí sa, čo sa bude v programe vykonávať.

Ak je program spustený s jediným argumentom `-help`, na štandardný výstup sa vypíše návod na použitie a potrebný formát gramatiky.

Ak je argumentov viac, prvý z nich musí byť `-g`, za ktorým nasleduje názov súboru s gramatikou. Ďalším argumentom, zapísaným po gramatike, môže byť argument `-a`, ktorý určuje, že na štandardný výstup sa vypíše výsledný automat vzniknutý z gramatiky. V tejto vetve sa volajú funkcie na spracovanie gramatiky a následne funkcie vytvárajúce automat s hlbokým zásobníkom, ktorý sa napokon vypíše na štandardný výstup.

Ďalšou možnosťou argumentu priamo po gramatike môže byť argument `-s`, ktorý programu hovorí, že sa okrem vytvorenia automatu bude spracovávať aj určitý reťazec, ktorý bude zapísaný na štandardný vstup alebo sa nachádza v súbore, ktorého názov je zapísaný hneď po argumente `-s`.

7.2.2 Funkcie

`void errorExit(string message)`

Funkcia volaná vždy, keď sa má ukončiť program s chybovým hlásením. Chybové hlásenie prestavuje parameter funkcie `string message` a toto hlásenie sa vypíše na chybový výstup. Následne ukončí program s návratovým kódom -1.

`void get_grammar(string grammarFile)`

Funkcia slúži na spracovanie gramatiky a je implementovaná ako metóda triedy `class stateGrammar`. Parameter `string grammarFile` predstavuje názov súboru, z ktorého sa gramatika načíta a spracuje. Názov je programu predaný pomocou argumentu pri spúšťaní.

Ak sa súbor podarí otvoriť, vo funkcii prebieha čítanie po jednotlivých riadkoch a gramatika sa postupne inicializuje. Program súbor číta po riadkoch. Štruktúra gramatiky a

správnosť jednotlivých riadkov sú kontrolované pomocou regulárnych výrazov a porovnávania reťazcov. Ak program nájde riadok zodpovedajúci jednej z položiek (L, V, W, T, S, P) a nasledujúci zápis definujúci položku je tiež správny, funkcia tento zápis spracuje a inicializuje zodpovedajúci atribút gramatiky.

V prípade, že gramatika nie je zapísaná správne, program sa ukončí a vypíše zodpovedajúce chybové hlásenie. V prípade, že má gramatika zlý formát, vypíše, že je tento formát nesprávny, v prípade, že je nesprávne zapísané pravidlo, vypíše, že je nesprávne zapísané pravidlo. Na konci funkcie je volaná funkcia na kontrolu naplnenia gramatiky.

`void check_if_full()`

Funkcia je volaná na konci funkcie `get_grammar(string grammarFile)` a kontroluje, či sa správne naplnili atribúty gramatiky. Ak je ktorýkoľvek z nich prázdny, program skončí a vyhodí chybové hlásenie, ktoré hovorí, že chýba niektorý z atribútov – teda to, že sa atribúty nenaplnili správne.

`void print_grammar()`

Funkcia je definovaná ako metóda triedy `class stateGrammar` a slúži na výpis gramatiky na štandardný výstup vo formáte popísanom vyššie.

`void parse_and_push(string str,vector<string> &array, string delimiter)`

Funkcia, ktorá rozdelí reťazec `string str` na jednotlivé symboly podľa `string delimiter` a následne ich povkladá do poľa `vector<string> &array`. Je volaná vo funkcii `void get_grammar (string grammarFile)` vždy, keď treba spracovať riadok v súbore gramatiky a rozdeliť ho na jednotlivé symboly/stavy. Je tiež volaná vo funkciách slúžiacich na spracovanie automatu vždy, keď reťazec treba rozdeliť na jednotlivé znaky a vložiť ich do poľa.

`void subtract_sets(vector<string> &from,vector<string> &what, vector<string> &where)`

Funkcia, ktorá od množiny `vector<string> &from` odráta množinu `vector<string> &what` a výsledok rozdielu vloží do množiny `vector<string> &where)`.

`bool correct_rule(string line)`

Funkcia na kontrolu správnosti pravidla gramatiky. Pomocou regulárneho výrazu skontroluje, či je pravidlo zapísané v správnom formáte, inak vedie ku chybovej hláške volanej z `void get_grammar(string grammarFile)`.

`void process_rule(string original, vector<struct gRule> &r)`

Funkcia, ktorá slúži na spracovanie pravidla gramatiky. Je volaná v prípade, že zápis pravidla prešiel kontrolou správnosti. Zápis spracuje a naplní štruktúru pravidla gramatiky správnymi údajmi.

`void print_dpda()`

Funkcia je definovaná ako metóda triedy `class deepPDA` a slúži na výpis automatu na štandardný výstup vo formáte popísanom nižšie.

`void delete_rules()`

Funkcia je implementovaná ako metóda triedy `class deepPDA`. Využíva sa na filtráciu množiny pravidiel – prechádza všetky pravidlá a vyberá všetky dostupné stavy. Následne v množine pravidiel necháva len tie pravidlá, ktoré vychádzajú z dostupných stavov. Nakoniec sa odstránia všetky duplikáty pravidiel a zostáva vyfiltrovaná výsledná množina pravidiel.

`vector <struct dpdaRule> get_rules(string stat)`

Funkcia implementovaná ako metóda triedy `class deepPDA`, ktorá slúži na nájdenie pravidiel, ktoré môžu byť následne použité pri danej konfigurácii. Pravidlá, ktoré vracia, vyhľadáva na základe stavu určeného parametrom `string stat`. Prechádza jednotlivé pravidlá a ak nájde stav, z ktorého pravidlo vychádza, ktorý je zhodný so stavom `string stat`, pravidlo pridá do poľa. Po prejdení všetkých pravidiel funkcia vráti toto pole.

`bool op_pop(vector <string> &ins, vector <string> &pds)`

Metóda triedy `class deepPDA`, ktorá implementuje operáciu pop. V parametroch sa jej predajú polia reprezentujúce symboly na vstupnej páske automatu `vector <string> &ins` a symboly na zásobníku automatu `vector <string> &pds)`, pričom operácia zabezpečí, že sa v oboch odstráni prvý symbol. Ak sa operácia podarí, teda prvý symbol oboch polí je rovnaký, funkcia vráti `true`. V inom prípade vráti `false`.

`bool op_expansion(vector <string> &pds, struct dpdaRule r, vector <string> nonterms)`

Metóda triedy `class deepPDA`, ktorá implementuje operáciu expanzie. V parametri sa jej predá pole `vector <string> &pds` predstavujúce zásobník automatu, v ktorom sa expanzia bude vykonávať, pravidlo `struct dpdaRule r`, na základe ktorého sa expanzia vykoná a pole `vector <string> nonterms` reprezentujúce neterminálne symboly gramatiky. Funkcia prechádza symboly na zásobníku, pričom hľadá neterminálny symbol v hĺbke určenej pravidlom `struct dpdaRule r`. Ak ho nájde a ide o neterminálny symbol zodpovedajúci symbolu, z ktorého pravidlo vychádza, vykoná operáciu expanzie – symbol nahradí symbolom alebo skupinou symbolov podľa pravidla. Ak sa operácia vydarí, vráti `true`, inak vracia `false`.

`bool get_back_config(struct configuration &config, vector <struct configuration> &cs)`

Metóda triedy `class deepPDA`, ktorá implementuje návrat k poslednej použitej vetviacej konfigurácii. Prvým parametrom je aktuálna konfigurácia, ktorá sa na základe návratu bude musieť prepísať. Druhým parametrom je pole, resp. zásobník obsahujúci konfigurácie, ku ktorým sa následne automat môže vrátiť.

Funkcia je volaná v prípade, že automat zistil, že ďalej pri aktuálnom stave nemôže pokračovať v spracovávaní jazyka a musí sa vrátiť do posledného stavu, o ktorom si myslí, že bol ešte správny. Opiera sa o položku konfigurácie `size_t count`, ktorá predstavuje index pravidla, ktoré bolo použité. Na začiaku navýši túto položku v poslednej konfigurácii na zásobníku.

Ak je hodnota položky väčšia alebo rovná počtu pravidiel v položke `vector <struct dpdaRule> rulesToBeUsed`, funkcia vykoná operáciu pop v zásobníku konfigurácií. Ak je

po tomto úkone zásobník prázdny, funkcia vráti `false`. Ak zásobník prázdny nie je, do aktuálnej konfigurácie sa priradí konfigurácia na vrchole zásobníka konfigurácií s navýšenou položkou `size_t count` o 1.

V opačnom prípade, teda v prípade, že `size_t count` nie je väčší alebo rovný počtu pravidiel, do konfigurácie sa len priradí posledná konfigurácia zo zásobníka.

V oboch prípadoch, kedy sa návrat podaril, funkcia vráti `true`.

`bool accept_or_reject(string inString, stateGrammar &grammar)`

Metóda triedy `class deepPDA`, ktorá spracováva reťazec na vstupe a vyhodnocuje, či reťazec patrí do jazyka generovaného gramatikou. Reťazec na vstupe je predaný funkcii pomocou prvého parametru `string inString`. Druhý parameter predstavuje gramatiku, z ktorej bol automat vytvorený.

Funkcia je volaná z hlavného tela programu vtedy, keď je programu predaný argument, ktorý značí, že sa má spracovať aj jazyk na vstupe. Na začiatku sa inicializuje prvá, počiatočná konfigurácia `struct configuration config`, z ktorej bude automat vychádzať. Ďalej sa definuje pole reprezentujúce zásobník vetviacich konfigurácií `vector <struct configuration> cs`, pole reprezentujúce zásobník automatu `vector <string> pds` a tiež pole reprezentujúce vstupnú pásku automatu `vector <string> ins`. Následne sa jazyk vyhodnocuje v cykle. Cyklus končí, keď sa automat dostane do koncového stavu.

V cykle sa vyhodnocuje, ktorá operácia bude vykonaná. Ak sú prvým symbolom na oboch zásobníkoch terminálne symboly, zavolá sa operácia pop. Ak operácia pop vráti `true` a je splnená podmienka prázdnej vstupnej pásky a prázdneho zásobníka, funkcia vracia `true`. Keď podmienka nie je splnená, prebehne kontrola, či nenastal jeden z chybných stavov. Týmito chybnými stavmi je napríklad prázdna vstupná páska ale neprázdny zásobník, prípadne prázdny zásobník ale neprázdna vstupná páska. Ak nastal jeden z týchto chybných stavov alebo operácia pop vrátila hodnotu `false`, funkcia volá funkciu zabezpečujúcu návrat k poslednej správnej konfigurácii, aby mal program možnosť použiť iné, lepšie pravidlo.

Ak však prvým symbolom nie je terminál, funkcia zavolá operáciu expanzie. V prípade, že sa daná konfigurácia ešte v zásobníku nevyskytuje, teda jej položka `size_t count` je rovná 0, a že počet možných pravidiel je väčší ako 1, sa táto konfigurácia vloží na vrchol zásobníka konfigurácií. Následne prebehne operácia expanzie. Ak operácia expanzie vráti hodnotu `true`, vyhodnotí sa nová, aktuálna konfigurácia. Keď však vráti `false`, zavolá sa funkcia zabezpečujúca návrat k poslednej vetviacej sa konfigurácii.

Ak funkcia návratu `bool get_back_config(struct configuration &config, vector <struct configuration> &cs)` vráti `true`, program pokračuje ďalej od aktuálnej konfigurácie. Vrátená hodnota `false` však znamená, že pre návrat už neexistuje žiadna ďalšia možná konfigurácia, a funkcia vráti hodnotu `false`.

`bool in_vector(vector <string> vec,string st)`

Funkcia je v programe volaná veľmi často. Slúži na zistenie, či sa reťazec `string st` nachádza v množine reťazcov `vector <string> vec`.

`int count_symbols(string s)`

Funkcia, ktorá slúži na výpočet počtu symbolov v reťazci `string s`. Symboly sú rozdelené znakom „.“. Je volaná vždy, keď je potrebné vypočítať dĺžku reťazca symbolov.

`bool check_string(vector<string> vec,string x)`

Funkcia, ktorá slúži na kontrolu, či všetky symboly z reťazca symbolov `string x` zodpovedajú symbolom abecedy gramatiky `vector<string> vec`.

`void dpda_init(deepPDA &dpda,stateGrammar &grammar)`

Funkcia slúžiaca na inicializáciu automatu s hlbokým zásobníkom. Inicializuje sa počitočný stav, počiatočný symbol a hĺbka automatu. Do množiny koncových stavov sa priradí „$“, do množiny všetkých stavov sa priradí koncový aj počiatočný stav. Do množiny vstupnej abecedy sa priradia terminály z gramatiky, do zásobníkovej abecedy sa priradia terminálne aj neterminálne symboly z gramatiky a špeciálny symbol „#“.

`void dpda_step1(deepPDA &dpda,stateGrammar &grammar)`

Funkcia implementujúca prvý krok algoritmu na konverziu stavovej gramatiky. Prechádzajú sa pravidlá stavovej gramatiky. Ak sa nájde pravidlo, v ktorom reťazec `fromString` zodpovedá počiatočnému symbolu gramatiky, štruktúra `struct dpdaRule` sa naplní zodpovedajúcimi údajmi a vloží sa do množiny pravidiel.

`void dpda_step2(deepPDA &dpda,stateGrammar &grammar)`

Funkcia implementujúca druhý krok algoritmu na konverziu stavovej gramatiky. Na začiatku sa pomocou funkcií `make_u()` a `make_v_2()` vygenerujú množiny reťazcov u a v. Tieto množiny sa prechádzajú a vyberajú sa kombinácie vyhovujúce podmienke $|uv| \leq n - 1$. Následne sa prechádzajú všetky stavy a neterminály, pričom všetky vyhovujúce kombinácie vedú k naplneniu štruktúry `struct dpdaRule`, ktorá sa nakoniec vloží do množiny pravidiel.

`void dpda_step3(deepPDA &dpda,stateGrammar &grammar)`

Funkcia implementujúca tretí krok algoritmu na konverziu stavovej gramatiky. Na začiatku sa pomocou funkcií `make_u()` a `make_v_3()` vygenerujú množiny reťazcov u a v. Opäť sa kontroluje platnosť podmienky $|uv| \leq n - 1$ a ďalších. Tentokrát sa štruktúra `struct dpdaRule` naplní dvojnásobne, vzhľadom na to, že v tomto kroku pribúdajú až 2 pravidlá na 1 vyhovujúcu kombináciu podmienok. Vzniká tu najviac duplikátov pravidiel.

`void dpda_step4(deepPDA &dpda,stateGrammar &grammar)`

Funkcia implementujúca štvrtý krok algoritmu na konverziu stavovej gramatiky. Prechádzajú sa všetky stavy z gramatiky a pri každom sa naplní štruktúra `struct dpdaRule`. Tá sa potom, ako pri všetkých krokoch, vloží do množiny pravidiel.

`string make_state(string state,string str)`

Funkcia, ktorá vytvorí zápis stavu automatu v tvare *stav,str*, ktorý sa následne priradzuje do stavu v štruktúre pravidla..

`vector<string> make_u(vector<string> nonterms,int limit)`

Funkcia, ktorá vygeneruje všetky možné kominácie reťazcov symbolov, pričom počet symbolov v reťazci je obmedzený parametrom funkcie `int limit`. Symboly, z ktorých sa gene-

rujú reťazce symbolov, sú neterminály gramatiky a sú funkcii predané pomocou parametra `vector<string> nonterms`. Funkcia je využitá vo funkcii `dpda_step2()` aj `dpda_step3()` na inicializáciu množiny U.

`vector<string> make_v_2(vector<string> nonterms,int limit)`

Funkcia, ktorá vygeneruje všetky možné kominácie reťazcov symbolov, pričom počet symbolov v reťazci je obmedzený parametrom funkcie `int limit`. Symboly, z ktorých sa generujú reťazce symbolov, sú neterminály gramatiky, ktoré sú funkcii predané pomocou parametra `vector<string> nonterms`, a tiež špeciálny symbol #. Funkcia je volaná vo funkcii `dpda_step2` na inicializáciu množiny V.

`vector<string> make_v_3(int limit)`

Funkcia, ktorá vygeneruje všetky reťazce zložené zo špeciálneho symbolu # do maximálneho počtu symbolov určeného parametrom `int limit`. Funkcia je volaná vo funkcii `dpda_step3` na inicializáciu množiny V.

`bool gStates(string u,string state)`

Funkcia volaná na kontrolu podmienok vo funkcii `dpda_step2` aj `dpda_step3`. Kontroluje, či sa stav (`string state`) vyskytuje v reťazci symbolov `string u`.

`string get_nonterminals(vector<string> nonterminals,string from)`

Funkcia, ktorá reprezentuje funkciu f(x) popísanú vyššie. Z reťazca symbolov `string from` vyberie a vráti len neterminálne symboly, ktoré vie posúdiť na základe ich prítomnosti vo vektore neterminálov z gramatiky predaného funkcii pomocou jej parametru `vector<string> nonterminals`. Je volaná vo funkcii `dpda_step2` pri vytváraní stavu.

`string prefix(string str,int limit)`

Funkcia volaná vo funkcii `dpda_step2`. Vracia symboly z reťazca symbolov `string str` v počte určenom `int limit`, pričom ich rozdeľuje podľa znaku „.“.

`void make_dpda(deepPDA &dpda,stateGrammar &grammar)`

Funkcia volaná priamo z `main()`, v ktorej sú volané všetky funkcie reprezentujúce jednotlivé kroky konverzie a funkcia na filtráciu pravidiel automatu.

`string make_string(vector<string> str)`

Funkcia, ktorá slúži na vytvorenie reťazca symbolov oddelených znakom „.“ z poľa `vector<string> str`.

`string get_language(string language)`

Funkcia volaná z hlavného tela programu, ktorá zo vstupného súboru s názvom predaným pomocou parametru `string language` prijme reťazec, ktorý následne automat spracuje.

7.3 Výstup

Výstupom programu je zásobníkový automat, prípadne vyhodnotenie, či zadaný reťazec patrí do jazyka spracovávaného týmto automatom.

Ak bol pri spustení použitý argument `-a`, na štandardný výstup sa tento vytvorený automat vypíše vo formáte:

Depth:
limit
States:
(stav1),(stav2),(stav3)...
Input alphabet:
symbol1,symbol2,symbol3...
PDA alphabet:
symbol1,symbol2,symbol3...
Start state:
počiatočný_stav
End states:
koncový_stav1,koncový_stav2,koncový_stav3...
Start symbol:
počiatočný_symbol
Rules:
pravidlo1
pravidlo2
...

---

kde

- *Depth* predstavuje hĺbku automatu
- *States* reprezentuje množinu všetkých stavov
- *Input alphabet* je vstupná abeceda
- *PDA alphabet* je zásobníková abeceda
- *Start state* označuje počiatočný stav
- *End states* predstavuje množinu koncových stavov
- *Start symbol* označuje počiatočný symbol na zásobníku
- *Rules* reprezentuje množinu pravidiel, pričom pravidlá sú na výstupe vo formáte $hbka < stav > symbol\_v\_zsobnku \rightarrow < stav > symbol(y)\, expanzie$ podľa zápisu v algoritme. Ak je reťazec symbolov zložený z viacerých symbolov, jednotlivé symboly sú oddelené bodkou, rovnako ako v prípade, keď sa v označení stavu nachádza viac ako 1 symbol.

Ak bol však pri spustení použitý argument `-s`, program vytvorí automat s hlbokým zásobníkom z danej stavovej gramatiky a následne pomocou tohto automatu spracuje zadaný jazyk. Toto spracovanie sa vykonáva vo funkcii `bool accept_or_reject(string inString, stateGrammar &grammar)`. Ak táto funkcia vráti hodnotu `true`, reťazec je automatom vyhodnotený za správny a teda patrí do jazyka. Na štandardný výstup sa vypíše

„ACCEPTED“. Naopak, ak automat zistí, že reťazec nepatrí do jazyka generovaného danou gramatikou, teda funkcia vráti hodnotu `false`, na štandardný výstup sa vypíše „REJECTED“.

Môj výstup

Výstupom programu môže byť napríklad obrázok 7.3, ktorý zobrazuje výstup vypísaný v prípade parametra `-a` aj `-s` vtedy, ak je vstupom gramatika generujúca jazyk $L = (0^n1^n)(0^m1^m)|n > 0, m >= 0$. Obrázok 7.4 zobrazuje oba prípady výstupu v prípade, že je vstupom gramatika, ktorá generuje jazyk $L = a^nb^nc^n|n > 0$.

```
dominika@dp-pc:~/BP/BP$ ./gta -g grammars/g_01.txt -a
Depth:
2
States:
(s),($),(p,S),(p,A),(q,A.A),(f,),(p,A.A),(f,S),(f,#),(f,A),(f,S.#),(f,#.#),(f,A.
#),(p,A.#),
Input alphabet:
0,1,
PDA alphabet:
0,1,S,A,#,
Start state:
s
End states:
($),
Start symbol:
S
Rules:
1<s>S -> <p,S>S
1<p,S>S -> <p,A>A
1<p,A>A -> <f,>0.1
1<p,A>A -> <p,A>0.A.1
1<p,S>S -> <q,A.A>A.A
1<f,A>A -> <p,A>A
1<p,A.A>A -> <f,A>0.1
1<p,A.A>A -> <p,A.A>0.A.1
1<q,A.A>A -> <p,A.A>A
1<p,A.#>A -> <f,#>0.1
1<p,A.#>A -> <p,A.#>0.A.1
1<f,A.#>A -> <p,A.#>A
2<p,A.A>A -> <f,A>0.1
2<p,A.A>A -> <p,A.A>0.A.1
2<q,A.A>A -> <p,A.A>A
1<f,>S -> <f,S>S
1<f,># -> <f,#>#
1<f,>A -> <f,A>A
1<f,#>S -> <f,S.#>S
1<f,#># -> <f,#.#>#
1<f,#>A -> <f,A.#>A
1<f,#.#># -> <$>#
dominika@dp-pc:~/BP/BP$ ./gta -g grammars/g_01.txt -s
0.1.0.0.0.0.0.1.1.1.1.1
ACCEPTED
```

Obr. 7.3: Gramatika generujúca jazyk $L = (0^n1^n)(0^m1^m)|n > 0, m >= 0$

```
dominika@dp-pc:~/BP/BP$ ./gta -g bp/grammar1.txt -a
Depth:
2
States:
(s),($),(p,S),(p,A.C),(q,A.C),(f,C),(f,),(f,S),(f,#),(f,A),(f,S.#),(f,#.#),(f,A.#),(f,C.#),
Input alphabet:
a,b,c,
PDA alphabet:
a,b,c,S,A,C,#,
Start state:
s
End states:
($),
Start symbol:
S
Rules:
1<s>S -> <p,S>S
1<p,S>S -> <p,A.C>A.C
1<f,C>C -> <f,>c
1<p,A.C>A -> <q,A.C>a.A.b
1<p,A.C>A -> <f,C>a.b
1<f,C.#>C -> <f,#>c
2<q,A.C>C -> <p,A.C>c.C
1<f,>S -> <f,S>S
1<f,># -> <f,#>#
1<f,>A -> <f,A>A
1<f,>C -> <f,C>C
1<f,#>S -> <f,S.#>S
1<f,#># -> <f,#.#>#
1<f,#>A -> <f,A.#>A
1<f,#>C -> <f,C.#>C
1<f,#.#># -> <$>#
dominika@dp-pc:~/BP/BP$ ./gta -g bp/grammar1.txt -s
a.a.a.b.b.b.c.c.c
ACCEPTED
```

Obr. 7.4: Gramatika generujúca jazyk $L = a^n b^n c^n | n > 0$

7.4 Preklad a spustenie

V `.zip` súbore sa nachádza program na konverziu gramatiky `gta.cpp`, súbor `makefile` na preklad a tiež 3 priložené testovacie gramatiky a príklady reťazcov jazykov, ktoré tieto gramatiky generujú.

Preklad

Po rozbalení `.zip` súboru sa pomocou príkazu `make` program preloží. Program je prekladaný príkazom `g++ -std=c++11 -Wall -Wextra -pedantic gta.cc -o gta`.

Spustenie

Keď už je program preložený, následne sa môže spustiť pomocou príkazu `./gta -g grammar.txt -a`, kde argument `-g` symbolizuje, že bude následovať textový súbor s gramatikou `grammar.txt`, namiesto ktorého sa využije súbor s gramatikou v správnom formáte (nesprávny formát vyvolá chybový výpis a koniec programu), a argument `-a`, ktorý symbolizuje, že program má na štandardný výstup vypísať len vytvorený automat.

Druhou možnosťou je spustenie programu pomocou príkazu `./gta -g grammar.txt -s [string.txt]`. Rovnako ako v prvom prípade, `-g` symbolizuje, že bude následovať tex-

tový súbor s gramatikou `grammar.txt`. `-s` symboolizuje, že program pomocou automatu vytvoreného z gramatiky skúsi spracovať reťazec, ktorý sa nachádza v súbore `string.txt`, prípadne sa tento reťazec načíta zo štandardného vstupu. Na štandardný výstup sa vypíše vyhodnotenie príslušnosti reťazca do jazyka.

Kapitola 8

Záver

Cieľom tejto práce bolo detailne sa zoznámiť s automatmi s hlbokým zásobníkom a ich vlastnosťami a tieto vlastnosti naštudovať, následne navrhnúť metódu syntaktickej analýzy založenú na týchto automatoch a využiť ju na syntaktickú analýzu navrhnutého jazyka. Spolu s automatmi s hlbokým zásobníkom bolo potrebné zoznámiť sa aj so stavovou gramatikou vzhľadom na jej úzke prepojenie s týmito automatmi pri ich vytváraní. Dôležitým krokom bolo dôkladné štúdium a pochopenie popisu algoritmu na konverziu gramatiky, ktorého tvorba predstavuje jadro tejto práce. Poslednou podstatnou časťou štúdia bola samotná syntaktická analýza, na ktorú sa vytvorený automat mal využiť. Všetky ciele práce boli splnené.

Na základe popisu algoritmu na konverziu stavovej gramatiky na automat s hlbokým zásobníkom bol vytvorený a popísaný samotný algoritmus spolu s pomocnými funkciami, ktoré využíva. Podľa vytvoreného algoritmu bol naimplementovaný program, ktorý túto konverziu dokáže vykonať a následne odstráni z množiny pravidiel automatu všetky nepotrebné pravidlá, teda tie, ktoré sa opakujú alebo vychádzajú z nedosiahnuteľných stavov. Nakoniec bol program rozšírený aj o možnosť časti syntaktickej analýzy, ktorá vyhodnocuje príslušnosť reťazca do jazyka. Túto rozpoznávaciu časť bolo možné na základe vytvoreného automatu vykonať. Program dokáže metódou syntaktickej analýzy zhora nadol vyhodnotiť príslušnosť zadaného reťazca do jazyka špecifikovaného zadanou stavovou gramatikou, avšak žiadny ďalší výstup vzhľadom na zatiaľ neznámu dostupnosť riešenia neposkytuje. Riešenie vykonáva syntaktickú analýzu s návratom, čo nie je najefektívnejší spôsob realizácie, avšak v úvodnej fáze rozpracovania zatiaľ jediný možný. Program dokáže spracovať akúkoľvek gramatiku, avšak v niektorých prípadoch, kedy gramatika nie je úplne správne definovaná, môže dôjsť k odmietnutiu aj správne zapísaného reťazca. V prípade zle definovaných pravidiel hrozilo zacyklenie programu, a z toho dôvodu má automat obmedzenú veľkosť zásobníka.

Z celkového hľadiska teda nejde o najefektívnejšie riešenie syntaktickej analýzy programovacích jazykov vzhľadom na možnosť pomerne časovo náročného výpočtu v prípade, že by sa počas behu programu návrat musel vykonávať príliš často z dôvodu komplexnosti programovacích jazykov. Vďaka svojej schopnosti rozpoznávať aj bezkontextové jazyky s kontextovými prvkami by však predstavoval riešenie „neuhladenosti“ aktuálneho ad hoc spôsobu spracovávania týchto kontextových prvkov programovacích jazykov [3]. Mohol by predstavovať istú normu, pomocou ktorej by sa tieto jazyky dali elegantnejšie a jednoduchšie spracovávať.

Momentálnym príkladom využitia stavovej gramatiky je napríklad práca zaoberajúca sa jej rozšírením na jej využitie v popise biomolekulárnych štruktúr a následné modelovanie

DNA a RNA štruktúr [2]. Stavová gramatika pre podobné prípady predstavuje ideálne riešenie, a automat s hlbokým zásobníkom by výrazne mohol uľahčiť kontrolu správnosti týchto štruktúr či podobných prípadov.

Efektívnejšie riešenie syntaktickej analýzy pomocou tohto automatu by bolo možné dosiahnuť deterministickým riešením. Vytvorený automat by mohol byť deterministický, ak by bola ekvivalentná gramatika dobre navrhnutá. Program by v budúcnosti mohol byť schopný určiť, či je zadaná gramatika vhodná, a teda či sa z nej dá vytvoriť deterministický automat s hlbokým zásobníkom. V prípade, že by bola gramatika taká, že by sa z nej dal vytvoriť deterministický automat, riešenie by znížilo časovú aj pamäťovú náročnosť. Program by si nemusel odpamätávať jednotlivé stavy z dôvodu návratu, čo by ušetrilo pamäť, a nemusel by návraty vôbec vykonávať, čo by ušetrilo procesorový čas potrebný na vykonanie syntaktickej analýzy. V prípade, že by gramatika nebola vhodná, program by mohol zadávateľa požiadať o prepis gramatiky do iného, lepšieho tvaru. Ideálne riešenie by mohla predstavovať „LL(k) verzia“ stavovej gramatiky, prípadne by táto verzia naviac mohla byť rozšírená o ďalšie informácie.

Ďalším možným efektívnejším riešením by bolo vytvorenie rozšíreného automatu s hlbokým zásobníkom, ktorý by dokázal vykonávať syntaktickú analýzu smerom zdola nahor, podobne ako je to pri klasickom zásobníkovom a rozšírenom zásobníkovom automate. Nie je však isté, či by toto riešenie bolo možné vzhľadom na hĺbku automatu a jeho schopnosť pracovať so symbolmi aj ďalej v zásobníku, nielen na jeho vrchole.

Keďže teória bezkontextových jazykov a ich spracovania je už detailne prepracovaná, bolo by nanajvýš vhodné detailne spracovať aj možnosti spracovania čiastočne kontextových jazykov. Podobne ako pri syntaktickej analýze využívajúcej zásobníkový automat, aj pri syntaktickej analýze založenej na automate s hlbokým zásobníkom by mohla vznikať určitá syntaktická štruktúra. Túto možnosť, rovnako ako aj predchádzajúce možnosti, je potrebné teoreticky spracovať aj prakticky overiť v budúcnosti.

Aktuálny program by mohol byť rozšírený o možnosť výpisu pravidiel použitých na spracovanie zadaného reťazca, pričom poradie pravidiel by si počas behu ukladal spolu s jednotlivými konfiguráciami a pri úspešnom spracovaní by čísla použitých pravidiel vypísal na výstup. Podobne ako pri klasickej syntaktickej analýze by mohol byť vytváraný derivačný strom, avšak tento strom by taktiež musel byť rozšírený o nejaké vlastnosti, napríklad o stavy.

Vzhľadom na to, že dobrá gramatika je základom pre dobrý zásobníkový automat, bolo by vhodné, aby bol program schopný upozorniť užívateľa, resp. zadávateľa gramatiky, na jej nevhodnosť. Užívateľ by tak mohol zvážiť úpravu gramatiky na rozumnejšiu verziu, alebo pokračovať aj napriek tomu, že by jeho gramatika nemusela viesť k správnemu výsledku analýzy alebo by mohla viesť k zacykleniu automatu.

Literatúra

[1] Grune, D. a Jacobs, C. J. H. *Parsing Techniques: A Practical Guide*. 2. vyd. Springer, 2008. ISBN 978-0-387-20248-8.

[2] Kumar, A., Kalra, N. a Garhwal, S. *Evolving state grammar for modeling DNA and RNA structures* [online]. 2018. Dostupné z: https://www.sciencepubco.com/index.php/ijet/article/view/8627/2976.

[3] Laurent, N. a Mens, K. *Taming Context-Sensitive Languages with Principled Stateful Parsing* [online]. 2016.

[4] Meduna, A. a Lukáš, R. *Formální jazyky a překladače: Studijní opora* [online]. 2006+revize 2009-2015. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIFJ-IT%2Ftexts%2FOporaIFJ.pdf&cid=12745.

[5] Meduna, A. a Soukup, O. *Modern Language Models and Computation*. 1. vyd. Springer, 2017. ISBN 978-3-319-63099-1.