VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PARALELNÍ SYNTAKTICKÁ ANALÝZA

PARALLEL PARSING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MAROŠ HOLKO**
AUTHOR

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

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Holko Maroš**
Program: Informační technologie
Název: **Paralelní syntaktická analýza**
Parallel Parsing
Kategorie: Překladače

Zadání:

1. Podrobně se seznamte s několika metodami syntaktické analýzy. Zaměřte se na metody, které jsou založeny na normálních formách gramatik.
2. Navrhněte paralelní verzi syntaktického analyzátoru, který je založen na těchto metodách. Diskutujte jeho přednosti a nedostatky.
3. Studujte užití navrženého analyzátoru v bodě 2. Navrhněte a implementujte syntaktický analyzátor na jeho bázi. Testujte výsledný analyzátor.
4. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman: Compilers: Principles, Techniques, and Tools. Addison Wesley; 2nd edition (August 31, 2006)

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Meduna Alexander, prof. RNDr., CSc.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 31. října 2018

Abstrakt

Práca sa zaoberá štúdiom niekoľkých metód syntaktickej analýzy, obzvlášť najmä Cocke-Younger-Kasami algoritmu. Ďalej je navrhnutý spôsob paralelizácie tohto algoritmu a jeho implementácia v jazyku C++. Na dosiahnutie paralelizácie boli použité vlákna. S prácou bola vytvorená aj konzolová aplikácia, v ktorej bol implementovaný paralelný CYK algoritmus. Zároveň bolo navrhnuté a implementované rozšírenie, ktoré zisťuje všetky postupnosti použitých pravidiel v prípade, že vstupný reťazec patrí do danej gramatiky. V závere sú diskutované jeho prednosti a nedostatky.

Abstract

The goal of this bachelor thesis is to create and implement a parallel version of a Cocke-Younger-Kasami algorithm, which is used for syntactic analysis. This algorithm works with context-free grammars, so a big part of this work is dedicated to context-free grammars and their transformation to the Chomsky normal form. Output of this thesis is console application in C++ which use threads for parallel processing. There is also an extension for finding all rule successions for given input string designed and implemented. In the end there is a discussion about program's advantages and disadvantages.

Kľúčové slová

syntaktická analýza, prekladač, Cocke-Younger-Kasami, paralelný CYK algoritmus, Chomského normálna forma, CNF, bezkontextová gramatika, rozbor pravidiel, C++

Keywords

syntax analysis, compiler, Cocke-Younger-Kasami, parallel CYK algorithm, Chomsky normal form, CNF, context-free grammar, rule succession, C++

Citácia

HOLKO, Maroš. *Paralelní syntaktická analýza.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Alexander Meduna, CSc.

Paralelní syntaktická analýza

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána prof. RNRr. Alexandra Medunu CSc.. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Maroš Holko
30. apríla 2019

Poďakovanie

Chcel by som poďakovať vedúcemu práce, pánovi prof. RNDr. Alexandrovi Medunovi CSc., za jeho poskytnutú odbornú pomoc pri vypracovávaní práce.

Obsah

Kapitola 1

Úvod

Syntaktická analýza je dôležitou súčasťou prekladačov v teórii informatiky. Podobne ako v prirodzenom jazyku zisťujeme, či je veta napísaná gramaticky správne, aj v informatike zisťujeme, či daný reťazec znakov patrí do určenej gramatiky. Lenže ako sa to dá v informatike využiť? Ako som spomínal, súvisí to s prekladačmi. Prekladač v informatike je program, ktorý načíta zdrojový program napísaný v zdrojovom jazyku a preloží ho na cieľový program napísaný v cieľovom jazyku, pričom zdrojový a cieľový program sú funkčne ekvivalentné. Napríklad z jazyka vyššej úrovne (C++) sa program prekladá na jazyk nižšej úrovne (strojový kód). Je to z toho dôvodu, lebo počítač rozumie práve jazyku nižšej úrovne.

Prekladač sa skladá zo šiestich častí vykonávajúcich nasledujúce činnosti v tomto poradí: lexikálna analýza, syntaktická analýza, sémantická analýza, generovanie vnútorného kódu, optimalizácia a generovanie cieľového kódu. Často je však ťažké niektoré časti od seba odlíšiť, lebo sú spolu priveľmi previazané (napr. syntaktická a sémanticá analýza).

Gramatika v informatike má jasne špecifikovanú podobu. Gramatika predstavuje množinu prepisovacích pravidiel v takom tvare, že ľavá strana pravidla sa prepíše na pravú stranu pravidla. Tak ako existujú rôzne typy jazykov, existujú aj rôzne silné gramatiky popisujúce tieto jazyky. Tieto rôzne triedy gramatík popisuje Chomského hierarchia. Programovacie jazyky sú väčšinou popísané pomocou bezkontextových gramatík a práve tým sa v tejto práci venujem. Každá bezkontextová gramatika sa dá previesť do normálnej formy, ktorá je ekvivalentná s pôvodnou gramatikou. Pozornosť som venoval hlavne Chomského (CNF) a Greibachovej normálnej forme (GNF).

Na syntaktickú analýzu sa často používa Cocke-Younger-Kasami algoritmus (skrátene CYK). Tento algoritmus pracuje s gramatikami v Chomského normálnej forme, na vstupe prijíma reťazec a na výstupe oznamuje, či daný reťazec patrí do jazyka generovaného danou gramatikou. Cieľom tejto práce bolo navrhnúť paralelnú verziu tohto algoritmu a ukázať tak, že algoritmus sa dá urýchliť, a teda sa dá urýchliť aj celá syntaktická analýza.

Paralelný CYK algoritmus bol implementovaný v jazyku C++. Výsledkom implementácie bola konzolová aplikácia, ktorá načítava zo súboru gramatiku v CNF a zo vstupu reťazec terminálov a na výstupe oznámi, či gramatika generuje daný reťazec alebo nie.

Často nás však zaujíma aj to, ktoré pravidlá gramatiky a v akom poradí máme použiť, aby sme dostali z počiatočného symbolu vstupný reťazec. Kvôli tomuto som navrhol a implementoval rozšírenie, ktoré si počas CYK algoritmu ukladá jednotlivé derivácie a ich podčasti v CYK tabuľke. Na konci sa z týchto podčastí zisťujú všetky kombinácie použitia pravidiel, ak reťazec patrí do danej gramatiky.

V závere sú zhodnotené merania vytvorenej aplikácie a návrhy na jej zlepšenie.

Kapitola 2

Základné pojmy

Na úvod si vysvetlíme niektoré základné pojmy, ktorých znalosť je nevyhnutná v ďalších kapitolách.

2.1 Definície základných pojmov

Definície v tejto sekcii boli prevzaté z [6, 5].

Definícia 2.1.1 *Abeceda* Σ je konečná neprázdna množina elementov, ktoré nazývame symboly.

Definícia 2.1.2 *Reťazec,* taktiež nazývaný *slovo*, nad abecedou Σ je konečná postupnosť symbolov z Σ.

Nech Σ je abeceda.

1. ε je reťazec nad abecedou Σ
2. ak x je reťazec nad Σ a $a \in \Sigma$, potom xa je reťazec nad abecedou Σ

Reťazec, ktorý neobsahuje žiadne symboly (prázdny reťazec), sa značí symbolom ε.

Definícia 2.1.3 Nech x je reťazec nad abecedou Σ. *Dĺžka reťazca* x, $|x|$ je definovaná:

1. ak $x = \varepsilon$, potom $|x| = 0$
2. ak $x = a_1 \dots a_n$, potom $|x| = n$ pre $n \geq 1$ a $a_i \in \Sigma$ pre všetky $i = 1, \dots, n$

Nad reťazcami sa dajú robiť rôzne operácie. Medzi najznámejšie operácie patrí konkatenácia, mocnina a reverzácia.

Definícia 2.1.4 Nech x a y sú dva reťazce nad abecedou Σ. *Konkatenácia reťazcov* x a y je reťazec xy. Konkatenácia dvoch reťzacov, kde práve jeden z nich je prázdny, je rovná neprázdnemu reťazcu ($x\varepsilon = \varepsilon x = x$).

Definícia 2.1.5 Nech x je reťazec nad abecedou Σ. Pre $i \geq 0$ je i-tá *mocnina* reťazca x, x^i definovaná:

1. $x^0 = \varepsilon$

2. pre $i \geq 1 : x^i = xx^{i-1}$

Definícia 2.1.6 Nech x je reťazec nad abecedou Σ. *Reverzácia* reťazca x, $reversal(x)$, je definovaná:

1. ak $x = \varepsilon$, potom $reversal(\varepsilon) = \varepsilon$
2. ak $x = a_1 \dots a_n$, potom $reversal(a_1 \dots a_n) = a_n \dots a_1$ pre $n \geq 1$ a $a_i \in \Sigma$ pre všetky $i = 1, \dots n$

Definícia 2.1.7 Nech Σ^* značí množinu všetkých reťazcov nad Σ. Každá podmnožina $L \subseteq \Sigma^*$ je *jazyk* nad Σ. Ďalej Σ^+ značí množinu $\Sigma^* - \{\varepsilon\}$.

Definícia 2.1.8 Jazyk L je *konečný*, ak L obsahuje konečný počet reťazcov, inak je *nekonečný*.

2.2 Časti prekladača

Prekladač je program, ktorý načíta zdrojový program napísaný v zdrojovom jazyku a preloží ho na cieľový program, ktorý je napísaný v cieľovom jazyku. Oba programy sú funkčne ekvivalentné. Uplatnenie nájdeme v situáciach, keď sa prekladá jazyk vyššej úrovne (napr. C/C++) na jazyk nižšej úrovne (napr. strojový kód). Robí sa to preto, lebo počítač rozumie práve jazyku nižšej úrovne. Bez prekladača by sme len ťažko vykonávali programy na počítači. Počas prekladu prekladač najprv skontroluje, či je zdrojový program zapísaný správne a až potom začne prekladať. V opačnom prípade prekladač zahlási chyby a preklad skončí neúspešne.

Preklad sa skladá zo šiestich častí. Týmito časťami sú: lexikálna analýza, syntaktická analýza, sémantická analýza, generovanie vnútorného kódu, optimalizácia a generovanie cieľového kódu. Tieto časti prebiehajú v poradí, v akom sú vymenované. Činnosti týchto častí budú vysvetlené v nasledujúcich podsekciách. [4]

2.2.1 Lexikálna analýza

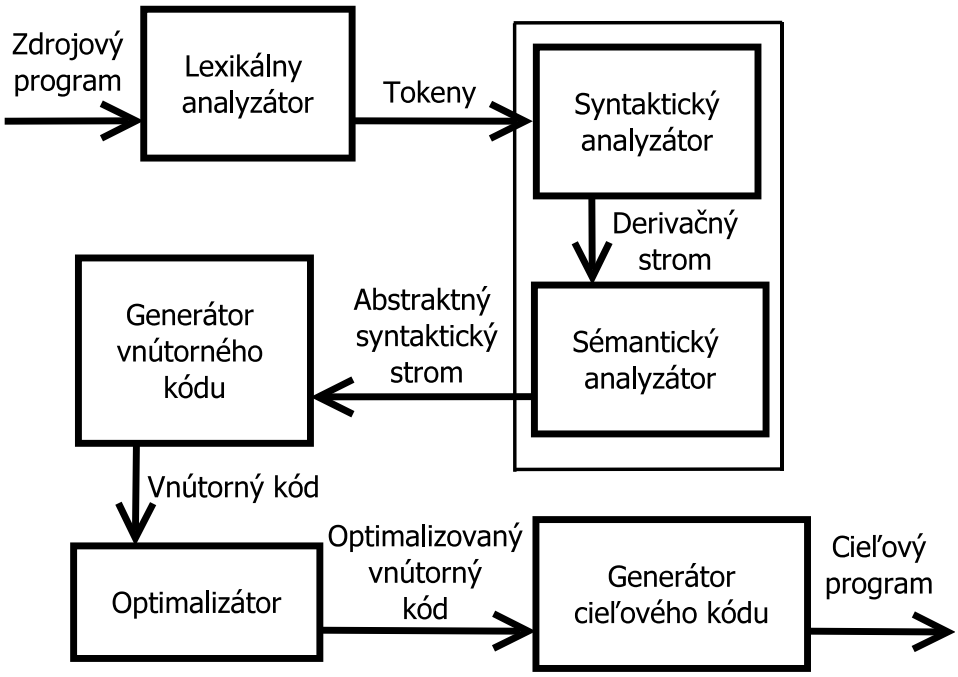
Lexikálnu analýzu vykonáva lexikálny analyzátor, taktiež nazývaný *scanner*. Lexikálny analyzátor rozbije program na jednotlivé lexémy. Lexémy predstavujú logicky oddelené entity, akými sú napríklad identifikátory, rezervované slová, kľúčové slová alebo celé čísla. Lexikálna analýza pomocou lexikálneho analyzátoru rozpoznáva jednotlivé lexémy tak, že číta sekvenciu znakov, ktoré tvoria lexému zo zdrojového programu. Po rozpoznaní lexém lexikálny analyzátor vytvorí lexikálne symboly (v angličtine nazývané *tokeny*), ktoré sú následne posielané syntaktickému analyzátoru. Výsledkom lexikálneho analyzátora je teda reťazec lexikálnych symbolov reprezentujúcich program.

V praxi je ale často lexikálny analyzátor realizovaný ako podprogram syntaktického analyzátora, ktorý ho volá, keď potrebuje ďalši lexikálny symbol pre svoju činnosť[1]. [7, 4]

2.2.2 Syntaktická analýza

Úlohou syntaktickej analýzy, ktorú vykonáva syntaktický analyzátor (angl. *parser*), je určiť syntaktickú štruktúru programu. To znamená, že z lexikálnych symbolov získaných od lexikálneho analyzátoru sa snaží nájsť postupnosť pravidiel gramatiky, pomocou ktorých sa

[1] Z toho pochádza pojem syntaxou riadený preklad.



Obr. 2.1: Schéma prekladača

dá vygenerovať slovo tvorené z lexikálnych symbolov. Túto postupnosť pravidiel je možné zobraziť graficky, a to v podobe stromu, ktorý sa nazýva derivačný strom. Listy v derivačnom strome sú tvorené práve lexikálnymi symbolmi. Podľa smeru konštrukcie derivačného stromu rozlišujeme dva spôsoby konštrukcie. Prvým spôsobom je konštrukcia zhora nadol, ktorá začína od koreňa a postupuje smerom k okrajom stromu. Druhý spôsob je zdola nahor, v ktorom sa postupuje naopak, a to smerom od okraja ku koreňu. Podľa stavby derivačného stromu rozlišujeme aj rovnaké dva typy syntaktických analyzátorov vykonávajúcich syntaktickú analýzu. Získaním derivačného stromu syntaktický analyzátor overí, že program je syntakticky správny a zároveň získa aj jeho syntaktickú štruktúru. [7, 4]

2.2.3 Sémantická analýza

Sémantická analýza kontroluje, či je program zapísaný sématicky správne. To zahŕňa najmä typovú kontrolu, ktorá overuje, či každý operátor v programe má operandy správnych typov. Správne typy sú udávané v špecifikácii jazyka. Ak typ nesedí, prekladač môže v tomto mieste prekladu typ opraviť na správny alebo zahlásiť chybu a preklad tak ukončiť. Sémantická analýza býva často súčasťou syntaktického analyzátora, keďže sú obe časti tesne previazané a niekedy nie je možné ich jednoznačne určiť. [7, 4]

2.2.4 Generovanie vnútorného kódu

Ako už bolo vyššie spomenuté, z výstupu syntaktickej a sémantickej analýzy sme získali derivačný strom, z ktorého chceme vygenerovať vnútorný kód. Kód sa nazýva vnútorný, lebo ešte nereprezentuje strojový kód daného počítača, ale len formu medzikódu medzi zdrojovým a cieľovým kódom. Prečo by sme ale chceli vytvárať nejaký medzikód, keď môžeme rovno generovať kód v cieľovom jazyku? Robíme to preto, lebo vnútorný kód nie je

závislý od žiadnej architektúry počítača a jeho prevod do strojového kódu cieľovej architektúry je jednoduchá úloha, zatiaľ čo predošlé fázy prekladu nie sú. Vnútorný kód môžeme navyše optimalizovať (zefektívniť ho) alebo ho generovať do kódov viacerých architektúr a nemusíme vykonávať aj predchádzajúce časti prekladu pre každú architektúru zvlášť. Takto môžeme ušetriť čas potrebný na preklad do viacerých architektúr.

Na reprezentáciu vnútorného kódu sa často používa trojadresný kód. Trojadresný kód má podobu $[O, X, Y, Z]$, kde O predstavuje operáciu, X je prvý operand, Y je druhý operand a Z je výsledok. Ak je jeden z elementov X, Y, Z vynechaný, ide vtedy o tzv. *nil* komponent. V tomto prípade sa pracuje len s neprázdnymi komponentami. [7, 4]

2.2.5 Optimalizácia

Optimalizácia pracuje s vnútorným kódom, pričom preorganizováva jednotlivé inštrukcie, aby celkový kód bežal efektívnejšie. Tento proces nie je výpočtovo triviálny, preto je v prekladačoch možnosť vypnúť ho. Rozlišujeme dva typy optimalizácie. Jeden prebieha len nad vnútorným kódom, to znamená, že nie je závislý od architektúry konkrétneho počítača. Druhý prebieha nad cieľovým kódom na konkrétnej platforme. [7, 4]

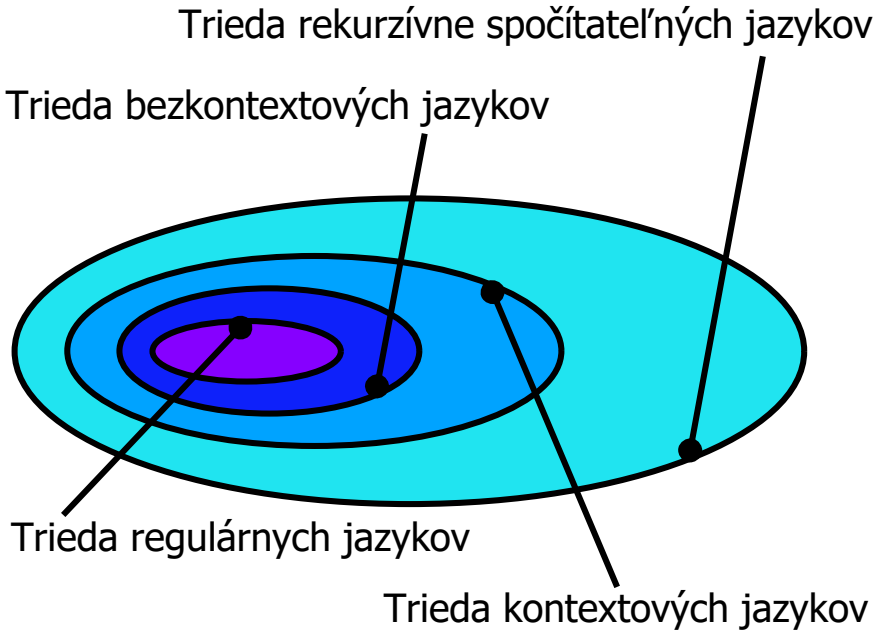
2.2.6 Generovanie cieľového kódu

Z (optimalizovaného) vnútorného kódu sa generujú sekvencie inštrukcií, ktoré sú z inštrukčnej sady cieľového stroja. Pre túto fázu je potrebné poznať informácie o cieľovom stroji, napríklad miesta v pamäti pre premenné programu. [7, 4]

Kapitola 3

Bezkontextový jazyk a jeho transformácie

Podľa Chomského hierarchie jazyky možno rozdeliť do štyroch tried (obr. 3.1), kde jazyky z iných tried majú rôznu vyjadrovaciu silu. V tejto kapitole sa budem venovať bezkontextovým jazykom a ich modelom. Na zápis gramatiky programovacích jazykov sa často využívajú práve bezkontextové jazyky. Medzi základné modely pre bezkontextové jazyky patria bezkontextové gramatiky a zásobníkové automaty. Zdroje, z ktorých som čerpal v tejto kapitole, sú [6, 5, 4, 3].



Obr. 3.1: Chomského hierarchia jazykov

Definícia 3.0.1 *Bezkontextová gramatika* je štvorica $G = (N, T, P, S)$, kde

- N je abeceda neterminálov
- T je abeceda terminálov a platí $N \cap T = \emptyset$

- P je konečná množina pravidiel v tvare $A \to x$, kde $A \in N$, $x \in (N \cup T)^*$
- $S \in N$ je počiatočný neterminál

Jednotlivé pravidlá sa nazývajú prepisovacie z toho dôvodu, že ľavá strana pravidla sa prepisuje na pravú stranu pravidla nezávisle od okolia (odtiaľ názov bezkontextová). Pravidlo tvaru $A \to \varepsilon$ sa nazýva ε-pravidlo, pri ktorom sa neterminál prepisuje za prázdny reťazec. V praxi to funguje tak, že vstupný reťazec je zložený z terminálov a my zisťujeme, či z počiatočného neterminálu sa dá tento vstupný reťazec vygenerovať pomocou pravidiel gramatiky. Použitie pravidla na zmenu symbolu v reťazci sa nazýva *derivačný krok*.

Definícia 3.0.2 Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Nech $u, v \in (N \cup T)^*$ a $p = A \to x \in P$. Potom uAv *priamo derivuje* uxv použitím p v G, zapísané ako $uAv \Rightarrow uxv[p]$ alebo $uAv \Rightarrow uxv$

Definícia 3.0.3 Nech L je jazyk. L je *bezkontextový jazyk*, ak existuje bezkontextová gramatika, ktorá L generuje.

Ak počas aplikovania pravidiel nahrádzame vždy najľavejší neterminál, hovoríme, že ide o *najľavejšiu deriváciu*.

Definícia 3.0.4 Nech $G = (N, T, P, S)$ je bezkontextová gramatika, nech $u \in T^*$, $v \in (N \cup T)^*$, $p = A \to x \in P$ je pravidlo. Potom uAv priamo derivuje uxv pomocou *najľavejšej derivácie* použitím pravidla p v G, zapísanej ako: $uAv \Rightarrow_{lm} uxv[p]$.

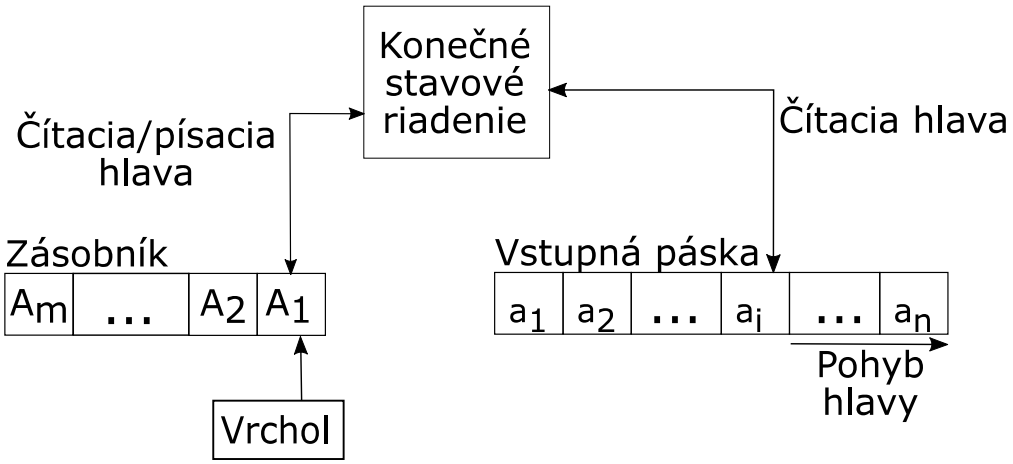
Podobne by sme mohli definovať aj najpravejšiu deriváciu. Druhým modelom pre bezkontextové jazyky je zásobníkový automat. Každá bezkontextová gramatika sa dá previesť na ekvivalentný zásobníkový automat a naopak.

Definícia 3.0.5 *Zásobníkový automat* je sedmica $M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde

- Q je konečná množina stavov
- Σ je vstupná abeceda
- Γ je zásobníková abeceda
- R je konečná množina pravidiel v tvare $Apa \to wq$, kde $A \in \Gamma, p, q \in Q, a \in \Sigma \cup \{\varepsilon\}, w \in \Gamma^*$
- $s \in Q$ je počiatočný stav
- $S \in \Gamma$ je počiatočný symbol na zásobníku
- $F \subseteq Q$ je množina koncových stavov

3.1 Normálne formy gramatík

Ako už vieme z predošlých sekcií, bezkontextová gramatika obsahuje na ľavej strane vždy jeden neterminál a na pravej strane môže obsahovať ľubovoľný počet terminálov a neterminálov. Lepšie by sa nám s gramatikou pracovalo, ak by existovali nejaké ustálené formy gramatík, kde by sme mali istotu, že gramatika bude vyzerať tak, ako očakávame. Preto sa v tejto sekcii budeme venovať dvom najznámejším normálnym formám bezkontextových gramatík, a to Chomského normálnej forme a Greibachovej normálnej forme. Ukážeme si aj postup, ako previesť bezkontextovú gramatiku do jednej z týchto foriem.



Obr. 3.2: Schéma zásobníkového automatu

3.1.1 Chomského normálna forma

Gramatika v Chomského normálnej forme má na pravej strane dva neterminály alebo jeden terminál. Táto forma je jednou z najčastejšie používaných foriem a je pomenovaná po svojom autorovi Noamovi Chomskom.

Definícia 3.1.1 Nech $G = (N, T, P, S,)$ je bezkontextová gramatika. G je v *Chomského normálnej forme*, ak každé pravidlo z P je v jednom z tvarov:

- $A \to BC$, kde $A, B, C \in N$
- $A \to a$, kde $A \in N, a \in T$

[6]

3.1.2 Greibachovej normálna forma

Greibachovej normálna forma je pomenovaná po Sheile Adele Greibachovej. V tejto forme sa musí na pravej strane nachádzať jeden terminál nasledovaný rôznym množstvom neterminálov. Môže nastať prípad, že za terminálom na pravej strane sa už nebude nachádzať žiadny neterminál. V tomto prípade ide tiež o validný zápis pravidla v Greibachovej normálnej forme.

Definícia 3.1.2 Nech $G = (N, T, P, S,)$ je bezkontextová gramatika. G je v *Greibachovej normálnej forme*, ak každé pravidlo z P má tvar:

- $A \to ax$, kde $A \in N, a \in T, x \in N^*$

[6]

3.2 Prevody bezkontextovej gramatiky

Gramatika často obsahuje niektoré pravidlá, ktoré sú nepotrebné pre špecifikáciu nejakého jazyka. Kvôli tomu je potom syntaktická analýza zbytočne zložitá a jej spracovávanie môže byť aj nejednoznačné. V nasledujúcom texte si preto ukážeme, ako môžeme gramatiku urobiť čistejšou a ľahšou pre analyzovanie.

3.2.1 Odstránenie nepotrebných symbolov

Gramatika môže obsahovať symboly, ktoré sú nepotrebné, lebo sa nedajú použiť pri generovaní reťazcov jazyka. Preto ich chceme odstrániť a pracovať len s potrebnými (ukončujúcimi) symbolmi.

Definícia 3.2.1 Nech $G = (N, T, P, S)$ je bezkontextová gramatika a $A \in N \cup T$. A je *ukončujúci symbol*, ak existuje $w \in T^*$ také, že $A \Rightarrow^* w$ v G, inak A je *neukončujúci symbol*.

Na odstránenie neukončujúcich symbolov z gramatiky G potrebujeme zostrojiť množinu V a vložiť do nej všetky terminály gramatiky G. Ďalej pre každé pravidlo r gramatiky G zistíme, či sa z neho dá vygenerovať symbol z množiny V. Ak sa dá, vložíme príslušný neterminál do množiny V. Toto opakujeme dovtedy, kým sa dajú pridávať neterminály do množiny V. V pseudokóde je algoritmus zobrazený v alg.1. Na záver už len odstránime z gramatiky G pravidlá, ktoré obsahujú symboly, ktoré nie sú vo V.

Algoritmus 1: Algoritmus odstránenia neužitočných symbolov

```
1  Vstup: gramatika G = (N, T, P, S)
2  Výstup: množina V obsahujúca všetky ukončujúce symboly z G
3  begin
4      V := T
5      repeat
6          if rhs(r) ∈ V* pre nejaké r ∈ P then
7              pridaj lhs(r) do V
8          end if
9      until žiadna zmena;
10 end
```

Najlepšie si to zobrazíme na príklade. Máme gramatiku, kde $a, i, o, (,)$ sú terminály a ostatné znaky neterminály.

$$S \to SoS \qquad S \to (S)$$
$$S \to SoA \qquad S \to i$$
$$S \to A \qquad B \to i$$
$$A \to AoA$$

Množina V bude na začiatku inicializovaná symbolmi: $o, i, (,)$. Potom začneme cyklicky prechádzať jednotlivé pravidlá a pridáme do množiny V symboly B a S, lebo z nich sa dá priamo vygenerovať symbol z množiny V. Pri ďalšej iterácii zistíme, že nič ďalšie sa už v množine V neobjaví. Symboly, ktoré nie sú v množine V, sú neukončujúce, v našom príklade je to symbol A.[4, 3]

3.2.2 Odstránenie nedosiahnuteľných symbolov

Ďalším problémom, ktorý môže nastať, je, že gramatika môže obsahovať nedosiahnuteľné symboly. To sú symboly, ktoré sa nevyskytujú v žiadnom reťazci odvodenom z danej gramatiky.

Definícia 3.2.2 Nech $G = (N, T, P, S)$ je bezkontextová gramatika a $X \in N \cup T$. Symbol X je *dosiahnuteľný*, ak $S \Rightarrow^* uXv$ v G pre $u, v \in (N \cup T)^*$, inak je *nedosiahnuteľný*.

Na zistenie, ktoré symboly sú v gramatike G dosiahnuteľné, si zostrojíme množinu W a vložíme do nej počiatočný symbol. Ďalej pre každé pravidlo, ktorého ľavá strana sa nachádza v množine W, vložíme do W pravú stranu daného pravidla. Toto opakujeme, kým sa výsledná množina W dosiahnuteľných symbolov mení. Rovnako ako pri odstraňovaní nepotrebných symbolov, aj tu odstránime pravidlá z gramatiky G, ktoré nie sú z množiny W.

Algoritmus 2: Algoritmus zistenia dosiahnuteľných symbolov

```
1  Vstup: gramatika G = (N, T, P, S)
2  Výstup: množina W obsahujúca všetky dosiahnuteľné symboly z G
3  begin
4      W := {S}
5      repeat
6          if lhs(r) ∈ W pre nejaké r ∈ P then
7              pridaj rhs(r) do W
8          end if
9      until žiadna zmena;
10 end
```

Znovu si to ukážeme na príklade. Máme gramatiku rovnakú ako v predošlom príklade. Množinu W inicializujeme počiatočným symbolom S. Cyklicky prejdeme všetky pravidlá a do W pridáme $o, A, (,), i$. Po druhej iterácii algoritmus skončí, lebo do W už nič nepribudlo. Jediný nedosiahnuteľný symbol je tak B, ktorý sa v množine W nenachádza.

Ak vykonáme oba predchádzajúce algoritmy, dostaneme gramatiku, ktorá obsahuje už len užitočné symboly. V našom príklade by po vykonaní oboch algoritmov zostali len tieto pravidlá: $S \to SoS$, $S \to (S)$, $S \to i$. [4, 3]

3.2.3 Odstránenie ε-pravidiel

Bezkontextové gramatiky často obsahujú aj ε-pravidlá, ktoré generujú prázdny reťazec. Pravidlá s ε na pravej strane nám slúžia na skracovávanie vygenerovaných vetných foriem. V syntaktickej analýze to nie je veľmi žiaduce, lebo často nám vzniká problém nejednoznačnosti v tom, ktoré pravidlo použiť. Odstránením týchto pravidiel by sa tak celý proces syntaktickej analýzy podstatne zjednodušil. Na lepšie pochopenie si najskôr potrebujeme vysvetliť pojem ε-neterminál. ε-neterminál je neterminál, z ktorého sa dá odvodiť prázdny reťazec.

Definícia 3.2.3 Nech $G = (N, T, P, S)$ je bezkontextová gramatika a $A \in N$. Symbol A je *ε-neterminál*, ak $A \Rightarrow^* \varepsilon$ v G.

Na zistenie ε-neterminálov si zostrojíme množinu E, do ktorej na začiatku vložíme ľavé strany pravidiel, ktoré na pravej strane majú ε. Následne pri iterovaní cez všetky pravidlá zisťujeme, či sa pravá strana pravidla nachádza v množine E. Ak sa tam nachádza, ľavú stranu toho pravidla vložíme tiež do E. Algoritmus sa skončí, keď sa už žiadny symbol nedá pridať do E.

Algoritmus 3: Algoritmus zistenia ε-neterminálov

Vstup: gramatika $G = (N, T, P, S)$
Výstup: množina E obsahujúca všetky ε-neterminály z G
begin
    $E := lhs(p) | p \in P, rhs(p) = \varepsilon$
    repeat
        if ***rhs(r)*** $\in E^*$ pre nejaké $r \in P$ **then**
            pridaj **lhs(r)** do E
        end if
    until žiadna zmena;
end

Teraz, keď už vieme, čo sú a ako nájsť ε-neterminály, môžeme si vysvetliť, ako transformovať gramatiku na gramatiku bez ε-pravidiel (alg. 4). Takto prekonvertovaná gramatika generuje všetky reťazce pôvodnej gramatiky okrem prázdneho reťazca, ktorý sa už bez ε-pravidiel nedá vygenerovať.

Algoritmus 4: Algoritmus odstránenia ε-pravidiel

Vstup: gramatika $G = (N, T, P, S)$
Výstup: gramatika $H = (N, T, P, S)$ taká, že L(H)=L(G)-{ε} a ${}_HP$ neobsahuje žiadne ε-pravidlá
begin
    ${}_HN =_G N$, ${}_HT =_G T$, ${}_HP = r | r \in_G P$; r nie je ε-pravidlo
    Určiť množinu $E \subseteq_G N$ obsahujúcu všetky ε-neterminály z G
    repeat
        if pre nejaké $r \in P$, ***rhs(r)*** $= x_0 A_1 x_1 \dots A_n x_n$ kde $A_i \in E, x_j \in ({}_G\Sigma - E)^*$, $X_i \in \{\varepsilon, A_i\}$, $1 \leq i \leq n$, $0 \leq j \leq n$, a $|x_0 x_1 x_2 \dots x_n| \geq 1$ **then**
            pridaj **lhs(r)** $\rightarrow x_0 X_1 x_1 \dots X_n x_n$ do ${}_HP$
        end if
    until žiadna zmena;
end

Aplikovanie algoritmu 4 si ukážeme na príklade. Máme gramatiku:

1. $S \rightarrow ASA$
2. $S \rightarrow aB$
3. $B \rightarrow b$
4. $B \rightarrow \varepsilon$
5. $A \rightarrow \varepsilon$

Na začiatok si zostrojíme množinu ε-neterminálov pomocou algoritmu 3. V tejto množine sa budú nachádzať symboly A, B. Do novej gramatiky bez ε-pravidiel vložíme všetky pravidlá zo starej gramatiky okrem ε-pravidiel. V našom príklade to budú prvé tri pravidlá. Následne pre každé pravidlo z novej gramatiky zistíme, či sa na pravej strane nachádza neterminál z množiny E. Ak sa tam taký neterminál nachádza, nahradíme ho prázdnym reťazcom (vymažeme ho). Toto vykonáme pre všetky kombinácie vyhovujúcich neterminálov na pravej strane pravidla.

Vezmime si prvé pravidlo $S \to ASA$. Na pravej strane sa dvakrát nachádza ε-neterminál A. Po nahradení prázdnym reťazcom nám vzniknú tri nové pravidlá: $S \to AS$, $S \to SA$, $S \to S$. V pravidle $S \to aB$ sa nachádza ε-neterminál B. Nové pravidlo, ktoré z neho vznikne, je $S \to a$. Z ďalších pravidiel nové pravidlá už nevzniknú, takže vo výslednej gramatike budú tieto pravidlá:

1. $S \to ASA$
2. $S \to AS$
3. $S \to SA$
4. $S \to S$
5. $S \to aB$
6. $S \to a$
7. $B \to b$

[3, 4, 5]

3.2.4 Odstránenie jednotkových pravidiel

Gramatiky môžu obsahovať pravidlá, ktoré len prepisujú jeden neterminál na iný neterminál. Príkladom môže byť pravidlo $S \to A$. Takéto pravidlá sa nazývajú *jednotkové pravidlá*. Tieto pravidlá len zbytočne zväčšujú gramatiku a komplikujú syntax jazyka, preto je žiaduce sa takýchto pravidiel zbaviť. Algoritmus prevodu gramatiky na gramatiku bez jednotkových pravidiel je zobrazený v alg. 5.

Definícia 3.2.4 Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Pravidlo $p \in P$ je *jednotkové pravidlo*, ak $rhs(p) \in N$.

Príklad: Máme gramatiku:

1. $S \to AB$
2. $S \to A$
3. $A \to aAb$
4. $A \to ab$

Jednotkové pravidlo je $S \Rightarrow A$. Pomocou neho sa dá odvodiť nové pravidlo $S \to aAb$, lebo $S \Rightarrow A \Rightarrow aAb$. Ďalšie pravidlo, ktoré sa dá odvodiť, je $S \to ab$, lebo $S \Rightarrow A \Rightarrow ab$.

Nová gramatika bez jednotkových pravidiel bude vyzerať takto:

1. $S \to AB$
2. $S \to aAb$
3. $S \to ab$
4. $A \to aAb$
5. $A \to ab$

[3, 4, 5]

Algoritmus 5: Algoritmus odstránenia jednotkových pravidiel

Vstup: gramatika $G = (N, T, P, S)$
Výstup: gramatika $H = (N, T, P, S)$ taká, že L(H)=L(G)-$\{\varepsilon\}$ a ${}_HP$ neobsahuje žiadne jednotkové pravidlá
begin
    ${}_HN =_G N$, ${}_HT =_G T$, ${}_HP = \emptyset$
    repeat
        if $A \Rightarrow^n B[r_1, r_2 \dots r_n] \Rightarrow x\ v\ G$, kde $A, B \in_G N, x \in ({}_GN \cup_G T)^* -_G N, 1 \leq n \leq \text{card}({}_GP)$ a každé r_j je jednotkové pravidlo, $1 \leq j \leq n$ **then**
            pridaj $A \to x$ do ${}_HP$
        end if
    until žiadna zmena;
end

3.2.5 Gramatika v správnom tvare

Po aplikovaní všetkých predošlých transformácií dostaneme gramatiku, ktorá je v tzv. *správnom tvare*. S takouto gramatikou sa ľahšie pracuje pri syntaktickej analýze, no nie je nevyhnutné používať len gramatiku v takomto tvare. Niektoré analyzátory používajú gramatiky obsahujúce práve ε-pravidlá.

Definícia 3.2.5 Nech $G = (N, T, P, S)$ je bezkontextová gramatika v *správnom tvare*, ak spĺňa tieto vlastnosti:

- $N \cup T$ obsahuje len užitočné symboly
- G neobsahuje ε-pravidlá
- G neobsahuje jednotkové pravidlá

[3, 4, 5]

3.3 Prevod bezkontextovej gramatiky do Chomského normálnej formy

V predošlej sekcii sme si vysvetlili, ako z bezkontextovej gramatiky spraviť gramatiku v správnom tvare, teraz si ukážeme algoritmus, ktorý konvertuje bezkontextovú gramatiku v správnom tvare na ekvivalentnú bezkontextovú gramatiku v Chomského normálnej forme. Algoritmus je zobrazený v alg. 6.

Algoritmus 6: Algoritmus transformácie bezkontextovej gramatiky do Chomského normálnej formy

Vstup: gramatika v správnom tvare $G = ({}_G\Sigma, {}_G R)$
Výstup: gramatika $H = ({}_H\Sigma, {}_H R)$ v Chomského normálnej forme taká, že $L(G) = L(H)$
begin
  ${}_H\Sigma = {}_G\Sigma$ a ${}_H R = \{r | r \in {}_G R, \mathbf{rhs}(p) \in {}_G\Delta \cup {}_G N^2\}$
  Vytvor novú množinu neterminálov O takú, že $O \cap {}_G\Sigma = \emptyset$ a $car(O) = card({}_G\Delta)$
  Vytvor bijektívne zobrazenie β z ${}_G\Sigma$ do $O \cap {}_G N$
  Vlož $\{\beta(a) \to a | a \in {}_G\Delta\} \cup \{A \to \beta(X_1)\beta(X_2) | A \to X_1X_2 \in {}_G R\}$ do ${}_H R$
  repeat
    if pre nejaké $n \geq 3, A \to X_1X_2X_3 \ldots X_{n-1}X_n \in {}_G R$ **then**
      pridaj $A \to \beta(X_1)\langle X_2 \ldots X_n\rangle, \langle X_2 \ldots X_n\rangle \to \beta(X_2)\langle X_3 \ldots X_n\rangle, \ldots, \langle X_{n-2}X_{n-1}X_n\rangle \to \beta(X_{n-2})\langle X_{n-1}X_n\rangle, \langle X_{n-1}X_n\rangle \to \beta(X_{n-1})(X_n)$ do ${}_H R$
    end if
  until žiadna zmena;
end

Algoritmus môže svojou činnosťou pridať do gramatiky nové neterminály, z ktorých niektoré môžu byť zbytočné. Na odstránenie týchto symbolov stačí znovu použiť algoritmus 1. [4]

Kapitola 4

Metódy syntaktickej analýzy

V tejto kapitole si predstavíme niektoré používané metódy syntaktickej analýzy. Ako už vieme zo skorších kapitol, rozlišujeme syntaktickú analýzu zhora nadol a zdola nahor. Počas tejto činnosti sa zostavuje syntaktický strom, ktorý, ak sa podarí zostaviť, značí, že program je syntakticky správny.

4.1 Metódy pracujúce zhora nadol

Jednou z metód pracujúcou zhora nadol sú LL-analyzátory založené na LL gramatikách. Prvé L v názve znamená, že syntaktický analyzátor číta jednotlivé tokeny zľava doprava a druhé L je v názve preto, lebo vytvára tak najľavejšie derivácie z počiatočného symbolu. Na vysvetlenie LL gramatiky si ale najprv potrebujeme vysvetliť pojmy *množina First, Empty, Follow* a *Predict*.

Definícia 4.1.1 Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Pre každé $x \in (N \cup T)^*$ je definovaná $First(x)$ ako: $First(x) = \{a : a \in T, x \Rightarrow^* ay; y \in (N \cup T)^*\}$.

Inými slovami, množina *First* vyjadruje množinu, ktorá obsahuje všetky terminály, ktorými sa môže začínať reťazec derivovateľný z nejakého reťazca x.

Definícia 4.1.2 Nech $G = (N, T, P, S)$ je bezkontextová gramatika.

$Empty(x) = \{\varepsilon\}$, ak $x \Rightarrow^* \varepsilon$; inak

$Empty(x) = \emptyset$, kde $x \in (N \cup T)^*$.

Definícia 4.1.3 Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Pre všetky $A \in N$ definujeme množinu $Follow(A)$: $Follow(A) = \{a : a \in T, S \Rightarrow^* xAay, x, y \in (N \cup T)^*\} \cup \{\$: S \Rightarrow^* xA, x \in (N \cup T)^*\}$

Definícia 4.1.4 Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Pre každé $A \to x \in P$ definujeme množinu $Predict(A \to x)$ ako:

- ak $Empty(x) = \{\varepsilon\}$, potom $Predict(A \to x) = First(x) \cup Follow(A)$
- inak ak $Empty(x) = \emptyset$, potom $Predict(A \to x) = First(x)$

V množine *Predict* pre dané pravidlo sa nachádzajú všetky terminály, ktoré môžu byť aktuálne najľavejšie vygenerované, ak pre ľubovoľnú vetnú formu použijeme dané pravidlo.

Definícia 4.1.5 Nech $G = (N, T, P, S)$ je bezkontextová gramatika. G je $LL - gramatika$, ak pre každé $a \in T$ a každé $A \in N$ existuje maximálne jedno A-pravidlo v tvare $A \to X_1 X_2 \dots X_n \in P$ a platí: $a \in Predict(A \to X_1 X_2 \dots X_n)$.

Treba podotknúť, že trieda jazykov generovaných LL-gramatikami je podmnožinou triedy jazykov generovaných bezkontextovými gramatikami. To znamená, že len niektoré bezkontextové gramatiky je možné previesť na ekvivalentnú LL-gramatiku. Tento prevod je možné uskutočniť pomocou transformácií faktorizácia a odstránenie ľavej rekurzie. Tieto prevody si tu už nebudeme vysvetľovať. [6, 4]

4.1.1 Rekurzívne zostupný LL syntaktický analyzátor

Tento syntaktický analyzátor číta vstup zľava doprava. Ďalej pre každé pravidlo je zostrojená množina *Predict* a každý neterminál je reprezentovaný vlastnou procedúrou. Množina *Predict* určuje, ktoré pravidlo má byť použité pre aktuálny symbol na vstupe. Pre zostrojenie syntaktického stromu táto metóda rekurzívne volá jednotlivé procedúry pre vstupné symboly a podľa toho, čo je na zásobníku a na vstupe, vracia hodnotu. Zanorovanie procedúr býva riešené implicitne na úrovni programovacieho jazyka. [4, 6]

4.1.2 Prediktívna syntaktická analýza riadená tabuľkou

Hlavná nevýhoda pre predošlý analyzátor je tá, že pre zmenu gramatiky, musíme upraviť aj príslušné procedúry pre neterminály. Preto si teraz predstavíme prediktívnu syntaktickú analýzu, ktorá je riadená tabuľkou. V tejto metóde pre zmenu gramatiky bude len nutné upraviť údaje v tabuľke. Zanorovanie v tejto metóde už nemôže byť riešené implicitne ako v predošlej metóde, ale explicitne na vlastnom zásobníku. [4, 6, 5]

4.2 Metódy pracujúce zdola nahor

Metódy zdola hore čítajú vstup vo forme tokenov zľava doprava, ale syntaktický strom tvoria od listov smerom ku koreňu.

4.2.1 Precedenčná syntaktická analýza

Precendenčná syntaktická analýza sa zvyčajne používa na vyhodnocovanie logických a aritmetických výrazov. Celý proces je riadený operátormi a ich prioritou v jednotlivých výrazoch. Využíva sa tu precedenčná tabuľka, ktorej riadky a stĺpce sú označené terminálmi a symbolom označujúcim koniec vstupu. V jednotlivých bunkách tabuľky je vždy pre dvojicu operátorov vyjadrené, ktorý operátor je nadradený a podľa toho sa vykonáva syntaktická analýza. [4, 6, 5]

4.2.2 LR syntaktický analyzátor

Pri analýze zdola nahor sa pri konštrukcii derivačného stromu hľadajú najpravejšie derivácie, od toho pochádza písmeno R v názve (prvé L znamená to isté ako pri LL analyzátore). LR syntaktické analyzátory sú postavené na LR tabuľkách. Ich výhodou je, že sú rýchle a jednoducho zvládajú zotavenie z chýb, lebo nikdy nevkladajú chybný symbol na zásobník. [4, 6, 5]

Kapitola 5

Cocke-Younger-Kasami algoritmus

V tejto kapitole si vysvetlíme metódu syntaktockej analýzy, ktorá nepracuje so zásobníkovým automatom ako väčšina metód ale využíva k svojej činnosti tabuľku, ktorú si zostavuje a vypĺňa počas behu.

5.1 Princíp sekvenčnej verzie algoritmu

Cocke-Younger-Kasami algoritmus (skrátene CYK alebo CKY) slúži na syntaktickú analýzu bezkontextových gramatík. Algoritmus pracuje metódou zdola hore, pričom konštruuje tabuľku, pomocou ktorej zisťuje, či vstupný reťazec patrí do danej gramatiky alebo nie [4, 2].

Na vstupe dostáva gramatiku v Chomského normálnej forme a vstupný reťazec terminálov. Najprv sa skonštruuje tabuľka o veľkosti n riadkov, kde n je dĺžka vstupného reťazca. Počet stĺpcov v prvom riadku je n a každý ďalší riadok má o jeden stĺpec menej než predchádzajúci riadok. Posledný riadok má preto len jeden stĺpec. Všetky bunky tabuľky sú inicializované na prázdnu hodnotu (znak $ v tabuľkách).

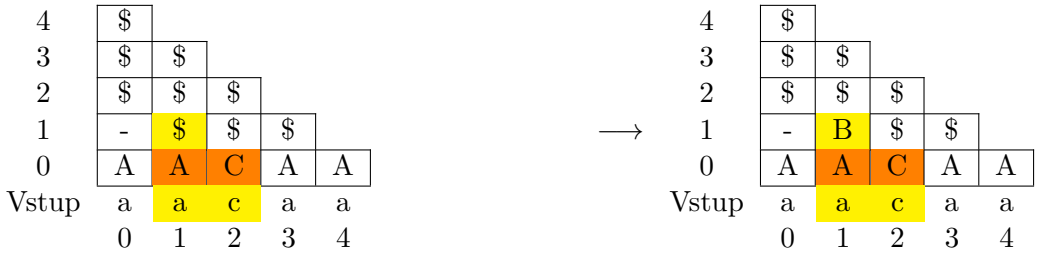
Príklad: Máme gramatiku, kde $S, A, B, C \in N$ a $a, c \in T$. Počiatočný neterminál je S.

1. $S \to AB$
2. $A \to BC|a$
3. $C \to AB|c$
4. $B \to AC|CA$

Ďalej máme vstupný reťazec *aacaa*. Tabuľka po inicializácii je zobrazená v tab. 5.1.

4	$				
3	$	$			
2	$	$	$		
1	$	$	$	$	
0	$	$	$	$	$
Vstup	a	a	c	a	a
	0	1	2	3	4

Tabuľka 5.1: Inicializovaná CYK tabuľka



Tabuľka 5.2: CYK tabuľka pri výpočte bunky [1,1]

Algoritmus následne pre každý terminál vo vstupe hľadá v gramatike pravidlo s príslušným terminálom na pravej strane. V prípade nájdenia sa uloží ľavá strana všetkých pravidiel, ktoré vyhovujú, do prvého riadka tabuľky (nultý riadok) do stĺpca, ktorého pozíciu určuje poradové číslo terminálu vo vstupe. Ak žiadne pravidlo nevyhovuje, uloží sa do bunky prázdna hodnota -, ktorá znamená, že bunka už bola spracovaná.

Ďalej algoritmus pracuje už len s neterminálovými pravidlami. V bunkách druhého riadka (index 1 v tabuľke) sa skúmajú vždy dvojice susediacich terminálov vo vstupe s pravými stranami pravidiel. Keďže už nepracujeme s terminálovými pravidlami, potrebujeme zistiť, z ktorých neterminálov sa dajú skúmané terminály vygenerovať. Na ich zistenie využijeme predošlý riadok, v ktorom sme zistili, akými neterminálmi sa dajú nahradiť jednotlivé terminály. Teraz môžeme tieto neterminály skonkatenovať a porovnať s neterminálmi na pravých stranách v pravidlách. V prípade zhody zistíme, že dané neterminály sa dajú nahradiť iným neterminálom a tento nový neterminál sa uloží do aktuálne spracovávanej bunky. Celý algoritmus pracuje tak, že veľký problém rozbije na menšie podproblémy, ktoré už vyriešil v predošlých krokoch.

Príklad: Pri výpočte bunky [1,1] (tab. 5.2) potrebujeme zistiť, ktorým neterminálom sa dá nahradiť terminál a a c. Vidíme, že terminál a na pozícii 1 vo vstupe sa dá nahradiť neterminálom A v bunke [0,1]. Terminál c (pozícia 2) sa nahradí neterminálom C (bunka [0,2]). Výsledok po konkatenácii AC sa nachádza vo štvrtom pravidle, takže môžeme túto dvojicu nahradiť neterminálom B.

V každom ďalšom riadku sa skúmaná n-tica susediacich terminálov zväčší o 1 voči predchádzajúcemu riadku.

Príklad: Výpočet bunky [2,0]. Skúmaná n-tica je aac. Algoritmus rozdelí reťazec na menšie podčasti všetkými rôznymi spôsobmi (v tomto prípade len dva) a/ac, aa/c. Pre každú dvojicu podčastí hľadá v tabuľke prislúchajúce neterminály, z ktorých sa dá daná podčasť vygenerovať (tab. 5.3). Terminál a sa dá vygenerovať z neterminálu A (bunka [0,0]) a dvojica ac z B (bunka [1,1]). Nájdené neterminály A a B znova skonkatenuje a porovná s pravými stranami pravidiel [1]. Toto sa vykoná pre všetky podčasti. Postup pre ďalšie riadky je analogický s predošlým, len s tým rozdielom, že s každým ďalším riadkom pribúda viac podčastí, čiže viac kombinácií na porovnanie.

Ak by sa neterminál skonkatenoval s prázdnou hodnotou -, znamená to, že pre túto dvojicu pravidlo neexistuje.

Príklad je uvedený v tab. 5.4, kde podčasti pre výpočet bunky [2,2] $C-$ (bunky [0,2] a [1,3]) a BA (bunky [1,2] a [0,4]) nie sú súčasťou žiadneho pravidla.

[1] Ak by niektorá bunka obsahovala viac neterminálov, tak sa všetky skonkatenované dvojice porovnajú s pravými stranami pravidiel.

4	$				
3	$	$			
2	$	$	$		
1	-	B	B	-	
0	A	A	C	A	A
Vstup	a	a	c	a	a
	0	1	2	3	4

4	$				
3	$	$			
2	$	$	$		
1	-	B	B	-	
0	A	A	C	A	A
Vstup	a	a	c	a	a
	0	1	2	3	4

Tabuľka 5.3: Podčasti CYK tabuľky pri výpočte bunky [2,0]

4	$				
3	$	$			
2	S,C	S,C	$		
1	-	B	B	-	
0	A	A	C	A	A
Vstup	a	a	c	a	a
	0	1	2	3	4

$\longrightarrow$

4	$				
3	$	$			
2	S,C	S,C	-		
1	-	B	B	-	
0	A	A	C	A	A
Vstup	a	a	c	a	a
	0	1	2	3	4

Tabuľka 5.4: Výpočet bunky [2,2] obsahujúcej prázdnu hodnotu -

Po vyplnení tabuľky (tab. 5.5), algoritmus zistí prítomnosť počiatočného neterminálu v bunke posledného riadka. Ak sa v ňom počiatočný neterminál nachádza (v našom prípade S), skúmaný reťazec patrí do danej gramatiky.

4	S,C				
3	B	B			
2	S,C	S,C	-		
1	-	B	B	-	
0	A	A	C	A	A
Vstup	a	a	c	a	a
	0	1	2	3	4

Tabuľka 5.5: Výsledná CYK tabuľka

Časová náročnosť algoritmu je v najhoršom prípade $O(n^3 \cdot |G|)$, kde n je dĺžka vstupného reťazca a G je veľkosť gramatiky. Priestorová zložitosť je n^3 (trojdimenzionálna tabuľka). Celý algoritmus by sa dal zhrnúť do pseudokódu inšpirovaného jazykom C uvedeného v alg. 7 [8].

Algoritmus 7: CYK PSEUDOKÓD

```
1  i = 0;    j = 0;
2  n =length of input string;
   // Spracovanie prvého riadka tabuľky
3  while (i < n) do
4      if (A → a_i ∈ R) then
5          add A to CYK[j,i];
6      else
7          add - to CYK[j,i];
8      end if
9      i++;
10 end while

   // Iterácia cez všetky zvyšné riadky
11 for (j = 1; j < n; j++) do
       // Iterácia cez všetky stĺpce v danom riadku
12     for (i = 0; i < n − j; i++) do
           // Iterácia cez všetky podčasti vstupného reťazca terminálov
13         for (k = 0; k < j; k++) do
14             if (B ∈ CYK[k, i] and C ∈ CYK[j − k − 1, i + k + 1]) then
15                 add A to CYK[j,i] for some A, B, C ∈ _G N;
16             else
17                 add - to CYK[j,i];
18             end if
19         end for
20     end for
21 end for

   // Zistenie či vstupný reťazec patrí do gramatiky
22 if (Starting symbol ∈ CYK[n, 0]) then
23     input string belong to grammar
24 else
25     input string does not belong to grammar
26 end if
```

5.2 Návrh paralelnej verzie algoritmu

Sekvenčný CYK algoritmus zapisuje v danom momente vždy do jednej bunky v jednom riadku nezávisle od ostatných buniek v danom riadku. Jednotlivé podčasti potrebné pre vyhodnotenie aktuálnej bunky sú uložené vždy v predchádzajúcich riadkoch. Z týchto predpokladov som vychádzal pri návrhu paralelnej verzie algoritmu.

Na začiatku sa podľa dĺžky vstupného reťazca určí maximálny počet súčasne bežiacich vlákien podľa vzorca $v = \frac{n}{2} + 1$, kde n je dĺžka vstupu a v počet vlákien. Pri vyhodnocovaní prvého riadka sa pre každú bunku vytvorí nové vlákno, ktorému sa predá ako parameter index stĺpca práve vyhodnocovanej bunky (index stĺpca je zhodný s pozíciou terminálu vo vstupnom reťazci). Pre vyhodnotenie prvého riadka bude potrebné spustiť dve sady s v vláknami. V prvej sade sa vytvorí práve v vlákien a v druhej $n - v$ vlákien. V tab. 5.6 sú uvedené sady vlákien pre prvý riadok pre vstupný reťazec dĺžky 9. Pre bunky v ďalších riadkoch sa vláknu predajú ako parametre indexy bunky tabuľky. Vlákno následne vyhodnotí podčasti skúmaného podreťazca a zapíše výsledok do tabuľky.

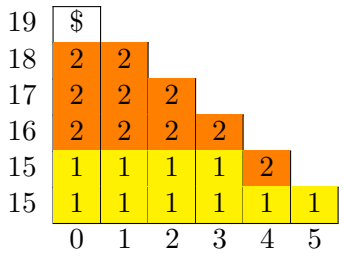
Urýchlenie spočíva v tom, že iterovanie cez všetky podčasti danej bunky môže prebiehať paralelne s iteráciami cez podčasti iných buniek tabuľky. Časová náročnosť algoritmu by teoreticky bola v tomto prípade o niečo väčšia než $O(n^2 \cdot |G|)$. Jediným problémom je zaistenie synchronizácie pre čítanie a zapisovanie vlákien rôznych riadkov do tej istej bunky.

0	$	$	$	$	$	$	$	$	$
Vstup	a	a	c	a	a	c	a	a	a
	0	1	2	3	4	5	6	7	8

Tabuľka 5.6: Sady vlákien pre prvý riadok tabuľky, kde dĺžka vstupu je 9

Skupiny vlákien budú bežať v sadách, to znamená, že akonáhle jedno vlákno zo sady skončí, ďalšie sa nemôže vytvoriť hneď, ale musí počkať na skončenie všetkých vlákien danej sady. Až potom môže toto vlákno začať ako súčasť novej sady. Počet súčasne bežiach vlákien nie je rovnaký počas celého algoritmu, ale môže sa znižovať, ak prvé vlákno z jednej sady a iné vlákno z tej istej sady majú index riadka líšiaci sa o viac ako 1. V tom prípade sa počet vlákien zmenší dvakrát ($v_{new} = \frac{v}{2}$). V príklade uvedenom v tab. 5.7 sada vlákien s označením 2 presahuje z riadku 15 až na riadok 18. V tomto prípade by sa po dokončení sady s číslom 2 znížil počet vlákien bežiacich v sade z aktuálnych 10 na polovicu. Znižovanie počtu vlákien je navrhnuté kvôli veľkým tabuľkám, v ktorých by vlákna spustené na konci sady zbytočne čakali na výpočet prvých vlákien tej istej sady.

Paralelná verzia v pseudokóde je uvedená v alg. 9 a činnost vlákien je uvedená vo funkciách v alg. 8.



Tabuľka 5.7: Dosiahnutie podmienky pre zníženie počtu vlákien

Algoritmus 8: Funkcie pre paralelný CYK pseudokód

```
// Funkcia pre vlákno spracúvajúce bunky prvého riadka
function launch_process[l] with param i:
if (A → a_i ∈ R) then
    add A to CYK[j,i];
else
    add - to CYK[j,i];
end if
```

```
// Funkcia pre vlákno spracúvajúce bunky všetkých riadkov okrem prvého
function launch_process[l] with param j, i:
for (k = 0; k < j; k++) do
    if (B ∈ CYK[k, i] and C ∈ CYK[j − k − 1, i + k + 1]) then
        add A to CYK[j,i] for some A, B, C ∈ _G N;
    else
        add - to CYK[j,i];
    end if
end for
```

Algoritmus 9: Paralelný CYK pseudokód

```
1  i = 0;    j = 0;
2  n =length of input string;
3  v = n/2 + 1; // Maximálny počet súčasne bežiacich vlákien
   // Spracovanie prvého riadka tabuľky
4  while (i < n) do
       // Vytvorenie vlákna pre každú bunku prvého stĺpca
5      for (l = 0; l < v; l++) do
6          if (i < n) then
               // Vytvorí nové vlákno pre výpočet bunky [0,i]
7              launch_thread[l] with param i;
8              i++;
9          else
10             goto end_init;
11         end if
12     end for
13 end while
14 end_init:
   // Iterácia cez všetky zvyšné riadky
15 for (j = 1; j < n; j++) do
       // Iterácia cez všetky stĺpce v danom riadku
16     for (i = 0; i < n − j; i++) do
17         if (is_some_thread_free()) then
               // Vytvorí nové vlákno pre výpočet bunky [j,i]
18             launch_thread[l] with params j,i;
19         else
20             wait_until_all_threads_are_free();
21             launch_thread[l] with params j,i;
22             continue;
23         end if
24         if (|j of first launched thread in batch −j of some other thread in the same
             batch | >= 2) then
25             v = v/2;
26         end if
27     end for
28 end for
29 if (Starting symbol ∈ CYK[n, 0]) then
30     input string belong to grammar
31 else
32     input string does not belong to grammar
33 end if
```

5.3 Návrh rozšírenia CYK algoritmu o rozbor pravidiel

Výsledkom Cocke-Younger-Kasami algoritmu je rozhodnutie, či vstupný reťazec terminálov patrí do danej gramatiky. Často nás však v prípade úspechu zaujíma aj rozbor pravidiel, ktorými sme dospeli k výsledku. Zaujímajú nás teda všetky poradia pravidiel, pomocou ktorých môžeme z počiatočného neterminálu vygenerovať vstupný reťazec.

Na dosiahnutie tohto je potrebné urobiť niekoľko zmien v CYK algoritme. Prvou zmenou je, že do tabuľky sa nebudú ukladať neterminály z ľavej strany pravidiel, ale priamo index pravidla. Pôvodný neterminál sa dá zistiť z ľavej strany pravidla s príslušným indexom v zozname všetkých pravidiel. Práve preto je potrebné pravidlá zapísané v tvare $A \to BC|a$ rozpísať na dve pravidlá $A \to BC$ a $A \to a$. Všetky pravidlá treba potom očíslovať (obr. 5.1).

1. $S \to AB|BC$
2. $A \to BA|a$
3. $B \to CC|b$
4. $C \to AB|a$

$\longrightarrow$

1. $S \to AB$
2. $S \to BC$
3. $A \to BA$
4. $A \to a$
5. $B \to CC$
6. $B \to b$
7. $C \to AB$
8. $C \to a$

Obr. 5.1: Rozpísanie gramatiky a očíslovanie pravidiel

Ďalej je treba vytvoriť štruktúru, ktorá bude uchovávať informácie o všetkých deriváciách, ktoré sa vykonajú počas CYK algoritmu. Dátová štruktúra je zobrazená v tab. 5.8. Derivácie sa ukladajú v momente, keď v CYK algoritme skonkatenované podčasti vyhovujú pravej strane niektorého z pravidiel. Do derivácie sa uloží pozícia práve vyhodnocovanej bunky a pozície buniek jednotlivých podčastí. Je potrebné uložiť aj indexy pravidiel, ktorých ľavé strany vystupujú ako podčasti v skonkatenovanom reťazci.

Štruktúra derivácie	
Pozícia výslednej bunky	Index pravidla výslednej bunky
Pozícia bunky s ľavou podčasťou	Index pravidla bunky ľavej podčasti
Pozícia bunky s pravou podčasťou	Index pravidla bunky pravej podčasti

Tabuľka 5.8: Dátová štruktúra pre deriváciu

Algoritmus nájdenia postupnosti pravidiel začína len vtedy, ak vstupný reťazec patrí do danej gramatiky. Zo všetkých derivácií sa nájdu len tie, ktoré majú ako *pozíciu výslednej bunky* najvrchnejšiu bunku tabuľky. Z týchto nájdených derivácií sa vyberú tie, ktoré majú *index pravidla výslednej bunky* taký, že dané pravidlo má na ľavej strane počiatočný neterminál. Pre každú z týchto derivácií sa vytvorí vlastný stav. Pojem stav v tomto texte predstavuje zoznam použitých pravidiel (na začiatku prázdny) a zásobník s ešte nespra-

covanými deriváciami. Každá z nájdených derivácií sa uloží do jedného stavu na zásobník derivácií. Následne je potrebné každý stav vyhodnotiť.

Vyhodnotenie stavu zahŕňa vybratie derivácie zo zásobníku. *Index pravidla výslednej bunky* sa uloží do zoznamu použitých pravidiel v danom stave. V zozname všetkých derivácií sa vyhľadajú všetky derivácie podľa dvojice hodnôt [*pozícia bunky pravej podčasti, index pravidla pravej podčasti*]. Zaujímajú nás derivácie, ktoré majú túto dvojicu rovnú dvojici [*pozícia bunky výsledku, index pravidla výsledku*]. To isté sa vykoná aj pre pozíciu ľavej bunky.

Vznikli nám dve množiny, ktoré obsahujú všetky rôzne derivácie z daného stavu pre ľavú a pravú podčasť. Následne urobíme N kópií súčasného stavu, kde N je počet prvkov množiny výsledku kartézskeho súčinu množín pravej a ľavej podčasti. Do každej kópie sa uloží na zásobník jeden prvok z výslednej množiny. Na zásobník derivácií sa derivácie vždy ukladajú v poradí pravá derivácia, ľavá derivácia. Je to z toho dôvodu, že skorším vybratím ľavej derivácie zo zásobníka a jej aplikovaním sa vždy bude nahrádzať najľavejší neterminál v reťazci. Kópia stavu sa potom uloží k ostatným nevyhodnoteným stavom, kde bude čakať na vyhodnotenie. Vyhodnocovanie aktuálneho stavu sa zruší, lebo z neho vzniklo mnoho nových stavov ktoré budú vyhodnotené neskôr.

Pri hľadaní derivácie podčasti sa môže stať, že daná derivácia sa v zozname všetkých derivácií nevyskytuje. Dôvodom je, že hľadaná derivácia obsahuje nahradenie neterminálu terminálom (terminálové pravidlo). V tomto prípade sa vytvorí nová derivácia, do ktorej sa uloží na pozíciu výslednej bunky hodnota hľadanej bunky a na index výsledného pravidla hľadané pravidlo. Na miesto podčastí sa uložia prázdne hodnoty, aby bolo pri ďalšom spracovávaní jasné, že derivácia už nemá podčasti. Pseudokód popísaného algoritmu je uvedený v alg. 10

Príklad: Máme gramatiku, kde $S, A, B, C \in N$ a $a, b, c \in T$. Počiatočný neterminál je S.

1. $S \to AB$
2. $S \to BC$
3. $A \to BA$
4. $A \to a$
5. $B \to CC$
6. $B \to b$
7. $C \to AB$
8. $C \to a$

Vstupný reťazex je *baaba*. Výsledok CYK algoritmu je uvedený v tab. 5.10 a výsledok s indexami pravidiel namiesto neterminálov v tab. 5.9. Zoznam všetkých derivácií je uvedený v tab. 5.2.

Začiatočné derivácie, ktoré budú na vrchole zásobníka sú derivácie vyznačené na obr. 5.2.

Činnosť algoritmu pre náš príklad je nasledovná. Zo zoznamu sa vyberie prvý stav obsahujúci na svojom zásobníku derivácií deriváciu s číslom 15 (obr 5.2). Derivácia 15 sa vyberie zo zásobníku a do zoznamu použitých pravidiel sa uloží číslo 2, lebo 2 je index

1.

Bunka	[1,0]	index	3
Ľavá	[0,0]	Ľavý index	6
Pravá	[0,1]	Pravý index	4

2.

Bunka	[1,0]	index	2
Ľavá	[0,0]	Ľavý index	6
Pravá	[0,1]	Pravý index	8

3.

Bunka	[1,1]	index	5
Ľavá	[0,1]	Ľavý index	8
Pravá	[0,2]	Pravý index	8

4.

Bunka	[1,2]	index	1
Ľavá	[0,2]	Ľavý index	4
Pravá	[0,3]	Pravý index	6

5.

Bunka	[1,2]	index	7
Ľavá	[0,2]	Ľavý index	4
Pravá	[0,3]	Pravý index	6

6.

Bunka	[1,3]	index	3
Ľavá	[0,3]	Ľavý index	6
Pravá	[0,4]	Pravý index	4

7.

Bunka	[1,3]	index	2
Ľavá	[0,3]	Ľavý index	6
Pravá	[0,4]	Pravý index	8

8.

Bunka	[2,1]	index	5
Ľavá	[0,1]	Ľavý index	8
Pravá	[1,2]	Pravý index	7

9.

Bunka	[2,2]	index	5
Ľavá	[1,2]	Ľavý index	7
Pravá	[0,4]	Pravý index	8

10.

Bunka	[3,1]	index	1
Ľavá	[0,1]	Ľavý index	4
Pravá	[2,2]	Pravý index	5

11.

Bunka	[3,1]	index	7
Ľavá	[0,1]	Ľavý index	4
Pravá	[2,2]	Pravý index	5

12.

Bunka	[3,1]	index	3
Ľavá	[1,1]	Ľavý index	5
Pravá	[1,3]	Pravý index	3

13.

Bunka	[3,1]	index	3
Ľavá	[2,1]	Ľavý index	5
Pravá	[0,4]	Pravý index	4

14.

Bunka	[3,1]	index	2
Ľavá	[2,1]	Ľavý index	5
Pravá	[0,4]	Pravý index	8

15.

Bunka	[4,0]	index	2
Ľavá	[0,0]	Ľavý index	6
Pravá	[3,1]	Pravý index	7

16.

Bunka	[4,0]	index	3
Ľavá	[0,0]	Ľavý index	6
Pravá	[3,1]	Pravý index	3

17.

Bunka	[4,0]	index	1
Ľavá	[1,0]	Ľavý index	3
Pravá	[2,2]	Pravý index	5

18.

Bunka	[4,0]	index	7
Ľavá	[1,0]	Ľavý index	3
Pravá	[2,2]	Pravý index	5

Obr. 5.2: Zoznam všetkých derivácií s vyznačenými počiatočnými deriváciami

Algoritmus 10: Rozbor pravidiel s použitím CYK algoritmu

```
1  struct_state {
2      used_rules
3      derivation_stack
4  }
5  struct_derivation {
6      cell_current
7      current_idx
8      cell_left
9      left_idx
10     cell_right
11     right_idx
12 }
13 List<used_rules> results; // Zoznam výsledných postupností
14 List<struct_derivation> derivations; // Zoznam všetkých derivácií
15 List<struct_state> states; // Zoznam stavov na vyhodnotenie
16 states = find_all_starting_states(); // Nájde všetky derivácie
       najvrchnejšej bunky tabuľky obsahujúce pravidlo s počiatočným
       neterminálom na ľavej strane a vytvorí nové stavy
17 for each state from states do
18     while state[i].derivation_stack not empty do
19         d = state[i].derivation_stack.pop;
20         state[i].used_rules.insert(d.current_idx);
21         right_set = find_derivations(d.cell_right, d.right_idx);
22         left_set = find_derivations(d.cell_left, d.left_idx);
23         create_states(right_set, left_set);
24     end while
25     results.insert(state[i].used_rules);
26 end for
27 print_results();
```

použitého pravidla v derivácii. Následne sa v zozname derivácií vyhľadá derivácia pravej podčasti s pozíciou [3,1] a s pravidlom 7 (ďalej uvádzané už len ako [3,1]:7).

Hľadaná derivácia je v tomto prípade len jedna a to na mieste 11. Celá derivácia sa uloží do zásobníka. To isté by sme spravili pre ľavú časť, ale v tomto prípade hľadáme deriváciu [0,0]:6. Táto derivácia sa v zozname nenachádza, lebo pochádza z prvého riadka CYK tabuľky. Preto vytvoríme novú deriváciu s týmito hodnotami (obr. 5.3). Takto vytvorenú deriváciu uložíme na zásobník derivácií.

Keďže zásobník nie je prázdny, vyberieme ďalšiu deriváciu z jeho vrchola (obr. 5.4). Do zoznamu použitých pravidiel uložíme index 6. Vidíme, že derivácia nemá žiadne podčasti, preto pokračujeme ďalšou iteráciou.

V ďalšej iterácii vyberieme deriváciu [3,1]:7, po ktorej ostane zásobník prázdny. Príslušné pravidlo uložíme do zoznamu použitých pravidiel a následne vyhľadáme deriváciu pravej podčasti [2,2]:5 (číslo 9 v obr. 5.2). Pre ľavú podčasť [0,1]:4 vytvoríme novú deri-

4	2,3,1,7				
3	-1	1,7,3,2			
2	-1	5	5		
1	3,2	5	1,7	3,2	
0	6	4,8	4,8	6	4,8
Vstup	b	a	a	b	a
	0	1	2	3	4

Tabuľka 5.9: CYK tabuľka s indexami pravidiel

4	S,A,C				
3	-	S,C,A			
2	-	B	B		
1	A,S	B	S,C	A,S	
0	B	A,C	A,C	B	A,C
Vstup	b	a	a	b	a
	0	1	2	3	4

Tabuľka 5.10: CYK tabuľka s neterminálmi

Bunka	[0,0]	index	6
Ľavá	[-1,-1]	Ľavý index	-1
Pravá	[-1,-1]	Pravý index	-1

Obr. 5.3: Derivácia prvého riadka vytvorená pre vloženie na zásobník

váciu, pretože je zase z prvého riadka CYK tabuľky. Nasledujúce iterácie sú analogické s predošlými.

Výsledná postupnosť pravidiel a odvodenie z počiatočného neterminálu je zobrazené na obr. 5.5.

Keďže existuje aj ďalší počiatočný stav ([4,0]:1), môžeme povedať, že existuje aspoň jeden ďalší rozbor. V našom prípade existuje už len jeden ďalší rozbor, lebo žiadne nové stavy nevznikli počas vykonávania algoritmu. Tento rozbor je zobrazený na obr. 5.6.

Bunka	[3,1]	index	7
Ľavá	[0,1]	Ľavý index	4
Pravá	[2,2]	Pravý index	5

⇒

Bunka	[0,0]	index	6
Ľavá	[-1,-1]	Ľavý index	-1
Pravá	[-1,-1]	Pravý index	-1

Obr. 5.4: Zásobník derivácií po spracovaní prvej derivácie (vrchol je označený ⇒)

$S \Rightarrow BC[2] \Rightarrow bC[6] \Rightarrow bAB[7] \Rightarrow baB[4] \Rightarrow baCC[5] \Rightarrow baABC[7] \Rightarrow baaBC[4] \Rightarrow baabC[6] \Rightarrow baaba[8]$

Obr. 5.5: Výsledný rozbor z počiatočného stavu [4,0]:2

$S \Rightarrow AB[1] \Rightarrow BAB[3] \Rightarrow bAB[6] \Rightarrow baB[4] \Rightarrow baCC[5] \Rightarrow baABC[7] \Rightarrow baaBC[4] \Rightarrow baabC[6] \Rightarrow baaba[8]$

Obr. 5.6: Výsledný rozbor z počiatočného stavu [4,0]:1

Kapitola 6

Implementácia

Táto kapitola sa zaoberá implementáciou programu a popisom použitých technológií.

Za implementačný jazyk som si zvolil jazyk C++, kvôli jeho rýchlosti a dynamickej alokácii pamäti. Program je koncipovaný ako terminálová aplikácia. Vývoj prebiehal v prostredí Linuxu, konkrétne Ubuntu 18.04. Testovanie potom prebiehalo na systémoch s Ubuntu 18.04 a Centos 7. Na preklad bol použitý prekladač G++ verzie 7.3.0 a 7.4.0. Hoci jazyk C++ poskytuje objektovo orientovanú paradigmu, rozhodol som sa nevyužiť tieto vlastnosti.

6.1 Prepínače programu

Beh programu je možné ovplyvniť niekoľkými prepínačmi, ktoré pozmeňujú množstvo výsledkov vypísaných na výstup programu.

- `-h/--help` program vypíše nápovedu
- `-f/--file filepath` určuje relatívnu cestu k súboru obsahujúcemu gramatiku
- `-l/--left` na výstupe zobrazí CYK tabuľku s neterminálmi
- `-i/--id` na výstupe zobrazí CYK tabuľku s poradovými číslami pravidiel
- `-s/--succession` na výstupe vypíše postupnosti všetkých pravidiel, z ktorých sa dá vygenerovať vstupný reťazec

Prepínače sú spracované vo funkcii `main()`, z ktorej sú následne volané funkcie `load_input_file()`, `cyk()` a `get_successions()` zabezpečujúce jednotlivé činnosti programu.

6.2 Členenie programu

6.2.1 Načítanie gramatiky zo súboru

Všetky funkcie, ktorých činnosť súvisí s načítaním gramatiky, sa nachádzajú v súbore s názvom `input.cpp/.h`. Aby sa súbor podarilo úspešne načítať, musí byť každé pravidlo v súbore uložené na osobitnom riadku. Ďalej neterminály môžu obsahovať len veľké písmená anglickej abecedy a terminály malé písmená anglickej abecedy a číslice. V súbore sa musí nachádzať aspoň jedno pravidlo, kde na ľavej strane je písmeno S (počiatočný neterminál).

Oddeľovač ľavej a pravej strany pravidla musí byť `->`. V prípade, že pravidlo na riadku nebude vyhovovať, preskočí sa, vypíše sa chybová hláška a pokračuje sa spracovaním ďalšieho riadka. Na skontrolovanie, či je pravidlo v správnom tvare, sú použité regulárne výrazy.

Pravidlá sú ukladané do dvoch asociatívnych polí [1], jedno pre terminálové pravidlá a druhé pre neterminálové. Ako kľúč je vždy ukladaná pravá strana pravidla, lebo podľa pravej strany sa bude v štruktúre vyhľadávať. V C++ je táto štruktúra reprezentovaná dátovým typom *std::map a std::multimap*. Na môj účel sa lepšie hodí multimap, lebo v nej musí byť dvojica kľúč-hodnota jedinečná v celej štruktúre (kľúče/pravé strany pravidla sa môžu opakovať) a v map musí byť každý kľúč jedinečný [1].

6.2.2 Paralelný Cocke-Younger-Kasami algoritmus

Algoritmus CYK je implementovaný vo funkcii `cyk()` v súbore `cyk.cpp/.h`. Pomocné funkcie pre vyhodnocovanie buniek CYK tabuľky sa nachádzajú v súbore `functions.cpp/.h`. Tabuľka je realizovaná ako vektor vektorov. Vektory prvej úrovne obsahujú rozdielny počet stĺpcov v každom riadku. Každá bunka obsahuje vektor dátového typu `int`, ktorý reprezentuje uložené indexy pravidiel v bunke. V C++ to vyzerá nasledovne: `vector<vector<vector <int> > > table`. Bunky sú na začiatku inicializované na hodnotu -2, ktorá označuje, že bunka ešte nebola spracovaná.

V `cyk()` funkcii sa z hlavného vlákna vytvárajú jednotlivé vlákna. Skupiny vlákien, ktoré bežia súčasne, sú ukladané do vektora vlákien. Návratové kódy jednotlivých vlákien tej istej sady sú ukladané do vektora `result_codes` dátového typu `int`. Príslušná pozícia vo vektore sa predá vláknu ako ukazateľ, cez ktorý vlákno vráti svoj návratový kód hlavnému vláknu. Po skončení sady sa vždy návratové kódy skontrolujú a v prípade chyby sa vykonávanie programu skončí s príslušným dôvodom. Toto je realizované vo funkcii `wait_for_batch()`. V tejto funkcii sa taktiež mažú už ukončené vlákna.

Vláknam, ktoré vyhodocujú terminálové pravidlá, je predaná funkcia `launch_T_process()`. V nej sa pomocou funkcie `get_string_at_position_from_string()` získa terminál zo vstupného reťazca. Funkcia `get_valid_T_rules()` potom nájde všetky vyhovujúce pravidlá z dátovej štruktúry *std::multimap* uchovávajúce terminálové pravidlá. Funkcia `save_rules_to_table()` ich následne uloží do tabuľky.

Pre vlákna spracúvajúce neterminálové pravidlá sa zavolá funkcia `launch_N_process()`. V nej sa vytvoria tri štruktúry `struct_cell_position` uchovávajúce pozíciu aktuálnej bunky a pozície buniek pre jednotlivé podčasti. Následne sa volá funkcia `evaluate_cell()`, ktorá vyhodnotí podčasti a nájde všetky vyhovujúce pravidlá. V prípade, že chceme po skončení programu vedieť aj postupnosti pravidiel, vytvoria sa v tejto funkcii aj štruktúry derivácií `struct_derivation`. Rovnako ako pri terminálových pravidlách, funkcia `save_rules_to_table()` ich ukladá do tabuľky.

V prípade, že bunka podčasti ešte nebola vyhodnotená, vlákno sa uspí na 10 mikrosekúnd a potom skúsi znova získať dáta pre podčasť. Takto to bude prebiehať, kým sa bunka podčasti nevyhodnotí iným vláknom.

Synchronizácia je zaistená jedným globálnym zámkom `mtx_cell` zdieľaným všetkými vláknami. Problém nastával v momente, keď jedno vlákno chcelo zapísať výsledok do bunky tabuľky (funkcia `save_rules_to_table()`) a iné vlákno potrebovalo dáta z tej istej bunky pre vyhodnocovanie vlastnej bunky (funkcia `wait_for_cell_evaluation()`). Iné synchronizačné problémy neboli nájdené, lebo každé vlákno pracuje so získanými dátami nezávisle od ostatných vlákien a výsledky zapisuje do tabuľky na pridelenú pozíciu.

[1] Dátová štruktúra zložená z dvojíc kľúč-hodnota.

6.2.3 Rozšírenie rozboru pravidiel

Funkcie spracúvajúce rozbor pravidiel sú v súbore `succession.cpp/.h` a ich pomocné funkcie sú v `succession_functions.cpp/.h`.

Na list stavov uvedený v pseudokóde v alg. 10 bola použitá dátová štruktúra zásobník (`stack<STRUCT_SUCCESSION>`). Štruktúra si na rozdiel od tej uvedenej v pseudokóde uchováva aj aktuálny reťazec, na ktorom si simuluje použitie jednotlivých derivácií. Vo funkcii `get_successions()` sa pre každý stav nachádzajúci sa v zásobníku stavov volá `try_succession()`.

V `try_succession()` sa pre každú deriváciu na zásobníku derivácií v danom stave uloží index pravidla a daná derivácia sa aplikuje na aktuálny reťazec vo funkcii `replace_leftmost_symbol()`. Ďalej sa vyhľadajú podčasti derivácie a z počtu ich rôznych podčastí sa určí, či sa uložia na zásobník derivácií v aktuálnom stave alebo sa z ich kombinácií vytvoria nové stavy a súčasný stav sa skončí. Všetky možnosti sú uvedené v tabuľke 6.1. Číslo -1 v tabuľke značí, že derivácia už nemá podčasť. Vložením na zásobník derivácií sa myslí zásobník v aktuálnom stave. Pri vytváraní nových stavov sa súčasný stav naklonuje a na jeho zásobník derivácií sa vloží jedna z možných kombinácií podčastí.

Ľavé podčasti	**Pravé podčasti**	**Akcia**
1	1	Vlož pravú a potom ľavú podčasť na zásobník derivácií
>=1	>=1	Vytvor z ich kombinácií nové stavy
-1	-1	Nerob nič
-1	1	Vlož pravú časť na zásobník derivácií
1	-1	Vlož ľavú časť na zásobník derivácií
>1	-1	Vytvor nové stavy z ľavých podčastí
-1	>1	Vytvor nové stavy z pravých podčastí

Obr. 6.1: Akcie pre všetky kombinácie rôznych počtov derivácií podčastí

Po vyprázdnení zásobníku derivácií sa ešte pre kontrolu porovná vygenerovný reťazec so vstupným a ak sa zhodujú, uloží sa postupnosť pravidiel vektora výsledkov (`vector<vector <int> > rule_successions`).

6.2.4 Návratové kódy programu

Program po skončení svojej činnosti vracia návratový kód. Podľa tohto kódu je možné určiť, či program skončil korektne alebo sa vyskytla nejaká chyba. Pri chybe sa naviac vypíše aj chybová hláška upresňujúca chybu. Typy chýb, ktoré sa môžu vyskytnúť, sú buď logického typu (nesprávna činnosť algoritmu) alebo systémové chyby (zlyhá alokácia pamäte, nepodarí sa otvoriť súbor). Chybové kódy:

- 0 program sa ukončil správne
- 1 vyskytla sa chyba počas vykonávania paralelného CYK algoritmu
- 2 programu boli zadané neznáme prepínače alebo ich zlá kombinácia
- 3 vyskytla sa chyba počas načítania gramatiky zo súboru
- 5 vyskytla sa chyba počas rozboru pravidiel

- 10 neočakávaná sémantická chyba počas rozboru pravidiel

6.3 Testovanie

Testovanie prebiehalo vždy po napísaní malej časti programu (funkcia alebo logický celok programu). Pre každú časť som sa snažil nájsť všetky možné stavy, ktoré som následne ošetril. Po dokončení implementácie bol program otestovaný aj ako celok.

Na overenie vyplnenej CYK tabuľky a správnosti paralelného CYK algoritmu som využil ručne vypočítané jednoduché príklady, ktorými som sa snažil pokryť špeciálne prípady, ktoré mohli nastať počas vykonávania algoritmu.

6.4 Známe nedostatky programu

Po skončení testovania nebola počas vykonávania CYK algoritmu známa žiadna závažná chyba. Pri vyhodnocovaní rozboru pravidiel pre veľké vstupy (aspoň 10 terminálov na vstupe) môže byť doba trvania programu aj niekoľko minút. Je to z toho dôvodu, lebo rozbor pravidiel sa na rozdiel od CYK algoritmu vykonáva sekvenčne jedným vláknom. Okrem toho je to ovplyvnené aj hľadaním všetkých možných kombinácií pravidiel. S každým ďalším terminálom na vstupe doba trvania programu rastie exponenciálne.

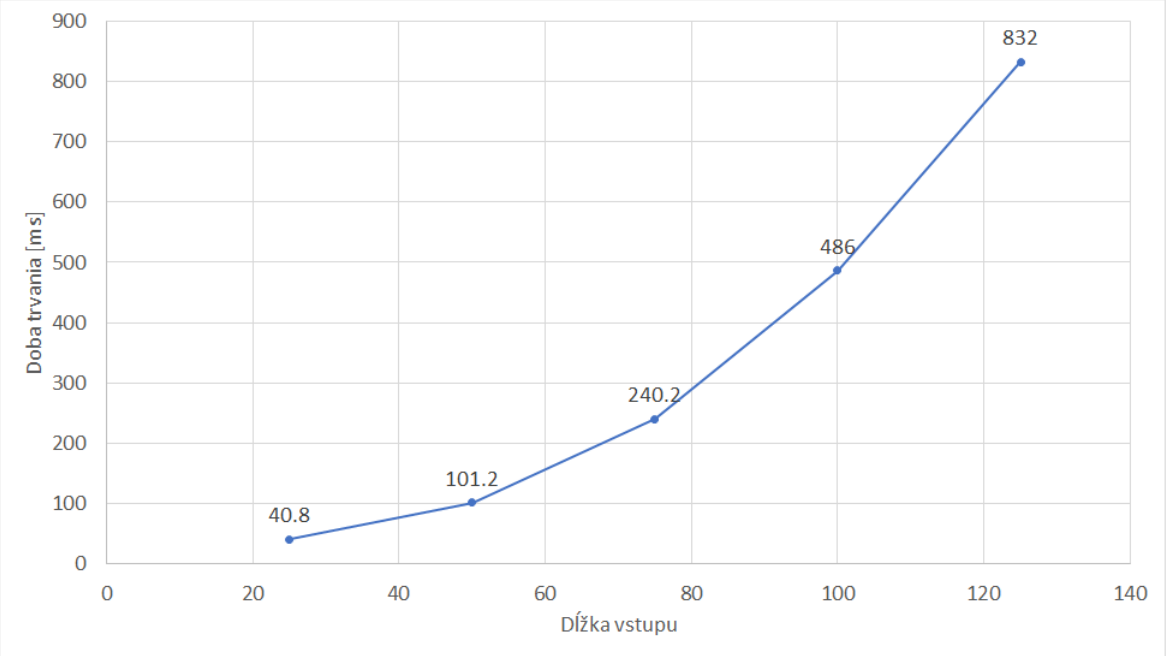
6.5 Zhodnotenie paralelného CYK programu

V tejto sekcii si ukážeme a zhodnotíme namerané výsledky implementovaného analyzátora. Výsledky boli namerané na dvoch rôznych počítačoch, jeden mal 2 fyzické a 4 logické jadrá na frekvencii 2,1 Ghz (označenie PC1 v grafoch) a druhý mal 6 fyzických jadier a 12 logických jadier na frekvencii 2,5 Ghz (označenie PC2 v grafoch). Pre dané vstupy boli jednotlivé merania vykonané niekoľkokrát a následne spriemerované.

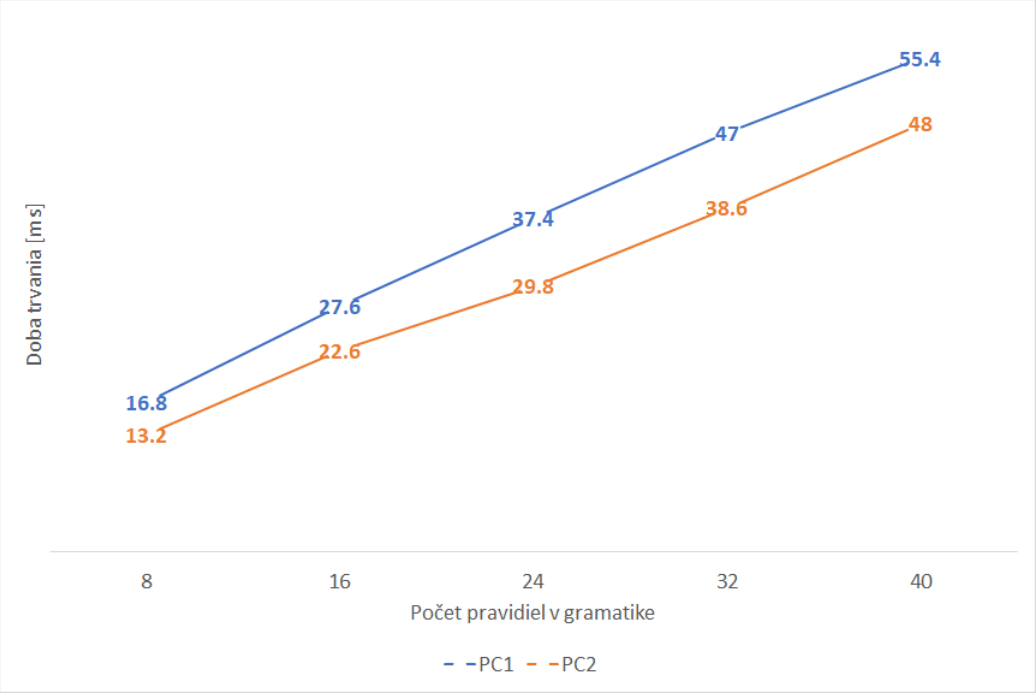
Pripomeňme si, že náročnosť CYK algoritmu je v najhoršom prípade $O(n^3 \cdot |G|)$. Z obr. 6.2 je možné vidieť ako so zväčšujúcim sa vstupným reťazcom narastá čas behu programu exponenciálne. Pre každé zdvojnásobenie vstupu doba behu programu narastie zhruba $1,7-2,5$-násobne. Gramatika v použitom príklade bola nasledovná:

1. $S \to AB$
2. $S \to BC$
3. $A \to BA$
4. $A \to a$
5. $B \to CC$
6. $B \to b$
7. $C \to AB$
8. $C \to a$

Ďalším faktorom vplývajúcim na dobu behu programu je veľkosť gramatiky (počet pravidiel). Zdvojnásobením pravidiel doba behu narastá len lineárne, ako je to zobrazené na obrázku 6.3.

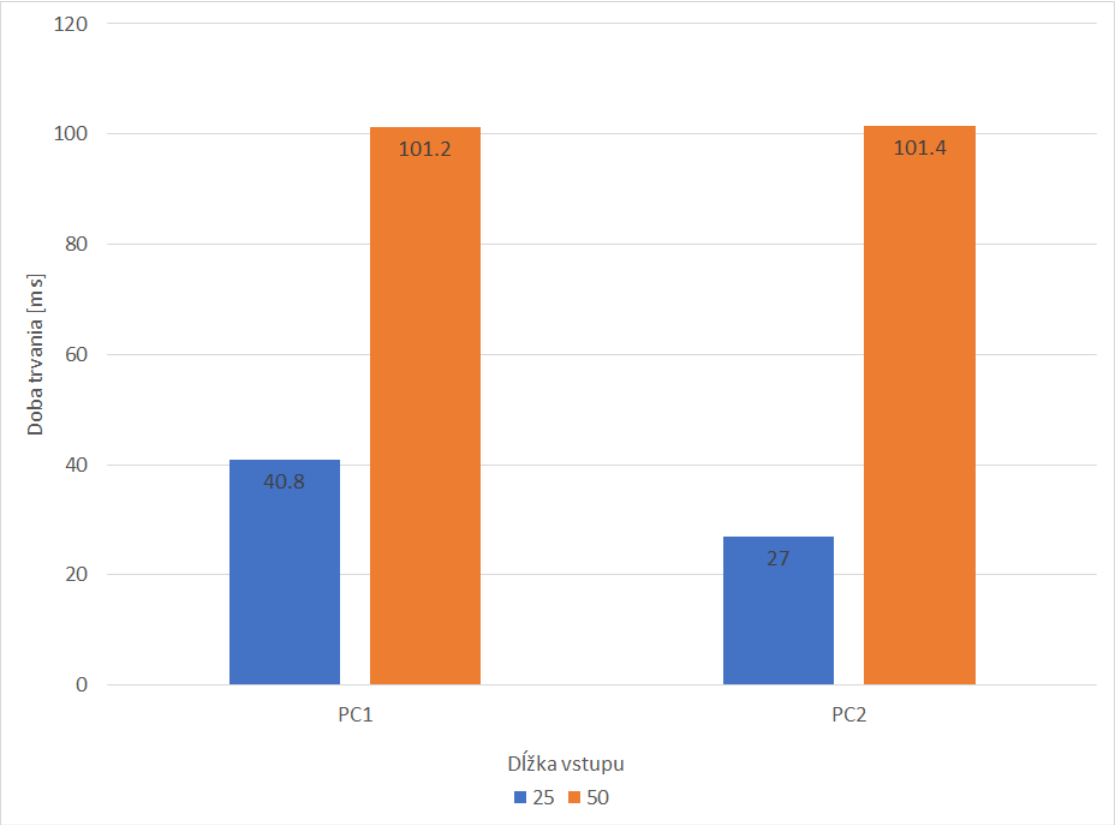


Obr. 6.2: Časová závislosť od dĺžky vstupu (PC1)



Obr. 6.3: Časová závislosť od veľkosti gramatiky

Na obrázku 6.4 je zobrazené porovnanie doby behu na dvoch počítačoch pre vstupy dĺžky 25 a 50. Z merania je zrejmé, že pre veľké vstupy (v našom prípade vstup dĺžky 50) sa doby behov na oboch počítačoch takmer vôbec nelíšia. To znamená, že počet vlákien CPU má vplyv na rýchlosť skôr pri kratších vstupných reťazcoch než pri dlhých vstupoch.

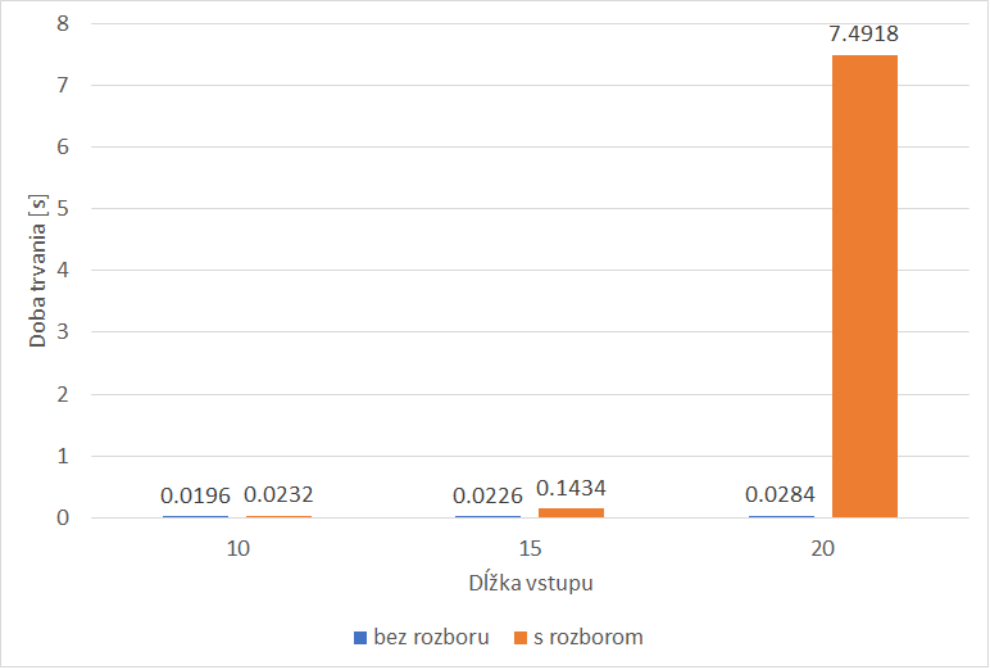


Obr. 6.4: Porovnanie rýchlostí na rôznych PC pre rôzne dĺžky vstupu

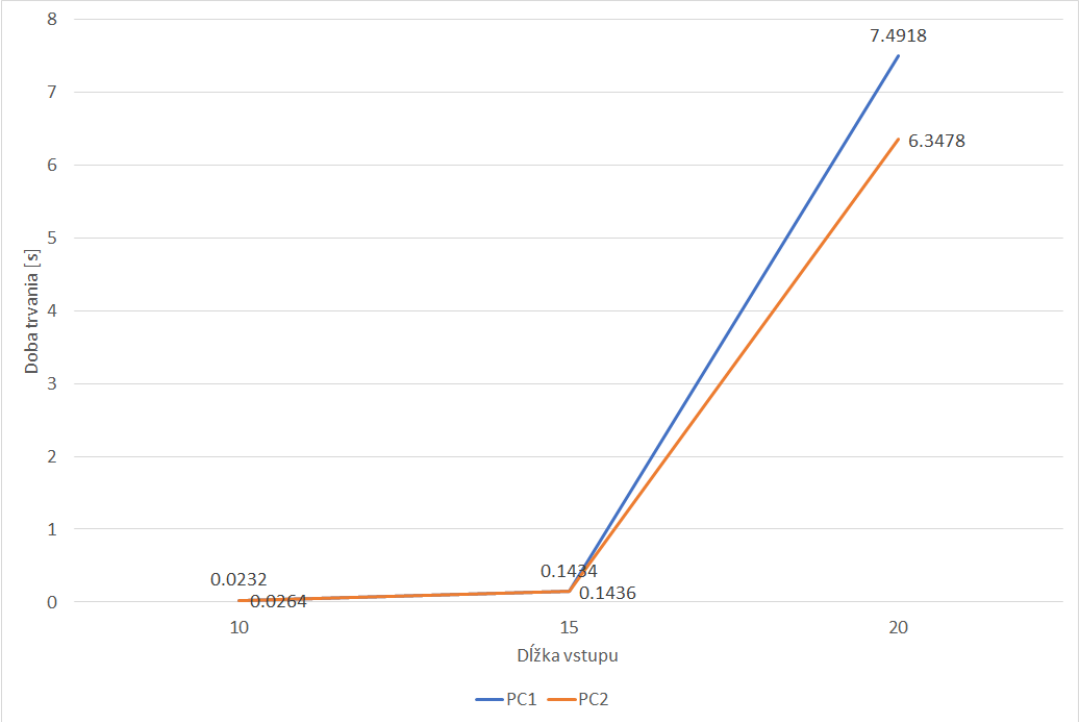
Treba zdôrazniť, že viac vlákien neznamená automaticky rýchlejší beh programu, lebo vytváranie, rušenie a prepínanie vlákien má nezanedbateľný vplyv na výkon procesora. Program by sa dal upraviť tak, že by na začiatku existoval nejaký pevný počet vlákien, ktoré by bežali v nekonečnom cykle a v prípade potreby výpočtu by sa im predali pozície buniek na výpočet. Po skončení výpočtu by stále bežali v cykle, no nevykonávali by nijakú činnosť. Počet vlákien by sa mohol zmenšovať rovnako, ako bolo navrnuté v tejto práci. Táto úprava by mohla ušetriť čas, ktorý zaberá neustále vytváranie a rušenie vlákien, ktoré vykonali svoju činnosť.

Teraz sa pozrieme na rozšírenie, ktoré zisťuje rozbory pravidiel a jeho dopad na dobu trvania programu. Nájdenie všetkých pravidiel, pomocou ktorých sa dá vygenerovať vstupný reťazec, má malý vplyv na výkon programu pre krátke vstupy. Na obrázku 6.5 pre vstupy do dĺžky 15 doba behu programu s rozšírením narástla len 6-krát oproti tej bez rozšírenia. Pre vstup dĺžky 20 doba behu programu narástla zhruba 264-krát oproti behu programu bez rozboru pravidiel. Z toho môžeme vidieť, že hľadanie rozborov pravidiel nie je triviálna úloha. Z grafu na obr. 6.6 je dobre vidieť, že s narastajúcim vstupom sa doba behu programu bude ďalej len exponenciálne zvyšovať. V budúcnosti je tu priestor na vylepšenie tohto roz-

šírenia paralelizovaním behu pre viac vlákien. Exponenciálna zložitosť by sa neeliminovala, no urýchlenie by malo zmysel.



Obr. 6.5: Vplyv rozboru pravidiel na výkon PC1



Obr. 6.6: Zložitosť rozboru pravidiel s narastajúcim vstupom

Kapitola 7

Záver

Cieľom tejto práce bolo navrhnúť paralelnú verziu syntaktickej analýzy využívajúcej algoritmus Cocke-Younger-Kasami. K tomu bolo potrebné naštudovať fungovanie jednotlivých častí prekladača. Ďalej bolo potrebné porozumieť bezkontextovým gramatikám a ich prevodom do normálnych foriem. Pozornosť som venoval najmä Chomského normálnej forme. Práca si vyžiadala podrobné štúdium Cocke-Younger-Kasami algoritmu, lebo jeho vstupom býva práve gramatika v Chomského normálnej forme. Tento algoritmus si počas svojho behu konštruuje tabuľku a je dopodrobna vysvetlený v tejto práci.

Ďalej bolo potrebné navrhnúť paralelnú verziu tohto algoritmu a ukázať tak, že syntaktickú analýzu je možné paralelizovať a tým ju urýchliť. Myšlienka môjho návrhu spočíva v tom, že výpočet jednej bunky sa vždy vykonáva samostatne, teda nezasahuje do iných buniek, len získava dáta z buniek vypočítaných v predošlom kroku. Na začiatku behu sa preto vytvorí $n/2+1$ vlákien, kde n je dĺžka vstupného reťazca. Vlákna vždy bežia v sadách, ktoré spracovávajú vždy susediace bunky. V prípade, že sada vlákien presahuje v CYK tabuľke aspoň cez dva riadky, počet súčasne bežiacich vlákien sa zníži na polovicu. Deje sa to z toho dôvodu, aby vlákna nemuseli zbytočne čakať na vlákna tej istej sady, ktoré spracovávajú bunky riadkov skôr vypočítaných.

Návrh som ešte rozšíril o nadstavbu, ktorá si počas stavby CYK tabuľky uchováva všetky derivácie, s pomocou ktorých po skončení CYK algoritmu zisťuje všetky rozbory pravidiel, ak reťazec patril do danej gramatiky. Následne po skončení CYK algoritmu sa pomocou derivácií zostavujú všetky ľavé rozbory pravidiel.

Výsledkom práce som vytvoril terminálovú aplikáciu v jazyku C++, ktorá načítava gramatiku zo súboru a vstupný reťazec zo štandarného vstupu. Na výstupe potom aplikácia oznámi, či vstupný reťazec je možné vygenerovať z danej gramatiky, a v prípade záujmu, aj všetky rozbory pravidiel. Aplikácia taktiež umožňuje zobraziť vyplnenú CYK tabuľku.

Beh výslednej aplikácie som porovnal na rôznych počítačoch a vyhodnotil som časové závislosti od vstupov programu a vplyv rošírenia na výkonnosť programu. V budúcnosti by bolo možné vytvoriť grafické GUI pre aplikáciu a zjednodušiť tak zadávanie vstupov. Ďalej by bolo možné paralelizovať rozšírenie, ktoré je vykonávané len sekvenčne.

Literatúra

[1] *C++ reference*. [Online; navštíveno 3.3.2018].
URL https://en.cppreference.com/w/

[2] Lange, M.; Leiß, H.: *To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm*. Ludwig-Maximilians-Universität München, Germany, [Online; navštíveno 1.03.2019].
URL https://www.informaticadidactica.de/index.php?page=LangeLeiss2009

[3] Meduna, A.: *Automata and Languages Theory and Applications*. Springer, 2000, ISBN 978-1-85233-074-3.

[4] Meduna, A.: *Elements of Compiler Design*. Auerbach Publications, 2008, ISBN 978-1-4200-6323-3.

[5] Meduna, A.: *Formal Languages and Computation Models and Their Applications*. CRC Press, 2014, ISBN 978-1-4665-1345-7.

[6] Meduna, A.; Lukáš, R.: Formálne jazyky a prekladače. [Online; navštíveno 18.3.2018].
URL https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FIFJ-IT%2Flectures&cid=12153

[7] Meduna, A.; Lukáš, R.: Formálne jazyky a prekladače IFJ Studijní opora. [Online; navštíveno 18.3.2018].
URL https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FIFJ-IT%2Ftexts%2FOporaIFJ.pdf&cid=12153

[8] Wikipedia: *CYK algorithm*. [Online; navštíveno 1.3.2019].
URL https://en.wikipedia.org/wiki/CYK_algorithm