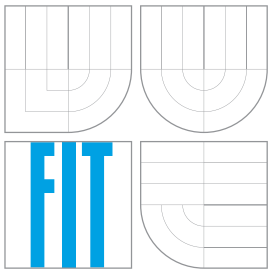
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PARALELNÍ VERZE OBECNÉ SYNTAKTICKÉ ANALÝZY

PARALLEL VERSIONS OF GENERAL PARSING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

TIBOR MIKITA

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. RNDr. ALEXANDER MEDUNA, CSc.

BRNO 2016

Zadání bakalářské práce/18920/2015/xmikit01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Mikita Tibor**
Obor: Informační technologie
Téma: **Paralelní verze obecné syntaktické analýzy**
Parallel Versions of General Parsing
Kategorie: Překladače

Pokyny:

1. Podrobně se seznamte s několika obecnými 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:

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

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Meduna Alexander, prof. RNDr., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato bakalářská práce se zabývá obecními metodami syntaktické analýzy. Autor studuje Cocke-Younger-Kasami algoritmus a přichází s návrhem paralelní verze. Motivací autora je zrychlení syntaktické analýzy založené na tomto algoritmu. Výsledkem práce je konzolová aplikace, která dokáže zjistit, zda vstupní řetězec patří do jazyka generovaného vstupní gramatikou, nebo nepatří, na základě navrženého paralelního algoritmu. Jako programovací jazyk byl zvolen jazyk C++. Pro dosažení paralelismu byla práce algoritmu rozdělena mezi několik vláken.

Abstract

This thesis deals with general parsing methods. The author studies the Cocke-Younger-Kasami algorithm and comes up with a design of a parallel version. The motivation of the author is the acceleration of parsing based on this algorithm. The result is a console application that can determine whether an input string belongs to a language generated by the input grammar or not on the basis of the designed parallel algorithm. For the programming language, C++ was chosen. To achieve parallelism the work of the algorithm was divided among several threads.

Klíčová slova

překladač, formální jazyky, syntaktická analýza, obecné metody, CYK algoritmus, paralelní programování, C++

Keywords

compiler, formal languages, parsing, general methods, CYK algorithm, parallel programming model, C++

Citace

MIKITA, Tibor. *Paralelní verze obecné syntaktické analýzy.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Meduna Alexander.

Paralelní verze obecné syntaktické analýzy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Alexandra Meduny. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tibor Mikita
10. května 2016

Poděkování

Tímto bych chtěl poděkovat panu prof. Medunovi za odborné vedení, cenné rady a připomínky, které mi poskytl při řešení této bakalářské práce.

Obsah

Kapitola 1

Úvod

Téma tejto bakalárskej práce patrí do kategórie formálnych jazykov a prekladačov. V dnešnej dobe sú prekladače neodmysliteľnou súčasťou programovania. Určite každý programátor má svoj obľúbený, alebo preferovaný programovací jazyk. Program, ktorý v tomto jazyku napíše, sa musí preložiť do podoby, ktorej bude počítač rozumieť. Prekladač je nástroj, ktorý číta zdrojový program napísaný v zdrojovom jazyku[1] a prekladá ho do cieľového programu, ktorý je napísaný v cieľovom jazyku[2]. Zdrojový a cieľový program sú vzájomne funkčne ekvivalentné, teda popisujú rovnakú výpočtovú úlohu, ktorá sa má vykonať.[1] Počítač už takto preloženému programu je schopný porozumieť a dokáže určitým spôsobom realizovať príkazy, ktoré od užívateľa, teda programátora, prostredníctvom tohto programu dostal.

Preložiť takýto zdrojový kód nie je triviálnou záležitosťou. Každý prekladač sa skladá z niekoľkých komplexných častí naväzujúcich na seba. Tieto časti nazývame fázy prekladu a každá z nich transformuje zdrojový program z jednej podoby do inej. Spolu ich je šesť a postupne sa vykonávajú v tomto poradí: lexikálna analýza, syntaktická analýza, sémantická analýza, generovanie vnútorného kódu, optimalizácia a generovanie cieľového kódu. Toto rozdelenie predstavuje logickú štruktúru. V praxi sa väčšinou syntaktická analýza spája so sémantickou do jedného celku, ktorý nesie názov syntaxou riadený preklad.[1] Táto práca sa venuje výhradne syntaktickej analýze.

Vstupom syntaktického analyzátora je postupnosť tokenov, ktorá sa vytvorí v predchádzajúcej fáze, teda v lexikálnej analýze. Na základe danej gramatiky sa vstup určitým spôsobom vyhodnotí, prebehne určitý proces a výsledkom je zväčša derivačný strom, z ktorého je neskôr možné vytvoriť vnútorný kód.

Syntaktická analýza má rôzne prístupy a metódy. K týmto metódam sa radia aj tzv. obecné metódy. Výhodou týchto metód je to, že budú fungovať nad celou množinou bezkontextových jazykov, teda dokážu obsiahnuť väčšiu množinu jazykov ako bežne používané syntaktické analyzátory. Avšak majú aj jednu zásadnú nevýhodu, a tou je pomalý prístup. Táto práca sa podrobnejšie venuje a skúma jednu konkrétnu obecnú metódu, ktorá nesie názov Cocke-Younger-Kasami algoritmus, tiež nazývaný CYK algoritmus. Túto metódu vynašli nezávisle traja páni v 70. rokoch 20. storočia, podľa ktorých bola táto metóda pomenovaná: John Cocke, Daniel Younger a Tadao Kasami.[2]

Cieľom tejto práce je navrhnúť a implementovať paralelnú verziu CYK algoritmu. Keďže sú obecné metódy vo všeobecnosti veľmi pomalé, použitie paralelného prístupu môže viesť k určitému zrýchleniu tejto syntaktickej analýzy, čo je aj hlavnou motiváciou autora. Vstu-

[1] Bežne sa tým myslí vyšší programovací jazyk ako je C, C++, Java a ďalšie.
[2] Typicky ide o strojový kód, prípadne jazyk symbolických inštrukcií.

pom naprogramovaného syntaktického analyzátora je reťazec tokenov a gramatika. Výstupom ale nie je derivačný strom ako zvyčajne. Ten je totiž hlavnou esenciou nasledujúcej fázy, a keďže sa táto práca zameriava len na syntaktickú analýzu, výstupom bude len jednoduchá odpoveď: áno alebo nie. Je vstupný reťazec syntakticky správne napísaný alebo nie je? Patrí vstupný reťazec do jazyka generovaného vstupnou gramatikou alebo nepatrí?

Kapitola 2 obsahuje základné definície a znalosti z oblasti formálnych jazykov a prekladačov, ktoré je nutné vedieť pre pochopenie problematiky, o ktorej sa v tejto práci píše. V kapitole 3 si predstavíme dva základné modely syntaktickej analýzy: bezkontextová gramatika a zásobníkový automat. Kapitola 4 sa venuje dvom primárnym normálnym formám gramatík: Chomského normálnej forme a Greibachovej normálnej forme. V kapitole 5 sa píše o metódach syntaktickej analýzy, ktoré sa bežne v praxi používajú. Taktiež sú tu definované špeciálne typy gramatík, ktoré sú pri programovaní syntaktických analyzátorov zvyčajne využívané: LL a LR gramatiky. Kapitola 6 popisuje obecné metódy syntaktickej analýzy a prezentuje niektoré existujúce algoritmy. V kapitole 7 je predstavený návrh PCYK algoritmu. Jedná sa o paralelnú verziu syntaktickej analýzy založenej na CYK algoritme. Popis implementácie navrhnutého algoritmu sa nachádza v kapitole 8. Kapitola 9 rozoberá testovanie aplikácie, dosiahnuté výsledky a porovnávanie algoritmov CYK a PCYK.

Kapitola 2

Zoznámenie sa s formálnymi jazykmi a prekladačmi

Predtým než sa začneme venovať samotnému jadru práce, je nevyhnutné vysvetliť si základné pojmy, ktoré sa budú v práci často vyskytovať. V tejto kapitole si definujeme potrebnú terminológiu z oblasti formálnych jazykov a prekladačov. Je nutné podotknúť, že sa predpokladajú základné znalosti z matematickej logiky, teórie množín, diskrétnej matematiky a teórie grafov. Všetky definície v podkapitole 2.1 sú prevzaté z [3][4]. Podkapitola 2.2 je prevzatá z [1].

2.1 Formálne jazyky

Jedným zo základných konceptov teórie formálnych jazykov je abeceda.

Definícia 2.1. *Abeceda* Σ je konečná, neprázdna množina elementov, ktoré nazývame *symboly*.

Konečnú postupnosť symbolov vybraných z abecedy Σ budeme nazývať reťazec nad abecedou Σ.

Definícia 2.2. Nech Σ je abeceda.

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

Symbol ε predstavuje tzv. *prázdny reťazec*, teda taký reťazec, ktorý neobsahuje žiadny symbol. Zápisom Σ^* budeme značiť množinu všetkých reťazcov nad abecedou Σ. Ak do tejto množiny nezaradíme prázdny reťazec, použijeme označenie Σ^+. Platí, že $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

Ak budeme hovoriť o dĺžke reťazca, budeme tým myslieť celkový počet symbolov v danom reťazci.

Definícia 2.3. Nech x je reťazec nad abecedou Σ. *Dĺžka* reťazca x, $|x|$, je definovaná:

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

Poznáme niekoľko operácií nad reťazcami. Asi jednou z najdôležitejších je konkatenácia, inak povedané zreťazenie.

Definícia 2.4. Nech x a y sú dva reťazce nad abecedou Σ. *Konkatenácia* x a y je reťazec xy.

Pre každý reťazec $x \in \Sigma^*$ platí $x\varepsilon = \varepsilon x = x$.
Nad reťazcami vieme uskutočniť aj unárne operácie. Jednou z nich je aj mocnina reťazca.

Definícia 2.5. Nech x je reťazec nad abecedou Σ. Pre $i \geq 0$, *i-tá mocnina* reťazca x, x^i, je rekurzívne definovaná:

1. $x^0 = \varepsilon$
2. $x^i = xx^{i-1}$, kde $i \geq 1$

Ak by sme potrebovali z určitého reťazca dostať reťazec napísaný v opačnom poradí, môžeme použiť operáciu reverzácie.

Definícia 2.6. Nech x je reťazec nad abecedou Σ. *Reverzácia* reťazca x, $reversal(x)$, je definovaná:

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

V súvislosti s reťazcom budeme často spomínať pojem podreťazec.

Definícia 2.7. Nech x a y sú dva reťazce nad abecedou Σ. Reťazec x je *podreťazcom* reťazca y, pokiaľ existujú reťazce z, z' nad abecedou Σ, pričom platí $zxz' = y$. Keď $x \notin \{\varepsilon, y\}$, potom x je *vlastným podreťazcom* reťazca y.

V tejto chvíli už máme všetky potrebné znalosti pre definovanie formálneho jazyka.

Definícia 2.8. Každá podmnožina $L \subseteq \Sigma^*$ je *formálny jazyk* (alebo len *jazyk*) nad abecedou Σ.

Odteraz budeme namiesto slovného spojenia formálny jazyk používať skrátene jazyk. Ak sa budeme baviť o jazyku, budeme tým myslieť formálny jazyk.

2.2 Prekladače

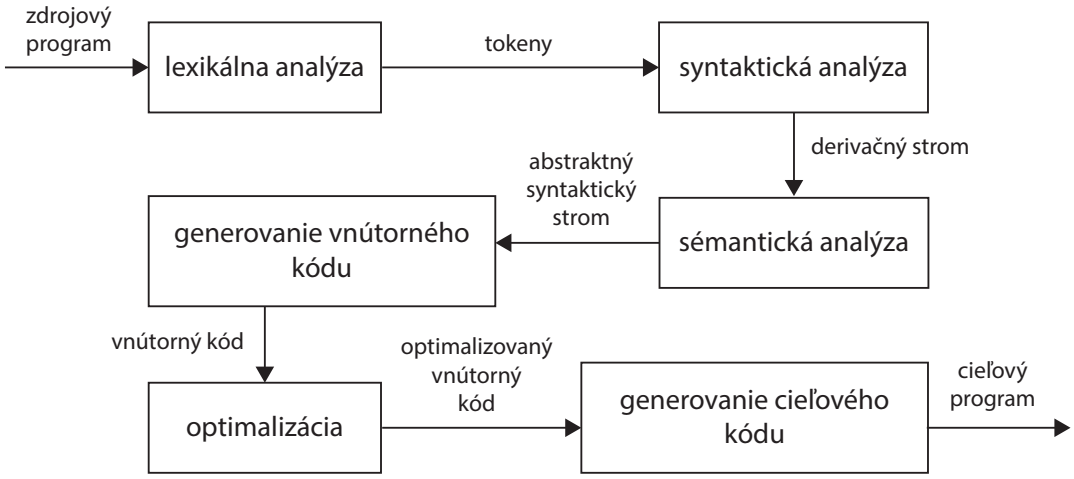
Teória formálnych jazykov má obrovské využitie nie len v lingvistike, ale aj v teoretickej informatike a v poslednej dobe dokonca aj v molekulárnej biológií. Táto teória obsahuje množstvo masívnych zjednodušení a abstrakcií a snaží sa tak priblížiť čo najviac prirodzenému jazyku.[5] My ju budeme využívať ako aparát pre písanie prekladačov.

Prekladač je metaprogram. Inak povedané, je to program, ktorý spracováva iný program.[6] Na vstup dostane zdrojový program napísaný v zdrojovom jazyku a preloží ho do cieľového programu, ktorý je napísaný v cieľovom jazyku. Tento proces sa dá rozdeliť na niekoľko častí, ktoré nazývame *fázy prekladu*:

- lexikálna analýza
- syntaktická analýza
- sémantická analýza

- generovanie vnútorného kódu
- optimalizácia
- generovanie cieľového kódu

Štruktúru prekladača a spôsob, akým sú jednotlivé komponenty pospájané, je možné vidieť na obrázku 2.1.



Obr. 2.1: Schéma jednotlivých komponentov prekladača

Na začiatku *lexikálna analýza* rozdelí zdrojový program na *lexémy*. Lexémy sú logicky oddelené lexikálne jednotky. Medzi lexémy patria napr. identifikátory, čísla, kľúčové slová, operátory, atď. Tieto lexémy prekladač reprezentuje pomocou *tokenov*. Token sa skladá z dvoch zložiek: typ a atribúty.

Úlohou *syntaktickej analýzy* je zistiť, či postupnosť tokenov, ktorú dostane od lexikálneho analyzátora, reprezentuje syntakticky správne napísaný program. Ako to robí? Syntax zdrojového jazyka je špecifikovaná *gramatickými pravidlami*. Podľa nich sa syntaktický analyzátor snaží odvodiť postupnosť tokenov, ktorú dostal na vstupe. Toto odvodzovanie nesie názov *derivácia*. Výsledkom je akýsi rozbor, čo je vlastne postupnosť použitých pravidiel. Graficky sa tento rozbor znázorňuje ako *derivačný strom*. Derivačný strom zodpovedá použitým pravidlám. Listy tohto stromu predstavujú jednotlivé tokeny. Každá dvojica rodičovský uzol-potomkovia v tomto strome reprezentuje určité pravidlo. Syntaktický analyzátor rad-radom vyberá vhodné pravidlá a postupne tak formuje derivačný strom.

Podľa toho, akým spôsobom dochádza ku konštrukcii stromu, rozlišujeme dva základné typy syntaktickej analýzy. Syntaktická analýza *zhora-nadol* vytvára derivačný strom od koreňa a postupuje smerom nadol až k listom. Naopak syntaktická analýza *zdola-nahor* vytvára derivačný strom od listov a postupuje smerom nahor až ku koreňu. Ak sa v tejto fáze podarí vyprodukovať derivačný strom, môžeme povedať, že zdrojový program je napísaný syntakticky správne. V opačnom prípade, ak derivačný strom nie je možné vytvoriť, zdrojový program je napísaný syntakticky nesprávne.

Vstupom *sémantického analyzátora* je derivačný strom. V tejto fáze sa kontrolujú sémantické aspekty programu. Možno najdôležitejšou kontrolou je kontrola typov, pri ktorej môže dochádzať k implicitnej konverzii typov tak, aby boli dotknuté operandy kompatibilné. Ak to nie je možné, dochádza k chybe. Ďalším príkladom sémantickej kontroly je

kontrola deklarácií premenných. Ako už bolo v úvode spomenuté, v praxi sú väčšinou syntaktická a sémantická analýza prepojené a tvoria jeden celok s názvom *syntaxou riadený preklad*. Výstupom je *abstraktný syntaktický strom*.

Nasleduje *generovanie vnútorného kódu*. Prečo je to vôbec potrebné? Výsledný vnútorný kód je uniformný, jednotný a nie je závislý na konkrétnom strojovom kóde. Ľahko sa s ním pracuje a hlavne optimalizuje. Priamy preklad z abstraktného syntaktického stromu do cieľového kódu môže byť zložitý a neprehľadný. Ako vnútorný kód sa zvyčajne používa *trojadresný kód*.

Optimalizátor upravuje vnútorný kód do efektívnejšej podoby. Výsledkom je *optimalizovaný vnútorný kód*. Táto fáza dokáže celý proces podstatne spomaliť, pretože dochádza k veľmi zložitým výpočtom. Preto prekladače disponujú možnosťou vypnúť optimalizácie a tak urýchliť celkový beh prekladu. Niektoré prekladače dokonca optimalizátor nemajú.

Ostáva *generovanie cieľového kódu*, čo už nie je také náročné. Dochádza tu k prevodu vnútorného kódu na postupnosť symbolických inštrukcií, ktoré vykonávajú činnosť popísanú zdrojovým programom. Výstupom je teda cieľový program napísaný v cieľovom jazyku. V praxi je to väčšinou strojový kód, prípadne jazyk symbolických inštrukcií.

Kapitola 3

Formálne modely syntaktickej analýzy

Tak ako už bolo napísané, táto práca sa zameriava na syntaktickú analýzu. Pripomeňme si, že úlohou syntaktickej analýzy je zistiť, či postupnosť tokenov získaná z lexikálnej analýzy predstavuje syntakticky správne napísaný zdrojový program. Zdrojový program je napísaný v zdrojovom jazyku. Väčšinou sa jedná o programovací jazyk. Každý programovací jazyk má presne a jasne definovanú syntax[1]. Napr. program napísaný v jazyku C sa skladá z funkcií, funkcia z deklarácií a príkazov, príkaz zase z výrazov a podobne. Syntax môžeme vyjadriť buď graficky pomocou syntaktických diagramov, ďalej pomocou BNF (Backus-Naur Form), alebo aj prostredníctvom gramatiky. Definície v tejto kapitole sú prevzaté z [4].

3.1 Bezkontextová gramatika

Gramatika má oproti ostatným metódam zápisu syntaxe obrovské výhody pre písanie prekladača. Gramatika vie veľmi precízne zadefinovať syntaktickú špecifikáciu programovacieho jazyka a pritom je to jednoduché, prehľadné a dá sa to ľahko pochopiť. Ďalšou výhodou je to, že pri písaní prekladača sa s gramatikou pohodlne pracuje a ak by sme do syntaxe jazyka chceli pridať nové konštrukcie, nie je s rozšírením gramatiky absolútne žiadny problém, a je to pomerne triviálna záležitosť. Gramatikou v tomto kontexte myslíme bezkontextovú gramatiku.[2]

Definícia 3.1. *Bezkontextová gramatika* (BKG) je štvorica $G = (N, T, P, S)$, kde:

- N je abeceda *neterminálov*
- T je abeceda *terminálov*, pričom $N \cap T = \varnothing$
- P je konečná množina *pravidiel* tvaru $A \to x$, kde $A \in N$, $x \in (N \cup T)^*$
- $S \in N$ je *počiatočný neterminál*

V syntaktickej analýze sú tokeny reprezentované terminálmi. Názov terminál je odvodený od anglického slova *terminate*, čo v preklade znamená ukončiť[1]. Derivácia končí, ak sa

[1] Syntax programovacieho jazyka je súbor pravidiel, ktoré popisujú formálnu štruktúru programu. Definuje identifikátory, čísla, výrazy, kľúčové slová, atď. Pomocou syntaxe dokážeme zistiť, či je program gramaticky správne napísaný.[7]

od počiatočného neterminálu dostaneme k reťazcu, ktorý bude pozostávať len z terminálov. Ak použijeme pravidlo $A \to x$, znamená to, že neterminál A má byť prepísaný na reťazec x. Na ľavej strane pravidla môže figurovať vždy len jeden neterminál. Z toho vyplýva, že terminál nemožno prepísať. Na pravej strane pravidla môžu byť terminály aj neterminály. Pravidlo v tvare $A \to \varepsilon$ nazývame *ε-pravidlo*. Ľavú stranu pravidla p budeme označovať ako $lhs(p)$ a pravú stranu ako $rhs(p)$.[3]

Neformálne povedané, v syntaktickej analýze dochádza k výberu vhodného pravidla, podľa ktorého sa v prípade prístupu zhora-nadol zmení jeden neterminál na reťazec terminálov a neterminálov, a v prípade prístupu zdola-nahor dochádza k tzv. *redukcii*, teda reťazec terminálov a neterminálov zameníme za jeden neterminál.

3.1.1 Derivácia

V gramatike je možné odvodzovať, tj. derivovať reťazce. Ak máme určitý reťazec terminálov a neterminálov, môžeme z neho odvodiť nový reťazec, ktorý vznikne nahradením jedného neterminálu z pôvodného reťazca za ľubovoľný reťazec terminálov a neterminálov. Formálne to môžeme definovať pomocou derivačného kroku a sekvencie derivačných krokov.

Definícia 3.2. Nech $G = (N, T, P, S)$ je BKG. Nech $u, v \in (N \cup T)^*$ a $p = A \to x \in P$. Potom, *uAv priamo derivuje uxv* za použitia p v G, zapisujeme:

$$uAv \Rightarrow uxv\ [p] \text{ alebo zjednodušene } uAv \Rightarrow uxv$$

Ak $uAv \Rightarrow uxv$ v G, môžeme povedať, že G vykonáva *derivačný krok* z uAv do uxv.

Definícia 3.3. Nech $u \in (N \cup T)^*$. G vykoná *nula derivačných krokov* z u do u, zapisujeme:

$$u \Rightarrow^0 u\ [\varepsilon] \text{ alebo zjednodušene } u \Rightarrow^0 u$$

Definícia 3.4. Nech $u_0, \ldots, u_n \in (N \cup T)^*$, $n \geq 1$ a $u_{i-1} \Rightarrow u_i\ [p_i]$, $p_i \in P$ pre všetky $i = 1, \ldots, n$, čo znamená:

$$u_0 \Rightarrow u_1\ [p_1]\ \Rightarrow u_2\ [p_2]\ \Rightarrow \cdots \Rightarrow u_n\ [p_n]$$

Potom G vykoná *n derivačných krokov* z u_0 do u_n, zapisujeme:

$$u_0 \Rightarrow^n u_n\ [p_1 \ldots p_n] \text{ alebo zjednodušene } u_0 \Rightarrow^n$$

Keď $u_0 \Rightarrow^n u_n\ [\pi]$ pre nejaké $n \geq 1$, potom u_0 *derivuje* u_n v G, zapisujeme:

$$u_0 \Rightarrow^+ u_n\ [\pi]$$

Keď $u_0 \Rightarrow^n u_n\ [\pi]$ pre nejaké $n \geq 0$, potom u_0 *derivuje* u_n v G, zapisujeme:

$$u_0 \Rightarrow^* u_n\ [\pi]$$

Ak to prenesieme do problému syntaktickej analýzy, funguje to nasledovne. Na začiatku je počiatočný neterminál, ktorý máme uvedený v samotnej definícií použitej gramatiky. Ten nahradíme za reťazec terminálov a neterminálov za použitia pravidla, kde na jeho ľavej strane sa nachádza práve spomínaný neterminál a na pravej strane je ten reťazec terminálov a neterminálov, ktorý nahradí počiatočný neterminál. Potom sa vyberie opäť

nejaký neterminál a hľadá sa pravidlo, ktoré by bolo možné uplatniť. Toto sa deje, kým nám neostane reťazec zložený len z terminálov.

Tu nastáva problém čisto z praktického hľadiska. Povedali sme, že sa počas derivovania vyberie nejaký neterminál a ten sa nahradí. Čo znamená nejaký? Je to nejednoznačné a nedeterministické, čo je v praxi obrovský problém. Preto sa zaviedli pojmy najľavejšia a najpravejšia derivácia, ktoré tento problém jednoducho vyriešia.

Definícia 3.5. Nech $G = (N, T, P, S)$ je BKG. Majme $u \in T^*$, $v \in (N \cup T)^*$ a pravidlo $p : A \to x \in P$. Potom uAv priamo derivuje uxv pomocou *najľavejšej derivácie* použitím pravidla p v G, zapisujeme:

$$uAv \Rightarrow_{lm} uxv \ [p]$$

Definícia 3.6. Nech $G = (N, T, P, S)$ je BKG. Majme $u \in (N \cup T)^*$, $v \in T^*$ a pravidlo $p : A \to x \in P$. Potom uAv priamo derivuje uxv pomocou *najpravejšej derivácie* použitím pravidla p v G, zapisujeme:

$$uAv \Rightarrow_{rm} uxv \ [p]$$

Neformálne to môžeme popísať tak, že pri najľavejšej derivácií sa prepíše vždy najľavejší neterminál a pri najpravejšej derivácií sa prepíše vždy najpravejší neterminál.

S deriváciou súvisia pojmy: užitočný symbol, neužitočný symbol, vetná forma, veta a generovaný jazyk.

Definícia 3.7. Nech $G = (N, T, P, S)$ je BKG. Symbol X je *užitočný*, keď $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$, kde $\alpha, \beta \in (N \cup T)^*$ pre nejaké $w \in T^*$, inak je symbol X *neužitočný*.

Definícia 3.8. Nech $G = (N, T, P, S)$ je BKG. Keď $S \Rightarrow^* x$, kde $x \in (N \cup T)^*$, potom x je *vetná forma* derivovaná gramatikou G. Keď navyše platí, že $x \in T^*$, x je *veta* derivovaná gramatikou G.

Definícia 3.9. Nech $G = (N, T, P, S)$ je BKG. *Jazyk generovaný* BKG G, $L(G)$, je definovaný:

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

Jazyk generovaný gramatikou G je množina všetkých viet derivovaných gramatikou G. Ak x je vstupný reťazec syntaktickej analýzy a G je vstupná gramatika, potom cieľom syntaktickej analýzy je zistiť, či platí tvrdenie $x \in L(G)$.

Ďalším problémom v praxi je nejednoznačnosť gramatiky. Môže nastať taká situácia, že syntaktický analyzátor dokáže pre jeden vstupný reťazec vytvoriť niekoľko rôznych derivačných stromov. Každý z týchto stromov bude korektný. Táto situácia je možno v teórii vítaná, ale v praxi to predstavuje zbytočné starosti.

Definícia 3.10. Nech $G = (N, T, P, S)$ je BKG. Keď existuje reťazec $x \in L(G)$ s viac ako jedným derivačným stromom, potom G je *nejednoznačná*. Inak G je *jednoznačná*.

3.2 Zásobníkový automat

Formálnym modelom syntaktickej analýzy je aj zásobníkový automat. K jeho definícií je dôležité poznať pojem konečný automat, ktorý sa využíva v lexikálnej analýze.

Definícia 3.11. *Konečný automat* (KA) je pätica $M = (Q, \Sigma, R, S, F)$, kde:

- Q je konečná množina *stavov*
- Σ je *vstupná abeceda*
- R je konečná množina *pravidiel* tvaru $pa \to q$, kde $p, q \in Q$, $a \in (\Sigma \cup \varepsilon)$
- $S \in Q$ je *počiatočný stav*
- $F \subseteq Q$ je množina *koncových stavov*

Zmyslom konečného automatu je mať jednoduchý model založený na konečnej množine stavov a výpočtových pravidiel, ktorý dokáže pracovať s nekonečnom. Fungovanie konečného automatu si vysvetlíme neformálnym popisom. Máme vstupnú pásku, ktorá pozostáva zo vstupnej abecedy. Dĺžka vstupnej pásky môže byť teoreticky nekonečná. Konečný automat postupne číta vstupnú pásku zľava doprava. Podľa toho, v akom stave sa aktuálne konečný automat nachádza a podľa toho, čo zo vstupnej pásky práve prečítal, sa rozhodne, do akého stavu pôjde. Pravidlo v tvare $pa \to q$ znamená, že ak sa konečný automat nachádza v stave p a čítacia hlava ukazuje na vstupnej páske na symbol a, môže konečný automat prejsť do stavu q s tým, že sa čítacia hlava posunie doprava. Jednoduchšie povedané, ak sa použije pravidlo v tvare $pa \to q$, konečný automat prečíta zo vstupu symbol a a prejde zo stavu p do stavu q. Môže nastať špeciálna situácia, keď $a = \varepsilon$. Potom pri prechode zo stavu p do stavu q nie je prečítaný zo vstupnej pásky žiadny symbol.

Zásobníkový automat je v podstate konečný automat rozšírený o potenciálne nekonečne veľký zásobník.

Definícia 3.12. *Zásobníkový automat* (ZA) 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* tvaru $Apa \to wq$, 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*

Rozdiel medzi zásobníkovým automatom a konečným automatom je ten, že zásobníkový automat navyše pri prechode nahrádza symbol na vrchole zásobníka za reťazec symbolov. Pravidlo v tvare $Apa \to wq$ znamená, že ak je aktuálny stav p, aktuálny symbol na vstupnej páske je a a symbol na vrchole zásobníka je A, potom zásobníkový automat môže prečítať symbol a zo vstupnej pásky, nahradiť symbol A za reťazec symbolov w na zásobníku a prejsť zo stavu p do stavu q.

Bezkontextová gramatika a zásobníkový automat sú ekvivalentné modely syntaktickej analýzy. Je to fakt, ktorý je dokázaný a existuje algoritmus na prevod bezkontextovej gramatiky na ekvivalentný zásobníkový automat. Jazyk generovaný touto gramatikou a jazyk prijímaný týmto automatom sú rovnaké. Pri písaní syntaktického analyzátora sa využívajú jak bezkontextová gramatika, tak zásobníkový automat.[3]

Kapitola 4

Normálne formy gramatiky

Pravidlá gramatiky môžu mať rôznu formu, teda podobu. Bezkontextová gramatika dovoľuje pravidlám mať na ľavej strane práve jeden neterminál. Avšak na pravej strane môžu byť terminály aj neterminály a ich rôzne kombinácie. Tých možností je naozaj veľa. Ukazuje sa, že je výhodné mať pravidlá v určitej predpísanej podobe, pretože sa s tým lepšie pracuje. Existuje niekoľko normálnych foriem, ktoré predpisujú tvar pravidiel gramatiky. Prečo je to užitočné? Prečo sa s tým lepšie pracuje? Pravdou je, že to má zmysel v teórii, ale aj v praxi. Predpísaný tvar pravidiel uľahčuje rôzne matematické dokazovania. Môžeme sa jednoducho spoliehať na to, že tvar pravidiel bude stále rovnaký, nikdy nie iný. V praxi sa normálne formy využívajú v niektorých metódach syntaktickej analýzy, ktoré to priam vyžadujú, inak nebudú fungovať správne. Existuje niekoľko normálnych foriem gramatiky. Medzi dve najpoužívanejšie patria Chomského normálna forma a Greibachovej normálna forma, ktoré si v tejto kapitole bližšie predstavíme. Definície v tejto kapitole sú prevzaté z [4].

4.1 Chomského normálna forma

Chomského normálna forma je pomenovaná po svojom autorovi, **Noamovi Chomskom**[8]. Pre gramatiku v tejto forme platí, že každé jej pravidlo má na pravej strane buď dva neterminály alebo jeden terminál. Žiadne ďalšie tvary nie sú povolené.

Definícia 4.1. Nech $G = (N, T, P, S)$ je BKG. Gramatika G je v *Chomského normálnej forme* (CNF) vtedy a len vtedy, ak každé pravidlo z P má jeden z dvoch tvarov:

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

Pomocou gramatiky v takejto forme nie je možné generovať prázdny reťazec. To nám v praxi až tak nevadí. Prostredníctvom pravidiel v tejto forme sa každou ďalšou deriváciou odvodí reťazec rovnako dlhý alebo dlhší ako reťazec prechádzajúci. Ďalej platí, že derivačný strom, ktorý vznikne na základe gramatiky v Chomského normálnej forme, je binárny strom a jeho výška sa rovná dĺžke generovaného reťazca.

Dôležitým poznatkom je fakt, že každá gramatika v Chomského normálnej forme je bezkontextová, a takisto pre každú bezkontextovú gramatiku existuje gramatika v Chomského normálnej forme[3]. Prevod bezkontextovej gramatiky do gramatiky v Chomského normálnej forme je možné vykonať podľa algoritmu 4.1[1].

[1] Algoritmus 4.1 je prevzatý z [3].

Algoritmus 4.1: Prevod BKG do gramatiky v CNF

Vstup: BKG $G = (N, T, P, S)$
Výstup: BKG $G' = (N', T, P', S)$ v CNF taká, že $L(G) = L(G')$

begin
  majme $W = \{a'$ pre všetky $a \in T\}$ a bijektívne zobrazenie $\beta : (N \cup T) \longrightarrow (W \cup N)$ definované ako $\beta(a) = a'$ pre všetky $a \in T$ a $\beta(A) = A$ pre všetky $A \in N$;
  inicializuj abecedu neterminálov $N' = (N \cup W)$ a množinu pravidiel $P' = \varnothing$;
  foreach a **in** T **do**
    pridaj $a' \to a$ do P';
  foreach $A \to a$ **in** P, *kde* $A \in N$, $a \in T$ **do**
    presuň $A \to a$ z P do P';
  foreach $A \to X_1X_2$ **in** P, *kde* $A \in N$ *a* $X_i \in (N \cup T)$ *pre* $i = 1, 2$ **do**
    pridaj $A \to \beta(X_1)\beta(X_2)$ do P';
    odstráň $A \to X_1X_2$ z P;
  repeat
    if $A \to X_1X_2X_3 \dots X_{n-1}X_n \in P$, *kde* $A \in N$, $X_i \in (N \cup T)$, $i = 1, \dots, n$ *pre nejaké* $n \geq 3$ **then**
      pridaj nové neterminály $< X_2 \dots X_n >, < X_3 \dots X_n >, \dots, < X_{n-2}X_{n-1}X_n >, < X_{n-1}X_n >$ do N';
      pridaj $A \to \beta(X_1) < X_2 \dots X_n >, < X_2 \dots X_n > \to \beta(X_2) < X_3 \dots X_n >, \dots, < X_{n-2}X_{n-1}X_n > \to \beta(X_{n-2}) < X_{n-1}X_n >, < X_{n-1}X_n > \to \beta(X_{n-1})\beta(X_n)$ do P';
      odstráň $A \to X_1X_2X_3 \dots X_{n-1}X_n$ z P;
  until *žiadna zmena*;
  odstráň všetky neužitočné symboly;

4.2 Greibachovej normálna forma

Okrem Chomského normálnej formy, niektoré metódy syntaktickej analýzy využívajú Greibachovej normálnu formu, ktorá je pomenovaná taktiež podľa jej autorky, **Sheily Greibachovej**[9]. Táto normálna forma predpisuje tvar pravidiel tak, aby na pravej strane bol jeden terminál nasledovaný reťazcom s ľubovoľným počtom neterminálov. Takže jeden terminál tam musí byť stále a to na prvej pozícií, a ďalšie symboly môžu byť už len neterminály, prípadne tam nemusia byť žiadne ďalšie symboly.

Definícia 4.2. Nech $G = (N, T, P, S)$ je BKG. Gramatika G je v *Greibachovej normálnej forme* (GNF) vtedy a len vtedy, ak každé pravidlo z P má nasledujúci tvar:

$$A \to ax, \text{ kde } A \in N,\ a \in T,\ x \in N^*$$

Gramatika v takejto normálnej forme nebude obsahovať žiadnu ľavú rekurziu.

Definícia 4.3. Nech $G = (N, T, P, S)$ je BKG. Pravidlo z P, ktoré má tvar $A \to Ax$, kde $A \in N$, $x \in (N \cup T)^*$ sa nazýva *ľavo rekurzívne pravidlo.*

Niektoré syntaktické analyzátory fungujú tak, že skúšajú postupne všetky pravidlá. Počet pravidiel, ktoré v daný okamžik vyhovujú analýze a je možné ich použiť, môže byť väčší. Potom sa musí vybrať len jedno pravidlo a zväčša je to prvé v tom zozname. Ak by to prvé vybrané pravidlo bolo ľavo rekurzívne, nastal by problém. Opäť by sa hľadalo pravidlo, ktoré by bolo možné použiť a získal by sa rovnaký zoznam ako predtým. Znova by sa vybralo to isté pravidlo a tak by to šlo dookola. Algoritmus by nikdy nemohol skončiť. Preto je ľavá rekurzia nežiadúcim prvkom gramatiky a preto je Greibachovej normálna forma tak užitočná. Rovnako ako to platí pre Chomského normálnu formu, ľubovoľnú bezkontextovú gramatiku je možné previesť do gramatiky v Greibachovej normálnej forme[3].

Kapitola 5

Bežne používané metódy syntaktickej analýzy

Existujú rôzne prístupy a metódy syntaktickej analýzy. Každá určitým spôsobom konštruuje derivačný strom. Podľa toho, akým spôsobom to robí, rozlišujeme dva základne typy syntaktickej analýzy: zhora-nadol a zdola-nahor.

Prístupom zhora-nadol sa konštrukcia derivačného stromu začína pri koreni, ktorý je označený počiatočným neterminálom. Z uzlov, ktoré sú označené neterminálmi, sa podľa pravidiel vytvárajú nové uzly. Ak nastane stav, že všetky doposiaľ neexpandované uzly sú označené len terminálmi, analýza končí. Na syntaktickú analýzu zhora-nadol sa môžeme pozerať ako na hľadanie najľavejšej derivácie pre vstupný reťazec. Do tejto skupiny metód patria napr. *rekurzívny zostup* a *nerekurzívna prediktívna syntaktická analýza.*[2]

Ak pri tvorbe syntaktického analyzátora použijeme prístup zdola-nahor, znamená to, že budeme vytvárať derivačný strom počínajúc listami, ktoré predstavujú terminály a postupne sa dopracujeme na vrchol stromu, teda ku koreňu, čo zasa predstavuje počiatočný neterminál. O tomto prístupe môžeme hovoriť ako o procese *redukovania* postupnosti terminálov, ktorá predstavuje vstupný reťazec, na počiatočný neterminál. V každom kroku redukcie sa určitá postupnosť symbolov, ktorá korešponduje s pravou stranou nejakého pravidla zamení za neterminál, ktorý sa nachádza na ľavej strane tohto pravidla. Hlavným problémom, ktorým sa táto analýza zaoberá, je nájdenie správneho pravidla, ktoré sa má aplikovať a kedy sa má aplikovať, resp. kedy má dôjsť k redukcii. Medzi tieto metódy patria napr. *shift-reduce syntaktická analýza* a *LR syntaktická analýza.*[2]

Niektoré analyzátory vyžadujú pre správny beh rôzne typy gramatík. Napr. spomenuté metódy s prístupom zhora-nadol dokážu pracovať len nad LL gramatikami. LR syntaktická analýza, ako už z názvu vyplýva, dokáže pracovať len nad LR gramatikami. V nasledujúcich sekciách si zadefinujeme jak LL, tak LR gramatiky a ukážeme si, aký je medzi nimi vzťah.

Definície v podkapitole 5.1 sú prevzaté z [3][10].

5.1 LL gramatika

Predtým, než si zadefinujeme samotnú LL gramatiku, je potrebné si vysvetliť, čo sú to množiny *FIRST*, *EMPTY*, *FOLLOW* a *PREDICT*. Tieto množiny priamo súvisia s gramatikou. Využijeme ich pri definícií LL gramatiky, ale majú využitie aj v samotnej analýze, kde sa podľa niektorých z týchto množín určí, ktoré pravidlo sa má aplikovať na základe vstupného symbolu.

Zápisom *FIRST(x)*, kde x je ľubovoľný reťazec symbolov gramatiky, máme na mysli množinu všetkých terminálov, ktorými môže začínať reťazec derivovateľný z x. Ak $x \Rightarrow^* \varepsilon$, potom ε patrí taktiež do množiny *FIRST(x)*.

Definícia 5.1. Nech $G = (N, T, P, S)$ je BKG. Pre každé $x \in (N \cup T)^*$ je definované *FIRST(x)* ako:

$$FIRST(x) = \{a : a \in T, x \Rightarrow^* ay, y \in (N \cup T)^*\}$$

Ďalšou množinou, ktorú využijeme je *EMPTY(x)*. Tá môže obsahovať len jediný symbol, ktorým je ε a to len vtedy, ak x derivuje ε. Inak ostane táto množina prázdna.

Definícia 5.2. Nech $G = (N, T, P, S)$ je BKG. Pre každé $x \in (N \cup T)^*$ je definované *EMPTY(x)* ako:

1. keď $x \Rightarrow^* \varepsilon$, potom $EMPTY(x) = \{\varepsilon\}$
2. keď $x \not\Rightarrow^* \varepsilon$, potom $EMPTY(x) = \varnothing$

Treťou užitočnou množinou je *FOLLOW(A)*, kde A je neterminál. Ide o množinu všetkých terminálov, ktoré sa vo vetnej forme môžu vyskytovať vpravo od A.

Definícia 5.3. Nech $G = (N, T, P, S)$ je BKG. Pre každé $A \in N$ je definované *FOLLOW(A)* ako:

$$FOLLOW(A) = a : a \in T, S \Rightarrow^* xAay, x, y \in (N \cup T)^* \cup \$: S \Rightarrow^* xA, x \in (N \cup T)^*$$

Dostávame sa k poslednej množine, ktorú získame tak, že využijeme vlastnosti predchádzajúcich troch množín.

Definícia 5.4. Nech $G = (N, T, P, S)$ je BKG. Pre každé $A \to x \in P$ je definované *PREDICT($A \to x$)* ako:

1. keď $EMPTY(x) = \{\varepsilon\}$, potom $PREDICT(A \to x) = FIRST(x) \cup FOLLOW(A)$
2. keď $EMPTY(x) = \varnothing$, potom $PREDICT(A \to x) = FIRST(x)$

S týmito znalosťami definujeme LL gramatiku nasledovne.

Definícia 5.5. Nech $G = (N, T, P, S)$ je BKG. G je *LL gramatika*, keď pre každé $a \in T$ a každé $A \in N$ existuje maximálne jedno pravidlo tvaru $A \to X_1X_2 \ldots X_n \in P$ a platí tvrdenie $a \in PREDICT(A \to X_1X_2 \ldots X_n)$.

Prečo názov LL gramatika? Čo to LL vlastne znamená? Prvé „L“ značí fakt, že pri syntaktickej analýze založenej na takejto gramatike sa vstupný reťazec číta zľava doprava. Druhé „L“ znamená, že sa bude hľadať najľavejšia derivácia.

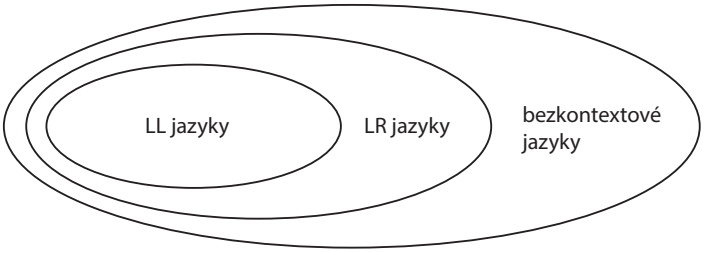
Povedali sme si, že LL gramatika sa využíva v metódach syntaktickej analýzy pracujúcej zhora nadol. Prečo je to tak? Vďaka LL gramatike môžu ľahko zistiť, ktoré pravidlo sa má použiť len na základe aktuálneho vstupného symbolu.

5.2 LR gramatika

LR syntaktická analýza sa radí k tzv. tabuľkovým metódam. Pracujú teda určitým spôsobom s tabuľkou. Postup, akým sa vytvára LR tabuľka si nebudeme uvádzať. Tých spôsobov je viac a môžeme vytvoriť rôzne tabuľky, ktoré majú každá iné vlastnosti. Ak pre nejakú gramatiku ale dokážeme vytvoriť LR tabuľku, potom môžeme túto gramatiku označiť ako *LR gramatiku.*[1]

Názov LR gramatika vznikol rovnakým spôsobom ako LL gramatika. Písmeno „L“ znamená, že vstupný reťazec sa číta zľava doprava. Písmeno „R“ vyjadruje hľadanie najpravejšej derivácie.

Aj keď je to zrejmé, LL gramatiky generujú LL jazyky a LR gramatiky generujú LR jazyky. Ďalej si pripomenieme, že bezkontextový jazyk je taký jazyk, ktorý je generovaný bezkontextovou gramatikou. Dôležitý poznatok je fakt, že množina LL jazykov je podmnožinou množiny LR jazykov a množina LR jazykov je zasa podmnožinou množiny bezkontextových jazykov[2]. Tento vzťah je znázornený na obrázku 5.1.



Obr. 5.1: Vzťah bezkontextových jazykov s LL a LR jazykmi

Je teda jasné, že všeobecne bezkontextové gramatiky sú silnejšie ak LR gramatiky, teda vedia popísať väčšie množstvo jazykov. No a LR gramatiky sú silnejšie ako LL gramatiky.

Kapitola 6

Obecné metódy syntaktickej analýzy

Metódy, ktoré patria do tejto kategórie, sa nazývajú preto *obecnými*, pretože sú aplikovateľné na celú množinu bezkontextových jazykov. Nie sú teda obmedzené len na LL alebo LR gramatiky, ako to je pri bežne používaných syntaktických metódach. Nie všetky obecné metódy budú fungovať nad ľubovoľnou bezkontextovou gramatikou, ale každý bezkontextový jazyk je popísaný minimálne jednou bezkontextovou gramatikou, nad ktorou budú tieto metódy aplikovateľné.[11] Napr. metóda Cocke-Younger-Kasami vyžaduje ako vstup bezkontextovú gramatiku v Chomského normálnej forme. Je ale dokázané, že každú bezkontextovú gramatiku je možné prepísať na ekvivalentnú bezkontextovú gramatiku v Chomského normálnej forme. Tento prevod je popísaný algoritmom 4.1.

Obecné metódy by sa v praxi veľmi používať nemali. Sú často pomalé a zaberajú veľa pamäte. Aj keď dnes už na pamäťovej zložitosti mnohokrát nezáleží, časová zložitosť je veľmi podstatná. V mnohých prípadoch si vystačíme s LL alebo LR gramatikami, pre ktoré existujú oveľa efektívnejšie metódy, ktoré sme si vymenovali v kapitole 5.

Avšak niekedy potrebujeme pracovať s takou bezkontextovou gramatikou, ktorá nepatrí ani do množiny LL gramatík, ani do množiny LR gramatík a na syntaktickú analýzu musíme použiť niektorú z obecných metód. V tejto kapitole si predstavíme niekoľko z nich.

Sekcie 6.1, 6.2, popis fungovania algoritmov v podsekciách 6.2.1 a 6.2.2 sú prevzaté z [11]. Obrázky v podsekcii 6.2.2 sú vytvorené na základe obrázkov z [12].

6.1 Metódy využívajúce spätné vyhľadávanie

Najprv si povieme niečo o metódach, ktoré využívajú spätné vyhľadávanie, tzv. *backtracking*. Dá sa povedať, že algoritmy týchto metód deterministicky simulujú nedeterministické syntaktické analyzátory. Pamäťová zložitosť metód so spätným vyhľadávaním v závislosti od dĺžky vstupného reťazca je lineárna (n). Avšak časová zložitosť je exponenciálna (c^n), čo je veľkou nevýhodou týchto metód a to je hlavný dôvod, prečo by sa tieto metódy nemali používať.

Princíp týchto metód je veľmi naivný až primitívny. Jednoducho povedané, skúšajú sa všetky možné kombinácie pravidiel. Poradie vyberaných pravidiel sa väčšinou upraví tak, aby pravidlá, ktoré majú väčšiu šancu, že budú aplikované, sú uprednostnené pred ostatnými. Ak sa dôjde do stavu, kedy nie je možné ďalej pokračovať, algoritmus sa vráti o niekoľko krokov dozadu tak, aby sa ocitol v stave, kde je iná alternatíva. Potom si vyberie

túto alternatívu a opäť skúša ísť čo najďalej. Niektoré algoritmy sú optimalizované tak, aby neprehľadávali celú vetvu, ak sa o nej už dopredu vie, že nemá správne riešenie. Existujú dve verzie metód využívajúce spätné vyhľadávanie. Jedna pracuje zhora-nadol a druhá zdola-nahor.

6.2 Tabuľkové metódy

Tieto metódy sú špeciálne tým, že nie sú založené na zásobníkovom automate ako väčšina metód syntaktickej analýzy. Ako už z názvu vyplýva, tieto metódy využívajú pre svoju činnosť tabuľky.

V nasledujúcich podkapitolách si predstavíme dve metódy, ktoré do tejto kategórie patria: Earleyho algoritmus a Cocke-Younger-Kasami algoritmus. Obe metódy majú kvadratickú pamäťovú zložitosť (n^2) a kubickú časovú zložitosť (n^3) v závislosti od dĺžky vstupného reťazca. Avšak ak Earleyho algoritmus bude spracovávať jednoznačnú gramatiku, potom bude časová zložitosť kvadratická.

6.2.1 Earleyho algoritmus

Tento algoritmus syntaktickej analýzy vymyslel a v roku 1968 popísal vo svojej dizertačnej práci **Jay Earley**. Sám autor ho označuje ako efektívnejší oproti algoritmom, ktoré využívajú spätné vyhľadávanie. Dokáže pracovať aj s nejednoznačnými gramatikami.[13]

Vstupom je bezkontextová gramatika $G = (N, T, P, S)$ a reťazec $x = a_1 a_2 \dots a_n \in T^*$. Princípom tohto algoritmu je vytváranie akýchsi objektov, ktoré budeme nazývať *položky*, a ich zaraďovanie do zoznamov. Každá položka má tvar $[A \to X_1 X_2 \dots X_k \cdot X_{k+1} \dots X_m, i]$, kde $A \in N$, $X_1 X_2 \dots X_m \in (N \cup T)^*$, pričom musí existovať pravidlo $A \to X_1 \dots X_m \in P$ a musí platiť $0 \leq i \leq n$. Bodka $(\cdot)$ medzi X_k a X_{k+1} je špeciálny symbol a nepatrí ani do N, ani do T. Číslo k môže byť ľubovoľné nezáporné kladné číslo, ale musí platiť $0 \leq k \leq m$. Ak bude $k = 0$, potom špeciálny znak $\cdot$ bude prvý symbol a teda pred ním už nebude žiadny iný symbol. Podobná situácia nastane, ak bude $k = m$, potom bude $\cdot$ posledný symbol a za ním už nič.

Syntaktická analýza prebieha tak, že sa postupne pre každé číslo j, $0 \leq j \leq n$, vytvoria zoznamy položiek I_j také, že položka $[A \to \alpha \cdot \beta, i]$ patrí do I_j, kde $A \in N$, $\alpha, \beta \in (N \cup T)^*$ pre $0 \leq i \leq j$, vtedy a len vtedy, ak pre nejaké $\gamma, \delta \in (N \cup T)^*$ platí $S \Rightarrow^* \gamma A \delta, \gamma \Rightarrow^* a_1 \dots a_i$ a $\alpha \Rightarrow^* a_{i+1} \dots a_j$. Ak v zozname I_j existuje položka $[A \to \alpha \cdot \beta, i]$, znamená to, že z reťazca α môžeme odvodiť časť vstupného reťazca od symbolu a_{i+1} po symbol a_j. Syntaktická analýza je úspešná, teda platí $x \in L(G)$, vtedy a len vtedy, ak v poslednom zozname I_n nájdeme položku $[S \to \alpha \cdot , 0]$.

6.2.2 Cocke-Younger-Kasami algoritmus

Cocke-Younger-Kasami algoritmus, inak nazývaný aj ako CYK alebo CKY algoritmus, je algoritmus obecnej syntaktickej analýzy pomenovaný po jeho troch autoroch: **John Cocke**[14], **Daniel Younger**[15] a **Tadao Kasami**[16]. Dá sa povedať, že táto metóda spadá do kategórie syntaktických metód pracujúcich zdola-nahor. Rovnako ako to platí pre Earleyho metódu, aj táto metóda je zaručene efektívnejšia ako metódy opísané v kapitole 6.1. Najväčšou výhodou CYK algoritmu je jeho jednoduchosť.

Vstupom nie je akákoľvek bezkontextová gramatika, ale musí to byť gramatika v Chomského normálnej forme a má to aj svoje opodstatnenie. Vstupnú gramatiku si označíme ako

$G = (N, T, P, S)$ a vstupný reťazec ako $x = a_1 a_2 \dots a_n \in T^*$.

Podstatou tohto algoritmu je vytvorenie množín neterminálov. Tieto množiny označujeme ako $CYK[i, j]$, kde $1 \leq i \leq j \leq n$. Neterminál A patrí do množiny $CYK[i, j]$ vtedy a len vtedy, ak platí $A \Rightarrow^* a_i \dots a_j$. Z toho vyplýva, že ak sa v množine $CYK[1, n]$ nachádza počiatočný neterminál S, tak platí $S \Rightarrow^* a_1 \dots a_n$ a teda $x \in L(G)$. Na začiatku sú všetky množiny prázdne a postupne sa dopĺňajú. Najprv sa určia množiny $CYK[i, i]$, do ktorých pridáme neterminál A vtedy a len vtedy, ak $A \to a_i \in P$. Potom ak platia tvrdenia, že $B \in CYK[i, j]$, $C \in CYK[j + 1, k]$ a $A \to BC \in P$, pridáme A do $CYK[i, k]$. Algoritmus končí, keď už nie je možné rozšíriť žiadnu ďalšiu množinu. Celý proces je popísaný v algoritme 6.1[1]. Pre lepšie pochopenie toho, ako CYK algoritmus funguje, je uvedený príklad 6.1.

Algoritmus 6.1: CYK algoritmus

Vstup: BKG $G = (N, T, P, S)$ a reťazec $x = a_1 a_2 \dots a_n \in T^*$

Výstup:

- PRIJAŤ, keď $x \in L(G)$
- ODMIETNÚŤ, keď $x \notin L(G)$

```
1  begin
2      inicializácia množín CYK[i, j] = ∅ pre 1 ≤ i ≤ j ≤ n;
3      for i = 1 to n do
4          if A → a_i ∈ P then
5              pridaj A do CYK[i, i];
6      repeat
7          if B ∈ CYK[i, j], C ∈ CYK[j + 1, k], A → BC ∈ P pre nejaké A, B, C ∈ N
             then
8              pridaj A do CYK[i, k];
9      until žiadna zmena;
10     if S ∈ CYK[1, n] then
11         PRIJAŤ;
12     else
13         ODMIETNÚŤ;
```

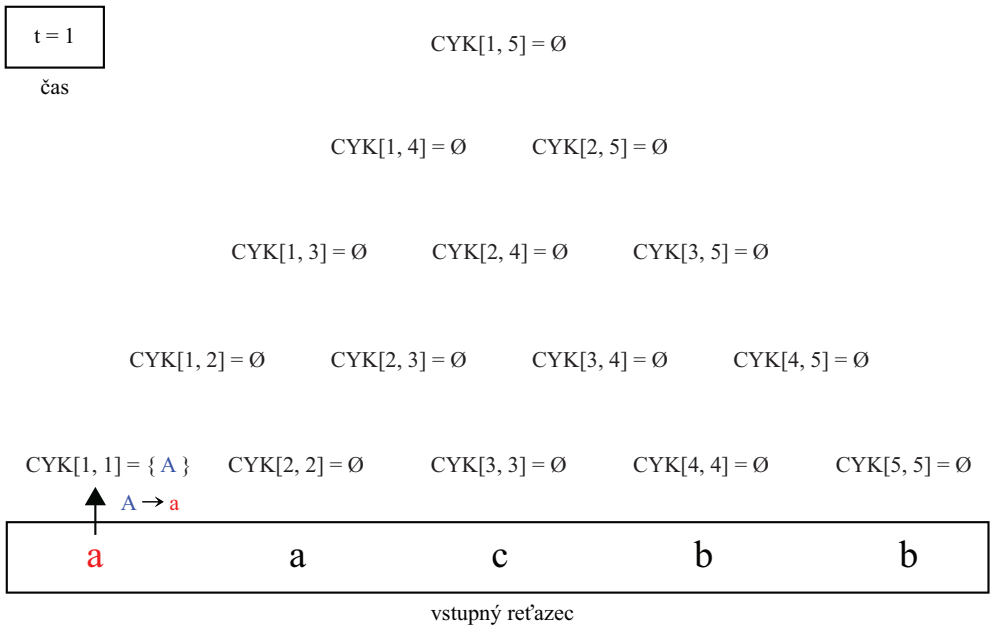
Príklad 6.1. Uvažujme BKG $G = (N, T, P, S)$ v CNF. Nech $N = \{A, B, C, S\}$ a $T = \{a, b\}$. Množina pravidiel P je nasledovná:

$$S \to AC$$
$$C \to SB$$
$$A \to a$$
$$B \to b$$
$$S \to c$$

[1] Algoritmus 6.1 je prevzatý z [1].

Vstupný reťazec je *aacbb*. Otázka znie, či daný vstupný reťazec patrí do jazyka $L(G)$. Použijeme metódu CYK podľa algoritmu 6.1.

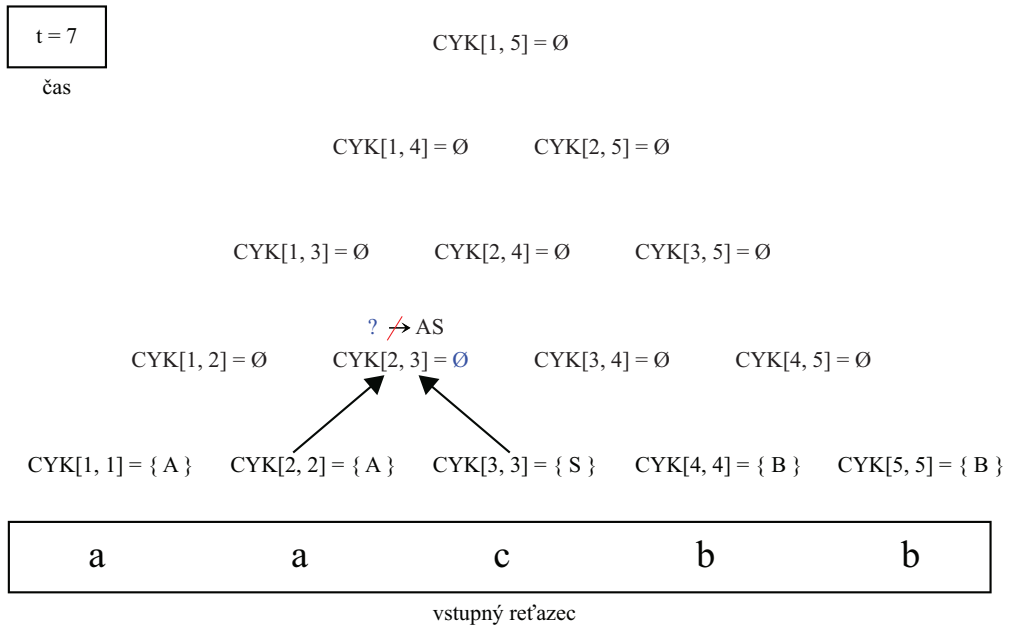
Najprv doplníme množiny v tvare $CYK[i,i]$, kde $1 \leq i \leq 5$. Do množiny $CYK[1,1]$ pridáme neterminál A, pretože existuje pravidlo $A \to a$ a zároveň $a_1 = a$. V čase $t = 1$ bude výpočet množín podľa obrázka 6.1. Ďalej do množiny $CYK[2,2]$ pridáme neterminál A, pretože existuje pravidlo $A \to a$ a zároveň $a_2 = a$. Analogicky budeme postupovať pri množinách $CYK[3,3]$, $CYK[4,4]$ a $CYK[5,5]$.



Obr. 6.1: Výpočet množiny $CYK[1,1]$ v čase $t = 1$ podľa algoritmu CYK

Všetky ďalšie množiny sú už závislé od predchádzajúcich množín. Napríklad množina $CYK[1,2]$ je závislá od dvojice množín $CYK[1,1]$ a $CYK[2,2]$. Prečo práve od tejto dvojice? Musí predsa platiť to, že z niektorého neterminálu z množiny $CYK[1,2]$ môžeme deriváciou dostať časť vstupného reťazca a to od 1. po 2. symbol, keďže cifry, ktorými označujeme túto množinu sú 1 a 2. Zároveň vieme, že $CYK[1,1]$ obsahuje také neterminály, z ktorých môžeme odvodiť 1. symbol vstupného reťazca a $CYK[2,2]$ zahŕňa neterminály, z ktorých zasa deriváciou dostaneme 2. symbol vstupného reťazca. Preto ak zoberieme nejaký neterminál z $CYK[1,1]$ a pridáme k nemu neterminál z $CYK[2,2]$, a existuje pravidlo, na ktorého pravej strane sa nachádzajú práve tieto dva neterminály v poradí zľava doprava, môžeme s určitosťou povedať, že z neterminálu na ľavej strane vybraného pravidla môžeme odvodiť časť vstupného reťazca od 1. po 2. symbol. To zodpovedá definícii množiny $CYK[1,2]$ a preto do tejto množiny neterminál z ľavej strany pravidla pridáme. A teraz konkrétne. Obidve množiny, na ktoré v tomto prípade prihliadame, obsahujú jeden neterminál A. Ak existuje pravidlo $p \in P$ také, že $rhs(p) = AA$, tak do množiny $CYK[1,2]$ pridáme $lhs(p)$. Lenže takéto pravidlo neexistuje, takže množina $CYK[1,2]$ ostane prázdna. Podobne to je aj s množinou $CYK[2,3]$. Keďže neexistuje pravidlo $p \in P$ také, že $rhs(p) = AS$, aj táto množina ostane prázdna. V čase $t = 7$ bude výpočet množín podľa obrázka 6.2.

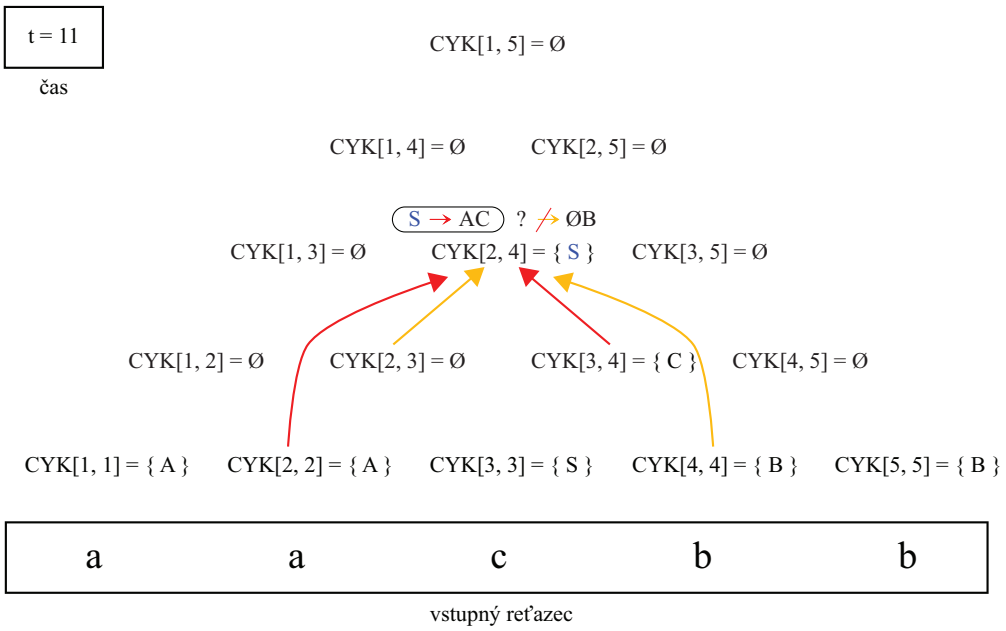
Do množiny $CYK[3,4]$ pridáme neterminál C, pretože $CYK[3,3]$ obsahuje neterminál S, $CYK[4,4]$ obsahuje neterminál B a existuje pravidlo $C \to SB \in P$. Množina $CYK[4,5]$ ostane prázdna.



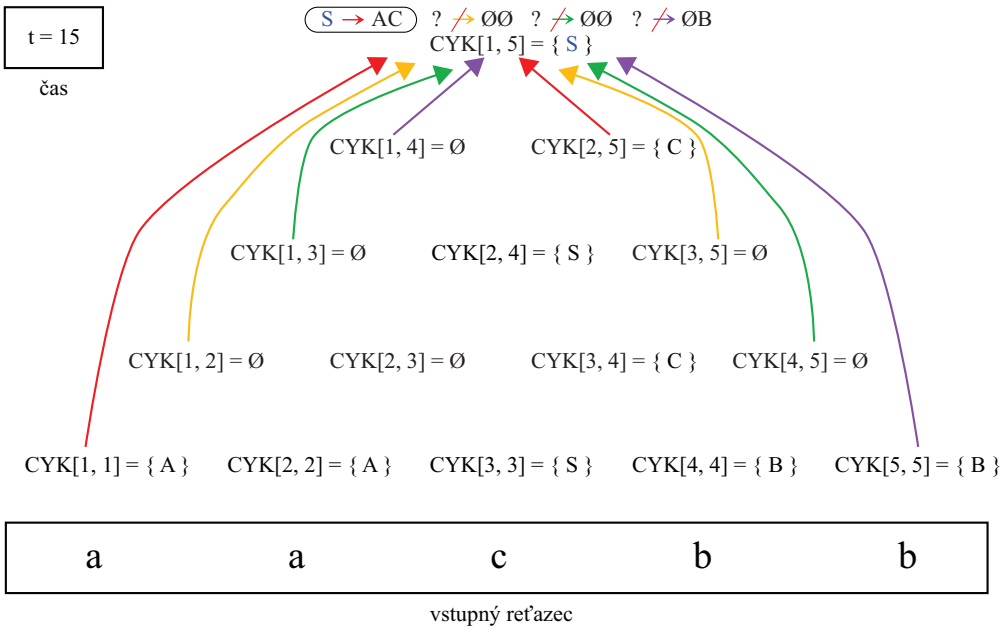
Obr. 6.2: Výpočet množiny $CYK[2,3]$ v čase $t = 7$ podľa algoritmu CYK

Čím ďalej, tým je výpočet náročnejší, pretože množiny začínajú byť závislé od viac množín ako doteraz. Napr. množina $CYK[2,4]$ je závislá od množín $CYK[2,2]$, $CYK[3,4]$ a $CYK[2,3]$, $CYK[4,4]$, ako je možné vidieť na obrázku 6.3. Je zrejmé, prečo práve tieto množiny. Sú len dve možnosti ako dosiahnuť to, aby sme do množiny $CYK[2,4]$ dostali neterminály, z ktorých dokážeme odvodiť časť vstupného reťazca od 2. po 4. symbol. Skúsime teda postupne obe možnosti, či nám túto aktuálne počítanú množinu rozšíria o nejaké neterminály. Pravidlo $p \in P$ také, že $rhs(p) = AC$, existuje a preto jeho ľavú stranu, teda neterminál S pridáme do množiny $CYK[2,4]$. Druhá dvojica množín, na obrázku 6.3 vyznačená žltými šípkami, do množiny $CYK[2,4]$ nič nepridá. Totižto množina $CYK[2,3]$ je prázdna a teda pravá strana hľadaného pravidla by obsahovala najviac jeden neterminál. To avšak nie je prípustné, pretože pracujeme s gramatikou v Chomského normálnej forme. Musí teda platiť, že na pravej strane pravidla môže byť práve jeden terminál alebo práve dva neterminály.

Takýmto spôsobom sa postupne doplnia všetky zvyšné množiny. Koniec výpočtu nastáva v čase $t = 15$ a konečnú podobu množín je možné vidieť na obrázku 6.4. Keďže sa počiatočný neterminál S nachádza v množine $CYK[1,5]$, platí tvrdenie $aacbb \in L(G)$.



Obr. 6.3: Výpočet množiny $CYK[2,4]$ v čase $t = 11$ podľa algoritmu CYK



Obr. 6.4: Výpočet množiny $CYK[1,5]$ v čase $t = 15$ podľa algoritmu CYK

Kapitola 7

Návrh paralelnej obecnej metódy

V tejto kapitole si predstavíme metódu, ktorú budeme označovať ako **PCYK** algoritmus. Označenie PCYK vzniklo spojením slova paralelný (anglicky *parallel*) – odtiaľ písmeno P – a skratkou CYK. V podstate sa jedná o modifikáciu CYK algoritmu, ktorý sme si demonštrovali v kapitole 6.2.2, do paralelnej podoby. Dôvodom pre výber práve CYK algoritmu je jeho jednoduchosť a možný priestor pre zrýchlenie. Určité množiny, ktoré sa v CYK algoritme vyskytujú, môžu byť vypočítané naraz, pretože nie sú na sebe závislé. Znamená to, že je možné použiť paralelizmus tak, aby sa tieto množiny počítali v rovnaký okamžik a tým môžeme dosiahnuť požadované zrýchlenie. Platí to len o niektorých množinách a sú tam isté podmienky, o ktorých si bližšie v tejto kapitole niečo povieme. Obrázky v podkapitole 7.3 sú vytvorené na základe obrázkov z [12].

7.1 Princíp fungovania algoritmu

CYK algoritmus pracuje v princípe sekvenčne. Postupne rozširuje množiny neterminálov, kým to je možné. Tieto množiny je možné roztriediť do skupín, ktoré budeme nazývať úrovne.

Definícia 7.1. Hovoríme, že množina $CYK[i, j]$ je na k-tej *úrovni*, ak platí:

$$j - i + 1 = k$$

Napr. množina $CYK[1, 1]$ je na 1. úrovni, pretože $1 - 1 + 1 = 1$. Množina $CYK[1, 5]$ je na 5. úrovni, pretože $5 - 1 + 1 = 5$.

Aby sme si uľahčili prácu s vysvetľovaním PCYK algoritmu, zadefinujeme si pojem pokrývať, ktorý budeme používať a ktorý hrá dôležitú úlohu v tejto metóde.

Definícia 7.2. Nech $G = (N, T, P, S)$ je BKG. Majme reťazec $x = a_1 a_2 \dots a_n \in T^*$. Hovoríme, že množina $CYK[i, j]$, kde $1 \leq i \leq j \leq n$, pokrýva reťazec $x' = a_i \dots a_j$ vtedy a len vtedy, ak pre každý neterminál $A \in CYK[i, j]$ platí, že $A \Rightarrow^* a_i \dots a_j$, pričom $A \in N$.

Je zrejmé, že reťazec x' je podreťazcom reťazca x. Z tejto definície ďalej vyplýva, že množiny na 1. úrovni pokrývajú reťazec s dĺžkou 1. Množiny na 2. úrovni pokrývajú reťazec s dĺžkou 2, atď.

CYK algoritmus sa dá rozdeliť v podstate na dve fázy. Najprv samozrejme dochádza k inicializácii množín, čo je jednoducho povedané vyprázdnenie množín, resp. nastavenie každej množiny na prázdnu množinu. Potom ale nasleduje prvá fáza, v ktorej dochádza

k počítaniu množín v tvare $CYK[i,i]$, kde $1 \leq i \leq n$, to znamená množín na 1. úrovni. Jednotlivé množiny na tejto úrovni nie sú na sebe závislé, pretože každá ma pridelený jeden symbol, podľa ktorého hľadá vyhovujúce pravidlá. Pre výpočet každej množiny teda stačí jeden vstupný symbol a konečná množina pravidiel. Ani jedna z týchto dvoch vecí sa v čase nemení. Znamená to, že tieto množiny môžeme určiť všetky naraz v jeden časový okamžik. Ako sa s tým vysporiada PCYK algoritmus, si vysvetlíme neskôr.

V druhej fáze CYK algoritmu dochádza k postupnému dopĺňaniu množín na vyšších úrovniach. Pôvodný algoritmus len uvádza, akým spôsobom sa majú tieto množiny počítať, avšak nehovorí o tom, kedy sa má počítať, ktorá množina. CYK algoritmus je napísaný nedeterministicky, čo je v praxi značnou nevýhodou a programátor sa s tým musí vysporiadať sám. Samozrejme platí, že ak je aktuálne počítaná množina závislá od iných množín, v tejto chvíli už musia byť práve tieto množiny určené. V PCYK algoritme si kladieme podmienku, že ak chceme určiť množinu na úrovni k, musíme najprv určiť množiny na všetkých úrovniach k', pre ktoré platí $1 \leq k' < k \leq n$. Jednoducho povedané, ak chceme určiť množinu na určitej úrovni, musia byť vypočítané všetky množiny na všetkých úrovniach nižších ako je úroveň, na ktorej sa nachádza množina, ktorú chceme práve vypočítať. Dôvod je prostý. Množiny sú závislé len od tých množín, ktoré sú na nižších úrovniach. Množiny na rovnakej úrovni nie sú na sebe závislé. Neplatí to len o prvej úrovni, ktorú sme už spomínali pri popise prvej fáze, ale platí to aj pre všetky ostatné úrovne. Podobne ako v prvej fáze, aj v druhej fáze môžeme jednotlivé množiny, ale na rovnakej úrovni, určovať naraz.

7.2 Použitý paralelizmus

Vypočítať všetky množiny na rovnakej úrovni je len teoreticky možné. Počet množín na rovnakej úrovni závisí od dĺžky vstupného reťazca. V praxi sa väčšinou do syntaktickej analýzy dostane zdrojový program rozdelený na tokeny. Token je jeden symbol a postupnosť tokenov, resp. symbolov je náš vstupný reťazec. Zdrojové programy sú zväčša veľmi rozsiahle, teda dĺžka takéhoto vstupného reťazca je veľmi veľká. Určite sa nebavíme o reťazci, zloženého z desiatok symbolov, ale oveľa viac. Keď si uvedomíme, že v dnešnej dobe sú bežné procesory, ktoré zvládnu vykonávať dva až šestnásť procesov, resp. vlákien zároveň, potom je očividné, že nemá zmysel počítať všetky množiny na rovnakej úrovni naraz. Bolo by to naivné, ak by sme pre každú množinu na rovnakej úrovni vytvorili jedno vlákno alebo proces, ktorý by túto množinu naplnil. V teórii je to ale možné a bol by to správny prístup, ktorý by dokázal rapídne zrýchliť pôvodný algoritmus, no v praxi musíme počítať s momentálne dostupnými zdrojmi a prostriedkami. Dnes by to prakticky vyzeralo tak, že by sa vytvoril obrovský počet vlákien, ktoré by procesor nestíhal vykonávať zároveň a drvivá väčšina vlákien by bola dosť dlhú dobu nečinná. Potom by dochádzalo k obrovskej réžii prepínania vlákien a zabralo by to veľa času. To by mohlo algoritmus razantne spomaliť.

Preto je PCYK algoritmus navrhnutý tak, že na začiatku sa určí množstvo množín, ktoré sa môže vykonávať zároveň. Prakticky to znamená počet vlákien, ktoré budú pracovať paralelne. Algoritmus dokáže počítať s toľkými vláknami, koľko sa mu na začiatku určí. Počas behu algoritmu už tento počet nie je možné zmeniť, ale na začiatku sa môže nastaviť na ľubovoľnú hodnotu. Potom to funguje tak, že každá úroveň sa rozdelí rovnomerne na menšie skupiny podľa toho, koľko vlákien sa môže naraz vykonávať. Určiť tieto skupiny nie je úplne triviálna záležitosť a takáto operácia sa musí vykonávať pred výpočtom každej úrovne. Ak budeme mať v jednej úrovni napr. 10 množín a naraz sa môžu počítať 3 množiny, potom 1. vlákno bude počítať 4 množiny, 2. vlákno 3 množiny a 3. vlákno 3 množiny. Ak sa posunieme o úroveň vyššie, potom to už bude dokonale rovnomerné a každé vlákno

bude počítať práve 3 množiny, samozrejme každé vlákno bude mať na starosti iné množiny, ale počet bude rovnaký. Už z tohto príkladu vidno, prečo to rozdelenie množín medzi všetky vlákna nebude úplne jednoduché a bude tam potrebný sofistikovaný prepočet.

7.3 Ukážka algoritmu

Algoritmus teda bude postupovať zdola-nahor a najprv určí množiny na 1. úrovni, potom na 2., 3., a tak ďalej, až sa dostane na najvyššiu n-tú úroveň. Vstup a výstup algoritmu je rovnaký, ako pri CYK algoritme až na počet vlákien, čo je novým vstupným parametrom. Podmienka, ktorá určí, či vstupný reťazec patrí do jazyka generovaného vstupnou gramatikou, ostáva taktiež rovnaká ako pri pôvodnom algoritme, teda keď počiatočný neterminál S je v množine $CYK[1, n]$, potom $x \in L(G)$. Celý priebeh výpočtu je popísaný v algoritme 7.1.

Príklad 7.1. Uvažujme BKG $G = (N, T, P, S)$ v CNF. Nech $N = \{A, B, C, S\}$ a $T = \{a, b\}$. Množina pravidiel P je nasledovná:

$$S \to AC$$
$$C \to SB$$
$$A \to a$$
$$B \to b$$
$$S \to c$$

Vstupný reťazec je *aacbb*. Otázka znie, či daný vstupný reťazec patrí do jazyka $L(G)$. Tentokrát ale použijeme metódu PCYK podľa algoritmu 7.1.

Vstup CYK a PCYK algoritmu sa líši v tom, že PCYK algoritmus navyše vyžaduje počet vlákien, teda počet množín, ktoré sa môžu počítať v jeden okamih. Pre tento príklad zvolíme ako túto konštantu číslo 2. Prvý krok, ktorý musíme urobiť je to, že rozdelíme množiny na 1. úrovni na 2 skupiny. Nebude to rozdelené dokonale rovnomerne, pretože na vstupe máme 5 symbolov. Tak teda 1. vláknu pridelíme prvé 3 množiny a 2. vláknu pridelíme zvyšné 2 množiny. Znamená to, že v $S[1]$ bude uložené číslo 3 a v $S[2]$ bude uložené číslo 2. Rozdelenie, ktoré sme si zvolili, je možné vidieť na obrázku 7.1.

Ďalšia časť algoritmu sa vykonáva paralelne. Pre každé vlákno sa vypočíta pomocná premenná f. Pre 1. vlákno bude $f = 0$, pre 2. vlákno bude $f = 3$. Premenná f slúži na to, aby sme presne určili poradie množiny na danej úrovni. Potom už môže každé vlákno počítať pridelené množiny nezávisle od iných vlákien. Ten výpočet množín je už rovnaký, ako pri CYK algoritme. Prejdeme celú množinu pravidiel a budeme hľadať také pravidlá, ktoré na pravej strane obsahujú len jeden terminál a tento terminál musí zodpovedať vstupnému symbolu na pozícii d, pričom $d = f +$ (poradie množiny v rámci danej skupiny). V čase $t = 1$ budú množiny vyzerať podľa obrázka 7.1.

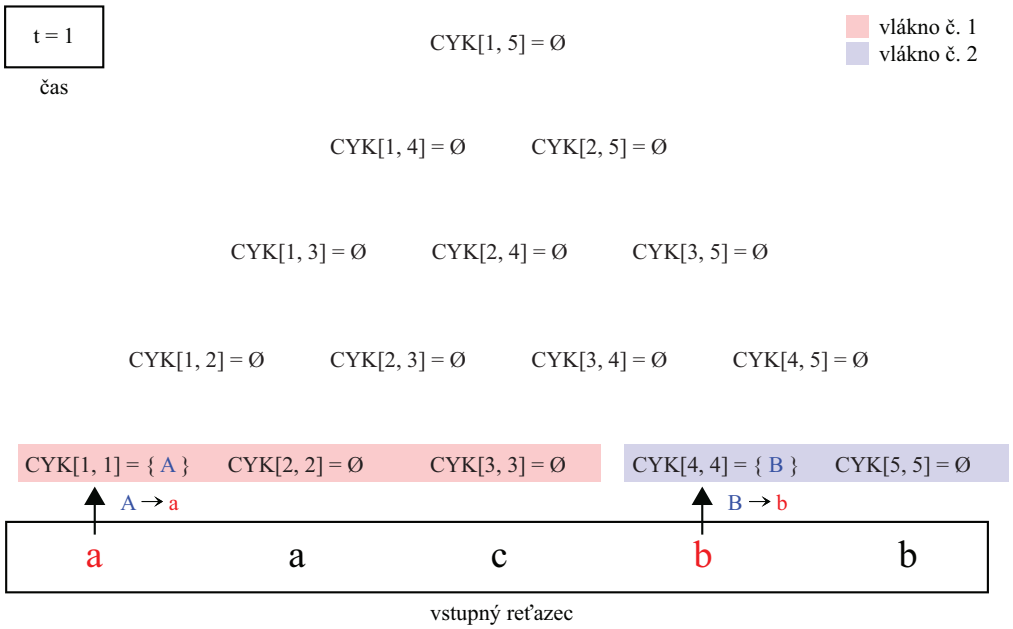
V čase $t = 2$ už budú naplnené aj množiny $CYK[2, 2]$ a $CYK[5, 5]$, to je jasné. Zaujímavé je ale to, že v čase $t = 3$ pribudne len jedná vypočítaná množina. Ako je to možné? Prečo len jedno vlákno bude aktívne? No povedali sme si, že ak chceme vypočítať množiny na vyššej úrovni, musíme mať najprv vypočítané všetky množiny na všetkých nižších úrovniach od úrovne, na ktorej chceme momentálne vypočítať množiny. Táto podmienka musí platiť vždy. V čase $t = 2$ ostáva už len jedna množina na 1. úrovni nevypočítaná a tou je $CYK[3, 3]$. Táto množina je pridelená 1. vláknu. Zatiaľ čo 1. vlákno bude pracovať na naplnení tejto množiny, 2. vlákno bude musieť čakať.

Algoritmus 7.1: PCYK algoritmus

Vstup: BKG $G = (N, T, P, S)$, reťazec $x = a_1 a_2 \dots a_n \in T^*$ a počet vlákien c
Výstup:

- PRIJAŤ, keď $x \in L(G)$
- ODMIETNÚŤ, keď $x \notin L(G)$

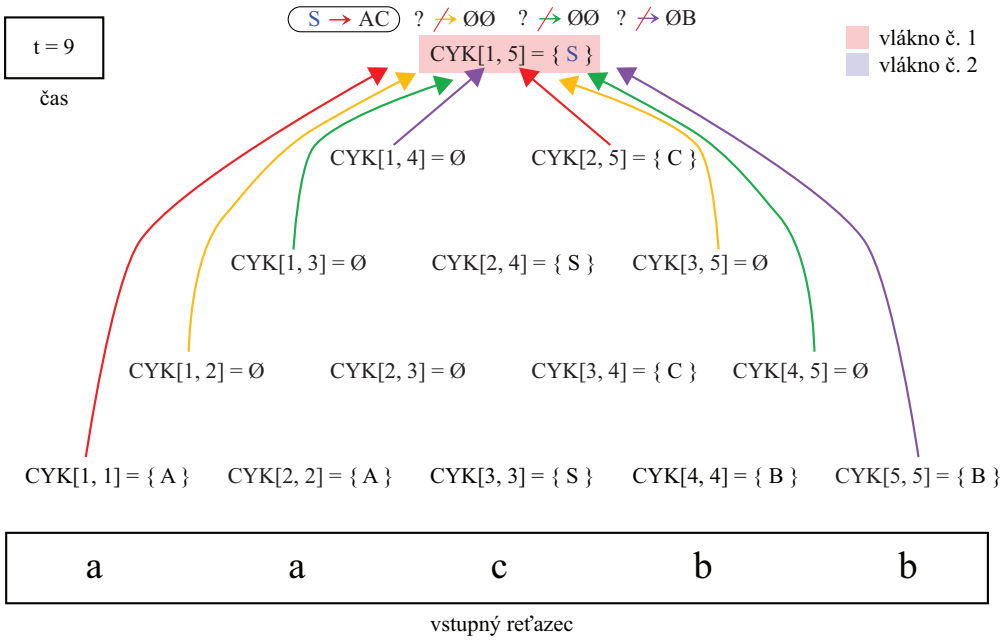
```
1  begin
2      inicializácia množín CYK[i, j] = ∅ pre 1 ≤ i ≤ j ≤ n;
3      rovnomerne rozdeľ množiny na 1. úrovni do c skupín a počet množín v týchto
        skupinách ulož jednotlivo do poľa S tak, že v S[1] bude uložený počet množín 1.
        skupiny pre 1. vlákno, v S[2] bude uložený počet množín 2. skupiny pre 2.
        vlákno, atď.;
4      for t = 1 to c do in parallel
5          f = 0;
6          for i = 1 to t − 1 do
7              f+ = S[i];
8          for i = 1 to S[t] do
9              d = i + f;
10             foreach p in P do
11                 if rhs(p) == a_d then
12                     pridaj lhs(p) do CYK[d, d];
13     for e = 2 to n do
14         rovnomerne rozdeľ množiny na úrovni e do c skupín a počet množín v týchto
            skupinách ulož jednotlivo do poľa S tak, že v S[1] bude uložený počet
            množín 1. skupiny pre 1. vlákno, v S[2] bude uložený počet množín 2.
            skupiny pre 2. vlákno, atď.;
15         for t = 1 to c do in parallel
16             f = 0;
17             for i = 1 to t − 1 do
18                 f+ = S[i];
19             for i = 1 to S[t] do
20                 d = i + f;
21                 for k = d to d + e − 2 do
22                     foreach B in CYK[d, k] do
23                         foreach C in CYK[k + 1, d + e − 1] do
24                             foreach p in P do
25                                 if rhs(p) == BC then
26                                     A = lhs(p);
27                                     pridaj A do CYK[d, d + e − 1];
```



Obr. 7.1: Výpočet množín $CYK[1,1]$ a $CYK[4,4]$ v čase $t = 1$ podľa algoritmu PCYK

Množiny na 1. úrovni máme už všetky vypočítané. Čím vyššia úroveň, tým sú množiny viac závislé od iných množín. To platilo pre CYK algoritmus a platí to aj pre PCYK algoritmus. Čo má každá úroveň v tomto prípade spoločné je to, že pred začiatkom výpočtu je potrebné určiť si, ktoré vlákno bude počítať ktoré množiny. V našom príklade to rozdelenie množín na 2. úrovni bude menej problematické ako na 1. úrovni, pretože 4 množiny môžeme rozdeliť jednoducho na 2 polovice. Potom už každé vlákno začne počítať pridelené množiny. Pravidlá, podľa ktorých sa množiny na vyšších úrovniach dopĺňajú, sú rovnaké ako v prípade CYK algoritmu. V algoritme 7.1 je to popísané striktne deterministicky, ale to už si nebudeme prechádzať úplne dopodrobna. Stačí vedieť, že stále platí tvrdenie, že množina s označením $CYK[1,2]$ má pokrývať reťazec od 1. po 2. symbol, teda v tomto prípade reťazec aa. Musíme sa teda pozrieť na množiny $CYK[1,1]$ a $CYK[2,2]$, ktoré dokopy pokrývajú práve spomenutý reťazec aa. Hľadáme pravidlo s pravou stranou zloženou z neterminálov AA. Také pravidlo neexistuje a preto množina $CYK[1,2]$ ostane prázdna. Ďalšie príklady výpočtu množín si už uvádzať nebudeme, pretože sa počítajú rovnakým spôsobom, ktorý platí aj pre CYK algoritmus.

Podstatná vec, ktorú si treba uvedomiť je to, že PCYK algoritmus využíva paralelné spracovanie, to znamená, že môžeme určité množiny vypočítať naraz v jeden okamih. Práve táto technika prispieva k zrýchleniu algoritmu, teda aspoň na teoretickej báze. Výsledok algoritmu je rovnaký, ako to bolo v príklade 6.1. Záverečný pohľad na množiny je možné vidieť na obrázku 7.2. Dôležitým poznatkom je, že PCYK algoritmu trvalo 9 časových jednotiek, kým vypočítalo poslednú množinu, pričom CYK algoritmu to trvalo 15 časových jednotiek. Zrýchlenie je na prvý pohľad poznateľné.



Obr. 7.2: Výpočet množiny $CYK[1,5]$ v čase $t = 9$ podľa algoritmu PCYK

Kapitola 8

Implementovaná aplikácia

Na základe PCYK algoritmu bola implementovaná aplikácia. Ide o syntaktický analyzátor, ktorého výstupom nie je derivačný strom ako to zvyčajne býva, ale len jednoduchá odpoveď: áno alebo nie. Aplikácia len rieši otázku, či patrí vstupný reťazec do jazyka generovaného vstupnou gramatikou alebo nepatrí.

V tejto kapitole si popíšeme, ako bola aplikácia navrhnutá a implementovaná. Najprv si povieme niečo o základných vlastnostiach programu, potom si popíšeme vstup a výstup aplikácie. Priblížime si, aké triedy boli v programe použité. Vysvetlíme si, ako bol vyriešený paralelizmus a nakoniec si povieme niečo o použitých dátových typoch.

8.1 Základné vlastnosti programu

Po preložení programu vznikne spustiteľný súbor s názvom **pcyk**. Program je napísaný v programovacom jazyku C++ a používa štandard C++11. Pri písaní programu boli použité techniky objektovo orientovaného programovania. Program pre svoju činnosť využíva len štandardné knižnice jazyka C++. Aplikácia bola vyvíjaná na operačnom systéme Ubuntu 15.10.

Program je rozdelený do niekoľkých častí. Každá časť (okrem hlavnej časti programu) je implementovaná ako trieda. Každá trieda je tvorená dvojicou: zdrojový súbor (prípona *.cpp*) a hlavičkový súbor (prípona *.hpp*). Dokopy sa v programe vyskytuje 12 tried a každá má svoju úlohu.

Hlavná časť programu je vykonávaná funkciou `main()`, ktorá sa nachádza v súbore `main.cpp`. Na začiatku sa získajú a analyzujú vstupné parametre. Na základe toho sa vytvoria objekty reprezentujúce gramatiku a vstupný reťazec. Tie potom putujú do ďalšieho objektu, ktorý zabezpečí priebeh celého PCYK algoritmu. Metóda sa spustí a získaný výsledok sa zapíše na výstup.

8.2 Vstup a výstup aplikácie

Vstupom aplikácie je reťazec, gramatika a číslo udávajúce počet vlákien. Reťazec má veľmi jednoduchú syntax. Jednotlivé symboly sú oddelené bielymi znakmi. Pri gramatike je to už trochu komplikovanejšie. Vstupnú gramatiku je možno zapísať rôznymi spôsobmi. Existuje niekoľko štandardov pre syntax gramatiky. Jeden zo štandardov definuje aj veľmi známy generátor syntaktických analyzátorov GNU bison[17]. Avšak implementovaná aplikácia po-

užíva vlastnú syntax pre vstupnú gramatiku, ktorá je popísaná touto BNF:

⟨grammar⟩ ⊨ ⟨rule⟩ | ⟨rule⟩ ⟨grammar⟩
⟨rule⟩ ⊨ ⟨rule-lhs⟩ ":" ⟨opt-ws⟩ ⟨rule-rhs⟩ ⟨opt-ws⟩ ";" ⟨line-end⟩
⟨rule-lhs⟩ ⊨ ⟨opt-ws⟩ ⟨non-terminal⟩ ⟨opt-ws⟩
⟨rule-rhs⟩ ⊨ ⟨non-terminal⟩ ⟨opt-ws⟩ "," ⟨opt-ws⟩ ⟨non-terminal⟩ | ⟨terminal⟩
⟨line-end⟩ ⊨ ⟨opt-ws⟩ ⟨EOL⟩ | ⟨line-end⟩ ⟨line-end⟩
⟨opt-ws⟩ ⊨ ⟨whitespace⟩ ⟨opt-ws⟩ | ϵ

Čo sa týka počtu vlákien, tak v prípade, že je zadané číslo 1, bude sa vykonávať pôvodný algoritmus CYK bez použitia techník paralelizmu. To isté nastane, ak sa tento počet neuvedie, keďže tento parameter nie je povinný. Všetky parametre sú popísané v tabuľke 8.1.

Krátky variant	**Dlhý variant**	**Vysvetlenie**	**Povinnosť parametra**
`-h`	`--help`	program vypíše na štandardný výstup pomocník, ktorý podáva informácie o aplikácii a vysvetľuje, ako ju používať	nie
`-s <súbor>`	`--string <súbor>`	súbor, v ktorom je uložený vstupný reťazec	áno
`-g <súbor>`	`--grammar <súbor>`	súbor, v ktorom je uložená vstupná gramatika	áno
`-o <súbor>`	`--output <súbor>`	súbor, do ktorého bude zapísaný výsledok algoritmu a v prípade zadaného prepínača `-d` aj ladiace informácie; ak sa nepoužije tento parameter, zapisovať sa bude na štandardný výstup	nie
`-t <číslo>`	`--threads <číslo>`	počet vlákien, ktoré sa zúčastnia na výpočte CYK množín; ak bude počet 1 alebo tento argument nebude vôbec zadaný, spustí sa pôvodná metóda CYK bez použitia paralelizmu	nie
`-d`	`--debug`	program vypíše ladiace informácie (počet vlákien, vstupný reťazec, dĺžka vstupného reťazca, vstupná gramatika, obsah množín po skončení algoritmu a čas trvania algoritmu v mikrosekundách)	nie

Tabuľka 8.1: Vymenovanie a popis vstupných parametrov programu

Výstupom aplikácie je jednoduchá odpoveď: áno alebo nie, resp. TRUE alebo FALSE. Zapisuje sa to buď na štandardný výstup alebo do súboru, ak bol zadaný príslušný vstupný

argument. Ak aplikácia beží v režime ladenia, výstupom sú aj rôzne výpisy, napr. vstupná gramatika, reťazec, vyplnené množiny CYK apod.

8.3 Popis jednotlivých tried

Trieda **InputParameters** kontroluje vstupné argumenty programu a uchováva ich. Pri programovaní tejto triedy bol použitý návrhový vzor *singleton*[1]. Dôležitou metódou je `get()`, ktorej parametrami sú počet argumentov programu a pole argumentov. Táto metóda ich analyzuje a potrebné informácie uloží do atribútov objektu.

Trieda **Parser** pripraví vstupný reťazec a vstupnú gramatiku do podoby, s ktorou potom ďalej program bude vedieť pracovať. Táto trieda implementuje rozhranie **IParser**. Použitím rozhrania sa možnosti aplikácie rozširujú, keďže je možné implementovať viacero tried, ktoré budú vytvárať objekty reprezentujúce vstupný reťazec a gramatiku. Na spôsobe konštrukcie týchto objektov nezáleží. Podstatné je, aby tieto triedy obsahovali 2 metódy a to `constructGrammar()` a `constructInputString()`, ktoré vrátia vytvorenú gramatiku a reťazec.

Vstupnú gramatiku zastupuje trieda **Grammar**. Jej atribútmi sú počiatočný symbol, abeceda terminálov, abeceda neterminálov a množina pravidiel, presne podľa definície 3.1. Okrem metód, ktorými získame (tzv. *getter*) alebo nastavíme (tzv. *setter*) atribúty objektu, sa v tejto triede vyskytuje metóda `addProduction()`, ktorá pridáva do gramatiky pravidlá. Navyše táto metóda vráti odkaz na vlastnú inštanciu. Je to z toho dôvodu, aby bolo možné použiť reťazové volanie tejto metódy (`addProduction()->addProduction()->...`). Poslednou metódou v tejto triede je metóda `isInChomskyNormalForm()`, ktorá vie zistiť, či je daná gramatika v CNF.

Pre symbol bola implementovaná trieda **Symbol**. Každý symbol musí byť buď terminál alebo neterminál. Je to vyriešené tak, že v tejto triede je atribút `terminal_`, ktorý má booleovskú hodnotu. Ak to je TRUE, potom je daný symbol terminál, inak je neterminál. Konštruktor tejto triedy je privátna metóda. Symboly je možné vytvárať pomocou statických metód `newTerminal()` a `newNonterminal()`, ktorým dáme reťazec – teda názov symbolu – a vrátia nám vytvorený terminál alebo neterminál, čo je len objekt triedy **Symbol**, ale už s nastaveným príznakom `terminal_`.

Trieda **Alphabet** je rozhranie pre abecedu. V syntaktickej analýze sa stretávame s dvoma typmi abecedy a to je abeceda terminálov a abeceda neterminálov. Každá ma svoju vlastnú triedu a to **TerminalAlphabet** a **NonterminalAlphabet**, pričom obidve triedy implementujú práve rozhranie **Alphabet**. Trieda **Alphabet** má jediný atribút `symbols_`, ktorého dátový typ je množina objektov triedy **Symbol**. Metóda `getSymbol()` je rovnaká pre obidve abecedy a jednoducho vráti jediný symbol podľa názvu. V čom sa abecedy líšia je to, že v prípade abecedy terminálov, chceme mať metódu, ktorá bude pridávať do inštancie terminály, no a v opačnom prípade to budú neterminály. Z toho dôvodu bolo navrhnuté spoločné rozhranie s virtuálnou metódou `addSymbol()`, ktorá do inštancie pridáva nové symboly.

Pre pravidlá gramatiky bola vytvorená trieda **Production**, ktorá ma 2 atribúty: `leftHandSide_` a `rightHandSide_`, teda ľavú a pravú stranu. Ľavá strana je objekt triedy **Symbol** a musí byť zadaná pri konštrukcii objektu. Pravá strana je vektor objektov triedy **Symbol** a pri vytvorení objektu je prázdna. Pomocou metódy `addSymbolToRHS()` je možné

[1] Singleton je návrhový vzor, ktorým dosiahneme to, že trieda bude mať maximálne jednu inštanciu a bude sa k nej pristupovať z jedného globálneho bodu[18].

pridávať symboly na pravá stranu, pričom stále sa nový symbol pridá na koniec, teda úplne napravo. Podobne ako to je vymyslené pri triede **Grammar** a jej metóde `addProduction()`, aj metóda `addSymbolToRHS()` vráti odkaz na vlastnú inštanciu, aby bolo možné využiť reťazové volanie tejto metódy.

Podstatná časť programu a síce samotný PCYK algoritmus je implementovaný v triede **PCYK**. Jej hlavnými atribútmi sú vstupný reťazec, vstupná gramatika a počet vlákien. Tieto tri atribúty sa inicializujú hneď pri konštrukcii objektu. Okrem týchto atribútov má ešte ďalšie pomocné atribúty potrebné pre beh algoritmu. Algoritmus sa spustí metódou `run()`, ktorej návratový typ je `bool` a teda výsledkom bude odpoveď na otázku, či patrí vstupný reťazec do jazyka generovaného vstupnou gramatikou. Na začiatku algoritmu sa overí, či je vstupná gramatika v CNF. Potom sa na základe počtu vlákien zistí, či má bežať algoritmus paralelne alebo nie. V prípade, že sa má použiť pôvodný algoritmus CYK, zavolá sa metóda `runOrigin()`, ktorá nepoužíva pri výpočte žiadny paralelizmus a je implementovaná podľa algoritmu 6.1, a hneď sa vráti jej výsledok. Ak je počet vlákien väčší ako 1, začnú sa vykonávať príkazy podľa algoritmu 7.1. Metóda `prepareSetsForEachThread()` rovnomerne rozdelí množiny na aktuálnej úrovni[2] a pre každé vlákno sa do vektora `rangeOfSetsForEachThread_` uloží dvojica čísel, ktorá určuje interval, podľa ktorého každé vlákno vie, ktoré množiny má v danej chvíli vypĺňať. Táto metóda sa volá na začiatku každej úrovne. Po vytvorení vlákien sa zavolá metóda `threadFunction()`. Na začiatku prebehne výpočet 1. úrovne za pomoci metódy `firstLevel()`. Potom nasleduje cyklus, ktorý pre každú ďalšiu úroveň zavolá metódu `higherLevel()`. Metódy `firstLevel()` a `higherLevel()` počítajú len jednu množinu podľa vstupného argumentu. Preto je potrebné, aby každé vlákno tieto metódy volalo viackrát, pre každú množinu, ktorá mu bola pridelená. Keď všetky vlákna skončia svoju prácu, zistí sa, či sa v množine $CYK[1,n]$, kde n je dĺžka vstupného reťazca, nachádza počiatočný neterminál. Ak áno, metóda vráti hodnotu TRUE, inak vráti FALSE.

Trieda **Utils** bola vytvorená pre všeobecne využiteľné metódy, napr. pre prácu s reťazcami. Obsahuje 2 metódy `removeWhiteSpaces()` a `explode()`. Obidve sú implementované ako statické metódy a teda ak ich chceme niekde zavolať, nie je potrebné vytvárať inštanciu tejto triedy. Prvá menovaná metóda dokáže odstrániť z reťazca všetky biele znaky. Druhá metóda vie rozdeliť reťazec na vektor podreťazcov podľa zadaného oddeľovača.

8.4 Riešenie paralelizmu

Paralelnú podobu algoritmu je možné docieliť pomocou viacero vlákien alebo procesov. Implementovaná aplikácia používa len štandardné knižnice a konkrétne pre účely paralelizmu sa využívajú štandardné vlákna, ktoré pribudli v norme C++11. Ak riešime paralelnú úlohu, problémom niekedy môže byť synchronizácia vlákien. V PCYK algoritme sa jasne uvádza, že keď vlákno dokončí svoju úlohu na aktuálnej úrovni, musí počkať za ostatnými vláknami, aby taktiež dokončili svoju prácu na tejto úrovni. V programe je to vyriešené pomocou triedy **condition_variable**, ktorá je štandardne deklarovaná v hlavičkovom súbore s rovnakým názvom. Pomocou objektu takejto triedy môžeme dočasne zastaviť vlákno, ktoré bude čakať na to, až mu iné vlákno oznámi, aby pokračovalo v práci. Blokovanie sa deje pomocou metódy `wait()`. V našom prípade to funguje nasledovne. Každé vlákno, ktoré dopočíta poslednú pridelenú množinu na aktuálnej úrovni, informuje hlavné vlákno pomocou metódy `notify_all()`, že už skončilo. Takéto vlákno potom čaká na pokyn od hlavného vlákna, aby mohlo pokračovať v dopĺňaní množín na vyššej úrovni. Hlavné vlákno infor-

[2] Aktuálna úroveň sa zistí pomocou atribútu `currentLevel_`.

muje ostatné vlákna až vtedy, ak dostane notifikáciu od každého vlákna, že dopočítalo svoje pridelené množiny. Takto zabezpečíme to, aby sa najprv vypočítali všetky množiny na jednej konkrétnej úrovni a až potom sa išlo počítať množiny na vyššej úrovni. To je jednou z podmienok navrhnutého PCYK algoritmu.

8.5 Dátové typy

Program pracuje len so štandardnými knižnicami. Namiesto obyčajných ukazovateľov na objekty, ktoré vznikli použitím operátora `new[]`, sa využívajú tzv. chytré ukazovatele (po anglicky *smart pointers*). Vďaka týmto špeciálnym ukazovateľom sa nemusí v celom programe zavolať ani raz operátor `delete[]`, ktorým by sa korektne uvoľnila pamäť pridelená operáciou `new[]`. Táto povinnosť programátora odpadá a preberajú ju práve chytré ukazovatele. Zdrojový kód je tým čistejší a čitateľnejší.

Kapitola 9

Testovanie a dosiahnuté výsledky

Táto kapitola sa zaoberá testovaním výslednej aplikácie, čo je veľmi dôležitou fázou vývoja akejkoľvek aplikácie. Ukážeme si, že implementovaný pôvodný CYK algoritmus funguje správne. Ďalej budeme porovnávať rýchlosti pôvodného a modifikovaného algoritmu. Nakoniec si povieme niečo o optimalizáciách a ako dokáže optimalizovaný program vymazať rozdiely medzi paralelným a sekvenčným algoritmom.

9.1 Prostredia pre testovanie a použité gramatiky

Testovanie prebiehalo na dvoch počítačoch. Ich základné parametre sú zapísané v tabuľke 9.1.

	PC1	**PC2**
Počet CPU	12	8
Frekvencia CPU (GHz)	2,8	2,6
Operačný systém	CentOS 6.7	Ubuntu 15.10
Preklad	g++-4.9 (GCC) 4.9.3	g++-4.9 (GCC) 4.9.3

Tabuľka 9.1: Základné vlastnosti prostredí, v ktorých prebiehalo testovanie

Pre potreby testovania boli použité nasledujúce bezkontextové gramatiky:

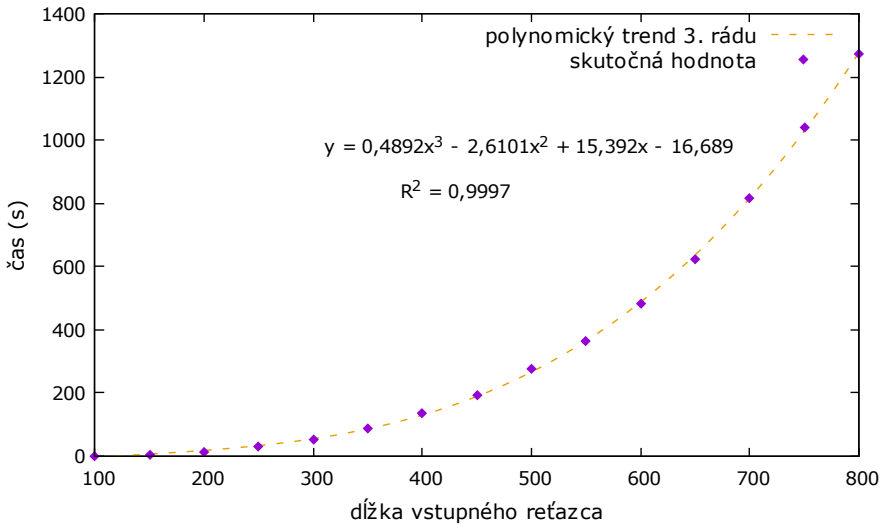
1. $G_1 = (N, T, P, S)$, $N = \{S, A, B, C, D, E\}$, $T = \{a, b\}$,
   $P = \{S \rightarrow CD,\ S \rightarrow BA,\ A \rightarrow BE,\ A \rightarrow a,\ B \rightarrow AS,\ B \rightarrow b,\ C \rightarrow a,$
   $D \rightarrow AB,\ E \rightarrow BB\}$

2. $G_2 = (N, T, P, S)$, $N = \{S, G, D, H, I, A, P, T, R, L\}$, $T = \{a, +, *, (,)\}$,
   $P = \{S \rightarrow GD,\ S \rightarrow HA,\ S \rightarrow IR,\ S \rightarrow a,\ G \rightarrow SP,\ D \rightarrow HA,\ D \rightarrow IR,$
   $D \rightarrow a,\ H \rightarrow DT,\ I \rightarrow LS,\ A \rightarrow IR,\ A \rightarrow a,\ P \rightarrow +,\ T \rightarrow *,\ R \rightarrow),$
   $L \rightarrow (\}$

9.2 Časová zložitosť CYK algoritmu

CYK algoritmus, z ktorého sa pri návrhu PCYK algoritmu vychádza, má kubickú časovú zložitosť (n^3) – to je fakt[11]. Výsledná aplikácia používa jak CYK, tak PCYK algoritmus. Obidva algoritmy sú navzájom ekvivalentné. Znamená to, že pri rovnakom vstupe dávajú rovnaké výsledky. Síce nám CYK algoritmus môže dávať správne výsledky, ale aby sme

zistili, či je algoritmus naozaj naprogramovaný správne, potrebujeme zistiť jeho časovú zložitosť. Keďže sa jedná o časovú zložitosť v závislosti od dĺžky vstupného reťazca, stačí ak budeme program opakovane spúšťať nad rovnakou gramatikou ale nad rôzne dlhými vstupnými reťazcami. Podľa obrázku 9.1 je zrejmé, že tam kubická zložitosť figuruje[1]. Týmto sme potvrdili správnosť naprogramovaného CYK algoritmu.



Obr. 9.1: Časová zložitosť CYK algoritmu – použitá gramatika G_1

9.3 Zrýchlenie pomocou vlákien

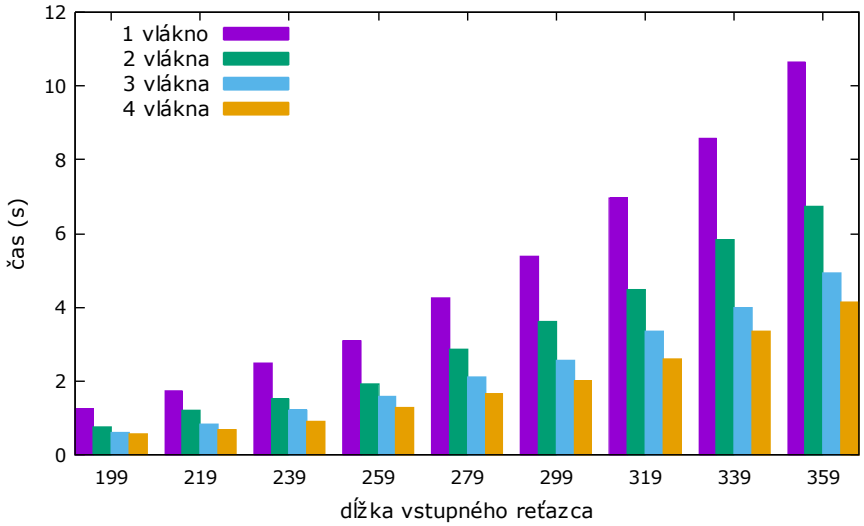
Počas testovania sa ukázalo, že pôvodná metóda CYK, ale aj navrhnutá metóda PCYK funguje správne. Ďalším krokom bolo ukázať, že paralelný algoritmus PCYK je rýchlejší ako pôvodný sekvenčný CYK algoritmus. Táto hypotéza sa overovala na počítači PC1 bez použitia optimalizácií. Výsledky sú pozitívne a je možné ich vidieť na obrázku 9.2. Pri použití paralelného algoritmu sa beh programu zrýchlil. Ďalšie zistenie, hoci zjavné, je to, že pri použití viac vlákien sa rýchlosť algoritmu zvyšuje.

9.4 Použitie optimalizácií

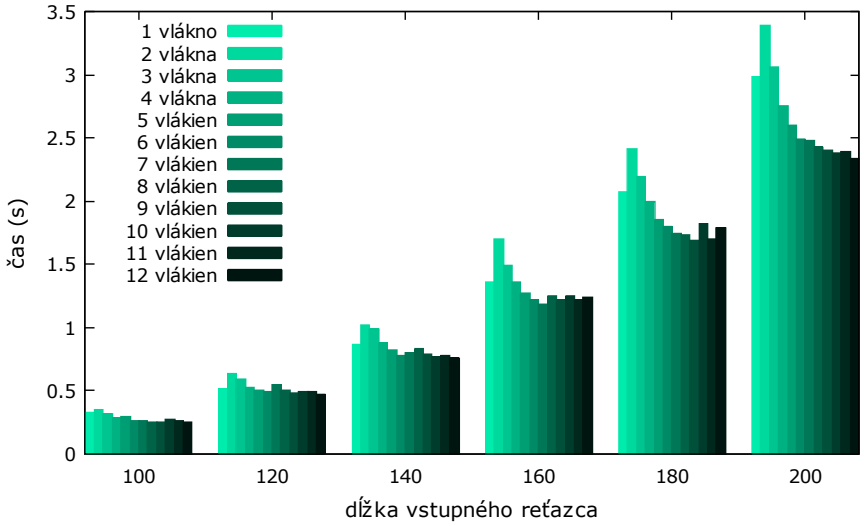
Všetky doposiaľ spomenuté testy sa vykonávali pomocou programu, ktorý bol preložený bez použitia optimalizácií. Ak ale použijeme optimalizácie, výhoda paralelizmu sa stráca. Už pri použití len dvoch vlákien dochádza dokonca k spomaleniu algoritmu oproti sekvenčnej verzii, viď obrázok 9.3. Naďalej platí tvrdenie, že ak použijeme viac vlákien, algoritmus sa vykoná rýchlejšie. Tie rozdiely v rýchlostiach pri rôznom počte vlákien už ale nie sú také výrazné.

Pri preklade aplikácie je možné pomocou prepínačov `-O1`, `-O2` alebo `-O3` dať vedieť prekladaču, aby sa pokúsil o optimalizácie programu[19]. Čím vyššie číslo použijeme, tým bude výsledný program viac optimalizovaný, čo v konečnom dôsledku znamená, že bude rýchlejší. Pri testovaní sa zistilo, že použitie prepínačov `-O1` a `-O2` dáva takmer rovnaké výsledky. V prípade použitia `-O3` už je vidno rozdiely.

[1]Program bol spúšťaný v testovacom prostredí PC1 a pri preklade sa nepoužila žiadna optimalizácia.

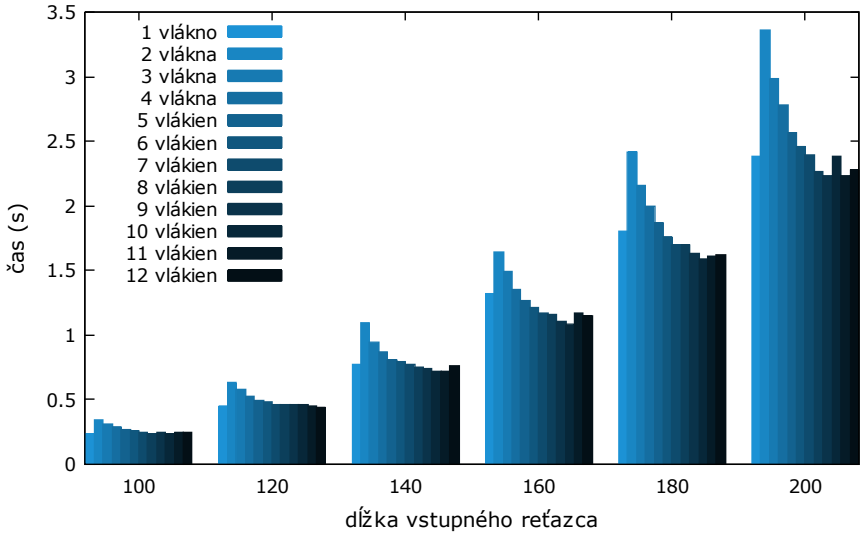


Obr. 9.2: Zrýchlenie pri použití viac vlákien – použitá gramatika G_2



Obr. 9.3: Použitie optimlizácií `-O1` – použitá gramatika G_1

Ak program optimalizujeme najviac, čo to ide, prídeme k zisteniu, že použitie paralelizmu sa v prípade študovaného algoritmu, a síce CYK algoritmu, veľmi neoplatí. Z obrázku 9.4 môžeme vyčítať, že rozdiel v rýchlosti pri použití jedného vlákna a dvoch vlákien je ešte výraznejší ako to bolo za použitia optimalizácií `-O1`. Zároveň je vidno, že 12 vlákien paralelného algoritmu dokáže vyhodnotiť vstup za takmer rovnaký čas ako 1 vlákno sekvenčného algoritmu. Pozorujeme len veľmi malé zrýchlenie. Môžeme skonštatovať, že sekvenčný algoritmus nie je až tak pomalší pri takto dobre zoptimalizovanom programe oproti paralelnému algoritmu.



Obr. 9.4: Použitie optimlizácií `-O3` – použitá gramatika G_1

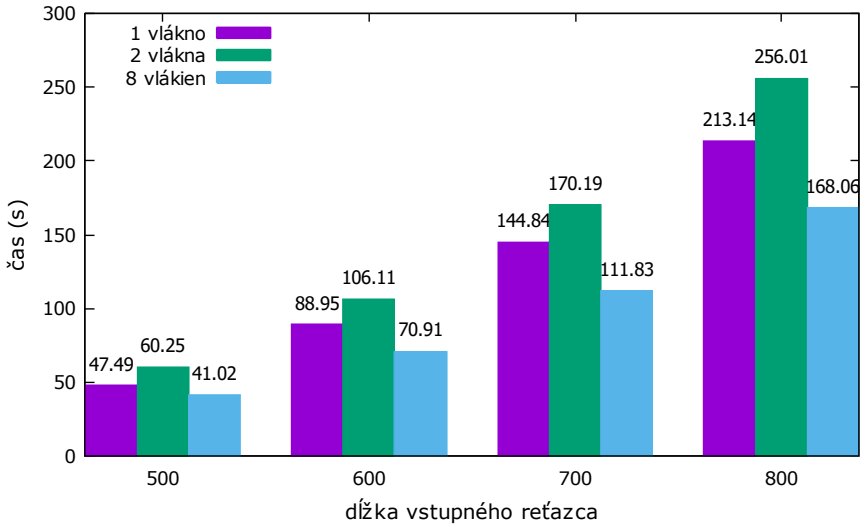
9.5 Obrovské vstupy

Doteraz spomenuté fakty boli zistené na základe testovania programu nad vstupným reťazcom s maximálnou dĺžkou 400 symbolov. Došli sme k záveru, že zrýchlenie pri väčšom počte vlákien možné síce je, ale nie je poznateľné. Čo sa stane, keď na vstup programu pošleme veľmi dlhý reťazec? Predstavme si, žeby sa syntaktická analýza vykonávala bez použitia paralelizmu 5 hodín. Teoreticky by v takomto prípade mohlo byť zrýchlenie za použitia paralelizmu už zjavné a užitočné.

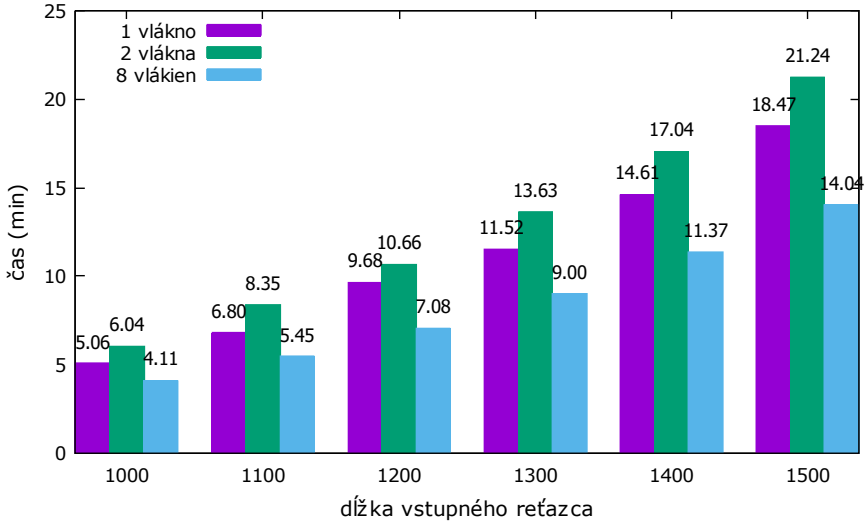
Zistilo sa, že 8 vlákien dokáže spracovať 800 symbolov dlhý reťazec o 45,08 sekúnd rýchlejšie ako 1 vlákno, viď obrázok 9.5. Percentuálne to je teda len o 21,05% rýchlejšie, pričom sa muselo použiť až 8 vlákien. Pri ešte väčších vstupoch, kde by analýza trvala omnoho dlhšie, už použitie paralelizmu význam má.

Ďalšie testy sa vykonali na počítači PC2, kde mohol bežať program neobmedzený čas. Vstupný reťazec bol o niečo dlhší a analýza sa vykonávala omnoho dlhšie. Dosiahnuté výsledky je možné pozorovať na obrázku[2] 9.6. Aj v tomto prípade bolo zrýchlenie za použitia 8 vlákien o približne 1/5 času. Ak sa teda vrátime k scenáru, že syntaktická analýza pomocou sekvenčného algoritmu CYK beží 5 hodín, potom pridaním paralelizmu by sa analýza vykonala za približne 4 hodiny. Ušetrili by sme tak hodinu času, no museli by sme použiť až 8 vlákien.

[2]V tomto grafe sa jednotka času zmenila zo sekundy na minútu kvôli lepšej čitateľnosti grafu.



Obr. 9.5: Obrovské vstupy (PC1) – použitá gramatika G_1



Obr. 9.6: Obrovské vstupy (PC2) – použitá gramatika G_1

9.6 Nevýhody paralelizmu

Paralelizmus v tomto prípade nie je až taký výhodný. Prečo je to tak? Prečo nedostávame dobré výsledky? Môže sa zdať, že pri použití viac vlákien, by mal program bežať rýchlejšie. To, že 2 vlákna by mali vykonať rovnakú prácu za dvakrát kratší čas ako 1 vlákno je len zdanie. V praxi sa vyskytujú určité problémy, ktoré si skúsime popísať.

Aby sme dosiahli paralelný beh algoritmu, použili sme koncept viacero vlákien. Treba si uvedomiť, že pri používaní viac vlákien dochádza taktiež k nevyhnutnej réžií pre prácu s týmito vláknami. Vytvorenie vlákien a takisto aj následné spojenie do hlavného vlákna trvá určitú dobu. To by nebol až taký problém, pretože k vytváraniu vlákien dochádza len na začiatku algoritmu a počas algoritmu sa nevytvárajú žiadne nové, ani nekončia existujúce vlákna. Jednoducho povedané, počas behu algoritmu ostáva počet vlákien stále rovnaký. Toto zdržanie sme ešte schopní tolerovať, pretože je konštantné, teda nezávisí od dĺžky vstupného reťazca, ani od vstupnej gramatiky. Pri väčšom vstupe sa toto spomalenie stratí.

Horšie je to so synchronizáciou vlákien. K nej dochádza počas celého algoritmu pomerne často. Nemôže predsa 2. vlákno počítať množiny na 3. úrovni, kým 1. vlákno počíta stále množiny na 2. úrovni. Musia sa navzájom čakať. Preto sú použité synchronizačné mechaniky tak, ako boli popísané v kapitole 8.4. Vlákna sa teda musia pozastaviť a neskôr znova spustiť a opäť nás to stojí drahocenný čas. Synchronizácia vlákien sa deje na každej úrovni a nie je to triviálna záležitosť. Ak označíme čas potrebný na synchronizáciu vlákien ako t_1, čas potrebný na vykonanie PCYK algoritmu bez započítania času potrebného na synchronizáciu vlákien ako t_2 a čas potrebný na vykonanie CYK algoritmu ako t_3, potom ak chceme aby PCYK algoritmus bol rýchlejší ako CYK algoritmus, musí platiť $t_1 + t_2 < t_3$. Premenná t_2 je vždy menšie číslo ako t_3, dôležitou zložkou je t_1.

Kapitola 10

Záver

Táto práca sa zaoberá syntaktickou analýzou. V 70. rokoch 20. storočia bola vymyslená obecná metóda, ktorá nesie názov Cocke-Younger-Kasami algoritmus (alebo len CYK algoritmus). Výhodou tejto metódy je to, že dokáže pracovať nad celou množinou bezkontextových jazykov. Nie je obmedzená len na LL alebo LR gramatiky. Veľkou nevýhodou je jej neefektívnosť a to, že je pomalá. Neskôr vznikli iné metódy syntaktickej analýzy, ktoré sa vykonávajú oveľa rýchlejšie a používajú sa dodnes. Síce dokážu pracovať len s LL alebo LR gramatikami, ale v súčasnosti to postačuje. Cieľom tejto práce bolo naštudovať si existujúce obecné metódy a zistiť, či sa pridaním paralelizmu do ich algoritmov môžu zrýchliť a konkurovať tak dnes používaným syntaktickým analyzátorom. Výsledkom práce je návrh paralelného algoritmu PCYK založeného na CYK algoritme a následne aj jeho implementácia.

Výhodami navrhnutého algoritmu sú paralelizmus, rýchlosť a striktný determinizmus. O paralelizme a o tom, či to je výhoda alebo nevýhoda by sa dalo polemizovať. Počas vývoja a následného testovania sa zistilo, že paralelizmus môže byť v istých situáciách nevýhodný. Tieto problémy boli popísané v kapitole 9.6. Zistilo sa, že niekedy dokáže PCYK algoritmus prekonať rýchlosť pôvodného CYK algoritmu, ale musia platiť určité podmienky. Musíme mať k dispozícii veľký počet vlákien, s ktorými môže PCYK algoritmus pracovať. PCYK algoritmus v takej podobe, v akej momentálne je, dokáže ušetriť mnoho času oproti CYK algoritmu len vtedy, ak na vstupe obdrží veľmi dlhý reťazec. Pôvodná myšlienka bola, aby sa PCYK algoritmus porovnával s dnes používanými analyzátormi, ale nakoniec sa ukázalo, že niekedy bol problém prekonať rýchlosť aj pôvodného sekvenčne pracujúceho CYK algoritmu. Navrhnutý algoritmus je napísaný striktne deterministicky. V každom kroku je jasné, čo sa má vykonať. To je veľkou výhodou pre jeho praktické využitie.

Vytvorený program je konzolovou aplikáciou, ktorej vstupom je reťazec symbolov, gramatika a počet vlákien, na ktorých má bežať PCYK algoritmus. Výstupom je jednoduchá odpoveď: áno alebo nie. Program rieší základnú otázku: patrí vstupný reťazec do jazyka generovaného vstupnou gramatikou alebo nepatrí? Program je objektovo orientovaný a teda jeho prípadná úprava nebude veľmi náročná.

Výsledná aplikácia vyžaduje pre beh algoritmu bezkontextovú gramatiku v Chomského normálnej forme. To je jednou z podmienok samotnej CYK metódy. Bez gramatiky práve v takejto forme by tento algoritmus nemohol nikdy fungovať. V rámci ďalšieho vývoja tohto projektu by sa mohla pridať možnosť automatického prevodu akejkoľvek bezkontextovej gramatiky do gramatiky v Chomského normálnej forme podľa algoritmu 4.1. Potom by vstupná gramatika nebola obmedzená normálnou formou, stačilo by, aby bola bezkontextová. Prevod by riešil samotný program.

Vstupná gramatika musí byť zapísaná vo forme, ktorej bude program rozumieť. Pre potreby tejto práce bola použitá vlastná syntax pre zápis gramatiky do súboru. Táto syntax je popísaná pomocou BNF v kapitole 8.2. Avšak v súčasnosti existuje niekoľko štandardov, ktoré sa obvykle využívajú pre zápis gramatiky. Možno by bolo dobré, ak by aj v rámci tohto projektu bol použitý niektorý zo štandardov. Samotný program už na to je predpripravený. Existuje v ňom trieda `IParser`, ktorá predstavuje rozhranie pre analýzu vstupnej gramatiky v súvislosti so správnym rozdelením pravidiel na ľavú a pravú stranu. Stačí potom implementovať toto rozhranie a následne ho v programe použiť. Takto implementované rozhranie už je schopné komunikovať so zvyškom programu a nie je potrebné kvôli tomuto meniť žiadne ďalšie triedy.

Ďalším vylepšením aplikácie by mohlo byť postupné vypisovanie množín s ich obsahom. Bolo by dobré, ak by sme videli ako sa množiny časom menia, ktoré neterminály sa do množín pridávajú a na základe akých pravidiel. Potom by sa táto aplikácia dala využiť aj ako učebná pomôcka. Už tento projekt sa týmto mal zaoberať, ale keďže sa autor práce zameral hlavne na rýchlosť algoritmu, vizuálna stránka aplikácie ostala nedoriešená. Ak by mal program zvládnuť aj názornú ukážku postupne sa meniacich množín, mohlo by to ohroziť rýchlosť vykonávajúceho sa algoritmu.

Ako už bolo viackrát spomenuté, výstupom programu je jednoduchá odpoveď: áno alebo nie, pravda alebo nepravda. Syntaktická analýza je len jednou z fáz komplexného prekladača. Síce sa jedná o najdôležitejšiu fázu, no bez ostatných komponentov sa v praxi nezaobíde. Výstupom bežne používajúcich syntaktických analyzátorov je derivačný strom. Tento strom predstavuje určitú štruktúru, podľa ktorej je nasledujúca fáza prekladu schopná vykonať určité operácie. Vytvorenie stromu je teda esenciálna záležitosť, ktorá v tomto projekte chýba. Ak by naprogramovaná aplikácia mala poslúžiť ako syntaktický analyzátor, ktorý by bol súčasťou niektorého prekladača, je potrebné dorobiť generovanie derivačného stromu. Algoritmus na generovanie derivačného stromu podľa doplnených CYK množín už bol vymyslený[11].

Ďalší vývoj, už ale mimo úprav implementovanej aplikácie, by sa mohol týkať samotného návrhu algoritmu. Bolo by vhodné formálne definovať časovú zložitosť. Ďalším krokom by mohol byť pokus o modifikáciu iných obecných metód syntaktickej analýzy. Mohlo by sa zistiť, či je možné upraviť aj niektoré iné obecné metódy do paralelnej podoby a či by to malo pozitívny dopad na rýchlosť vykonávania algoritmu.

Prínosom tejto práce je zistenie, že nie vždy je úprava algoritmu do podoby využívajúcej paralelizmus výhodná. Prvotné zdanie môže klamať. Aj keď v teórií to môže nádherne fungovať, v praxi nastávajú isté problémy. CYK algoritmus vykonáva pomerne triviálne operácie. Paralelizmus sa v tomto prípade veľmi neoplatí, pretože veľa času sa stráca práve pri práci s viacero vláknami, ako to bolo popísané v kapitole 9.6. Paralelné programovanie nie je úplne jednoduchou záležitosťou. V konkrétnych prípadoch si je potrebné uvedomiť, či to vôbec zmysel má. Možno sa v budúcnosti vymyslí iná technika paralelného programovania, ktorá by navrhnutému algoritmu dokázala pomôcť a zlepšiť tak terajšiu situáciu. Treba podotknúť, že v tomto projekte sa využívala len štandardná knižnica jazyka C++ pre prácu s vláknami. V súčasnosti existuje niekoľko knižníc, ktoré problém paralelizmu riešia možno efektívnejšie, napr. OpenMP[20] alebo Open MPI[21].

Literatura

[1] Meduna, A. *Elements of compiler design.* Boca Raton, [FL]: Auerbach Publications, 2008. ISBN 978-1-4200-6323-3.

[2] Aho, A. V., Lam, M. S., Sethi, R. a Ullman, J. D. *Compilers: principles, techniques & tools.* Second Edition. Boston, [MA]: Addison Wesley, 2007. ISBN 0-321-48681-1.

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

[4] Meduna, A. *Automata and Languages: Theory and Applications.* London: Springer, 2000. ISBN 81-8128-333-3.

[5] Jäger, G. a Rogers, J. Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences* [online]. 2012, roč. 367, č. 1598, s. 1956–1970 [cit. 2016-04-23]. Dostupné z: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3367686/>.

[6] Cunningham, W. Meta Programming. *Cunningham & Cunningham, Inc.* [online]. February 7, 2012 [cit. 2016-04-24]. Dostupné z: <http://c2.com/cgi/wiki?MetaProgramming>.

[7] Tucker, A. B. a Noonan, R. *Programming Languages: Principles and Paradigms.* New York, [NY]: McGraw-Hill, 2002. ISBN 0-07-238111-6.

[8] Chomsky, N. On certain formal properties of grammars. *Information and Control* [online]. 1959, roč. 2, č. 2, s. 137–167 [cit. 2016-04-24]. ISSN 0019-9958. Dostupné z: <http://www.sciencedirect.com.ezproxy.lib.vutbr.cz/science/article/pii/S0019995859903626>.

[9] Greibach, S. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *Journal of the ACM (JACM)* [online]. 1965, roč. 12, č. 1, s. 42–52 [cit. 2016-04-24]. ISSN 0004-5411. Dostupné z: <http://dl.acm.org.ezproxy.lib.vutbr.cz/citation.cfm?doid=321250.321254>.

[10] Meduna, A. a Lukáš, R. Syntaktická analýza shora dolů. *Formální jazyky a překladače* [online]. 2015 [cit. 2016-04-24]. Dostupné z: <https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IFJ-IT/lectures/Ifj-anim-pdf.zip?cid=9356>. Učebné materiály.

[11] Aho, A. V. a Ullman, J. D. *The Theory of Parsing, Translation, and Compiling: Volume I: Parsing.* Englewood Cliffs, [NJ]: Prentice-Hall, 1972. ISBN 0-13-914556-7.

[12] Meduna, A. a Lukáš, R. Normální formy a vlastnosti bezkontextových jazyků. *Formální jazyky a překladače* [online]. 2015 [cit. 2016-04-24]. Dostupné z: <https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IFJ-IT/lectures/Ifj-anim-pdf.zip?cid=9356>. Učebné materiály.

[13] Earley, J. *An Efficient Context-Free Parsing Algorithm*. Pittsburgh, [PA], 1968 [cit. 2016-04-28]. Dostupné z: <http://reports-archive.adm.cs.cmu.edu/anon/anon/usr/ftp/scan/CMU-CS-68-earley.pdf>. Dizertačná práca. Carnegie Mellon University, Carnegie Institute of Technology, College of Engineering and Science, Computer Science Department. Vedoucí práce Robert Floyd.

[14] Cocke, J. a Schwartz, J. T. *Programming languages and their compilers: Preliminary notes*. New York, [NY]: Courant Institute of Mathematical Sciences, 1970.

[15] Younger, D. H. Recognition and parsing of context-free languages in time n^3. *Information and Control* [online]. 1967, roč. 10, č. 2, s. 189–208 [cit. 2016-04-23]. ISSN 0019-9958. Dostupné z: <http://www.sciencedirect.com.ezproxy.lib.vutbr.cz/science/article/pii/S001999586780007X>.

[16] Kasami, T. *An efficient recognition and syntax-analysis algorithm for context-free languages*. Urbana, [IL]: University of Illinois, 1966.

[17] Free Software Foundation. Bison 3.0.4: Rules Syntax. *The GNU Operating System and the Free Software Movement* [online]. 23 January 2015 [cit. 2016-04-24]. Dostupné z: <https://www.gnu.org/software/bison/manual/html_node/Rules-Syntax.html#Rules-Syntax>.

[18] Gamma, E. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, [MA]: Addison-Wesley, 1995. ISBN 0-201-63361-2.

[19] Free Software Foundation. Optimize Options - Using the GNU Compiler Collection (GCC). *GCC, the GNU Compiler Collection* [online]. 1988-2016 [cit. 2016-04-25]. Dostupné z: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>.

[20] OpenMP Architecture Review Board. *OpenMP.org* [online]. 1998-2015 [cit. 2016-04-25]. Dostupné z: <http://openmp.org/wp/>.

[21] The Open MPI Project. *Open MPI: Open Source High Performance Computing* [online]. 2004-2016 [cit. 2016-04-25]. Dostupné z: <https://www.open-mpi.org/>.