BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

A REDUCTION OF FINITELY EXPANDABLE DEEP PUSHDOWN AUTOMATA

REDUKCE KONEČNĚ EXPANDOVATELNÉHO HLUBOKÉHO ZÁSOBNÍKOVÉHO AUTOMATU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. LUCIE CHARVÁT**
AUTOR PRÁCE

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

BRNO 2017

Master's Thesis Specification/6086/2016/xdvora1f

Brno University of Technology - Faculty of Information Technology

Department of Information Systems | Academic year 2016/2017

Master's Thesis Specification

For: **Dvořáková Lucie, Bc.**
Branch of study: Information Systems
Title: **Deep Pushdown Automata and Their Restricted Versions**
Category: Compiler Construction

Instructions for project work:

1. Study deep pushdown atomata and their restricted versions. Consult this study with your advisor.
2. Based upon instructions given by your advisor, define and study finitely expandable deep pushdown atomata with a restricted number of non-input symbols.
3. Study the properties of automata defined in 2. Compare their power with unrestricted versions of finitely expandable deep pushdown atomata.
4. Discuss applications of automata defined in 2. Consider complicated syntax structures that are non-context-free. Describe their parsing based on automata from 2.
5. Implement and test the parsing methods developed in 4.
6. Summarize the results. Discuss the future investigation concerning this project.

Basic references:

- Meduna, A.: Automata and Languages, Springer, London, 2000
- Rozenberg, G. and Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1 through 3, Springer, 1997, ISBN 3-540-60649-1
- Aho, A. V., Sethi, R., Ullman, J. D. : Compilers : principles, techniques, and tools, Addison-Wesley, 2nd ed., 2007, ISBN: 0321486811
- Meduna Alexander: Deep Pushdown Automata, Acta Informatica, roč. 2006, č. 98, DE, pp. 114-124

Requirements for the semestral defense:
Items 1 and 2.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Meduna Alexander, prof. RNDr., CSc.**, DIFS FIT BUT
Beginning of work: November 1, 2016
Date of delivery: May 24, 2017

L.S.

Dušan Kolář
Associate Professor and Head of Department

Abstract

For a positive integer n, n-expandable deep pushdown automata always contain no more than n occurrences of non-input symbols in their pushdowns during any compilation. As its main result, the present thesis demonstrates that those automata are as powerful as the same automata with #, which always appears solely as the pushdown bottom, and a single pushdown non-input symbol. An infinite hierarchy of language families follows from this result.

Abstrakt

Pro přirozené číslo n, n-expandovatelné hluboké zasobníkové automaty vždy obsahují maximálně n výskytů nevstupních symbolů v jejich zásobníku v průběhu jakékoli kompilace. Jako hlavní výsledek, tato práce demonstruje, že tyto automaty mají stejnou vyjadřovací sílu jako automaty s #, nacházející pouze na dně zásobníku, a jediným dalším nevstupním symbolem. Z tohoto závěru vyplývá nekonečná hierarchie jazyků přijímaných těmito automaty.

Keywords

Deep Pushdown Automata, Finite Expandability, Reduction, Non-Input Pushdown Symbols

Klíčová slova

Hluboké zásobníkové automaty, konečná expanze, redukce, nevstupní zásobíkové symboly

Reference

CHARVÁT, Lucie. *A Reduction of Finitely Expandable Deep Pushdown Automata.* Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Meduna Alexander.

A Reduction of Finitely Expandable Deep Pushdown Automata

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of prof. RNDr. Alexander Meduna, CSc. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Lucie Charvát
September 17, 2017

Contents

Chapter 1

Introduction

In essence, deep pushdown automata represent language-accepting models based upon new stack-like structures, which can be modified deeper than on their top. As a result, these automata can make expansions deeper in their pushdown lists as opposed to ordinary pushdown automata, which can expand only the very pushdown top. At present, the study of deep pushdown automata represent a vivid trend in formal language theory (see [10, 11, 1, 15]). The present theses makes a contribution to this trend.

This thesis narrows its attention to n-expandable deep pushdown automata, where n is a positive integer. In essence, during any computation, their pushdown lists contain #, which always appears as the pushdown bottom, and no more than $n - 1$ occurrences of other non-input symbols. As its main result, the thesis demonstrates how to reduce the number of their non-input pushdown symbols different from # to one symbol, denoted by \$, without affecting the power of these automata. Based on this main result, the thesis establishes an infinite hierarchy of language families resulting from these reduced versions of n-expandable deep pushdown automata. More precisely, consider n-expandable deep pushdown automata with pushdown alphabets containing #, \$, and input symbols. The thesis shows that $(n + 1)$-expandable versions of these automata are stronger than their n-expandable versions, for every positive integer n. In addition, it points out that these automata with # as its only non-input symbol characterize the family of regular languages.

After the correctness of the reduction has been proven, the thesis brings its attention to practical application of the result. The reduced finitely expandable deep pushdown automaton parses the same language as its non-reduced counterpart. The thesis therefore focuses on typical languages accepted by this kind of automata, especially the ones that are context-sensitive. After analysis of the syntax structure of the languages, the thesis demonstrates practical use of the automata during parsing of structures that are commonly used in programming languages. We show implementation of a simulator that is capable of parsing and generating an output to showcase the conversion and the parsing process.

The thesis is organized as follows. Chapter 2 gives definitions needed to follow the thesis. Chapter 3 establishes the results sketched above. Chapter 4 then shows an example of how these automata can be applied as language acceptors and detailed explanation of the implementation. At last, Chapter 5 includes final thoughts on the work done within this thesis.

Chapter 2

Preliminaries and Definitions

This chapter will gradually introduce terms necessary to follow this thesis. In the first section, we will go over the basic terminology and establish notion of formal languages. The following sections then provide detailed explanation of a hierarchy of formal grammars and languages they can generate together with automata that are then able to accept such languages. The last section covers the special types of grammars and automata that are not as commonly known but are important to fully understand the concept of this thesis and unify the definitions that might differ across the publications. If not stated otherwise, the definitions used in this chapter are based on the ones that are provided in [8, 14, 16].

2.1 Language and Regular Set

In this section we introduce the basic terms that relate to formal language theory and how to create and manipulate language in general.

2.1.1 Formal Language

The base for introducing a concept of the language are notions an *alphabet* and a *string*. An alphabet is a nonempty set of elements called *symbols of the alphabet*. In some cases we can even work with infinite alphabets, but in the following applications we will use finite alphabets only.

A *string* (also a *word* or a *sentence*) over given alphabet is any finite sequence of symbols of the alphabet. An empty sequence of symbols, i.e. the sequence with no symbol is called an empty string. The empty string will be denoted by ε. We can formally define strings over an alphabet Σ in the following manner:

1. The empty string ε is a string over Σ,
2. if x is a string over Σ and $a \in \Sigma$, then xa is string over Σ,
3. y is a string over Σ if and only if y can be constructed using rules (1) and (2).

Let x and y be strings over Σ. The *concatenation* of strings x and y is the string xy (the string y is attached to the string x). The concatenation is obviously associative, i.e., $x(yz) = (xy)z$, but not commutative, $xy \neq yx$.

The *length of a string* is a nonnegative integer representing number of symbols in the string. The length of the string x is denoted by $|x|$. If $x = a_1a_2...a_n, a_i \in \Sigma$ for $i = 1, ...n$, then $|x| = n$. The length of the empty string equals zero, i.e. $|\varepsilon| = 0$.

A *power* operator, a^n, where a is a string and n is an integer denotes n concatenations of string a. $a^n = aa^{n-1}$ and $a^0 = \varepsilon$, so $|a^n| = n|a|$.

Let Σ be an alphabet. By the symbol Σ^* we denote the set of all strings over alphabet Σ including the empty string, by the symbol Σ^+ we denote the set of all strings over alphabet excluding the empty sting, i.e. $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. The set L, such that $L \subseteq \Sigma^*$ (eventually $L \subseteq \Sigma^+$, if $\varepsilon \in L$), is called a *language* L over alphabet Σ. Thus it means that a language is any subset of strings over the given alphabet. A string x, $x \in L$, is called the sentence of the language L.

A *language family* is a set of languages that share a defined property. The property used in this thesis is a model of the language. Language family described by model M is $L(M)$.

2.1.2 Regular Set

Let Σ be a finite alphabet. The class of regular sets is the smallest class of languages which contains the sets $\varnothing$, $\{\varepsilon\}$, $\{a\}$ for all symbols a and which is closed under the following operations: union, product and iteration. More precisely, one can recursively define *regular sets* over the alphabet Σ as follows:

1. $\varnothing$ (the empty set) is a regular set over Σ
2. $\{\varepsilon\}$ (the set containing only the empty string) is a regular set over Σ
3. $\{a\}$ for all $a \in \Sigma$ is regular set over Σ
4. if P and Q are regular sets over Σ, then $P \cup Q$, $P \cdot Q$ and P^* where $P \cdot Q = \{xy \mid x \in P \wedge y \in Q\}$, $P^* = \bigcup_{n=0}^{\infty} P^n$, $P^0 = \{\varepsilon\}$, and $P^n = \underbrace{P \cdot P \cdot \ldots \cdot P}_{n\text{-times}}$ are also regular sets over Σ
5. Regular sets are just the sets which arise by application of the rules 1–4.

2.1.3 Regular Expression

In order to simplify notation, we also introduce notion of *regular expression*. Regular expressions over Σ are recursively defined as

1. $\varnothing$ is a regular expression denoting the regular set $\varnothing$,
2. ε is a regular expression denoting the regular set $\{\varepsilon\}$,
3. a is a regular expression denoting the regular set $\{a\}$ for all $a \in \Sigma$,
4. if p, q are regular expressions denoting the regular sets P and Q, then
   (a) $(p + q)$ is regular expression denoting the regular set $P \cup Q$,
   (b) (pq) is regular expression denoting the regular set $P \cdot Q$,
   (c) (p^*) is regular expression denoting the regular set P^*.
5. There are no other regular expressions over Σ.

2.2 Formal Grammar

Grammar describes a set of rules that applied can generate a language. Base on the structure of the rules we can determine how powerful the grammar can be and what type of languages it could potentially generate.

2.2.1 Definition

Grammar is a quadruple

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

where

- N is a finite alphabet of nonterminals
- T is a finite alphabet of terminals, $N \cap T = \varnothing$
- $R \subset (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$ is a finite set of rewriting rules
- $S \in N$ is the start symbol of the grammar

Let $G = (N, T, R, S)$ be a grammar and let λ and μ be strings from $(N \cup T)^*$. The strings λ and μ are in binary relation $\Rightarrow_G$ called the *direct derivation* if and only if the strings λ and μ can be written in the form

$$\lambda = \gamma\alpha\delta$$

$$\mu = \gamma\beta\delta$$

where γ and δ are strings from $(N \cup T)^*$ and $\alpha \to \beta$ is a rewriting rule from P. If the strings λ and μ satisfy the relation of direct derivation, then we will write $\lambda \Rightarrow_G \mu$ and we say that the string μ can be directly derived the string λ in the grammar G. When it is clear which grammar we are talking about, we shall drop an index under symbol $\Rightarrow$.

2.2.2 Grammars as Language Generators

Let $G = (N, T, P, S)$ be a grammar and let λ and μ be strings from $(N \cup T)^*$. The strings λ and μ are in binary relation $\Rightarrow^+$ called *derivation* if and only if there is a sequence of direct derivations $v_{i-1} \Rightarrow v_i$ $i = 1, ..., n$, $n \geq 1$ such that:

$$\lambda = v_0 \Rightarrow v_1 \Rightarrow ... \Rightarrow v_{n-1} \Rightarrow v_n = \mu$$

This sequence is called the *derivation of length n*. If $\lambda \Rightarrow^+ \mu$ then we say that string μ is *generated* from the string λ in G. The relation $\Rightarrow^+$ is obviously the transitive closure of the relation direct derivation $\Rightarrow$. The symbol $\Rightarrow^n$ denotes n power of relation $\Rightarrow$.

If for strings λ and μ hold $\lambda \Rightarrow^+ \mu$ or $\lambda = \mu$ in G then we write $\lambda \Rightarrow^* \mu$. The relation $\Rightarrow^*$ is the transitive and reflexive closure of the relation of direct derivation $\Rightarrow$.

Let $G = (N, T, P, S)$ be a grammar. The string $\alpha \in (N \cup T)^*$ is called the *sentential form* if $S \Rightarrow^* \alpha$, i.e. the string α is generated from the start symbol S. The sentential form composed from terminal symbols only is called the *sentence*. *Language* $L(G)$, generated by the grammar G is the set of all sentences $L(G)$:

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

2.2.3 Chomsky Hierarchy

The *Chomsky hierarchy* uses restrictions on grammar rewrite rules to classify languages into four types, type 0, type 1, type 2 and type 3.

- *Type 0* also called *Unrestricted grammars* (U) follow the definition as

$$\alpha \to \beta, \alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*, \beta \in (N \cup \Sigma)^*$$

  and generate the family of *recursively enumerable languages*, **RE**. Example of such grammar is

$$\begin{aligned} G &= (\{A, B\}, \{a, b\}, R, S) \\ R &= \{A \to AbB \mid a, \\ &\quad AbB \to baB \mid BAbB, \\ &\quad B \to b \mid \varepsilon\} \end{aligned}$$

- *Type 1* also called *Context sensitive grammars* (CS) have rules in the following form,

$$\alpha A \beta \to \alpha \gamma \beta, A \in N, \alpha, \beta \in (N \cup \Sigma)^*, \gamma \in (N \cup \Sigma)^+ \text{ or } S \to \varepsilon$$

  The rules of context-sensitive grammar take inconsideration the surrounding symbols (the context) of the expanding non-input symbol. Context sensitive grammars generate the family of *context sensitive languages*, **CS**. Example of such grammar is

$$\begin{aligned} G &= (\{A, S\}, \{0, 1\}, R, S) \\ R &= \{S \to 0A1 \mid 01, \\ &\quad 0A \to 00A1 (\alpha = 0, \beta = \varepsilon, \gamma = 0A1) \\ &\quad A \to 01 \end{aligned}$$

- Rules of *type 2* also called *Context-free grammars* (CF) are in form of

$$A \to \gamma, A \in N, \gamma \in (N \cup \Sigma)^*$$

  Compared to context sensitive grammar the non-input symbol A can be expanded independently on the symbols surrounding it. Context-free grammars generate the family of *context-free languages*, **CF**. Example of grammar is

$$\begin{aligned} G &= (\{S\}, \{0, 1\}, R, S) \\ S &\to 0S1 \mid \varepsilon \end{aligned}$$

- Rules of *type 3* also called *right-linear grammars* are of form,

$$A \to xB \text{ or } A \to x; A, B \in N, x \in \Sigma^*$$

  Right-linear grammar can be transformed to the special right-linear grammar with rules of the form:

$$A \to aB \text{ or } A \to a; A, B \in N, a \in \Sigma \text{ or } S \to \epsilon$$

This type of grammar is called the *regular grammar*, specifically right-regular grammar. Hence grammars of type 3 are also called *regular grammars* **REG**. Example of type 3 grammar is

$$\begin{aligned} G = (\{A, B\}, \{a, b, c\}, R, A) \\ A \to aaB \mid ccB \\ B \to bB \mid \varepsilon \end{aligned}$$

2.3 Automata

Automata are the finite representation of formal language which can be infinite. Automata are categorized base on languages they can accept and similarly to grammars can be categorized based on Chomsky hierarchy. Automata are defined by sets of states and rules that determine the behaviour of the automaton and therefore the languages it can accept.

In this section we will introduce three basic types of automaton. The first is the finite-state machine that has no extra memory and can only accept regular languages. The pushdown automaton, part of which is a pushdown (stack) that allows temporal save of certain symbols and therefore expands the set of languages this automaton can accept to context-free languages. Lastly the Turing machine that includes rewritable tape giving the automaton largest acceptance power.

2.3.1 Finite-State Machine

A *finite-state machine* is an automaton with no extra memory and decision whether it accepts language or not is solely based on fact if there is a suitable transition to a next state. Therefore it can only recognise regular languages.

A finite-state machine (FSM) M is a quintuple

$$M = (Q, \Sigma, R, s, F)$$

where

- Q is a set of states;
- Σ is the finite input alphabet;
- $R \subset Q \times \Sigma \times Q$ is a finite set of transition rules;
- $s \in Q$ is a starting state;
- $F \subset Q$ is a finite set of accepting states.

A *computation step* of FSM, $\vdash$, is an application of a transition rule from R, i.e.,

$$paw \vdash qw \text{ if } (pa \to q \in R)$$

where $w \in \Sigma^*$. $\vdash^+$ is a transitive closure, $\vdash^*$ is a reflexive transitive closure, and $\vdash^k$, for $k \geq 0$, is the kth power of the binary relation $\vdash$.

The language accepted by M is

$$L(M) = \{w \mid w \in \Sigma^*, sw \vdash^* f\varepsilon, f \in F\}$$

The family **FSM** of languages accepted by FSM is the same family as the one that is generated by regular grammars, i.e., **FSM** = **REG**.

2.3.2 Pushdown Automaton

A *pushdown automaton*, as the name suggest, contains a pushdown to store certain values. This possibility increases the acceptance power of an automaton to be able to accept context-free languages.

A pushdown automaton (PDA) M is a 7-tuple

$$M = (Q, \Sigma, \Gamma, R, s, S, F)$$

- Q, Σ, s and F are defined like in the finite automaton;
- Γ is a pushdown alphabet, Σ is a subset of Γ, there is a bottom symbol $\#$ in $\Gamma \setminus \Sigma$;
- $S \in \Gamma$ is the start pushdown symbol;
- The transition relation R is a finite set that contains elements of

$$(Q \times (\Gamma \setminus (\Sigma \cup \{\#\}))) \times (Q \times (\Gamma \setminus \{\#\})^+)$$

  and of

$$(Q \times \{\#\}) \times (Q \times (\Gamma \setminus \{\#\})^* \{\#\})$$

A *configuration* of a pushdown automaton M is a triplet $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma$ where

- q is a current state of the control unit.
- w is yet unread part of input string; the first symbol of w is under the reading head. If $w = \varepsilon$, then all the symbols of input string have been already read.
- α is a content of the stack. Unless otherwise indicated, we represent the stack as a string, whose left most symbol corresponds to the top of the stack. If $\alpha = \varepsilon$, then the stack is empty.

A *transition* of a pushdown automaton M is represented by a binary relation $\vdash$ which is defined on a set of configurations of the pushdown automaton M. The relation

$$(q, aw, S\alpha) \vdash (q', w, \gamma\alpha)$$

holds, if $(q, a, S, q', \gamma) \in R$ for some $q \in Q$, $a \in (\Sigma \cup \varepsilon)$, $w \in \Sigma^*$, $Z \in \Gamma$ and $\alpha\gamma \in \Gamma^*$

We interpret the relation $\vdash$ this way. If the pushdown automaton M is in a state q and a symbol S is at the top the stack, then after reading an input symbol $a \neq \varepsilon$, the automaton can move to a state q', whereas the reading head moves to right and the symbol S from the top of the stack is replaced by a string γ. If $a = \varepsilon$, the reading head does not move, which means, that the automaton transitions to a new state and a new content of the stack is not determined by the next input symbol. This type of transition is called *ε-transition*. Note that ε-transition can happen also when all the input symbols have been already read.

The relations $\vdash^i$, $\vdash^+$, $\vdash^*$ are defined as usual. The relation $\vdash^+$, resp. $\vdash^*$ is transitive resp. transitive and reflexive closure of the relation $\vdash$, $\vdash^i$ is i-th power of the relation $\vdash$, $i \geq 0$.

An *initial configuration* of a pushdown automaton is of the form (s, w, S) for $w \in \Sigma^*$, e.g., the automaton is in the initial state s, the string w is on the input tape and the start symbol S is on the stack. A final configuration is of form (q, ε, α), where $q \in F$ is the final state and $\alpha \in \Gamma^*$.

The language accepted by M is

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

The family **PDA** of languages accepted by PDA is the same family as the one that is generated by context-free grammars, i.e., **PDA** = **CF**.

2.3.3 Turing Machine

A Turing machine consists of a finite-state control unit, unidirectionally unbounded tape and reading / writing head. In one computation step, a Turing machine first reads the symbol under the head. Consequently, depending upon the symbol currently read and the state of the control unit, it can rewrite the read symbol, change the state and move the head right or left by one cell.

A *Turing machine* (TM) is a 7-tuple of the form $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where:

- Q is a finite set of control states,
- Σ is a finite set of symbols called input alphabet, $\Delta \notin \Sigma$,
- Γ is a finite set of symbols, $\Sigma \subset \Gamma$, $\Delta \in \Gamma$,
- δ is a partial function $(Q \setminus \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times (\Gamma \cup \{L, R\})$, where $L, R \notin \Gamma$, called the transition function,
- q_0 is the initial state, $q_0 \in Q$ and
- q_{accept} is the final accepting state, $q_{accept} \in Q$
- q_{reject} is the final rejecting state, $q_{reject} \in Q$

The symbol Δ denotes the so-called *blank* (empty symbol) which occurs in tape areas, that haven't been used yet (but can be also written on tape later).

A *configuration* of the tape is a pair consisting of an infinite string representing the tape contents and a head position on this string—more precisely, it is an element of the set $\{\gamma\Delta^\omega \mid \gamma \in \Gamma^*\} \times \mathbb{N}$. We write the configuration of the tape as $\Delta xyz\underline{z}\Delta x\Delta\Delta...$ (the underline marks the head position). A *configuration of the machine* consists of the control state and the tape configuration—formally, it is an element of the set $Q \times \{\gamma\Delta^\omega \mid \gamma \in \Gamma^*\} \times \mathbb{N}$.

For an arbitrary string $\gamma \in \Gamma^\omega$ and a number $n \in \mathbb{N}$ we denote by γ_n the symbol on the position n of the string γ and by $s_b^n(\gamma)$ the string that is formed from γ by substituting γ_n for b.

A *computation step* of a TM M is defined as a binary relation ${}_M\vdash$ such that $\forall q_1, q_2 \in Q$ $\forall \gamma \in \Gamma^\omega$ $\forall n \in \mathbb{N}$ $\forall b \in \Gamma$:

- $(q_1, \gamma, n) \vdash (q_2, \gamma, n+1)$ for $\delta(q_1, \gamma_n) = (q_2, R)$, i.e., an operation of *a move to the right* when γ_n is under head
- $(q_1, \gamma, n) \vdash (q_2, \gamma, n-1)$ for $\delta(q_1, \gamma_n) = (q_2, L)$ and $n > 0$, i.e., an operation of *a move to the left* when γ_n is under head
- $(q_1, \gamma, n) \vdash (q_2, s_b^n(\gamma), n-1)$ for $\delta(q_1, \gamma_n) = (q_2, b)$, i.e., an operation of *writing* b when γ_n is under head

The language accepted by M is

$$L(M) = \{w \mid (q_0, \Delta w \Delta\Delta \ldots, 1) \vdash^* (q_{accept}, x, i), x \in \{\gamma\Delta^\omega \mid \gamma \in \Gamma^*\}, i \in \mathbb{N}\}$$

The family **TM** of languages accepted by TM is the same family as the one that is generated by unrestricted grammars, i.e., **TM** = **RE**.

2.4 Special Types of Grammars and Automata

This section cover the grammars and automata that are not as commonly known but as important to understand the outcome of this thesis. The section wants to make sure reader understands why these structures and/or modification are necessary and provides context between them.

2.4.1 State Grammar

A *state grammar* G is a context-free grammar extended by additional mechanism that influences the choice of the rule during every derivation step. If application of rule always takes place within the first n occurrences of nonterminals, G is referred to as n-limited.

A *state grammar* is quintuple

$$G = (N, W, T, R, S)$$

where

- N is a finite alphabet of nonterminals
- W is a finite set of states
- T is a finite alphabet of terminals, $N \cap T = \varnothing$
- $R \subseteq (W \times N) \times (W \times (N \cup T)^+)$ is finite set of relations
- $S \in N$ is the start symbol

Instead of $(q, A, p, v) \in P$, we write $(q, A) \to (p, v) \in R$ throughout. For every $z \in (N \cup T)^*$, define

$$states_G(z) = \{q \in W \mid (q, A) \to (p, v) \in R, A \text{ is a nonterminal in } z\}$$

If $(q, A) \to (p, v) \in R$, $x, y \in V^*$, $states_G(x) = \varnothing$, then G makes a *derivation step* from (q, xAy) to (p, xvy), symbolically written as

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

In addition, if n is a positive integer and xA contains n or less nonterminals, we say that $(q, xAy) \Rightarrow (p, xvy)$ is n-limited, symbolically written as

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

In the standard manner we extend $\Rightarrow$ to $\Rightarrow^m$, $m \geq 0$. Based on $\Rightarrow^m$ we can define $\Rightarrow^+$ and $\Rightarrow^*$.

Let $n \in \mathbb{N}+$ and $\alpha, \beta \in (W \times (N \cup T)^+)$. To express that every derivation step in $\alpha \Rightarrow^m \beta$, $\alpha \Rightarrow^+ \beta$ and $\alpha \Rightarrow^* \beta$ is n-limited, we write $\alpha_n \Rightarrow^m \beta$, $\alpha_n \Rightarrow^+ \beta$ and $\alpha_n \Rightarrow^* \beta$ respectively.

The language of G, $L(G)$, is defined as

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

Moreover, we define for every $n \geq 1$

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

A derivation of the form $(q, S)_n \Rightarrow^* (p, w)$, where $q, p \in W$ and $w \in T^*$, represents a successful n-limited generation of w in G.

2.4.2 Matrix Grammars

A *matrix grammar* defines a sequence of rules that must be applied in given order instead just one as in standard manner. This enables the grammar to expand the set of languages it can generate.

A matrix grammar is a quintuple $G = (N, T, M, S, F)$, where

- N is a finite alphabet of nonterminals
- T is a finite alphabet of terminals, $N \cap T = \varnothing$
- $M = m_1, m_2, ..., m_n$ is a finite set of finite sequences of context-free rules (i.e., for $1 \leq i \leq n$, $m_i = (A_{i,1} \to w_{i,1}, A_{i,2} \to w_{i,2}, \ldots, A_{i,r_i} \to w_{i,r_i})$ for some $r_i \geq 1$, $A_{i,j} \in N$, $w_{i,j} \in (N \cup T)^*, 1 \leq j \leq r_i$)
- $S \in N$ is the start symbol
- F is a subset of the rules occurring in the matrices m_i, $1 \leq i \leq n$

Now, let $m = (A_1 \to w_1, A_2 \to w_2, \ldots, A_r \to w_r) \in M$. We say that $y \in (N \cup T)^*$ is derived from $x \in (N \cup T)^+$ by m (and write $x \Rightarrow y$), if there exist words $x_1, x_2, \ldots, x_{r+1}$ such that

1. $x = x_1$ and $y = x_{r+1}$
2. for all $0 \leq i \leq r$:
   (a) either $x_i = x_i' A_i x_i''$ and $x_{i+1} = x_i' w_i x_i''$, or
   (b) A_i does not occur in x_i, $x_{i+1} = x_i$ and $A_i \to w_i \in F$

In the standard manner we extend $\Rightarrow$ to $\Rightarrow^n$, $n \geq 0$. Based on $\Rightarrow^n$ we can define $\Rightarrow^+$ and $\Rightarrow^*$. The language $L(G)$ generated by G is defined in standard way as

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

In this thesis we use an alteration of this grammar *matrix grammars of finite index*. A matrix grammar of index k for a positive integer is one whose sentential form must not contain more than k nonterminals in any derivation step, otherwise the derivation is stopped and not successful. In the following explanation, we will use $\mathbf{MG}_k$ in order to denote a family of languages generated by matrix grammars of index k.

2.4.3 Deep Pushdown Automaton

A *deep pushdown automaton* works very similarly to the regular pushdown automaton main difference being the ability to access values deeper on the stack. This option can significantly enlarge the acceptance power of such automaton since now it can accept certain subsets of context-sensitive languages. More precisely, they give rise to an infinite hierarchy of language families coinciding with the hierarchy resulting from n-limited state grammar.

A deep pushdown automaton (deep PDA) is a 7-tuple

$$M = (Q, \Sigma, \Gamma, R, s, S, F)$$

where

- Q, Σ, Γ, s, S and F are defined like in the pushdown automaton
- In what follows, $N = \Gamma \setminus (\Sigma \cup \{\#\})$. The transition relation R is a finite set that contains elements of
$$(\mathbb{N} \times Q \times N \times Q \times (\Gamma \setminus \{\#\})^+)$$
and of
$$(\mathbb{N} \times Q \times \{\#\} \times Q \times (\Gamma \setminus \{\#\})^*\{\#\})$$
where $I = \{i : 1 \leq i \leq n\}$ for some n. We will write $mqA \to pv \in R$ instead of $(m, q, A, p, v) \in R$.

A *configuration* of M is any member of $Q \times \Sigma^* \times (\Gamma \setminus \{\#\})^*\{\#\}$. Let Ξ denote the set of all configurations of M. Next, we define three binary relations over Ξ — ${}_p\vdash$, ${}_e\vdash$, and $\vdash$. Let $q, p \in Q$, $x \in \Sigma^*$, $z \in (\Gamma \setminus \{\#\})^*\{\#\})$.

1. Let $a \in \Sigma$; then, $(q, ax, az)\ {}_p\vdash\ (q, x, z)$.
2. Let $mqA \to pv \in R$, $z = uAw$, $u \in (\Gamma \setminus \{\#\})^*$, u contains $m-1$ occurrences of symbols from N, either $A \in N$, $v \in (\Gamma \setminus \{\#\})^+$ and $w \in (\Gamma \setminus \{\#\})^*\{\#\}$ or $A = \#$, $v \in (\Gamma \setminus \{\#\})^*\{\#\}$, and $w = \varepsilon$; then, $(q, x, uAw)\ {}_e\vdash\ (p, x, uvw)$.
3. Let $\alpha, \beta \in \Xi$; $\alpha \vdash \beta$ if and only if $\alpha\ {}_p\vdash\ \beta$ or $\alpha\ {}_e\vdash\ \beta$.

Intuitively, in ${}_p\vdash$ and ${}_e\vdash$, p and e stand for *pop* and *expansion*, respectively. Consider 2 above; to express that $(q, x, uAw)\ {}_e\vdash\ (q, x, uvw)$ is made according to $mqA \to pv$, write $(q, x, uAw)\ {}_e\vdash\ (p, x, uvw)\ [mqA \to pv]$. If $\alpha, \beta \in \Xi$, $\alpha \vdash \beta$ in M, we say that M makes a *move* from α to β. In the standard manner, extend ${}_e\vdash$, ${}_p\vdash$, and $\vdash$ to ${}_e\vdash^i$, ${}_p\vdash^i$, and $\vdash^i$, respectively, for $i \geq 0$; then, based on ${}_e\vdash^i$, ${}_p\vdash^i$, and $\vdash^i$, define ${}_e\vdash^+$, ${}_e\vdash^*$, ${}_p\vdash^+$, ${}_p\vdash^*$, $\vdash^+$, and $\vdash^*$. The language of M, $L(M)$, is defined as

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

The acceptance power of the deep pushdown automata that make expansion of depth m or less, where $m \geq 1$, are equivalent to m-limited state grammars so these automata accept a proper subfamily of the language family accepted by deep pushdown automata that make expansions of depth $m+1$ or less, thus creating infinite hierarchy of language families occurring between the families of context-free and context-sensitive languages. However, there always exists some context-sensitive languages that cannot be accepted by any deep pushdown automata that make expansions of depth n or less, for every positive integer n. Rigorous proof of these statements can be found in section 18.2 Accepting Power of [16].

2.4.4 Finitely Expandable Deep Pushdown Automaton

An *n-expandable deep pushdown automaton* is an 8-tuple

$$M = (Q, \Sigma, \Gamma, R, s, S, F, n)$$

where all components are as for the original deep PDA, and n is a positive integer. Expanding transition can only be applied in a step $x_e \vdash y$ if y will not contain more than n non-input symbols. Otherwise either some other transition must be applied or the computation stops without accepting. A deep PDA that is n-expandable for some n is called *finitely expandable*.

This modification is very important since it ensures the number of non-input symbol on stack is never infinite, witch would be possible given a regular deep PDA. It would not be possible to create automaton with the reduction introduced in this thesis without this condition.

Let $n, r \in \mathbb{N}$, ${}_n\mathbf{DPDA}$ denotes the language family accepted by n-expandable deep pushdown automata. ${}_n\mathbf{DPDA}_r$ denotes the language family accepted by n-expandable deep pushdown automata with # and no more than $(r-1)$ non-input pushdown symbols.

In Section 3.1 of the thesis, we provide a proof that finitely expandable deep pushdown automata has the same acceptance power as matrix grammars of finite index.

2.5 Normal and Reduced Forms

Normal and reduced forms of grammars and automata typically refers to their simpler and more restricted subset. Having a grammar or automaton in such a well-specified form can significantly ease construction of proofs and reasoning about these entities [17]. Since the main result of the thesis contributes to this effort in the terms of finitely expandable deep pushdown automata, this section defines and describes applicability of the most commonly used normal and reduced forms of context-free grammars (Chomsky and Greibach) and regular automata (minimal FSM).

2.5.1 Chomsky Normal Form

A context-free grammar $G = (N, T, P, S)$ is in Chomsky normal form when every rule from P is of the form $A \to BC$ or $A \to a$, where a is a terminal, and A, B, and C are non-terminals. Further B and C are not the starting non-terminals. Additionally, a special rule $S \to \varepsilon$ is permitted where S is the start non-terminal so a language generated by the grammar can contain an empty string. In [3], it is shown that every context-free grammar can be transformed into such form. The main idea of the algorithm that converts a grammar to this form involves:

1. Determining all nullable variables and getting rid of all ε-productions
2. Getting rid of all non-terminal unit productions
3. Breaking up long productions
4. Moving terminals to unit productions

The key advantage is that in Chomsky normal form, every derivation of a string of n letters has exactly $2 \cdot n - 1$ steps. Therefore one can determine if a string is in the language

generated by the grammar by exhaustive search of all its derivations. This form of grammar is practically applied in Cocke-Younger-Kasami algorithm [4, 18, 12] for recognition and parsing of context-free languages (including non-deterministic ones). Another application of the form can be seen in proof of the so-called Pumping Lemma [2] that is practically used to show that a given language is not context-free.

2.5.2 Greibach Normal Form

A context-free grammar $G = (N, T, P, S)$ is said to be in Greibach normal form if every production rule is of the form $A \to ax$, where a is a terminal, A in non-terminal, and x is a string of over non-terminals (including empty string). The proof that every context-free grammar can be transformed into this form is shown in [7]. The main idea of the algorithm that converts a grammar to this form involves:

1. Removal of a left-recursion and getting rid of all ε-productions
2. Finding a linear order on a set of non-terminals
3. Substitution of productions with respect to the order from the previous step so all rules are in form $A \to ay$ where a is a terminal, A in non-terminal, and y is a string over terminals and non-terminals.
4. Transformation of the productions $A \to ay$, $A \in N$, $a \in T$, $y \in (N \cup T)^*$ to $A \to ax$, $x \in N^*$.

The key advantage of Greibach Normal Form is that a grammar in such a form produces precisely one symbol of a generated string in each of its derivation steps. This fact was used in order to prove that every context-free grammar can be recognized by a pushdown automaton that works in real time (i.e., without ε-transitions) and thus efficiently parsed [7].

2.5.3 Minimal Finite State Machine

For each regular language that can be accepted by a finite state machine, there exists a minimal finite state machine with a minimum number of states which is unique (except that states can be given different names) [9]. Given a deterministic finite state machine there are three classes of states that can be removed from the original machine without affecting the language it accepts in order to minimize it:

- Non-terminating states – States for which there does not exist a string that takes the machine to its final state.
- Unreachable states – States that are not reachable from the initial state of the machine, for any input string.
- Non-distinguishable states – States are those that cannot be distinguished from one another for any input string.

Finite state minimization is usually done in three steps, corresponding to the removal / merger of the relevant states in the order presented above. The minimal DFA ensures minimal computational cost for tasks such as pattern matching. Moreover, since the minimal finite state machine is unique, the minimization can be used, e.g., to show that two regular expressions match the same strings.

Chapter 3

Result

This chapter focuses on the main result of the thesis. Its first part establishes the infinite hierarchy of language families that coincides with the hierarchy resulting from the matrix grammars of finite index. This proof is based on paper by Meduna [13]. Next section then contains an original proof that reduced version of the n-expandable deep pushdown automata have the exact same accepting power as their non-reduced counterparts. This proof is also being published as a paper in Schedae Informaticae [6].

3.1 Hierarchy of Language Families Generated by Matrix Grammars

At start, we establish a lemma that helps simplify the upcoming proof. Its rigorous proof can be found in [13].

Lemma 3.1.1 *Any n-expandable deep PDA M can accept every $w \in L(M)$ so all expansions precede all pops during the accepting process without any loss of generality.*

Next, we present a theorem and a simplified version of its proof that was introduced in [13] as it contains details that are necessary for understanding of following explanation.

Theorem 3.1.1 *For all integers $n > 0$, matrix grammars of index n generate the same class of languages that is accepted by n-expandable deep push down automata, i.e.,* $\mathbf{MG}_n = {}_n\mathbf{DPDA}$.

Proof 3.1.1 *In the following text, we assume that all the transitions we define below will be defined for all possible depths, i.e. for all positions from one to n. Let $G = (N, T, M, S)$ be a matrix grammar of index n. Our deep PDA is $M = (\{s\} \cup (M \times \{1, 2, \ldots, \ell\}), T, \{S\} \cup N \cup T, R, s, S, \{s\}, n)$, where ℓ is the maximal number of rules in a matrix of G. The only component left to define is the transition function R. For every matrix $m : A_1 \to v_1, A_2 \to v_2, \ldots, A_k \to v_k$ from M it contains the following transitions where m_i denotes the element (m, i) from the set of states:*

$$sA_1 \to m_1v_1, m_1A_2 \to m_2v_2, m_2A_3 \to m_3v_3, \ldots$$

$$\ldots, m_{k-2}A_{k-2} \to m_{k-1}v_{k-2}, m_{k-1}A_k \to sv_k$$

The states m_i are used only in these transitions simulating m, thus there is only one possible transition for each of these states. This means that the automaton either executes

the entire sequence of transitions corresponding to the matrix m, or it will stop without accepting the word.

To see that G and M are equivalent, let us look at the sentential forms of a derivation and the corresponding stack contents. Both start with S. As stated above, rules of the grammar and expanding transitions of the deep PDA enact exactly the same changes. Thus as long as we do not use any pops, the sentential forms and the corresponding stack contents are identical, and it is obvious that every terminal word that can be derived by the grammar can also be generated on the stack. Lemma 3.1.1 shows that all computations of M can be normalized in this manner. Finally, after finishing the simulation of a matrix, M is always in its final state s and can thus accept the word on the stack if it consists only of terminals and matches the input word. Therefore the deep PDA accepts the same language that the grammar generates.

For showing the inverse inclusion, we need a more sophisticated argumentation. We will construct a matrix grammar for a given deep PDA, and again the sentential forms and the stack contents will be in close correspondence. However, since the grammar does not distinguish between different positions of non-terminals while the automaton does, we need to store information about their position in the non-terminals themselves. Whenever new ones are introduced or when one is rewritten to a string of only terminals, the following ones need to be updated, because their positions change. This will be ensured by putting all the rules involved in one matrix.

We now proceed to define a matrix grammar simulating an n-expandable deep PDA $(Q, \Sigma, \Gamma, R, s, S, F, n)$. The set of non-terminals will be $N \times \{2, \ldots, n\} \cup N \times Q \cup \{\varepsilon\} \times \{2, \ldots, n\} \cup \{S\}$, where the second component will contain the position of a given stack symbol, if it is not the first one, or it will contain the state, if the symbol is the first one. The start symbol will be S, and the matrix $(S \to (S, s)(\varepsilon, 2)(\varepsilon, 3) \cdots (\varepsilon, n))$ is the only one containing a rule rewriting S. The symbols with ε are placeholders for further non-terminals.

For the matrices simulating the automaton's transitions, the simplest case is when exactly one non-terminal is on the right-hand side of the rule. Then the positions of non-terminals in the other parts of the sentential form are not affected. So for every transition $kpA \to quBv$ with $uv \in T^*$ and we add the matrices $((X, p) \to (X, q), (A, k) \to u(B, k)v)$ for all non-terminals X and all k between two and n. The first rule changes the state that is stored in the sentential form's first non-terminal, the second rule does the expansion at position k. If $k = 1$, then the matrix is singleton and contains the rule $(A, p) \to u(B, q)v$, which rewrites the non-terminal and changes the state in a single step.

If a rule of the grammar produces more than one non-terminal, then the depth in the simulated stack of all the non-terminals to the right of that application side is changed. Therefore these positions need to be updated accordingly. The problem here is that we do not know, how many non-terminals there are. This is where the function of the symbols containing ε becomes clear. With their presence, we always have exactly n non-terminals during the simulation of the deep PDA, and thus the matrices can be designed with this condition.

So for a transition $kpA \to qu_0B_1u_1B_2 \ldots u_{l-1}B_\ell u_\ell$, first note that it can only be executed, if there are at most $n - \ell + 1$ nonterminals on the stack; otherwise the condition of being finitely expandable is violated. This, on the other hand, means that the current sentential form must contain at least $l - 1$ placeholders containing ε. We test for this by starting the matrices with sequences $(X, i) \to (X, i), (\varepsilon, i + 1) \to (\varepsilon, i + 1)$, which do not change anything but establish that the first placeholder is at position $i+1$. As a second step, all the

nonterminals from positions $k+1$ to i are moved $l-1$ positions to the right and replaced by the nonterminal put there by the deep PDA's expansion. This is done by sequences $(Y,k+j) \to (B_{j+1},k+j)u_{j+1}, (Z,k+j+\ell-1) \to (Y,k+j+\ell-1)$ for the first $\ell-1$ nonterminals, which are replaced by the new ones from the expanding transition. The following ones are copied by sequences $(Y,m) \to (Y,m)$,$(Z,m+\ell-1) \to (Y,m+\ell-1)$; here the first rule checks, which nonterminal is at the original position, the second rule copies it $l-1$ positions to the right. This process must start from the right side in order not to delete any nonterminal before it has been copied. Finally in position k we must apply $(A,k) \to u_0(B_1,k)u_1$ and in position one $(X,p) \to (X,q)$ to update the state.

Summarizing, the matrices for simulating $kpA \to qu_0B_1u_1B_2 \ldots u_{l-1}B_\ell u_\ell$ are the following:

$$
\begin{array}{l}
((X,i) \to (X,i),\\
(\varepsilon,i+1) \to (\varepsilon,i+1),\\
\left.\begin{array}{l}(Y_m,m) \to (Y_m,m),\\ (Z_m,m+\ell-1) \to (Y_m,m+\ell-1),\end{array}\right\} \text{ for } m \in i,\ldots,k+\ell,\\
\left.\begin{array}{l}(Y_m,m) \to (B_{m-k+1},m)u_{m-k+1},\\ (Z_m,m+\ell-1) \to (Y_m,m+\ell-1),\end{array}\right\} \text{ for } m \in k+\ell-1,\ldots,k+1,\\
(A,k) \to u_0(B_1,k)u_1, (X',p) \to (X',q))
\end{array}
$$

One matrix is defined for each possible combination of $X, X', Y_m \in N$, i such that $n \geq i > k$, $Z \in N \cup \{\varepsilon\}$. If $k=1$, then the last line is condensed into one single rule $(A,p) \to u_0(B_1,q)u_1$. Notice that the number of nonterminals in the sentential form remains constant during application of these matrices and thus the condition of being of finite index is complied with.

Finally, there remains the case where a non-terminal is expanded to a string of terminals only. In this case one new placeholder must be introduced, and the position of the nonterminals right of the rule application must be decreased by one. Let the transition be $kpA \to qu$ with $u \to T^*$. Again, we start by testing for the border between nonterminals and placeholders by $(X,i) \to (X,i)$, $(\varepsilon,i+1) \to (\varepsilon,i+1)$. Then we can go from left to right, first update the state, then simulate the rule application, and finally move the other nonterminals to the left and insert a new ε. The matrices are:

$$
\begin{array}{l}
((X,i) \to (X,i),\\
(\varepsilon,i+1) \to (\varepsilon,i+1),\\
(X',p) \to (X',q), (A,k)\varepsilon u,\\
(Y_m,m) \to (Y_m,m-1), \text{ for } m \in k+1,\ldots,i,\\
(X,i) \to (X,i-1)(\varepsilon,i))
\end{array}
$$

In this case, there is also the possibility that we have n nonterminals present and no ε. In that case the first line simply contains $(X_n) \to (X_n)$, which establishes that the last position is occupied by a nonterminal. For $k=1$ there is a little modification necessary, because deleting the first nonterminal would also delete the state of the deep PDA. Therefore the second line is $(A,p) \to u$, $(Y_2,2) \to (Y_2,q)$, and the third line is only for m from 3 to i. Notice that the rule $(X,i) \to (X,i-1)(\varepsilon,i)$, which increases the number of nonterminals, is

applied only after the nonterminal at position k has been deleted. In this way the condition of being of finite index is complied with also here.

When removing the left-most non-terminal, however, there is also the possibility that it is the only one. In this case, this would be the last expansion of a non-terminal, and the simulation should stop. The only matter left to resolve, when the last non-terminal is deleted, is the fact that in the end of the simulation, in addition to the word of terminals, we have the remaining placeholders. They should be removed after the last expansion of a non-terminal in an accepting computation. Such an expansion can only take place in the first position, otherwise there are other non-terminals left that would be expanded further. Therefore we add for all transitions removing the first non-terminal matrices that do this expansion and in addition remove placeholders from positions 2 *to* n*; this way these matrices are only applicable if the non-terminal that is expanded is really the last one. Formally, for every transition* $1pA \to qu$ *with* $u \in T^*$ *we add the matrix* $((A,p) \to u, (\varepsilon,2) \to \lambda, \ldots, (\varepsilon,n) \to \lambda)$*, if* q *is a final state. This way, also the state of the deep PDA disappears, if it was final, and the grammar's derivation terminates with the same string of terminals that are on the stack of the simulated deep PDA in a computation according to the pattern of Lemma 3.1.1. Here it must be recalled that context-free matrix grammars of finite index have the same generative power whether there are deleting rules or not, see Lemma 3.1.2 in [5].*

Here the derivation graphs do not correspond to each other as directly as above. However, the correspondence is still rather evident. Let us define the following mapping that projects the compound nonterminals to their first component and deletes the placeholders:

$$\varrho(x) = \begin{cases} X & x = (X,i), X \in N, i \in \{1,2,\ldots,n\} \cup Q \\ \lambda & x = (\varepsilon,i), i \in \{1,2,\ldots,n\} \\ x & x \in T \end{cases}$$

Obviously, the deep PDA's stack contents at the start of a computation are the same as the image under ϱ *of the sentential form obtained after applying the only matrix applicable in the beginning, namely* $(S \to (S,s)(\varepsilon,2)(\varepsilon,3)\cdots(\varepsilon,n))$*. Also the state stored in the sentential form is equal to the one of the deep PDA. As the explanations throughout the definition of our matrix grammar illustrate, these two features equality of stack contents and the sentential form's image under* ϱ *plus the equality of states are preserved by simultaneous application of an expanding transition in the deep PDA and the corresponding matrix on the sentential form. Due to the one-to-one correspondence between expanding transitions and matrices (that always have to apply all of their rules) it is clear that any computation according to Lemma 3.1.1, before popping is started, leads to a terminal string on the stack that can also be generated by the grammar, if the current state is final. The inverse is equally obvious and thus the two devices are equivalent.* □

For matrix grammars of finite index it is known that they give rise to an in finite hierarchy of classes of languages. For every positive integer n, the class of languages generated by matrix grammars of index n is properly contained in the class of languages generated by matrix grammars of index $n+1$, see Theorem 3.1.7 in [5]. Since these classes are equal to the ones accepted by finitely expandable deep PDAs, also these devices induce an in finite hierarchy.

Corollary 3.1.1 *For all integers* $n > 0$*, the class of languages accepted by* n*-expandable deep PDAs is properly contained in the class of languages accepted by* $n+1$*-expandable deep PDAs.*

3.2 Reduction of Finitely Expandable Deep Pushdown Automata

In this section, we establish Lemma 3.2.1, which captures the main result of this thesis.

Lemma 3.2.1 *Let $n \in \mathbb{N}$. For every n-expandable deep PDA M, there exists an n-expendable deep PDA M_R such that $L(M) = L(M_R)$ and M_R contains only two non-input pushdown symbols—$\$$ and $\#$.*

Proof 3.2.1 Construction. *Let $n \in \mathbb{N}$. Let*

$$M = (Q, \Sigma, \Gamma, R, s, S, F)$$

be an n-expandable deep PDA. Recall that rules in R are of the form $mqA \to pv$, where $m \in \mathbb{N}$, $q, p \in Q$, either $A \in N$ and $v \in (\Gamma \setminus \{\#\})^+$ or $A = \#$ and $v \in (\Gamma \setminus \{\#\})^*\{\#\}$, where $\#$ denotes the pushdown bottom.

Let $\$$ be a new symbol, $\$ \notin Q \cup \Gamma$, and let homomorphisms f and g over Γ^* be defined as $f(A) = A$ and $g(A) = \$$, for every $A \in N$, and $f(a) = \varepsilon$ and $g(a) = a$, for every $a \in (\Sigma \cup \{\#\})$. Next, we construct an n-expandable deep PDA

$$M_R = (Q_R, \Sigma, \Sigma \cup \{\$, \#\}, R_R, s_R, \$, F_R)$$

by performing 1 through 4, given next:

1. *Add $m\langle q; uAz\rangle\$ \to \langle p; uf(v)z\rangle g(v)$ to R_R and add $\langle q; uAz\rangle$, $\langle p; uf(v)z\rangle$ to Q_R if $mqA \to pv \in R$, $u, z \in N^*$, $|u| = m - 1$, $|z| \leq n - m - 1$, $m \in \mathbb{N}$, $q, p \in Q$, $A \in N$, and $v \in (\Gamma \setminus \{\#\})^+$;*

2. *Add $m\langle q; u\rangle\# \to \langle p; uf(v)\rangle g(v)\#$ to R_R and add $\langle q; u\rangle$, $\langle p; uf(v)\rangle$ to Q_R if $mq\# \to pv\# \in R$, $u \in N^*$, $|u| = m - 1$, $m \in \mathbb{N}$, $q, p \in Q$, and $v \in (\Gamma \setminus \{\#\})^*$;*

3. *Set $s_R = \langle s; S\rangle$;*

4. *Add all $\langle p; u\rangle$ to F_R, where $p \in F$, $u \in N^*$.*

Later in this proof, we demonstrate that $L(M) = L(M_R)$.

Basic Idea. *States in Q_R include not only the states corresponding to the states in Q but also strings of non-input symbols. Whenever M pushes a non-input symbol onto the pushdown, M_R records this information within its current state and pushes $\$$ onto the pushdown instead.*

As previously mentioned in Lemma 3.1.1, any n-expandable deep PDA M can accept every $w \in L(M)$ so all expansions precede all pops during the accepting process. Without any loss of generality, we assume that M and M_R work in this way in what follows, too.

To establish $L(M) = L(M_R)$, we prove the following four claims.

Claim 3.2.1 *Let $(s, w, S\#) \vdash^j (q, v, x\#)$ in M, where $s, q \in Q$, $w, v \in \Sigma^*$, and $x \in (\Gamma \setminus \{\#\})^*$. Then, $(\langle s; S\rangle, w, \$\#) \vdash^* (\langle q; f(x)\rangle, v, g(x)\#)$ in M_R, where $\langle s; S\rangle$, $\langle q; f(x)\rangle \in Q_R$, and $g(x) \in (\Sigma \cup \{\$\})^*$.*

Proof 3.2.2 *This claim is proved by induction on $j \geq 0$.*

Basis. *Let $j = 0$, so $(s, w, S\#) \vdash^0 (s, w, S\#)$ in M, where $s \in Q$ and $S \in N$. Then, from 3 in the construction, we obtain*

$$(\langle s; S\rangle, w, \$\#) \vdash^0 (\langle s; S\rangle, w, \$\#)$$

in M_R, so the basis holds.

Induction Hypothesis. *Assume there is $i \geq 0$ such that Claim 3.2.1 holds true for all $0 \leq j \leq i$.*

Induction Step. *Let $(s, w, S\#) \vdash^{i+1} (q, w, x\#)$ in M, where $x \in (\Gamma \setminus \{\#\})^*$, $s, q \in Q$, $w \in \Sigma^*$. Since $i + 1 \geq 1$, we can express $(s, w, S\#) \vdash^{i+1} (q, w, x\#)$ as*

$$(s, w, S\#) \vdash^i (p, w, x_0A_1x_1...A_m...A_kx_k\#)$$

$$\vdash (q, w, x_0A_1x_1...A_{m-1}x_{m-1}y_0B_1y_1...B_\ell y_\ell x_m A_{m+1}...x_{k-1}A_kx_k\#)$$

$$[mpA_m \to qy_0B_1y_1...B_\ell y_\ell]$$

where $A_1, ..., A_k, B_1, ..., B_\ell \in N$ and $x_0x_1...x_k, y_0y_1...y_\ell \in \Sigma^*$. By the induction hypothesis, we have

$$(\langle s; S\rangle, w, \$\#) \vdash^* (\langle p; A_1...A_m...A_k\rangle, w, x_0\$x_1\$...\$x_k\#)$$

Since $mpA_m \to qy_0B_1y_1...B_\ell y_\ell \in R$, we also have

$$m\langle p; A_1...A_m...A_k\rangle\$ \to \langle q; A_1...A_{m-1}B_1...B_\ell A_{m+1}...A_k\rangle y_0\$y_1\$...\$y_\ell \in R_R$$

(see 1 in the construction). Thus,

$$(\langle p; A_1...A_m...A_k\rangle, w, x_0\$x_1\$...\$x_k\#) \vdash$$

$$(\langle q; A_1...A_{m-1}B_1...B_\ell A_{m+1}...A_k\rangle, w, x_0\$x_1\$...\$x_{m-1}y_0\$y_1\$...\$y_\ell x_m\$...\$x_k\#)$$

$$[m\langle p; A_1...A_m...A_k\rangle\$ \to \langle q; A_1...A_{m-1}B_1...B_\ell A_{m+1}...A_k\rangle y_0\$y_1\$...\$y_\ell]$$

Analogically, we can prove the induction step for the case when # is rewritten. Let $(s, w, S\#) \vdash^{i+1} (q, w, Sx\#)$ in M, where $x \in (\Gamma \setminus \{\#\})^*$, $s, q \in Q$, $w \in \Sigma^*$. Since $i + 1 \geq 1$, we can express $(s, w, S\#) \vdash^{i+1} (q, w, Sx\#)$ as

$$(s, w, S\#) \vdash^i (p, w, Sx_0A_1x_1...A_kx_k\#)$$

$$\vdash (q, w, Sx_0A_1x_1...A_kx_ky_0B_1y_1...B_\ell y_\ell\#)$$

$$[mp\# \to qy_0B_1y_1...B_\ell y_\ell\#]$$

where $A_1, ..., A_k, B_1, ..., B_\ell \in N$ and $x_0x_1...x_k, y_0y_1...y_\ell \in \Sigma^*$. By the induction hypothesis, we have

$$(\langle s; S\rangle, w, \$\#) \vdash^* (\langle p; SA_1...A_k\rangle, w, \$x_0\$x_1\$...\$x_k\#)$$

Since $mp\# \to qy_0B_1y_1...B_\ell y_\ell\# \in R$, we also have

$$m\langle p; SA_1...A_k\rangle\# \to \langle q; A_1...A_kB_1...B_\ell\rangle\$y_0\$y_1\$...\$y_\ell \in R_R$$

(see 2 in the construction). Thus,

$$(\langle p; SA_1...A_k\rangle, w, \$x_0\$x_1\$...\$x_k\#) \vdash$$

$$(\langle q; SA_1...A_kB_1...B_\ell\rangle, w, \$x_0\$x_1\$...\$x_ky_0\$y_1\$...\$y_\ell\#)$$

$$[m\langle p; SA_1...A_k\rangle\# \to \langle q; SA_1...A_kB_1...B_\ell\rangle\$y_0\$y_1\$...\$y_\ell\#]$$

Therefore, Claim 3.2.1 holds true. □

Claim 3.2.2 *$L(M) \subseteq L(M_R)$.*

Proof 3.2.3 *Consider Claim 3.2.1 for $v = \varepsilon$, $q \in F$, and $x = \varepsilon$. Under this consideration Claim 3.2.1 implies Claim 3.2.2.* □

Claim 3.2.3 *Let $(\langle s; S\rangle, w, \$\#) \vdash^j (\langle q; A_1...A_k\rangle, v, x\#)$ in M_R, where $s_R = \langle s; S\rangle$, $\langle q; A_1...A_k\rangle \in Q_R$, $w, v \in \Sigma^*$, $A_1, ..., A_k \in N$, $x = x_0\$x_1\$...\$x_k$, and $x_0...x_k \in \Sigma^*$. Then, $(s, w, S\#) \vdash^* (q, v, x_0A_1x_1...A_kx_k\#)$ in M, where $s, q \in Q$.*

Proof 3.2.4 *This claim is proved by induction on $j \geq 0$.*

Basis. *Let $j = 0$, so $(\langle s; S\rangle, w, \$\#) \vdash^0 (\langle s; S\rangle, w, \$\#)$ in M_R, where $s_R = \langle s; S\rangle$. From 3 in the construction, we have*

$$(s, w, S\#) \vdash^0 (s, w, S\#)$$

in M, so the basis holds.

Induction Hypothesis. *Assume there is $i \geq 0$ such that Claim 3.2.3 holds true for $0 \leq j \leq i$.*

Induction Step. *Let $(\langle s; S\rangle, w, \$\#) \vdash^{i+1} (\langle q; A_1...A_k\rangle, w, x_0\$x_1\$...\$x_k\#)$ in M_R, where $\langle q; A_1...A_k\rangle \in Q_R$, $A_1, ..., A_k \in N$, $w \in \Sigma^*$, and $x_0...x_k \in \Sigma^*$. Since $i + 1 \geq 1$, we can express*

$$(\langle s; S\rangle, w, \$\#) \vdash^{i+1} (\langle q; A_1...A_k\rangle, w, x_0\$x_1\$...\$x_k\#)$$

as

$$(\langle s; S\rangle, w, \$\#) \vdash^i (\langle p; A_1...A_m...A_k\rangle, w, x_0\$x_1\$...\$x_k\#)$$

$$\vdash (\langle q; A_1...A_{m-1}B_1...B_\ell A_{m+1}...A_k\rangle, w, x_0\$x_1\$...\$x_{m-1}y_0\$y_1\$...\$y_\ell x_m\$...\$x_k\#)$$

$$[m\langle p; A_1...A_m...A_k\rangle\$ \to \langle q; A_1...A_{m-1}B_1...B_\ell A_{m+1}...A_k\rangle y_0\$y_1\$...\$y_\ell]$$

By the induction hypothesis, we obtain

$$(s, w, S\#) \vdash^i (p, w, x_0A_1x_1...A_m...A_kx_k\#)$$

Since $m\langle p; A_1...A_m...A_k\rangle\$ \to \langle q; A_1...A_{m-1}B_1...B_\ell A_{m+1}...A_k\rangle y_0\$y_1\$...\$y_\ell \in R_R$, we also have $mpA_m \to qy_0B_1y_1...B_\ell y_\ell \in R$ as follows from 1 in the construction. We obtain

$$(p, w, x_0A_1x_1...A_m...A_kx_k\#)$$

$$\vdash (q, w, x_0A_1x_1...A_{m-1}x_{m-1}y_0B_1y_1...B_\ell y_\ell x_mA_{m+1}...x_{k-1}A_kx_k\#)$$

$$[mpA_m \to qy_0B_1y_1...B_\ell y_\ell]$$

Analogically, we can prove the case when # is expanded. Let $(\langle s; S\rangle, w, \$\#) \vdash^{i+1} (\langle q; SA_1...A_k\rangle, w, \$x_0\$x_1\$...\$x_k\#)$ in M_R, where $\langle q; SA_1...A_k\rangle \in Q_R$, $A_1, ..., A_k \in N$, $w \in \Sigma^*$, and $x_0...x_k \in \Sigma^*$. Since $i + 1 \geq 1$, we can express

$$(\langle s; S\rangle, w, \$\#) \vdash^{i+1} (\langle q; SA_1...A_k\rangle, w, \$x_0\$x_1\$...\$x_k\#)$$

as

$$(\langle s; S\rangle, w, \$\#) \vdash^i (\langle p; A_1...A_k\rangle, w, \$x_0\$x_1\$...\$x_k\#)$$

$$\vdash (\langle q; SA_1...A_kB_1...B_\ell\rangle, w, \$x_0\$x_1\$...\$x_ky_0\$y_1\$...\$y_\ell\#)$$
$$[m\langle p; SA_1...A_k\rangle\# \to \langle q; A_1...A_kB_1...B_\ell\rangle y_0\$y_1\$...\$y_\ell\#]$$

By the induction hypothesis, we obtain

$$(s, w, S\#) \vdash^i (p, w, Sx_0A_1x_1...A_m...A_kx_k\#)$$

Since $m\langle p; A_1...A_m...A_k\rangle\$ \to \langle q; A_1...A_{m-1}B_1...B_\ell A_{m+1}...A_k\rangle y_0\$y_1\$...\$y_\ell \in R_R$, *we also have* $mpA_m \to qy_0B_1y_1...B_\ell y_\ell \in R$ *as follows from 2 in the construction. We obtain*

$$(p, w, Sx_0A_1x_1...A_kx_k\#)$$
$$\vdash (q, w, Sx_0A_1x_1...x_{k-1}A_kx_ky_0B_1y_1...B_\ell y_\ell\#)$$
$$[mp\# \to qy_0B_1y_1...B_\ell y_\ell\#]$$

Therefore, Claim 3.2.3 holds true. □

Claim 3.2.4 $L(M_R) \subseteq L(M)$.

Proof 3.2.5 *Consider Claim 3.2.3 with* $v = \varepsilon$, $\langle q; A_1...A_k\rangle \in F_R$, *and* $x = \varepsilon$. *Under this consideration, Claim 3.2.3 implies Claim 3.2.4.* □

As $L(M) \subseteq L(M_R)$ *(see Claim 3.2.2) and* $L(M_R) \subseteq L(M)$ *(see Claim 3.2.4),* $L(M_R) = L(M)$. *Thus, Lemma 3.2.1 holds.* □

The next example illustrates the construction described in the previous proof.

Example 3.2.1 *Take this two-expandable deep PDA* $M = (\{s, q, p\}, \{a, b, c\}, \{a, b, c, A, S, \#\}, R, s, S, \{f\})$

$$\begin{aligned} R = \{&1sS \to qAA,\\ &1qA \to fab,\\ &1fA \to fc,\\ &1qA \to paAb,\\ &2pA \to qAc\} \end{aligned}$$

By the construction given in the proof of Lemma 3.2.1, we construct $M_R = (Q_R, \{a, b, c\}, \{a, b, c, \$, \#\}, R_R, \langle s; S\rangle, \$, \{\langle f; A\rangle, \langle f; \varepsilon\rangle\})$, *where* $Q_R = \{\langle s; S\rangle, \langle q; AA\rangle, \langle f; A\rangle, \langle f; \varepsilon\rangle, \langle p; AA\rangle\}$ *Figure 3.1.*

$$\begin{aligned} R_R = \{&1\langle s; S\rangle\$ \to \langle q; AA\rangle\$\$,\\ &1\langle q; AA\rangle\$ \to \langle f; A\rangle ab,\\ &1\langle f; A\rangle\$ \to \langle f; \varepsilon\rangle c,\\ &1\langle q; AA\rangle\$ \to \langle p; AA\rangle a\$b,\\ &2\langle p; AA\rangle\$ \to \langle q; AA\rangle\$c\} \end{aligned}$$

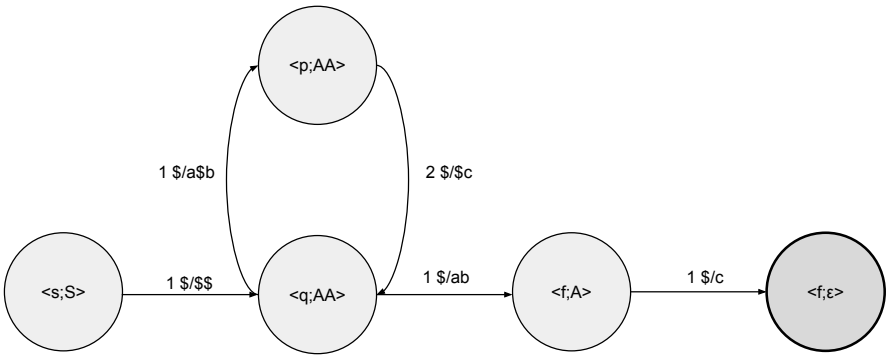

Figure 3.1: 2-expandable deep pushdown automaton accepting language $a^n b^n c^n$.

For instance, M_R makes

$$
\begin{aligned}
(\langle s;S\rangle, aabbcc, \$\#) &\ {}_e\vdash (\langle q;AA\rangle, aabbcc, \$\$\#) && [1\langle s;S\rangle\$ \to \langle q;AA\rangle\$\$]\\
&\ {}_e\vdash (\langle p;AA\rangle, aabbcc, a\$b\$\#) && [1\langle q;AA\rangle\$ \to \langle p;AA\rangle a\$b]\\
&\ {}_p\vdash (\langle p;AA\rangle, abbcc, \$b\$\#)\\
&\ {}_e\vdash (\langle q;AA\rangle, abbcc, \$b\$c\#) && [2\langle p;AA\rangle\$ \to \langle q;AA\rangle\$c]\\
&\ {}_e\vdash (\langle f;A\rangle, abbcc, abb\$c\#) && [1\langle q;AA\rangle\$ \to \langle f;A\rangle ab]\\
&\ {}_p\vdash (\langle f;A\rangle, cc, \$c\#)\\
&\ {}_e\vdash (\langle f;\varepsilon\rangle, cc, cc\#) && [1\langle f;A\rangle\$ \to \langle f;\varepsilon\rangle c]\\
&\ {}_p\vdash (\langle f;\varepsilon\rangle, \varepsilon, \#)
\end{aligned}
$$

3.3 Acceptance Power

In this section, we will evaluate acceptance power of the reduced finitely expandable deep pushdown automata.

Theorem 3.3.1 *For all $n \geq 1$, ${}_n\mathbf{DPDA} = {}_n\mathbf{DPDA}_2$.*

Proof 3.3.1 *This theorem follows from Lemma 3.2.1.* □

Corollary 3.3.1 *For all $n \geq 1$, ${}_n\mathbf{DPDA}_2 \subset {}_{n+1}\mathbf{DPDA}_2$.*

Proof 3.3.2 *This corollary follows from Theorem 3.3.1 in this thesis and Corollary 3.1.1.* □

Can we reformulate Theorem 3.3.1 and Corollary 3.3.1 in terms of ${}_n\mathbf{DPDA}_1$? The answer is no as we show next.

Lemma 3.3.1 *Let $M = (Q, \Sigma, \Gamma, R, s, S, F)$ be a deep PDA with $\Gamma \setminus \Sigma = \{\#\}$. Then, there is a right-linear grammar G such that $L(G) = L(M)$.*

Proof 3.3.3 *Let $M = (Q, \Sigma, \Gamma, R, s, S, F)$ with $\Gamma \setminus \Sigma = \{\#\}$. Thus, every rule in R is of the form $1q\# \to px\#$, where $q, p \in Q$, $x \in \Sigma^*$. Next, we construct a right-linear grammar $G = (Q, \Sigma, P, s)$ so $L(M) = L(G)$. We construct P as follows:*

1. *For every $1q\# \to px\# \in R$, where $p, q \in Q$, $x \in \Sigma^*$, add $q \to xp$ to P;*
2. *For every $f \in F$, add $f \to \varepsilon$ to P.*

A rigorous proof that $L(M) = L(G)$ is left to the reader. □

Theorem 3.3.2 **REG** $= {}_1\mathbf{DPDA}_1 = {}_n\mathbf{DPDA}_1$, *for any $n \geq 1$.*

Proof 3.3.4 *Let $n \geq 1$.* **REG** $\subseteq {}_1\mathbf{DPDA}_1 = {}_n\mathbf{DPDA}_1$ *is clear. Recall that right-linear grammars characterize* **REG**, *so* ${}_n\mathbf{DPDA}_1 \subseteq$ **REG** *follows from Lemma 3.3.1. Thus,* **REG** $= {}_n\mathbf{DPDA}_1$. □

Corollary 3.3.2 **REG** $= {}_1\mathbf{DPDA}_1 = {}_n\mathbf{DPDA}_1 \subset {}_n\mathbf{DPDA}_2$, *for all $n \geq 2$.*

Proof 3.3.5 *Let $n \geq 1$. As obvious, ${}_1\mathbf{DPDA}_1 = {}_n\mathbf{DPDA}_1 \subseteq {}_n\mathbf{DPDA}_2$. Observe that*

$$\{a^n b^n \mid n \geq 1\} \in {}_n\mathbf{DPDA}_2 \setminus {}_n\mathbf{DPDA}_1$$

Therefore, Corollary 3.3.2 holds. □

Chapter 4

Application

Since we have proven that the reduction of the pushdown alphabet of n-expandable deep pushdown automaton is possible, we can follow up with exemplary application. First of all we will analyze the syntax structures that are accepted by the automaton with the main focus on structures that are not context-free and create automata with a set of rules capable to accept such structures.

Afterwords, we will move to the implementation part of the thesis which provides possibility to create a finitely expandable deep pushdown automata. Moreover, the implementation allows analysis of the individual rule application and the transfer of the symbols in the pushdown.

4.1 Analysis of Syntax Structures

Deep pushdown automaton is able to parse any context-free language or any language that is subset of the context-free languages, but so does regular pushdown automaton, so this will not be the main interest of this analysis.

An example of a family of languages that are not context-free is the family of context-sensitive languages. This family is a super set of the family of context-free languages. There is infinite amount language families between the families of context-free and context-sensitive languages. There always exists some context-sensitive language that cannot be accepted by any deep pushdown automata that make expansion of depth n or less, for every positive integer n. Naturally larger the value n is, the larger the set of languages accepted by the automaton becomes.

Based on this information, the main restrictive factor of the n-expandable deep pushdown automata is the n which then also becomes the main restriction in finitely expandable deep pushdown automata with reduced pushdown alphabet. We will have to determine the value of n that is necessary to parse chosen languages.

4.1.1 Pumping Lemma Theorem

To prove that examples we are using in this section of this thesis are not context-free, we will use *Pumping Lemma Theorem* [2]. The introduced form of Pumping Lemma Theorem states conditions that are necessary for language to be context-free or sub set of a context-free language. If language does not fulfill these conditions, the language is considered to be at least context-sensitive.

Lemma 4.1.1 *Let L be a context-free language. Then there is a constant k such that if $z \in L$ and $|z| \geq k$, then z can be written in the form:*

$$z = uvwxy, vx \neq \varepsilon, |vwx| \leq k$$

and for all $i \geq 0$ is $uv^i wx^i y \in L$.

Proof 4.1.1 *Let $L = L(G)$ and $G = (N, T, R, S)$ be a grammar in CNF.*

1. *First we will prove this implication :*

   If $A \Rightarrow^+ \alpha$ then $|\alpha| \leq 2^{m-1}$, where m is the number of vertexes of the longest path in the corresponding derivation tree.

   For the proof we use the induction

   (a) *$m = 2$*

   (b) *Consider that the statement holds for some m and the longest path contains $m + 1$ vertices. Then the rule of form $A \to BC$ was applied in the first step of a derivation and we can apply the induction hypothesis on the sub-trees with the roots B and C. We obtain:*

   $$|\alpha| \leq 2^{m-1} + 2^{m-1} = 2^{(m-1)+1} = 2^{(m+1)-1}$$

2. *Let $|N| = n$ and $k = 2^{n+1}$. Consider an arbitrary sentence z such that $|z| \geq k$. Then the longest path in the corresponding derivation tree contains at least $n + 1$ vertices and necessarily at least 2 from them are marked by the same nonterminal. Denote this nonterminal by symbol A.*

The strings v, x can not be empty because the applied rule is of form $A \to BC$. Now consider the derivation of the string z of form: $S \Rightarrow^* uAy \Rightarrow^+ uvAxy \Rightarrow^+ uvwxy = z$ This means that in the grammar G there is also the derivation: $S \Rightarrow^* uAy \Rightarrow^+ uvAxy \Rightarrow^+ uvvAxxy \Rightarrow^+ uv^2wx^2y$, because $A \Rightarrow^+ w$, and thus the derivation $S \Rightarrow^* uv^i wx^i y$ for arbitrary $i > 0$, which is to be proved. □

4.1.2 Non-Context-Free Languages

Example of non-context-free language is

$$L = \{a^n b^n c^n \mid n \leq 0\}$$

Language L cannot be generated by any context-free grammar or parsed by regular pushdown automaton. However, as we have seen in the example, it can be parsed using deep pushdown automaton and there for using the deep pushdown automaton's reduced form. Now we will show that the given example contradicts the Pumping Lemma Theorem.

If we assume L is context-free language and choose a string $z = a^k b^k c^k$ from L. The string holds true to condition that string should be longer than a value k since $|z| = 3k$. According to Pumping Lemma Theorem string z includes substring vwx so that $|vwx| \leq k$ and $vx \neq \varepsilon$. If we fulfil both these conditions there are just 5 different possibilities what substring vwx can consist of. (1) Only of symbols a, (2) only of symbols b, (3) only of symbols c, (4) symbol(s) a followed by symbol(s) b, or (5) symbol(s) b followed by symbol(s) c.

The substring can not consist of all 3 symbols since the length has to be less or equal to k. We can see that if we choose i other than one, the number of some symbols will increase while the number of others won't. This contradicts the language where the number of the symbols is equal. Therefore, the language L is not context-free language.

Another typical example of context-sensitive language is

$$L_{dup} = \{ss \mid s \in \{a, b\}^*\}$$

If we assume language L_{dup} is a context-free language and choose a string $z = a^k b^k a^k b^k$ from L_{dup}, the condition that $|z| \geq k$ hold true since $|z| = 4k$. Since the substring of z $vwx \leq k$, there is 7 different of possibilities of symbols that substring vwx may contain. It can contain only one type of symbol, either a or b from the first half of the string or a or b from the second half of the string. If that was the case and i was anything else than 1, one s would become longer than the other and therefore could not be same.

Another possibility would be the substring vwx would consist of a and b from the first half or a and b from the second half. Similarly as in the first case the by increasing the value of i the length of s would no longer be same and therefore would contradict the structure of the language.

The last possibility would be if the substring vwx was located in the middle of the string z. If we would have change the value of the i we would get a string $a^k b^j a^m b^k$, which clearly showcase that the number of a in first half of string is not the same as number of a in the second half, which contradicts with the language structure once again. Thus, language L_{dup} is not context-free language.

4.1.3 Non-Context-Free Code

In practical application, we can find the language L_{dup} in many forms in many programing languages. For example, in most programming languages, declaration of a variable can be done in separate part of code than its assignment by a particular value. Typically, the value cannot be assigned unless the variable was previously declared. Context-free grammar has no way of knowing whether the variable was in fact declared and therefore extra look-up tables have to be created to ensure the proper syntax of the code.

The above described problem is demonstrated in the following pseudo code:

```
variable dog;
variable cat;
variable cow;
func {
    dog = value;
    cat = value;
    cow = value;
}
```

This structure could be generic for any code that has declaration separated from assigning of the value. More specifically, this structure is typical, e.g., for constructor of an object in object-oriented languages.

The automaton parsing this code must ensure that every *value* assigned to an *variable* was previously declared. We have created a finitely expandable deep pushdown automaton that is able to parse such structure:

$M = (Q, \Sigma, \Gamma, R, s, S, \{f\})$
$Q = \{s, p, f, g, \langle r; a\rangle, ..., \langle r; z\rangle, \langle t; a\rangle, ..., \langle t; z\rangle, \langle u; a\rangle, ..., \langle u; z\rangle\}$
$\Sigma = \{func, \{, \}, ;, , var, val, =, a, ..., z\}$
$\Gamma = \{func, \{, \}, ;, , var, val, =, a, ..., z, \#, P, I, N, V, I', S\}$

$$
\begin{aligned}
R = \{ & 1sS \to pVP, \\
& 2pP \to pfunc\{P\}, \\
& 2pP \to pI = N; P, \\
& 2pN \to pval, \\
& 2pI \to \langle r; a\rangle aI', \\
& 1\langle r; a\rangle V \to pvaraV, \\
& 2pI' \to \langle t; a\rangle aI', \\
& 1\langle t; a\rangle V \to paV, \\
& 2pI' \to \langle u; a\rangle a, \\
& 1\langle u; a\rangle V \to pa; V, \\
& ... \\
& 2pI \to \langle r; z\rangle zI', \\
& 1\langle r; z\rangle V \to pvarzV, \\
& 2pI' \to \langle t; z\rangle zI', \\
& 1\langle t; z\rangle V \to pzV, \\
& 2pI' \to \langle u; z\rangle z, \\
& 1\langle u; z\rangle V \to pz; V, \\
& 2pP \to g\varepsilon, \\
& 1gV \to f\varepsilon\}
\end{aligned}
$$

$F = \{\langle f; \varepsilon\}$

Basic Idea. The restriction of the automaton is that the order of declaration matches the order of assigned values in order to mimic the structure of L_{dup} language. Additionally, no more then one assignment of the value to certain variable is possible. First four rules of the automaton $1sS \to pVP$, $2pP \to pfunc\{P\}$, $2pP \to pI = N; P$ and $2pN \to pval$ create the outlining structure of the code. Based on the expansion of P the assigning of the value can be put anywhere in the code.

The automaton individually checks each letter of a name of the assigned variable, whether it can mach any name created out of lower-case letters within declaration section. There is a visible pattern to the following rules, one rule adds a letter to the assigned variable and leads the automaton to unique state. The second rule then adds the same letter to the declaration variable and leads the automaton back to state where other letter can be added. The first two rule for every letter $2pI \to \langle r; a\rangle aI'$ and $1\langle r; a\rangle V \to pvaraV$ create non-input symbol I' that is than used to expand the entire name of the variable. The second two rules $2pI' \to \langle t; a\rangle aI'$ and $\langle t; a\rangle V \to paV$ then continue expansion of individual letters of the name. When the name ends using last two rules $2pI' \to \langle u; a\rangle a$ and $1\langle u; a\rangle V \to pa; V$ which get rid of the non-input symbol I' and so it is possible to start generating another variable again

The last two rules $2pP \to g\varepsilon$ and $1gV \to f\varepsilon$ are to eliminate the last non-input symbol P and bring the automaton to final states.

Similarly, with a simple extension of the set of rules, the automaton could check if the type of the variable matches the type of the assigned value, or be able to accept value assigning in mixed matched order.

Now, one can convert the created automaton into the following reduced form:

$$
\begin{aligned}
M_R &= (Q_R, \Sigma, \Gamma_R, R_R, s, S, \{f\}) \\
Q &= \{s, p, f, g, \langle r;a\rangle, ..., \langle r;z\rangle, \langle t;a\rangle, ..., \langle t;z\rangle, \langle u;a\rangle, ..., \langle u;z\rangle\} \\
\Sigma &= \{func, \{, \}, ;, , var, val, =, a, ..., z\} \\
\Theta &= \{func, \{, \}, ;, , var, val, =, a, ..., z, \#, \$\} \\
R &= \{1\langle s;S\rangle\$ \to \langle\langle p;VP\rangle\$\$, \\
&\quad 2\langle p;VP\rangle\$ \to \langle p;VP\rangle func\{\$\}, \\
&\quad 2\langle p;VP\rangle\$ \to \langle p;VINP\rangle\$ = \$;\$, \\
&\quad 2\langle p;VINP\rangle\$ \to \langle p;VP\rangle val, \\
&\quad 2\langle p;VINP\rangle\$ \to \langle\langle r;a\rangle;VI'NP\rangle a\$, \\
&\quad 1\langle\langle r;a\rangle;VI'NP\rangle\$ \to \langle p;VI'NP\rangle vara\$, \\
&\quad 2\langle p;VI'NP\rangle\$ \to \langle\langle t;a\rangle;VI'NP\rangle a\$, \\
&\quad 1\langle\langle t;a\rangle;VI'NP\rangle\$ \to \langle p;VI'NP\rangle a\$, \\
&\quad 2\langle p;VI'NP\rangle\$ \to \langle\langle u;a\rangle;VNP\rangle a, \\
&\quad 1\langle\langle r;a\rangle;VNP\rangle\$ \to \langle p;VNP\rangle a;\$, \\
&\quad ... \\
&\quad 2\langle p;VINP\rangle\$ \to \langle\langle r;z\rangle;VI'NP\rangle z\$, \\
&\quad 1\langle\langle r;z\rangle;VI'NP\rangle\$ \to \langle p;VI'NP\rangle varz\$, \\
&\quad 2\langle p;VI'NP\rangle\$ \to \langle\langle t;z\rangle;VI'NP\rangle z\$, \\
&\quad 1\langle\langle t;z\rangle;VI'NP\rangle\$ \to \langle p;VI'NP\rangle z\$, \\
&\quad 2\langle p;VI'NP\rangle\$ \to \langle\langle u;z\rangle;VNP\rangle z, \\
&\quad 1\langle\langle r;z\rangle;VNP\rangle\$ \to \langle p;VNP\rangle z;\$, \\
&\quad 2\langle p;VP\rangle\$ \to \langle g;V\rangle\varepsilon, \\
&\quad 1\langle g;V\rangle\$ \to \langle f;\varepsilon\rangle\varepsilon\} \\
F &= \{\langle f;\varepsilon\rangle\}
\end{aligned}
$$

The reduced form can be also seen in Figure 4.1.

4.2 Implementation

The implementation of simulator and reduction of n-expandable deep pushdown automata is done in `Java` and takes in consideration the time and space complexity of the structure and the methods needed to simulate the parsing of input string. The basic structure of the program is shown in UML diagram that is depicted in Figure 4.2.

Since the aim of this thesis is to show both non-reduced and reduced version of the automaton has the same acceptance power, the parsing is not automatic and has to receive the order of rules that will be applied. However, this is enough to showcase both automaton are capable of accepting the non context-free language and generate output to illustrate it.

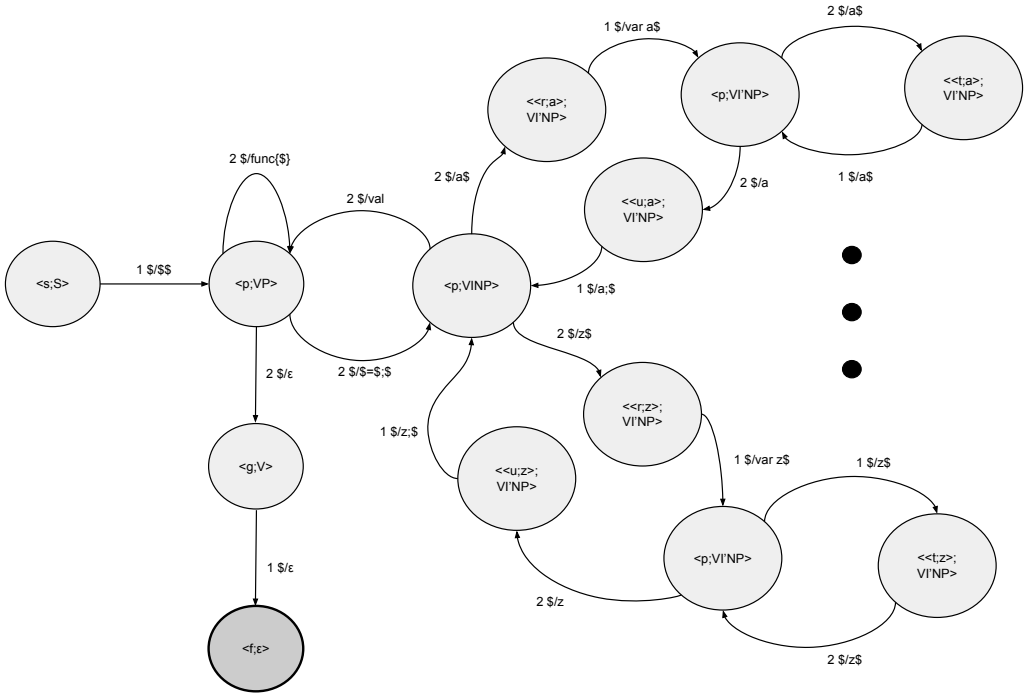

Figure 4.1: Automaton accepting the pseudo code.

4.2.1 Deep Stack

In order to be able to properly simulate the parsing of an input, one has to have a customized stack that allows accessing elements located deeper on the stack. The structure `DeepStack` includes the functionality of a regular stack but also methods enabling access to deeper elements:

- `push(element, depth)` pushes element to certain depth and returns the reference to the object where the element is saved. This feature will come in handy when creating the deep pushdown.
- `pop(depth)` returns an element from certain depth and removes it from the stack.
- `push(element, previous element)` pushes element under the element that was specified by the reference to it. It then returns reference to the new added element. Again this feature will come in handy when creating the deep pushdown.
- `pop(previous element)` returns an element based on his reference and removes it from the stack.

4.2.2 Special Structures

A special structure `PDSymbol` was created to define each symbol of the pushdown alphabet. Each symbol is defined by its name and the fact if its non-input, input or special bottom

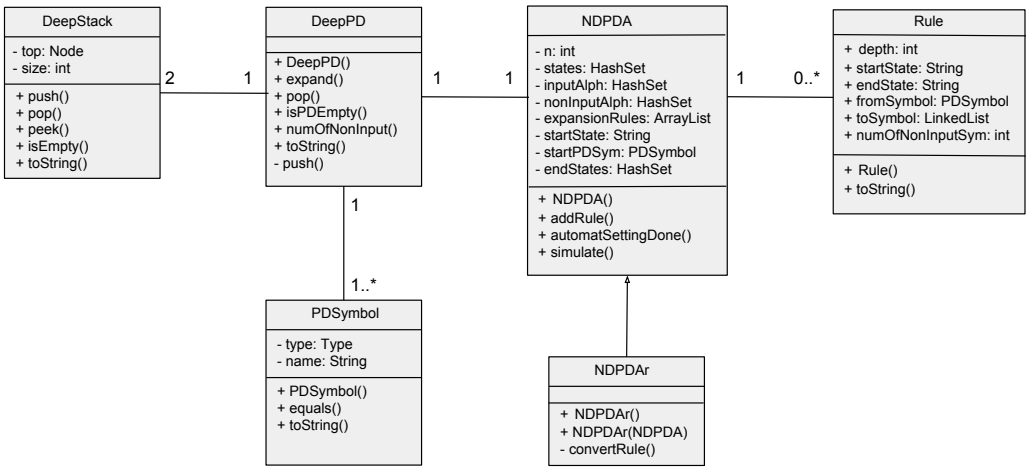

Figure 4.2: UML diagram of the implementation of the deep PDA.

symbol #. The `toString()` method than provides convenient way to label each element on the pushdown by their type:

$$\{a : \text{TERMINAL}\}$$

4.2.3 Deep Pushdown

Deep pushdown is then created using two deep stacks. The *main stack* stores both the input and non-input symbols and the *reference stack* stores only references to the non-input symbols located on the main stack. This structure makes it simple to find non-input symbol in certain depth and then using the reference to find the location of the non-input symbol among the larger stack. Since the number of input symbols can be far larger then the number of the non-input symbols this structure creates and effective way to find the searched symbols. The structure `deepPD` includes two main methods and couple of supporting methods.

One of the main methods is `expand()` that manages both of the stacks when expanding when adding both non-input and input symbols. Example of expansion is in the Figure 4.3.

Since throughout the entire thesis we used the assumption that the operation of the pop can be used after all the expansion have been made, the method `pop` just pops the top symbol of the main stack. If the expansions have been done correctly there should be just the special bottom symbol located on both on the main and the reference stack.

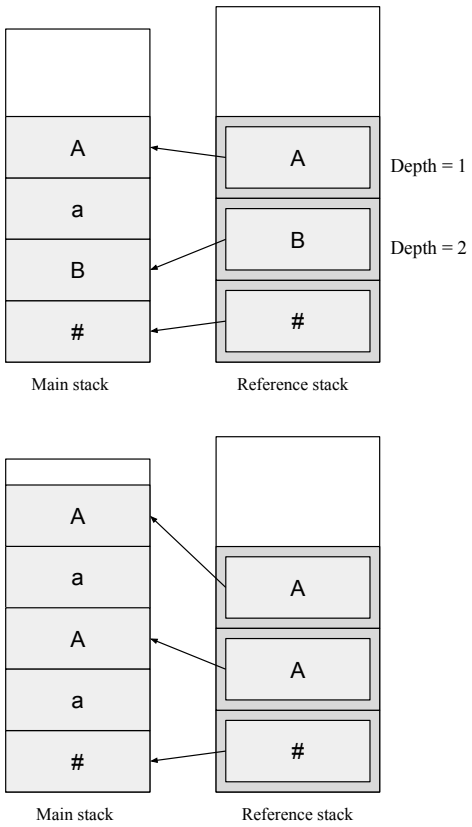

Figure 4.3: Expansion using the rule $2pB \rightarrow qAa$.

The supportive methods includes function such as `isPDEmpty()` that checks if just the main stack contains only special symbol #, function `numOfNonInput()` that returns number of non-input symbols on the stack, which can be used to check if a rule can be applied without breaking the rule of maximum of n non-input symbols on the pushdown and function `isExpansionDone()` that checks if there is only the special symbol # on the reference stack, therefore the pops can follow.

4.2.4 Deep Pushdown Automaton

The n-expandable deep pushdown automaton structure `NDPDA` is created after specifying the n, a starting state, an initial pushdown symbol, a set of final states and a set of rules.

Expansion rules are added using method `addRule(depth, from state, non-input symbol, to state, list of strings of input, non-input symbols)`. The function not only creates object of a rule `Rule` but also parses it to deduce the non-input symbols and states to create the complete definition of the created automaton.

While the rules are being added there is no way of telling which elements are the input symbols until we are sure we have the complete set of the non-input symbols so we created the method `automatonSettingDone()`. The method differentiates the non-input symbol based on the fact they have never been used on the left side of the expansion rule. The method also converts the string of non-input and input symbols of the right side of the rule into an set of `PDSymbols`. The method gets called automatically when the parsing begins or

can be call explicitly if we want to have the complete definition of the automaton without the actual parsing.

The method `simulation()` is a very simple parsing method where the user has to specify the order in which the rules have to be applied. It is however enough to proof the that the automaton indeed does accept the input string. The functionality of the simulation is as follows:

1. Gradually select the rules from the input and check:
   (a) if the rule actually exist in automaton,
   (b) if the beginning state matches the current state of the automaton,
   (c) if the rule doesn't brake the n-extensibility rule.
2. Call the expansion of the deep pushdown using the given rule and checks if the expansion was successful.
3. If all the expansions have been applied, simulation attempts to pop all of the input symbols left on the pushdown.
4. If the pushdown now contains only the bottom symbol # the parsing was successful.

4.2.5 Deep Pushdown Automaton Reduction

A subclass `NDPDAr` of class `NDPDA` was created to be able to restrict the number of non-input symbols in the pushdown alphabet to only one special symbol \$ and the bottom symbol #. This automaton can be created the same way at the `NDPDA` automaton by specifying the n, start state, initial pushdown symbol, set of final states and set of rules or it can be created by converting the `NDPDA` into the `NDPDAr`. The conversion between the two automata is based on the construction defined earlier in the theses.

For every rule $mqA \to pv \in R$ `convertRule()` is applied, and rule $m\langle q; uAz\rangle\$ \to \langle p; uf(v)z\rangle g(v)$ is created for every possible u and z where $u, z \in N^*$, $|u| = m - 1$, $|z| \leq n - m - 1$, $m \in \mathbb{N}$, $q, p \in Q$, $A \in N$, and $v \in (\Gamma \setminus \{\#\})^+$. This is achieved by creating all the permutation of all the non-input symbols (except the special symbol #) of maximum length of $n-2$ and minimum length of $m-1$. The left side of the rule is than created with the original depth m, a new state that is combination of the original state q and the new created sequence where the non-input symbol A is placed in the mth position and instead of the non-input symbol, special symbol \$. The right side of the rule is again created by new state that consist of combination of the original state p and the new sequence gained by permutation, with all the non-input symbols from v added to the mth place. The state is followed by v where the non-input symbols have been replaced by special symbol \$.

Throughout this conversion all the new created states are added to new set of states, starting state is $s_R = \langle s; S\rangle$ and if p state is in the set of the original final states the new state $\langle p; u\rangle$ is added to the new set of final states.

For example the rule $1qA \to fab$ of 2-expandable deep PDA with only two non-input symbols A and S will be converted to rules:

1. $1\langle q; A\rangle\$ \to \langle f; \varepsilon\rangle$
2. $1\langle q; AA\rangle\$ \to \langle f; A\rangle ab$
3. $1\langle q; AS\rangle\$ \to \langle f; S\rangle ab$

Except for this new constructor `NDPDAr` public method are identical to the ones in `NDPDA`.

4.3 Testing

In the testing part of application we will create previously mentioned automaton accepting the language $L = \{a^n b^n c^n \mid n \leq 0\}$ and automaton that is able to detect if the variable was defined before we try assigning a value. We will then use them to parse example languages and show what exactly happens on the pushdown during the parsing.

4.3.1 Creating Simple Automaton

First, we create a new 3-expandable `NDPDA` that is able to parse language

$$L = \{a^n b^n c^n \mid n \leq 0\}$$

with a starting state s, starting pushdown symbol S and set of end states $\{f\}$

```
NDPDA automaton = new NDPDA(3, "s", "S", [ "f" ]);
```

then add the individual rules

```
automaton.addRule("s", "S", "q", ["A", "A"]);
automaton.addRule("q", "A", "f", ["a", "b"]);
automaton.addRule("f", "A", "f", ["c"]);
automaton.addRule("q", "A", "p", ["a", "A", "b"]);
automaton.addRule(2, "p", "A", "q", ["A", "c"]);
```

If depth is not specified it's automatically set as 1. This is the output structure of automaton after adding all the rules:

```
M = (Q, Σ, Γ, R, s, S, F)
Q = {p, q, s, f}
Σ = {b, a, c}
Γ = {b, a, c, #, A, S}
R = {
    1: 1sS->qAA
    2: 1qA->fab
    3: 1fA->fc
    4: 1qA->paAb
    5: 2pA->qAc
}
F = {f}
```

Now we can run the simulation using

```
automaton.simulate("aabbcc", rulesNum);
```

where *„aabbcc“* is the input string and `rulesNum` is list of numbers indicating which rules have to be used in our case $[1, 4, 5, 2, 3]$. The output then prints the applied rules and the state of the main push down after every applied rule:

Input string: aabbcc

```
{S:NONTERMINAL}
{#:BOTTOM}
```

```
Successfully applied: 1sS->qAA
{A:NONTERMINAL}
{A:NONTERMINAL}
```

{#:BOTTOM}

Successfully applied: 1qA->paAb
{a:TERMINAL}
{A:NONTERMINAL}
{b:TERMINAL}
{A:NONTERMINAL}
{#:BOTTOM}

Successfully applied: 2pA->qAc
{a:TERMINAL}
{A:NONTERMINAL}
{b:TERMINAL}
{A:NONTERMINAL}
{c:TERMINAL}
{#:BOTTOM}

Successfully applied: 1qA->fab
{a:TERMINAL}
{b:TERMINAL}
{b:TERMINAL}
{A:NONTERMINAL}
{c:TERMINAL}
{#:BOTTOM}

Successfully applied: 1fA->fc
{a:TERMINAL}
{a:TERMINAL}
{b:TERMINAL}
{b:TERMINAL}
{c:TERMINAL}
{c:TERMINAL}
{#:BOTTOM}

Expansion phase done!
Poping phase done!

=> Automaton M does accept given input.

Printing of the automaton using the method `toString()` can be called anytime but the method `automaton.automatSettingDone()` should be called beforehand to make sure all the input symbols are identified.

As previously mentioned the `NDPDAr` automaton can be created by conversion of the previous `NDPDA` automaton. More precisely using constructor

```
NDPDAr reducedAutomaton = new NDPDAr(automaton);
```

This specific conversion outputs all of the rules created based on the originals:

```
Converting rule: 1sS->qAA
        1<s;S>$ -> <q;AA>$$
        1<s;SA>$ -> <q;AAA>$$
        1<s;SS>$ -> <q;AAS>$$
    Converting rule: 1qA->fab
        1<q;A>$ -> <f;>ab
        1<q;AA>$ -> <f;A>ab
        1<q;AS>$ -> <f;S>ab
    Converting rule: 1fA->fc
        1<f;A>$ -> <f;>c
        1<f;AA>$ -> <f;A>c
        1<f;AS>$ -> <f;S>c
    Converting rule: 1qA->paAb
        1<q;A>$ -> <p;A>a$b
        1<q;AA>$ -> <p;AA>a$b
        1<q;AS>$ -> <p;AS>a$b
    Converting rule: 2pA->qAc
        2<p;AA>$ -> <q;AA>$c
        2<p;SA>$ -> <q;SA>$c
```

Giving us a new automaton specified as

```
M = (Q, Σ, Γ, R, sS, $, F)
Q = {sS, sSA, qAS, pAS, qAAA, f, fAA, fS, qAA, qA, pA, pAA,
     sSS, qAAS, fAS, fA, qSA, pSA}
Σ = {b, a, c}
Γ = {b, a, c, #, $}
R = {
     1:      1sS$–>qAA$$
     2:      1sSA$–>qAAA$$
     3:      1sSS$–>qAAS$$
     4:      1qA$–>fab
     5:      1qAA$–>fAab
     6:      1qAS$–>fSab
     7:      1fA$–>fc
     8:      1fAA$–>fAc
     9:      1fAS$–>fSc
     10:     1qA$–>pAa$b
     11:     1qAA$–>pAAa$b
     12:     1qAS$–>pASa$b
     13:     2pAA$–>qAA$c
     14:     2pSA$–>qSA$c
}
F = {f, fA, fS}
```

Now we want to run the same simulation using the input string „*aabbcc*“ we have to adjust the set of rules that will be used to $(1, 11, 13, 5, 7)$ since the entire set of rules has changed during the conversion. We can now see and compare the parsing of the identical string using the reduced automaton.

Input string: aabbcc

{$:NONTERMINAL}
{#:BOTTOM}

Successfully applied: 1sS$->qAA$$
{$:NONTERMINAL}
{$:NONTERMINAL}
{#:BOTTOM}

Successfully applied: 1qAA$->pAAa$b
{a:TERMINAL}
{$:NONTERMINAL}
{b:TERMINAL}
{$:NONTERMINAL}
{#:BOTTOM}

Successfully applied: 2pAA$->qAA$c
{a:TERMINAL}
{$:NONTERMINAL}
{b:TERMINAL}
{$:NONTERMINAL}
{c:TERMINAL}
{#:BOTTOM}

Successfully applied: 1qAA$->fAab
{a:TERMINAL}
{a:TERMINAL}
{b:TERMINAL}
{b:TERMINAL}
{$:NONTERMINAL}
{c:TERMINAL}
{#:BOTTOM}

Successfully applied: 1fA$->fc
{a:TERMINAL}
{a:TERMINAL}
{b:TERMINAL}

{b:TERMINAL}
{c:TERMINAL}
{c:TERMINAL}
{#:BOTTOM}

Expansion phase done!
Poping phase done!

=> Automaton M does accept given input.

We can see the expansions are very similar, which only confirms the fact that they both have the same accepting power.

4.3.2 Variable Declaration Checking

Using the previous approach we will create a 5-expandable deep PDA with reduced push-down alphabet that will be able to accept pseudo code that declares variable and then assigns it a value. For example:

```
variable dog;
variable cat;
variable cow;
func {
    dog = value;
    cat = value;
    cow = value;
}
```

We will start by creating a regular 5-expandable deep PDA and adding individual rules same way as we did in the creation of the previous automaton. That will create the following structure:

```
M = (Q, Σ, Γ, R, s, S, F)
Q = {<r;a>, <u;y>, <u;u>, <t;x>, <r;z>, <u;q>, <t;t>, <r;v>,
    <u;m>, <t;p>, <r;r>, <u;i>, <t;l>, <r;n>, <u;e>, <t;h>,
    <r;j>, <u;a>, <t;d>, <r;f>, <r;b>, <u;z>, <u;v>, <t;y>,
    <u;r>, <r;w>, <t;u>, <u;n>, <r;s>, <t;q>, <u;j>, <r;o>,
    <t;m>, <u;f>, <r;k>, <t;i>, <u;b>, <r;g>, <t;e>, <r;c>,
    <t;a>, <t;z>, <t;v>, <r;x>, <u;w>, <t;r>, <r;t>, <u;s>,
    <t;n>, <r;p>, <u;o>, <t;j>, <r;l>, <u;k>, <t;f>,<r;h>,
    <u;g>, <t;b>, <r;d>, <u;c>, f, g, <r;y>, <t;w>, <u;x>,
    <r;u>, <t;s>, <u;t>, p, <r;q>, <t;o>, s, <u;p>, <r;m>,
    <t;k>,<u;l>, <r;i>,<t;g>, <u;h>, <r;e>, <t;c>, <u;d>}
Σ = {m, z, g, t, func, a, n, {, ;, h, u, b, o, var, i,
    val, v, c, p, }, =, j, w, d, q, k, x, e, r, l, y, f, s}
Γ = {m, z, g, t, func, a, n, {, ;, h, u, b, o, var, i,
    val, v, c, p, }, =, j, w, d, q, k, x, e, r, l, y, f, s,
    #, P, I, N, V, I', S}
R = {
    1:  1sS–>p V P
    2:  2pP–>p func { P }
    3:  2pP–>p I = N ; P
    4:  2pN–>p val
```

```
    5:  2pI–><r;a> a I'
    6:  1<r;a>V–>p var a V
    7:  2pI'–><t;a> a I'
    8:  1<t;a>V–>p a V
    9:  2pI'–><u;a> a
    10: 1<u;a>V–>p a ; V
    11: 2pI–><r;b> b I'
    12: 1<r;b>V–>p var b V
    13: 2pI'–><t;b> b I'
    14: 1<t;b>V–>p b V
    15: 2pI'–><u;b> b
    16: 1<u;b>V–>p b ; V
    .
    .
    .
    149: 2pI–><r;y> y I'
    150: 1<r;y>V–>p var y V
    151: 2pI'–><t;y> y I'
    152: 1<t;y>V–>p y V
    153: 2pI'–><u;y> y
    154: 1<u;y>V–>p y ; V
    155: 2pI–><r;z> z I'
    156: 1<r;z>V–>p var z V
    157: 2pI'–><t;z> z I'
    158: 1<t;z>V–>p z V
    159: 2pI'–><u;z> z
    160: 1<u;z>V–>p z ; V
    161: 2pP–>g
    162: 1gV–>f
}
F = {f}
```

Now that automaton is created we can reduce it, keep in mind that the reduction is done exactly as it is specified in the construction, so wast majority of rules will be never used. The reduction of unused rules of the automaton is not part of this thesis.

```
M = (Q, Σ, Γ, R, sS, $, F)
Q = {gNIS, <t;k>NI'SI', gNIV, <t;p>SI'PN, <t;p>SI'PI, gNII,
    <r;m>VSS, <r;m>VSV, gNIN, gNIP, <u;k>I'SI', <t;z>VVNI',
    pVPI'II', pINI'I', <t;p>SI'PV, pIIII', <t;p>SI'PS, gNI',
    <t;p>SI'PP, <t;m>SI', <t;o>VPNI', <t;e>VI'NS, <r;m>VVN,
    <t;f>II'VI', <r;m>VVP, <t;e>VI'NN, <r;m>VVS, <r;a>II'I',
    <t;e>VI'NP, <t;p>SI'PI', <r;m>VVI, <t;p>SI'NI, <t;e>VI'NI,
    <t;l>VPSI',...}
Σ = {m, z, g, t, func, a, n, {, ;, h, u, b, o, var, i, val,
    v, c, p, }, =, j, w, d, q, k, x, e, r, l, y, f, s}
Γ = {m, z, g, t, func, a, n, {, ;, h, u, b, o, var, i, val,
    v, c, p, }, =, j, w, d, q, k, x, e, r, l, y, f, s, #, $}
R = {
```

```
        1:   1sS$–>pVP $ $
        2:   1sSP$–>pVPP $ $
        3:   1sSS$–>pVPS $ $
        4:   1sSV$–>pVPV $ $
        5:   1sSI$–>pVPI $ $
        6:   1sSN$–>pVPN $ $
        7:   1sSI'$–>pVPI' $ $
        8:   1sSPP$–>pVPPP $ $
        9:   1sSPS$–>pVPPS $ $
        10:  1sSPV$–>pVPPV $ $
        11:  1sSPI$–>pVPPI $ $
        .
        .
        .
        41868:  1gVI'NI$–>fI'NI
        41869:  1gVI'NN$–>fI'NN
        41870:  1gVI'NI'$–>fI'NI'
        41871:  1gVI'I'P$–>fI'I'P
        41872:  1gVI'I'S$–>fI'I'S
        41873:  1gVI'I'V$–>fI'I'V
        41874:  1gVI'I'I$–>fI'I'I
        41875:  1gVI'I'N$–>fI'I'N
        41876:  1gVI'I'I'$–>fI'I'I'
    }
    F = {fPPS, fSVV, fPPP, fSVS, fII, fPPN, fSVP, fI'SI', fIN,
        fSVN, fIP, fPPI, fSVI, fIS, fISN, fIV, fISI...}
```

We can now show the behaviour of the pushdown while parsing the example pseudo code.

```
Input string:
var dog;
func {
    dog = val;
}
```

{$:NONTERMINAL}
{#:BOTTOM}

Successfully applied:
1sS$->pVP $ $
{$:NONTERMINAL}
{$:NONTERMINAL}
{#:BOTTOM}

Successfully applied:
2pVP$->pVP func { $ }
{$:NONTERMINAL}
{func:TERMINAL}
{{:TERMINAL}
{$:NONTERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Successfully applied:
2pVP$->pVINP $ = $; $
{$:NONTERMINAL}
{func:TERMINAL}
{{:TERMINAL}
{$:NONTERMINAL}
{=:TERMINAL}
{$:NONTERMINAL}

{;:TERMINAL}
{$:NONTERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Successfully applied:
2pVINP$-><r;d>VI'NP d $
{$:NONTERMINAL}
{func:TERMINAL}
{{:TERMINAL}
{d:TERMINAL}
{$:NONTERMINAL}
{=:TERMINAL}
{$:NONTERMINAL}
{;:TERMINAL}
{$:NONTERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Successfully applied:
1<r;d>VI'NP$->pVI'NP var d $
{var:TERMINAL}
{d:TERMINAL}
{$:NONTERMINAL}
{func:TERMINAL}
{{:TERMINAL}
{d:TERMINAL}
{$:NONTERMINAL}
{=:TERMINAL}
{$:NONTERMINAL}
{;:TERMINAL}
{$:NONTERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Successfully applied:
2pVI'NP$-><t;o>VI'NP o $
{var:TERMINAL}
{d:TERMINAL}
{$:NONTERMINAL}
{func:TERMINAL}
{{:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{$:NONTERMINAL}
{=:TERMINAL}
{$:NONTERMINAL}
{;:TERMINAL}
{$:NONTERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Successfully applied:
1<t;o>VI'NP$->pVI'NP o $
{var:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{$:NONTERMINAL}
{func:TERMINAL}
{{:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{$:NONTERMINAL}
{=:TERMINAL}
{$:NONTERMINAL}
{;:TERMINAL}
{$:NONTERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Successfully applied:
2pVI'NP$-><u;g>VNP g
{var:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{$:NONTERMINAL}
{func:TERMINAL}
{{:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{g:TERMINAL}
{=:TERMINAL}
{$:NONTERMINAL}
{;:TERMINAL}
{$:NONTERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Successfully applied:
1<u;g>VNP$->pVNP g ; $
{var:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{g:TERMINAL}
{;:TERMINAL}
{$:NONTERMINAL}

{func:TERMINAL}
{{:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{g:TERMINAL}
{=:TERMINAL}
{$:NONTERMINAL}
{;:TERMINAL}
{$:NONTERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Successfully applied:
2pVNP$->pVP val
{var:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{g:TERMINAL}
{;:TERMINAL}
{$:NONTERMINAL}
{func:TERMINAL}
{{:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{g:TERMINAL}
{=:TERMINAL}
{val:TERMINAL}
{;:TERMINAL}
{$:NONTERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Successfully applied:
2pVNP$->pVP val
{var:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{g:TERMINAL}
{;:TERMINAL}
{$:NONTERMINAL}
{func:TERMINAL}
{{:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{g:TERMINAL}
{=:TERMINAL}
{val:TERMINAL}
{$:NONTERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Successfully applied:
2pVP$->gV
{var:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{g:TERMINAL}
{;:TERMINAL}
{$:NONTERMINAL}
{func:TERMINAL}
{{:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{g:TERMINAL}
{=:TERMINAL}
{val:TERMINAL}
{;:TERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Successfully applied:
1gV$->f
{var:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{g:TERMINAL}
{;:TERMINAL}
{func:TERMINAL}
{{:TERMINAL}
{d:TERMINAL}
{o:TERMINAL}
{g:TERMINAL}
{=:TERMINAL}
{val:TERMINAL}
{;:TERMINAL}
{}:TERMINAL}
{#:BOTTOM}

Expansion phase done!
Poping phase done!

=> Success: Automaton M does accept given input.

4.3.3 More Examples

Examples mentioned previously and more can be found and ran from file `NDPDA/RunExample.java` using methods

- `runExample1()` which creates an automaton `NDPDA` to parse language $a^n b^n c^n$, then converts it to `NDPDAr` and runs exemplary parsing of a string *aabbcc*.
- `runExample2()` which creates an automaton `NDPDA` that parses the code for checking if variable was declared before assigning value. Than running exemplary parsing of pseudo code.
- `runExample3()` creates a simplified version of previous automaton, specifying that the name of the variable can be only created by multiple symbols a. Than runs conversion to `NDPDAr` and parses an exemplary pseudo code.

Chapter 5

Conclusion

In the present thesis, we showed a way of reduction of finitely expandable deep PDAs with respect to the number of non-input pushdown symbols. This has been done by saving the information of the non-input symbols in the states of the automaton and substituting them on a pushdown using special symbol \$. We have proven the newly acquired automaton has the same acceptance power as its non-reduced version and therefore they can be interchanged. This new finding was accepted by the international journal dedicated to computer science and its mathematical foundations *Schedae Informaticae* and will be published.

As part of the analysis of the non context-free languages, an automaton that can parse such structure was created. This automaton enables the possibly of parsing a context sensitive pseudo code resembling common structure in modern programming languages. It has also been use to showcase the functionality of the implementation part of this thesis. Implemented part includes a robust structure to test and demonstrate the construction process of the reduction and parsing of the input string. It provides a simple interface to define, reduce and run an automaton.

Before closing this thesis, we suggest some open problem areas related to this subject for the future investigation.

1. Can we reduce these automata with respect to the number of states?

2. Can we simultaneously reduce them with respect to the number of both states and non-input pushdown symbols?

3. Can we achieve the reductions described above in terms of general deep PDAs, which are not finitely expandable? As a matter of fact, Lemma 2 holds for these automata, so it can be considered as a preliminary result related to this investigation area.

4. Can we formally define the relation between state grammar and matrix grammars of finite index and corollary the relation between deep pushdown automaton and n-expandable pushdown automaton?

Bibliography

[1] Arratia, A.; Stewart, I. A.: Program Schemes with Deep Pushdown Storage. In *Proc. of Logic and Theory of Algorithms, CiE 2008*, *Lecture Notes in Computer Science*, vol. 5028. Springer. 2008. ISBN 978-3-540-69405-2. pp. 11–21. doi:10.1007/978-3-540-69407-6_2.

[2] Bar-Hillel, Y.; Perles, M. A.; Shamir, E.: On Formal Properties of Simple Phrase Structure Grammars. *Zeitschrift fur Phonetik, Sprachwissenschaft und Kommunikationsforschung*. vol. 14. 1961: pp. 143–172.

[3] Chomsky, N.: On certain formal properties of grammars. *Information and Control*. vol. 2, no. 2. 1959: pp. 137 – 167. ISSN 0019-9958. doi:http://dx.doi.org/10.1016/S0019-9958(59)90362-6. Retrieved from: http://www.sciencedirect.com/science/article/pii/S0019995859903626

[4] Cocke, J.: *Programming Languages and their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University. 1969. ISBN B0007F4UOA.

[5] Dassow, J.; Paun, G.: *Regulated Rewriting in Formal Language Theory*. Springer-Verlag Berlin Heidelberg. 1989. ISBN 9783642749346.

[6] Dvorakova, L.; Meduna, A.: A Reduction of Finitely Expandable Deep Pushdown Automata. *Schedae Informaticae*. In press.

[7] Greibach, S. A.: A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM*. vol. 12, no. 1. January 1965: pp. 42–52. ISSN 0004-5411. doi:10.1145/321250.321254. Retrieved from: http://doi.acm.org/10.1145/321250.321254

[8] Harrison, M. A.: *Introduction to Formal Language Theory*. Addison-Wesley. 1978. ISBN 9780201029550.

[9] Hopcroft, J. E.; Motwani, R.; Ullman, J. D.: *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.. 2006. ISBN 0321462254.

[10] Kalra, N.; Kumar, A.: Fuzzy state grammar and fuzzy deep pushdown automaton. *Journal of Intelligent and Fuzzy Systems*. vol. 31, no. 1. 2016: pp. 249–258. doi:10.3233/IFS-162138.

[11] Kalra, N.; Kumar, A.: State Grammar and Deep Pushdown Automata for Biological Sequences of Nucleic Acids. *Current Bioinformatics*. vol. 11, no. 4. 2016: pp. 470–479. doi:10.2174/1574893611666151231185112.

[12] Kasami, T.: An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages. Technical report. 1966.
Retrieved from: http://hdl.handle.net/2142/74304

[13] Leupold, P.; Meduna, A.: Finately Expandable Deep PDAs. In *Proc. of Automata, Formal Languages and Algebraic Systems 2008*. World Scientific. 2010. ISBN 9789814317603. pp. 113–123.

[14] Meduna, A.: *Automata and Languages: Theory and Applications*. Springer. 2000.

[15] Meduna, A.: Deep Pushdown Automata. *Acta Inf.*. vol. 42, no. 8-9. 2006: pp. 541–552. doi:10.1007/s00236-006-0005-0.

[16] Meduna, A.; Zemek, P.: *Regulated Grammars and Automata*. Springer. 2014.

[17] Rozenberg, G.: *Handbook of Formal Languages, Volume 1 Word, Language, Grammar*. Springer Berlin Heidelberg. 1997. ISBN 978-3-642-63863-3. doi:10.1007/978-3-642-59136-5.
Retrieved from: https://dx.doi.org/10.1007/978-3-642-59136-5

[18] Younger, D. H.: Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*. vol. 10, no. 2. 1967: pp. 189 – 208. ISSN 0019-9958. doi:http://dx.doi.org/10.1016/S0019-9958(67)80007-X.
Retrieved from:
http://www.sciencedirect.com/science/article/pii/S001999586780007X