BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FINAL SENTENTIAL FORMS AND THEIR APPLICATIONS

KONCOVÉ VĚTNÉ FORMY A JEJICH APLIKACE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. TOMÁŠ KOŽÁR**
AUTOR PRÁCE

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

BRNO 2022

Department of Information Systems (DIFS) Academic year 2021/2022

Master's Thesis Specification

24245

Student:	**Kožár Tomáš, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Mathematical Methods
Title:	**Final Sentential Forms and Their Applications**
Category:	Theoretical Computer Science

Assignment:

1. Study selected topics of formal languages, including grammatical sentential forms.
2. Based upon the supervisor's instructions, introduce final context-free sentential forms and study them.
3. Based upon the supervisor's suggestion, consider a variety of applications that make use of final context-free sentential forms, such as linguistic applications. Discuss their advantages and disadvantages.
4. Based upon the supervisor's recommendation, implement selected applications from item 3. Compare them with other existing software tools.
5. Summarize the work and discuss its future development.

Recommended literature:

- Meduna, A.: Automata and Languages, Springer, London, 2000, ISBN 978-1-85233-074-3
- Rozenberg, G. and Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1-3, Springer, 1997, ISBN 3-540-60649-1

Requirements for the semestral defence:

- Items 1 through 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Meduna Alexander, prof. RNDr., CSc.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 18, 2022
Approval date:	October 26, 2021

Abstract

Context-free grammars are one of the most used formal models in formal language theory. They have many useful applications, but for many applications, they lack expressive power. We introduce a final language F. When a sentential form of the context-free grammar G belongs to the F, it becomes a final sentential form. By the erasion of the nonterminals from the final sentential forms, we receive a language of G finalized by $F, L(G, F)$. We prove that for each recursively enumerable language L, there exists context-free grammar G, such that $L = L(G, F)$, with $F = \{w\#w^R \mid w \in \{0,1\}^*\}$, where w^R is a reversal of w. When a regular language is used as F, no increase in generative power compared to context-free grammars is achieved. We show multiple applications of the final sentential forms in the fields of the linguistics and bioinformatics.

Abstrakt

Bezkontextové gramatiky sú jedny z najpoužívanejších formálnych modelov v teórii formálnych jazykov. Majú mnoho užitočných použití, no pre mnoho aplikácii nemajú dostatočnú vyjadrovaciu silu. Preto zavádzame koncový jazyk F. Keď vetná forma bezkontextovej gramatiky G patrí do jazyka F, stáva sa konečnou vetnou formou. Odstránením neterminálov z konečných vetných foriem získavame jazyk gramatiky G ukončený jazykom $F, L(G, F)$. Dokazujeme, že pre každý rekurzívne vyčísliteľný jazyk existuje jazyk $L(G, F)$, kde $F = \{w\#w^R \mid w \in \{0,1\}^*\}$ a w^R je obrátené slovo w. Keď ako F použijeme regulárny jazyk, nedosiahneme žiadne zvýšenie vyjadrovacej sily oproti bezkontextovým gramatikám. Ukazujeme viacero aplikácii koncových vetných foriem v oblastiach lingvistiky a bioinformatiky.

Keywords

formal grammars, sentential forms, minimal linear grammars, recursively enumerable languages, Cocke-Younger-Kasami, CYK

Kľúčové slová

formálne gramatiky, vetné formy, minimálne lineárna gramatiky, rekurzívne vyčísliteľné jazyky, Cocke-Younger-Kasami, CYK

Reference

KOŽÁR, Tomáš. *Final Sentential Forms and Their Applications.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. RNDr. Alexander Meduna, CSc.

Rozšírený abstrakt

Témou tejto práce sú koncové vetné formy bezkontextových gramatík. Bezkontextové gramatiky predstavujú v obore teórie formálnych jazykov jeden z najpoužívanejších formálnych modelov. Aj keď majú bezkontextové gramatiky množstvo zásadných praktických aplikácii, pre veľké množstvo aplikácii je ich vyjadrovacia sila nedostačujúca. Preto sa v tejto práci venujeme ukončovaniu vetných foriem. Toto ukončenie nám poskytuje mechanizmus, ktorým dokážeme zmeniť spôsob konštrukcie výsledného jazyka. Koncové vetné formy sú také vetné formy, ktoré dokážeme po transformácii priradiť do koncového jazyka F. Vyjadrovacia sila koncových vetných foriem záleží na výbere jazyka F. V práci ukazujeme, že ak je $F = \{w\#w^R \,|\, w \in \{0,1\}^*\}$, môžeme vyjadriť ľubovoľný rekurzívne vyčísliteľný jazyk. Ak ako F zvolíme regulárny jazyk, vyjadrovacia sila zostane oproti vyjadrovacej sile bezkontextových gramatík nezmenená.

Na začiatku práce definujeme základné pojmy ako je napríklad symbol, slovo a jazyk. Definované pojmy predstavujú fundamentálny základ teórie formálnych jazykov. Ďalej predstavujeme viaceré formálne modely. Konečné automaty, ktoré reprezentujú formálne jazyky pomocou prijímania jednotlivých slov. Ďalej viacero verzií gramatík, ktoré formálne jazyky reprezentujú pomocou generovania jednotlivých slov. Predstavené modely je možné medzi sebou porovnávať na základe ich vyjadrovacej sily. Ako základ pre porovnanie slúži Chomského hierarchia, no sú zavedené aj ďalšie porovnania, ktoré z tejto hierarchie nie sú jasné.

Samotným jadrom práce sú koncové vetné formy. Tie definujeme pomocou množiny W, koncového jazyka F, slabej identity ω a bezkontextovej gramatiky G. Množina W je podmnožinou všetkých terminálov a neterminálov gramatiky G. Koncový jazyk F je definovaný práve nad množinou W. Slabá identita ω zo vstupného slova vymaže všetky symboly, ktoré nepatria do definovanej množiny. Bezkontextová gramatika G slúži na generovanie vetných foriem. Na to, aby slovo generované bezkontextovou gramatikou patrilo do výsledného jazyka, musí obsahovať len terminálne symboly. Vetné formy sú ľubovoľné slová, ktoré bezkontextová gramatika môže vygenerovať. Môžu teda obsahovať terminály aj neterminály. Aby sa vetná forma gramatiky G stala koncovou vetnou formou, aplikujeme na ňu slabú identitu ω podľa množiny W a skúsime ju priradiť do jazyka F. Ak po aplikácii slabej identity vetná forma patrí do F, jedná sa o koncovú vetnú formu. Pre získanie výsledného jazyka $L(G, F)$ vymažeme z koncových vetných foriem všetky neterminály.

V práci skúmame dva druhy konečného jazyka F. Ako prvé dokazujeme, že na to, aby jazyk $L(G, F)$ mohol vyjadriť ľubovoľný rekurzívne vyčísliteľný jazyk, stačí nám práve jeden koncový jazyk F, a to $F = \{w\#w^R \,|\, w \in \{0,1\}^*\}$. Na druhej strane, ak ako F zvolíme regulárny jazyk, nevedie to k žiadnej zmene vyjadrovacej sily oproti bezkontextovým gramatikám. Oba tieto tvrdenia sú v práci rigorózne dokázané.

Pred tým než ukážeme aplikácie koncových vetných foriem, musíme definovať spôsob, ako zistiť či je dané slovo súčasťou jazyka $L(G, F)$. Tento problém sa rieši pomocou syntaktickej analýzy. Ako základ pre koncové vetné formy využívame bezkontextové gramatiky. Preto aj pri syntaktickej analýze ako základ využívame algoritmus pre syntaktickú analýzu bezkontextových gramatík. Algoritmus, ktorý sme vybrali sa nazýva Cocke-Younger-Kasami (ďalej len CYK). Tento algoritmus požaduje vstupnú gramatiku v Chomského normálovej forme. To znamená, že každé prepisovacie pravidlo môže mať na pravej strane buď jeden terminál alebo dva neterminály. Najväčším problémom pri syntaktickej analýze koncových vetných foriem je zmazanie neterminálov z koncových vetných foriem pri vytváraní jazyka $L(G, F)$. Algoritmus CYK upravujeme tak, aby zvládol rekonštruovať pôvodnú vetnú formu. Keďže pôvodná vetná forma musí byť aj koncovou vetnou formou,

pre každý symbol uchovávame históriu redukcií, na základe ktorej zrekonštruujeme pôvodnú vetnú formu a testujeme, či je aj koncová.

Algoritmus CYK požaduje vstupnú gramatiku v Chomského vetnej forme. My však zavádzame ďalšie obmedzenia, aby bola syntaktická analýza uskutočniteľná. Žiaden neterminál z W sa nesmie vyskytovať na ľavej strane prepisovacieho pravidla a pravá strana prepisovacieho pravidla nesmie obsahovať len neterminály z W. Tieto obmedzenia sú nutné, ináč by sme mohli rekonštruovať vetné formy donekonečna.

Popísaná syntaktická analýza je implementovaná vo forme programu za účelom demonštrácie koncových vetných foriem na konkrétnych aplikáciách. Tento program je implementovaný v jazyku Python. Užívateľ vo vstupnom súbore definuje vstupnú bezkontextovú gramatiku a množinu W, z ktorej sa odvodí jazyk F. Algoritmus CYK vyžaduje gramatiku v Chomského normálovej forme. Pre užívateľa by bolo náročné a nepraktické tvoriť gramatiku v tejto normálovej forme. Preto je súčasťou implementácie algoritmus na transformáciu bezkontextovej gramatiky do Chomského normálovej formy. Následne je implementovaná modifikácia algoritmu CYK pre koncové vetné formy.

V práci uvádzame niekoľko aplikácii koncových vetných foriem. Pri aplikáciách sa zaujímame o problémy, ktoré nie je možné reprezentovať pomocou bezkontextových gramatík. Prvou aplikáciou je jazyk, ktorý korešponduje k jazyku $L = \{ww \mid w \in \Sigma^*\}$, kde Σ je abecedou. Tento jazyk je jedným z klasických príkladov jazyka, ktorý nie je možné vyjadriť pomocou bezkontextových gramatík. Ďalšou aplikáciou je jazyk, ktorý obsahuje gramaticky správne a taktiež pravdivé anglické vety o prarodičoch. Tento jazyk koresponduje s jazykom $\{a^n b^n c^n \mid n \geq 0\}$. Ďalšou aplikáciou je jazyk, ktorý obsahuje také dvojice binárnych čísel s rovnakým počtom bitov, že hodnota ľavého čísla je väčšia ako toho pravého. Posledná aplikácia je z oblasti bioinformatiky. Jedná sa o zjednodušenú reprezentáciu sekundárnej štruktúry, anglicky nazývanej pseudoknot.

Final Sentential Forms and Their Applications

Declaration

I hereby declare that this master's thesis was prepared as an original work by the author under the supervision of prof. RNDr. Alexander Meduna, CSc. The supplementary information about application of formal languages in bioinformatics was provided by Ing. Ivana Burgetová, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Tomáš Kožár
May 17, 2022

Acknowledgements

Chcel by som sa hlavne poďakovať vedúcemu práce za jeho odborné konzultácie a rady, bez ktorých by táto práca nevznikla. Ďalej by som sa chcel poďakovať svojej mame, za jej snahu, výchovu a trpezlivosť. V neposlednej rade by som chcel poďakovať všetkým kamarátom, ktorý mi pomáhali vyhýbať sa písaniu tejto práce.

Contents

Chapter 1

Introduction

Theoretical informatics represent a huge part of the information technology. Specifically, the field of the formal language theory. Formal language theory studies mainly properties, complexity, and the means of representation of the formal languages. Informally, formal languages are sets consisting of words—also called strings. To represent formal languages, there are two major methods. Automata and grammars. Automata are used to represent computation and to accept formal languages. On the other hand, grammars act only as language generators. There are many different versions of automata and grammars. These versions differ in the mechanism they use, and in the restrictions that are used to constrain them. The main property of mentioned formal models is the language that they represent. We can classify languages based on their complexity. For example, take two languages. The first one consists of words consisting of odd numbers of xs. The second one consists of words, such that each word is triple of some input, some algorithm, and the output of said algorithm for the mentioned input. Notice that the first language is trivial compared to the second one. For the first language, we need to use a formal model that is able to somehow count an odd number of occurrences of the symbol x. For the second one, we need a formal model that is able to produce an output of an algorithm for any correct input.

The set of all the languages that we can represent by some formal model is called *language family*. Since the language families are sets, we can compare them to each other by a subset or superset relation. Based on the subsets of the language families, we can say that some formal models are more powerful than others. Intuitively, the more powerful the language family we want to represent, the more complex the formal model we need. Apart from making the model itself more complex, we can combine multiple weaker models to raise their generative power.

The most important language family is the family of *recursively enumerable languages*. This language family represents all of the problems, that we are able to solve algorithmically. Meaning that we are guaranteed to get the result. In this thesis, we prove, that by combining the context-free grammars and a single minimal-linear language, we are able to represent the whole recursively enumerable language family.

The theoretical informatics and formal language theory rely on mathematical notation and formal definitions. The basic definitions are formal, but they often vary slightly across different sources. Therefore, Chapter 2 is dedicated to the basic definitions that are needed throughout this thesis. At first, we define a symbol, an alphabet, a word, and a language. These definitions are absolutely crucial in the formal language theory. Next, we describe the formal models that we use to represent the languages in this thesis. The finite automata work as language acceptors, meaning that they get a word as an input and they decide, whether the input word belongs to the language that they represent. The grammars are language generators. They work by iterative application of their rules that rewrite their left-side to their right-side, in order to generate words. The language of some grammar is a set of all the words it generates. In this thesis, context-free grammars are of great interest. Therefore, we describe their properties in greater detail.

In this thesis, we introduce the notion of *final sentential forms*, which are defined in Chapter 3. Let W be a set of symbols and F be a *final language*. We use context-free grammar G to generate the *sentential forms*. Context-free grammar uses nonterminal and terminal symbols during the derivation. The nonterminal cannot occur in the words of the generated language. Only terminals. The sentential forms are any words that G generates, even ones with nonterminals. To obtain *final sentential forms*, we take the individual sentential forms and erase all the symbols that do not belong to W. If the resulting word belongs to the F, the original sentential form is a final sentential form. To obtain the resulting language $L(G, F)$ of the final sentential forms, we erase all the nonterminals from the final sentential forms. The power of the language family of $L(G, F)$ depends on the selection of the finalizing language F. We have proven that if $F = \{w\#w^R \mid w \in \{0, 1\}^*\}$, we achieve the recursively enumerable language family. On the other hand, if we choose regular language as F, the resulting power is the same as the power of the context-free grammars. Both of these claims are rigorously proven.

The most important property of the language models is the language they represent. For the language, the most important problem is the membership problem. Does the word we provide belong to the given language? Chapter 4 describes the syntax analysis which is used to decide the membership problem. For context-free grammars, we describe the algorithm Cocke-Younger-Kasami (CYK for short). This algorithm works with the context-free grammars in Chomsky normal form, but any context-free grammar can be converted to such grammar. The reason we selected this algorithm is, that it can be modified for other grammars as well. We introduce a modification of the CYK algorithm, such that the resulting algorithm is able to perform syntax analysis of the final sentential forms. The biggest challenge of such modification is the erasion of the nonterminals from the final sentential forms. We introduce restrictions on the underlying context-free grammars in order to simplify the syntax analysis itself.

As part of this thesis, we implement the described modification of the CYK algorithm for the final sentential forms. The purpose of this implementation is to demonstrate the final sentential forms on the practical applications. The first part of Chapter 5 describes the implementation of syntax analysis of the final sentential forms. For the syntax analysis, we need to provide the context-free grammar, set W, and the input word. The input grammar and the set W are specified in a JSON file. The JSON filename and the input word are provided through a command-line interface. The algorithm CYK that is used requires the input context-free grammar to be in Chomsky normal form. Therefore, the transformation of the context-free grammar to Chomsky normal form is also implemented.

In the second part of Chapter 5, we introduce the applications of the final sentential forms. For these applications, we are interested only in non-context-free languages. The first application corresponds to the formal language $L_1 = \{ww \mid w \text{ is a word }\}$. The second one corresponds to the language $L_2 = \{a^n b^n c^n \mid n \geq 0\}$. The third application contains pairs of the binary numbers with the same number of bits, where the value of the left binary number is greater than the value of the right binary number. The last application is from the field of bioinformatics and it represents the secondary structure of the RNA. None of these languages can be represented by the context-free grammar.

Chapter 2

Languages, automata and grammars

Before the actual subject of this thesis can be presented, we need to define basic notions of formal language theory. The formal language theory studies formal languages, their properties, and the ways that these languages can be formally represented. Natural languages are complex, with many rules regarding the form of the individual words, sentence structure, and so on. However, there are many exceptions to these rules. Sometimes, even linguists cannot agree on some disputes regarding natural languages. On the contrary, the formal languages are exact and are represented by the formal models. The definitions presented in this chapter are crucial to understand the topic of final sentential forms. It is assumed that the reader is familiar with basic set theory, such as sets, subsets, inclusions, and set operators.

All of the definitions, if not explicitly stated otherwise, are taken from [4].

2.1 Symbols, alphabets, and words

The smallest unit in formal language theory is a symbol. We can think of symbols as letters in natural languages. The alphabets represent a domain of symbols, that can be used to construct words. As in natural languages, there is no single alphabet. Different languages can be made of different alphabets.

Definition 2.1. An **alphabet** is a finite, nonempty set of elements, which are called **symbols**.

Consequently, the words over the alphabet can be created as sequences of symbols from said alphabet. The special case of the word, an empty word, is denoted by ε and contains no symbols.

Definition 2.2. Let Σ be an alphabet. A **word** is recursively defined as

- ε is the word over Σ and
- if x is the word over Σ and $a \in \Sigma$, then ax is the word over Σ

A **string** is synonymous with a word and can be used interchangeably. In this thesis, the term word is used. Since the words are sequences of symbols, we can define their length. The length of the word is a number of all symbols in the word.

Definition 2.3. Let x be a word over an alphabet Σ. A **length** of x denoted as $|x|$ is defined as

- if $x = \varepsilon$, then $|x| = 0$
- if $x = a_1, ..., a_n$, for some $n \geq 1$, where $a_i \in \Sigma, 1 \leq i \leq n$, then $|x| = n$.

Many operations can be performed on the words. Some of those operations are used in this thesis and defined below. Specifically, *concatenation, reversal, prefix* and *suffix* of words.

Definition 2.4. Let x and y be words over an alphabet Σ. Then, xy is the **concatenation** of x and y.

The operation of concatenation is denoted by the operator $\cdot$, but is usually omitted.

Definition 2.5. Let x be a word over an alphabet, Σ. The **reversal** of x, $reversal(x)$, is defined as

- if $x = \varepsilon$, then $reversal(x) = \varepsilon$
- if $x = a_1, ..., a_n$, for some $n \geq 1$, where $a_i \in \Sigma, 1 \leq i \leq n$, then $reversal(x) = a_n...a_1$.

Reversal of $x, reversal(x)$, can also be denoted as x^R.

Definition 2.6. Let x and y be words over an alphabet Σ. Then, x is a **prefix** of y if there exists a word $z \in \Sigma^*$, such that $xz = y$.

For word y, $prefix(y, i)$ denotes prefix of y of length i for all $0 \leq i \leq |y|$.

Definition 2.7. Let x and y be words over an alphabet Σ. Then, x is a **suffix** of y if there exists a word $z \in \Sigma^*$, such that $zx = y$.

For word y, $suffix(y, i)$ denotes suffix of y of length i for all $0 \leq i \leq |y|$.

Definition 2.8. Let x and y be two words over an alphabet Σ. Then, x is a **subword** of y if there exist two words, z, z', over Σ so $zxz' = y$.

2.2 Languages

Now let us take the concept of the words and define languages. Informally speaking, a language is any set of words. Languages can be finite or infinite. In format language theory, we are usually interested in infinite languages. Languages also differ in their complexity. Consider languages $L_1 = \{x \,|\, x \text{ is a prime number}\}$ and $L_2 = \{w \,|\, w \text{ is any word}\}$. Intuitively, language L_1 is more complex than language L_2. To represent language L_1, we need a formal model that is able to decide which numbers are prime. On the other hand, language L_2 is trivial, since any word belongs to it. The more complex languages we want to represent, the more complex formal models need to be used. The *generative power* (power for short) of a formal model represents the complexity of the languages it can represent in comparison to the other models. Models may have same power, meaning they are equivalent. Some models are more/less powerful than others. Some models may have mutually incomparable power.

Let us consider an alphabet Σ. Let Σ^* denote the set of all words over Σ and set $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. Notice that Σ^+ is the set of all nonempty words over Σ. Now, the definition of the language can be formalized.

Definition 2.9. Let Σ be an alphabet, and let $L \subseteq \Sigma^*$. Then, L is a **language** over Σ.

By this definition, $\emptyset$ and $\{\varepsilon\}$ are languages over any alphabet. However, notice that $\emptyset \neq \{\varepsilon\}$. As languages are defined as sets, notions concerning sets also apply to languages.

Definition 2.10. A **cardinality** of a language $L, card(L)$ denotes the number of words that L contains.

For example, $card(\emptyset) = 0$ and $card(\{\varepsilon\}) = 1$, since ε is also a word. Therefore, mentioned non equivalence $\emptyset \neq \{\varepsilon\}$ holds true. A special case of cardinality is infinity. Based on this, the languages can be divided into two groups. Finite and infinite languages.

Definition 2.11. Let L be a language. L is **finite** if $card(L) = n$ for some $n \geq 0$; otherwise, L is **infinite**.

Let us consider operations of union, intersection, and difference of sets. Since the languages are defined as sets, mentioned operations are straightforward for languages. For two languages L_1 and L_2, these operations are defined as

$$\begin{aligned} L_1 \cup L_2 &= \{x \,|\, x \in L_1 \text{ or } x \in L_2\} \\ L_1 \cap L_2 &= \{x \,|\, x \in L_1 \text{ and } x \in L_2\} \\ L_1 - L_2 &= \{x \,|\, x \in L_1 \text{ and } x \notin L_2\} \end{aligned}$$

2.3 Finite automata

We have introduced the notion of the symbols, alphabets, words, and languages. However, until now, we described the languages as mostly mathematical concept described by the sets. The theory of formal languages studies many formal models that are used to describe the languages. The main groups of these models are the language acceptors and the language generators. Automata, in our case the finite automata, are language acceptors. Each finite automaton consists of an input tape and a set of states. The input tape contains a word. Based on the symbols read from the input tape, the finite automaton can transition (move) between its states. The input tape is read from left to right and finite automaton can read each symbol on the tape only once. When the finite automaton reads the whole tape and its current state is a so-called *final state*, the input word is accepted. Otherwise, it is rejected. Using described mechanism, the finite automaton can either accept or reject the input word. There are many more different types of automata, that work in different ways. However, in this thesis, only finite automata are of interest.

Definition 2.12. A **finite automaton** is quintuple

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

where

- Q is a finite set of states

- Σ is an input alphabet such that $\Sigma \cap Q = \emptyset$
- $R \subseteq Q(\Sigma \cup \varepsilon) \times Q$ is a relation
- $s \in Q$ is a start state
- $F \subseteq Q$ is a set of final states.

Members of R are called *computational rules*, or simply, *rules*. R is referred to as a *finite set of rules*. For any rule $(pa, q) \in R$, where $p, q \in Q, a \in \Sigma \cup \varepsilon$ we write $pa \rightarrow q$.

As mentioned earlier, a finite automaton transitions between its states. To describe the current configuration of the finite automaton, we need to know an unread part of the input tape and its current state. A word that combines the current state and the unread input is called *configuration* of the finite automaton.

Definition 2.13. Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton. A **configuration** of M is a word χ satisfying $\chi \in Q\Sigma^*$.

Using this notion, we can now describe a *move* of a finite automaton. Let $pa \rightarrow q \in R$ be any computational rule of finite automaton. Define injection lhs from R to $Q\Sigma^*$ as $lhs(pa \rightarrow q) = pa$ and injection rhs from R to Q as $rhs(pa \rightarrow q) = q$.

Definition 2.14. Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton. If $lhs(r)y$ is a configuration of M, where $r \in R, y \in \Sigma^*$, then M makes a **move** from $lhs(r)y$ to $rhs(r)y$ according to r and is denoted as $lhs(r)y \vdash rhs(r)y \;\; [r]$, or simply $lhs(r)y \vdash rhs(r)y$.

The following definition extends a single move to the sequence of moves.

Definition 2.15. Let $M = (Q, \Sigma, R, s, F)$ be a finite automata.

1. Let χ be any configuration of M. M makes **zero moves** from χ to χ according to ε, written as $\chi \vdash^0 \chi \;\; [\varepsilon]$.
2. Let there exist a sequence of configurations $\chi_0, ..., \chi_n$ for some $n \geq 1$ such that $\chi_{i-1} \vdash \chi_i \;\; [r_i]$, where $r_i \in R, 1 \leq i \leq n$; that is $\chi_0 \vdash \chi_1 \;\; [r1] \vdash \chi_2 \;\; [r_2] \vdash ... \vdash \chi_n \;\; [r_n]$. Then, M makes n moves from χ_0 to χ_n according to $r_1...r_n$ written as $\chi_0 \vdash^n \chi_n \;\; [r_1...r_n]$.

Let χ and χ' be two configurations of M.

3. If there exists $n \geq 1$ so $\chi \vdash^n \chi' \;\; [\rho]$ in M, then $\chi \vdash^+ \chi' \;\; [\rho]$.
4. If there exists $n \geq 0$ so $\chi \vdash^n \chi' \;\; [\rho]$ in M, then $\chi \vdash^* \chi' \;\; [\rho]$.

Using defined notions of sequences of moves, words accepted by a finite automaton can be defined. By extending this definition, a language accepted by the finite automaton is defined.

Definition 2.16. Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton and $w \in \Sigma^*$. If there exists an **accepting computation** of form $sw \vdash^* f$, where $f \in F$, then M accepts w. The **language accepted by** M, denoted by $L(M)$, is defined as

$$L(M) = \{w \,|\, w \in \Sigma^* \text{ and } sw \vdash^* f, f \in F\}$$

A finite automaton defined in Definition 2.12 can have two potentially unwanted properties. It can contain so-called ε-moves that do not read any symbol and it can work in nondeterministic way.

Definition 2.17. Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton. M is an **ε-free finite automaton** if for all $r \in R, lhs(r) \in Q\Sigma$.

Any finite automaton can be converted to an equivalent ε-free finite automaton. Meaning that both automata accept the same language. An algorithm for this conversion can be found in [4] and is not discussed in this thesis.

Definition 2.18. Let $M = (Q, \Sigma, R, s, F)$ be an ε-free finite automaton such that for all $r, r' \in R, r \neq r'$ implies $lhs(r) \neq lhs(r')$. Then, M is a **deterministic finite automaton** (DFA for short).

A finite automaton that does not fulfill the property of DFA is called **nondeterministic finite automaton** (NFA for short). Similarly to ε-free finite automaton, there exists an algorithm that converts any ε-free finite automaton to an equivalent DFA. Therefore, for every NFA there exists an equivalent DFA. All the algorithms for conversions can be found in [4].

2.4 Grammars

Grammars are formal models that represent language generators and work by iterative rewriting of the their sentential forms. Grammar usually starts with a single nonterminal—*start symbol*. Then, grammar applies its rules to rewrite a left-side to a right-side of said rule. By repetition of the application of the rules, grammar generates the sentential forms. When the generated sentential form contains only terminals, it is considered to be a word and belongs to the language generated by said grammar. In this way, the grammars generates individual words of their languages.

Definition 2.19. A **unrestricted grammar** is a quadruple $G = (V, T, P, S)$, where

- V is a total alphabet;
- T is an alphabet of terminals such that $N \cap T = \emptyset$;
- $P \subseteq V^+ \times V^*$ is a finite relation;
- $S \in V - T$ is a start symbol.

Set $N = V - T$. N is an alphabet of nonterminals such that $N \cap T = \emptyset$. In some definitions, the quadruple $G = (N, T, P, S)$ is used. Members of P are called *productions*, *rewriting rules* or rules for short. Instead of $(u, v) \in P, u \in V^+, v \in V^*$ we write $u \to v$. For brevity, we often denote $u \to v \in P$ by a unique label p as $p : u \to v$ and we use p and $u \to v$ interchangeably. Define the injection lhs from P to V^+ as $lhs(u \to v) = u$ and injection rhs from P to V^* as $lhs(u \to v) = v$.

Since grammars are language generators, they contain a mechanism that allows them to generate words of a language. An unrestricted grammar $G = (V, T, P, S)$ uses its productions to generate individual words. Consider word xuy, where $x, y \in V^*$ and a production $p : u \to v \in P$. By using p, G directly derives xvy from xuy.

Definition 2.20. Let $G = (V, T, P, S)$ be an unrestricted grammar, $p \in P$, and $x, y \in V^*$. Then, $xlhs(p)y$ **directly derives** $xrhs(p)y$ according to p in G; symbolically $xlhs(p)y \Rightarrow xrhs(p)y\ \ [p]$, where $[p]$ can be omitted.

The following definition generalizes a direct derivation to a sequence of n derivation steps for $n \geq 0$.

Definition 2.21. Let $G = (V, T, P, S)$ be an unrestricted grammar.

1. For any $u \in V^*$, G makes a **zero-step derivation** from u to u according to ε, which is written as $u \Rightarrow^0 u\ \ [\varepsilon]$.
2. Let $u_0, ..., u_n \in V^*$, for some $n \geq 1$, such that $u_{i-1} \Rightarrow u_i\ [p_i]$, where $p_i \in P$, for $i = 1, ..., n$; that is, $u_0 \Rightarrow u_1\ \ [p_1] \Rightarrow u_2\ \ [p_2] \Rightarrow ... \Rightarrow u_n\ \ [p_n]$. Then G makes an n-step derivation from u_0 to u_n according to $p_1...p_n$, written as $u_0 \Rightarrow^n u_n\ \ [p_1...p_n]$.
3. If there exists $n \geq 1$ so $v \Rightarrow^n w\ \ [\pi]$ in G, then v **properly derives** w according to π in G written as $v \Rightarrow^+ w\ \ [\pi]$.
4. If there exists $n \geq 0$ so $v \Rightarrow^n w\ \ [\pi]$ in G, then v **derives** w according to π in G written as $v \Rightarrow^* w\ \ [\pi]$.

Derivations are often written without specifying productions $[\pi]$. Now that notion of derivations is defined, the sentential form, a sentence, and a language generated by a unrestricted grammar can be defined.

Definition 2.22. Let $G = (V, T, P, S)$ be a unrestricted grammar. If $S \Rightarrow^* w$ in G, then w is a **sentential form** of G. Set of all sentential forms generated by G is defined as $\phi(G) = \{w \in V^* \mid S \Rightarrow^* w\}$. A sentential form w, such that $w \in T^*$ is a **sentence** generated by G. The **language generated** by G, $L(G)$ is the set of all sentences that G generates; formally,

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

Restricted grammars

Unrestricted grammars represent the most general and most powerful grammars due to no restriction of the left-side of the rules. However, these grammars are not practical since working with such rules is difficult. As a result, more types of grammar are defined by the restrictions of the left-hand side of the rules. Sometimes, even the right-sides of the rules are restricted.

A *context-sensitive* grammar can rewrite only a single nonterminals, but it can select these nonterminals based on the surrounding symbols. These surrounding symbols are considered to be a context. A *context-free grammar* (CFG for short) can rewrite only single nonterminals, but these nonterminals cannot be selected based on the context. Otherwise, the mechanics of these grammars are identical to the unrestricted grammars.

Definition 2.23. A unrestricted grammar $G = (V, T, P, S)$ is a **context-sensitive** grammar if all rules $\alpha u \beta \rightarrow \alpha v \beta \in P$ satisfy $u \in V - T, \alpha, \beta \in V^*$ and $v \in V^+$. In addition P may contain the production $S \rightarrow \varepsilon$ and in this case S does not occur on the right side of any production of P.

Definition 2.24. A unrestricted grammar $G = (V, T, P, S)$ is a **context-free** grammar if all rules $u \to v \in P$ satisfy $u \in V - T, v \in V^*$.

Definition 2.25. A context-free grammar $G = (V, T, P, S)$ is a **regular** grammar if all rules $u \to v \in P$ satisfy $u \in V - T, v \in T((V - T) \cup \{\varepsilon\})$.

Definition 2.26. A context-free grammar $G = (V, T, P, S)$ is **linear** if no more than one nonterminal appears of the right-side of the rules from P. Formally, $P \subseteq (V - T) \times T^*(V - T)T^* \cup T^*$.

Definition 2.27. A linear grammar $G = (V, T, P, S)$ is a **minimal linear** if $V = T \cup \{S\}$ and $S \to \# \in P$, with $\# \in T$, is the only production with no nonterminal on the right side, and it is assumed that $\#$ does not occur in any other production. Formally, $P \subseteq \{S\} \times (T^*\{S\}T^* \cup \#)$.

The definition of the context-sensitive grammar is taken from [8]. The definitions of the linear and minimal grammars are taken from [9].

Definition 2.28. In this thesis, a minimal linear grammar $G = (N, T, P, S)$ is called a **palindromial grammar** if $|P| \geq 2$, and every rule of the form $S \to xSy$, where $x, y \in T$ satisfies $x = y$.

For instance, $H = (\{S, 0, 1, \#\}, \{0, 1, \#\}, \{S \to 0S0, S \to 1S1, S \to \#\}, S)$ is a palindromial grammar.

Notice that all of the grammars defined in Definitions 2.23 to 2.28 are more restricted versions of the unrestricted grammars. Therefore, these grammars are also unrestricted grammars. Due to this property, all of the notions defined for the unrestricted grammars apply to these grammars as well.

Queue grammars

The unrestricted grammars work by rewriting parts of the sentential forms directly in their position. However, we can modify the rewriting mechanism by which sentential forms are generated to obtain new kinds of grammars. One such example are the *queue grammars*. The queue grammars introduce a concept of states into the derivation. Similarly to the finite automata. A derived word consists of a sentential form over a total alphabet suffixed by a symbol representing a state in the queue grammar. The queue grammar always rewrites the leftmost symbol of the sentential form to some sequence of symbols that is appended after the sentential form. In front of the state symbol. This principle can be interpreted as a queue, hence the name. The queue grammar is formally defined as follows.

Definition 2.29. A **queue grammar** (see [2]) is a sextuple, $Q = (V, T, U, D, s, P)$, where

- V is an total alphabet of symbols;
- T is an alphabet of terminals such that $T \subseteq V$;
- U is an alphabet of states;
- D is an alphabet of final states such that $D \subseteq U$;
- $s \in (V - T)(U - D)$;

- $P \subseteq (V \times (U - D)) \times (V^* \times U)$ is a finite relation such that for for every $a \in V$, there exists an element $(a, b, z, c) \in P$.

Compared to the unrestricted grammars, queue grammars have more complex rules. Each rule is in the form $(a, b, z, c) \in P$, where $a \in V, b \in U - D, z \in V^*, c \in U$. The semantics of these rules are following. If the derivation is in state b, take the leftmost symbol a of the sentential form and erase it. Append z to the end of the sentential form and change the state symbol to c. An interesting property of the derivation is, that once a final state from D is reached, no further rule can be applied. This property is caused by the definition of P.

Definition 2.30. Let $Q = (V, T, U, D, s, P)$ be a queue grammar. If $u, v \in V^*U$ such that $u = arb; v = rzc; a \in V; r, z \in V^*; b, c \in U$; and $(a, b, z, c) \in P$, then u **directly derives** v according to (a, b, z, c) in Q; symbolically $u \Rightarrow v$ $[(a, b, z, c)]$ in Q or, simply, $u \Rightarrow v$. In the standard manner, extend $\Rightarrow$ to $\Rightarrow^n$, where $n \geq 0$; then, based on $\Rightarrow^n$, define $\Rightarrow^+$ and $\Rightarrow^*$.

Definition 2.31. The language of Q, $L(Q)$, is defined as $L(Q) = \{w \in T^* \mid s \Rightarrow^* wf$, where $f \in D\}$.

An example of the queue grammar is presented below.

Example 2.1. Let $G = (\{A, a, b\}, \{a, b\}, \{\overline{e}, \overline{f}\}, \{\overline{f}\}, A\overline{e}, P)$ be a queue grammar, where

$$P = \{1 : (A, \overline{e}, bAa, \overline{e}), 2 : (A, \overline{e}, \varepsilon, \overline{f}), 3 : (a, \overline{e}, a, \overline{e}), 4 : (b, \overline{e}, b, \overline{e})\},$$

and

$$\begin{aligned} A\overline{e} &\Rightarrow bAa\overline{e}\,[1] \Rightarrow Aab\overline{e}\,[4] \Rightarrow abbAa\overline{e}\,[1] \Rightarrow bbAaa\overline{e}\,[3] \Rightarrow bAaab\overline{e}\,[4] \\ &\Rightarrow Aaabb\overline{e}\,[4] \Rightarrow aabb\overline{f}\,[2] \end{aligned}$$

is an example derivation of $aabb \in L(G)$. The language generated by G is $L(G) = \{a^n b^n \mid n \geq 0\}$.

Example 2.1 is taken from the [1].

Definition 2.32. A **left-extended queue grammar** is a sextuple, $Q = (V, T, U, D, s, P)$, where V, T, U, D, and s have the same meaning as in a queue grammar. $P \subseteq (V \times (U - D)) \times (V^* \times U)$ is a finite relation (as opposed to an ordinary queue grammar, this definition does not require that for every $a \in V$, there exists an element $(a, b, z, c) \in P$). Furthermore, assume that $\# \notin V \cup U$. If $u, v \in V^*\{\#\}V^*U$ so that $u = w\#arb; v = wa\#rzc; a \in V; r, z, w \in V^*; b, c \in U$; and $(a, b, z, c) \in P$, then $u \Rightarrow v[(a, b, z, c)]$ in G or, simply $u \Rightarrow v$. In the standard manner, extend $\Rightarrow$ to $\Rightarrow^n$, where $n \geq 0$; then, based on $\Rightarrow^n$, define $\Rightarrow^+$ and $\Rightarrow^*$. The language of $Q, L(Q)$, is defined as $L(Q) = \{v \in T^* \mid \#s \Rightarrow^* w\#vf$ for some $w \in V^*$ and $f \in D\}$.

The left-extended queue grammar extends the queue grammars by recording the history of derivation. Every rewritten symbol is shifted left over $\#$. In this way, Q records the history of derivation.

Left-extended queue grammars play a crucial role in the proofs in this thesis.

2.5 Properties of context-free grammars

In formal language theory, context-free grammars are commonly studied, since CFGs represent a simple model that can represent many useful applications, such as programming languages. CFGs have countless properties. Some of these properties are specific to CFGs, but some are not.

The main purpose of the grammars, in general, is a generation of languages. However, grammar does not necessarily generate non-empty language. Consider grammar $G = (\{S, A\}, \{a\}, \{S \rightarrow A, A \rightarrow AA\}, S)$. It is apparent that the set of sentential forms of $G, \phi(G)$ is infinite, but the generated language of $G, L(G)$ is empty. There is no use for such grammar. This example of G is an extreme case. Grammars can contain only some parts of them that are useless. For example nonterminals, that cannot be derived to the sequence of terminals.

Definition 2.33. Let $G = (V, T, P, S)$ be a context-free grammar, and $X \in V - T$. X is **terminating** if there exists $w \in T^*$, such that $X \rightarrow^* w$ in G; otherwise, X is **nonterminating**.

Notice, that once a nonterminating symbol occurs in some sentential form, it can never be derived to a word of terminals. Therefore, the existence of such symbols in CFGs is unwanted.

Another instances of unwanted symbols in CFGs are *inaccessible* symbols. These symbols do not occur in any sentential form.

Definition 2.34. Let $G = (V, T, P, S)$ be a context-free grammar, and $X \in V$. X is **accessible** if $S \Rightarrow^* uXv$ in G, for some $u, v \in V^*$; otherwise, X is **inaccessible**.

When both nonterminating and inaccessible symbols are put together, we get useless.

Definition 2.35. Let $G = (V, T, P, S)$ be a context-free grammar, and $X \in V$. X is **useful** if $S \Rightarrow^* uXv \Rightarrow^* uxv$ in G, for some $u, v \in V^*$ and $x \in T^*$.; otherwise, X is **useless**.

Equivalently, X is useful if it is accessible and terminating. Otherwise, X is useless. Since useless symbols serve no purpose in CFGs, we want to remove them. Algorithms for removing useless symbols from CFGs can be found in [4].

In two previous definitions, we described symbols that are unwanted in CFGs. Of course, when we want to remove these symbols from CFG, we also have to remove any rules where these symbols occur. However, the main emphasis is on these symbols themselves. In the next two definitions, we define the forms of rules that are undesirable in CFGs.

Definition 2.36. Let $G = (V, T, P, S)$ be a context-free grammar and $A \in V - T$. A is an **erasable nonterminal** if $A \Rightarrow^* \varepsilon$ in G. G is a **ε-free context-free grammar** if it contains no erasable nonterminals.

ε-free context-free grammars are also-called **propagating**. An existence of ε-nonterminals implies existence of so-called *ε-rules*. Rules in the form of $X \rightarrow \varepsilon$ that erase nonterminals. ε-rules can be useful in the construction of grammar. However, such rules are problematic in the practical use of grammars, for example in an implementation.

Removal of ε-rules is possible. Consider CFG $G = (\{S, X, a, b\}, \{a, b\}, \{S \rightarrow aXa, S \rightarrow bXb, X \rightarrow aXa, X \rightarrow bXb, X \rightarrow \varepsilon\}, S)$. The language generated by G is $L(G) = \{ww^R \mid w \in \{a, b\}^+\}$. G contains single ε-rule—$X \rightarrow \varepsilon$. To remove this rule, add $S \rightarrow aa, S \rightarrow bb, X \rightarrow$

$AA, X \to bb$ to G. These rules simulate the application of $X \to \varepsilon$ later in the derivation. Notice that removal of ε-rules introduces new rules into a CFG, therefore it increases its size.

Notice, that $\varepsilon \notin L(G)$. This property of $L(G)$ is deliberately enforced by introduction of X in G and by having $S \to aXa$ and $S \to bSb$ as only rules that rewrites start symbol in G. Consider $H = (\{S, a, b\}, \{a, b\}, \{S \to aSa, S \to bSb, S \to \varepsilon\}, S)$, and $L(H) = L(G) \cup \varepsilon$. The ε-rule $S \to \varepsilon$ becomes non erasable in H. Therefore, not all ε-rules can be erased from CFGs. An exact algorithm for ε-rules removal can be seen in [4].

Definition 2.37. Let $G = (V, T, P, S)$ be a context-free grammar. A production, $p \in P$, is a **unit production** if $rhs(p) \in N$. If G contains no unit productions, it is **unit-free** context-free grammar.

Intuitively, we understand that unit rules are useless in CFGs. They only transform one nonterminal into another. All of the unit productions can be removed. Let $G = (V, T, P, S)$. Assume that G contains unit rules. For every $A, B \in V - T$, such that $A \to B \in P$ is a unit rule, add $\{A \to \alpha \,|\, B \Rightarrow \alpha\}$ to P. Repeat this step until no unit rule is left in G. An exact algorithm for unit rules removal can be seen in [4].

Both ε-rules and unit rules introduce an unwanted property to CFGs—*cycling*.

Definition 2.38. Let $G = (V, T, P, S)$ be a context-free grammar. G contains **cycle**, if $A \Rightarrow^+ A$ in G.

If a CFG contains cycle, it is called **cycling** CFG. Consider a CFG $G = (\{A, a\}, \{a\}, \{1 : A \to a, 2 : A \to A, 3 : A \to AA, 4 : A \to \varepsilon\}, A)$. G generates language $L(G) = \{a^i \,|\, i \in \mathbb{N}_0\}$ and contains unit and ε-rule. G cycles in two ways.

$$A \Rightarrow AA[3] \Rightarrow A[4]$$
$$A \Rightarrow A[2]$$

2.5.1 Derivation trees

For any grammar, the most important property is the language that it generates. However, not only the words that comprise the generated language are important. We are also interested in the way that these words are generated. In the case of the grammars, we are interested in the sequence of the derivation steps from the start symbol to the final word. A representation for this derivation is needed. Remember, that for some derivation $S \Rightarrow^+ \alpha[\gamma], \gamma$ represents labels of rules, in the order in which they were applied. But it doesn't capture which nonterminal was rewritten. In CFGs, the order of the applications of rules is also irrelevant. Therefore, this mechanism cannot be used to represent derivations.

A *derivation tree* is a representation of the derivation in the CFGs that can be easily presented graphically. It captures individual derivation steps but is invariant to the order of these steps. The derivation tree is a tree, such that the root node is the start symbol, all the nodes are nonterminals, and the leaf nodes are terminals. Informally, each node can have multiple child nodes. A root node is a node that is not a child node of any other node. A leaf has no child nodes. The child nodes of a parent node represent a result of the direct derivation of the parent node. An example of the derivation tree can be seen in Figure 2.1. A thorough definition of the derivation tree can be found in [4].

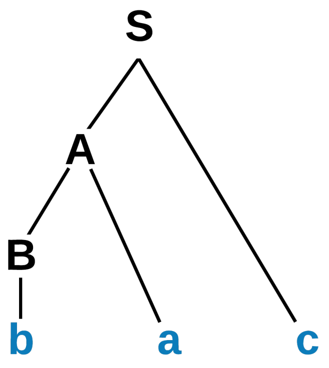


Figure 2.1: Derivation tree for a derivation of bac in a context-free grammar $G = (\{S, A, B\}, \{a, b, c\}, \{S \to Ac, A \to Ba, B \to b\}, S)$.

Now that the derivation trees are defined, we can define another property of CFGs—ambiguity. Remember that a language is a set of words. Sets can only contain unique values. However, a CFG can generate a single word in multiple different ways. Different order of application of the applied rules is not considered to be a different way of generation of the same word. Consider a CFG $G = (\{S, A, a\}, \{a\}, \{S \to A, S \to aS, S \to a, A \to aA, A \to a\}, S)$. G generates a language $L(G) = \{a^n \,|\, n \geq 1\}$. Notice that for every $w \in L(G)$, there are two ways it can be generated. Let $w = aa$. G derives w as $S \Rightarrow aS \Rightarrow aa$ or as $S \Rightarrow A \Rightarrow aA \Rightarrow aa$. It is impossible to say, which derivation was used to generate w. We can distinguish between these two derivations by the usage of the derivation trees, which are different for each of these derivations. This property is called *ambiguity*.

Definition 2.39. Let $G = (V, T, P, S)$ be a context-free grammar. If there exists word $w \in L(G)$, such that we can construct more than one derivation tree, G is **ambiguous**. Otherwise, G is **unambiguous**.

The ambiguity of CFGs is unwanted. However, there is no general way to get rid of ambiguity from the CFGs. Even the problem to decide whether a given CFG is ambiguous or not is undecidable. Meaning that there is no algorithm that is able to decide, whether the given grammar is ambiguous or not. Therefore we cannot remove the ambiguity from CFGs, since in general, we cannot even detect it.

Ambiguity may impose a problem during a syntax analysis (also-called parsing). Syntax analysis decides whether a given word is part of a language generated by some grammar. In general, this problem is called a *membership problem*. During syntax analysis of a CFG, we are effectively trying to find a derivation tree for the given word. In ambiguous CFGs, we are trying to find one of the possible derivation trees. It does not matter which one we find.

2.5.2 Chomsky normal form

In the grammars, the forms of the productions can be restricted, while preserving their expressive power. Such restrictions may allow us to work with said grammars in a more suitable way. These restrictions are often called *normal forms*. A grammar that does not satisfy any normal form is usually smaller than an equivalent grammar in some normal form. Meaning that in order to satisfy the normal form, the number of the rules usually increases.

In the case of CFGs, one of the studied normal forms is *Chomsky normal form*. Before we define Chomsky normal form, let us define a *proper* CFG. A proper form of CFGs represents much weaker restriction than normal forms.

Definition 2.40. Let $G = (V, T, P, S)$ be a context-free grammar and $A \in V - T$ satisfying the following properties:

- V contains only useful symbols (see Definition 2.35);
- G is ε-free (see Definition 2.36);
- G is unit-free (see Definition 2.37).

Then, G is a **proper context-free grammar**.

In the previous part of this chapter, we described why the useless symbols, ε-rules, and unit rules as undesirable in CFGs. Algorithms for their removal can be found in [4]. Remember that the rule that rewrites the start symbol to ε cannot be removed. Therefore, for every CFG G, there exists a proper CFG P such that $L(P) = L(G) - \{\varepsilon\}$.

Definition 2.41. Let $G = (V, T, P, S)$ be a context-free grammar. G is in **Chomsky normal form** if every rule, $p \in P$, satisfies $rhs(p) \in (T \cup (V - T)^2)$.

A CFG is in Chomsky normal form if the right-side of all the rules are either two nonterminals or a single terminal. Any CFG G can be transformed into a CFG $\overline{G}$ in Chomsky normal form, such that $L(G) = L(\overline{G})$. The algorithm for this transformation works in the following way. It takes a proper CFG $G = (V, T, P, S)$ as an input. Remember that a proper CFG does not contain rules, that have either a single nonterminal or epsilon on the right-side. Formally, for all $p \in P, rhs(p) \notin (V - T) \cup \{\varepsilon\}$ holds.

Begin the transformation of G to a CFG in Chomsky normal form $\overline{G} = (\overline{V}, T, \overline{P}, S)$. Set $W = \{\overline{a} | a \in T\}, \overline{V} = V \cup W$. CNF allows occurrence of the terminals in the right-sides of rules only as single terminal. In case of other occurrences of terminals, these terminals must be replaced by nonterminal, that can only be rewritten to said terminal. Define bijection β that maps every nonterminal to itself, and every terminal $a \in T$ to a nonterminal from $W, \overline{a}$. Formally, β is defined as bijection from V to $W \cup (V - T)$ as $\beta(a) = \overline{a}$ for all $a \in T$ and $\beta(A) = A$ for all $A \in V - T$. Bijection β does the replacement of terminals to their dedicated nonterminals. For each nonterminal $a \in T$, add $\overline{a} \to a$ to $\overline{P}$. Since the G is proper, right-side of each rule must be of length 1, 2 or more than 2. If the length of right-side of rule is 1, it must be single terminal, since proper CFG cannot contain single nonterminal as right-side. If the length of right-side of rule is 2, it can be easily added to $\overline{P}$. For every $p : A \to X_1X_2 \in P, A \in V - T; X_1, X_2 \in V$, add $A \to \beta(X_1)\beta(X_2)$ to $\overline{P}$. Any of symbols X_1 and X_2 can be a terminal. In such case, p would not satisfy CNF. Therefore, bijection β is used.

The last case of length or right-side of the rule is more than 2. Such sequences must be divided into multiple parts. For $A \to X_1X_2...X_n \in P; A \in V - T, X_i \in V, 1 \leq i \leq n, n \geq 3$, introduce new nonterminals $\langle X_2...X_n\rangle, \langle X_3...X_n\rangle, ...\langle X_{n-1}X_n\rangle$ and add them to $\overline{V}$. Add rules $A \to \beta(X_1)\langle X_2...X_n\rangle, \langle X_2...X_n\rangle \to \beta(X_2)\langle X_3...X_n\rangle, ..., \langle X_{n-1}X_n\rangle \to \beta(X_{n-1})\beta(X_n)$ to $\overline{P}$. Notice that these rules satisfy CNF. In this way, $A \Rightarrow X_1X_2...X_n[A \to X_1X_2...X_n]$ is simulated in $\overline{G}$.

An algorithm for this transformation is written as pseudocode in Algorithm 2.1.

Algorithm 2.1 An algorithm for transformation of a proper CFG to a CFG in Chomsky normal form.

Input
    A proper CFG $G = (V, T, P, S)$.

Output
    A CFG $\overline{G} = (\overline{V}, T, \overline{P}, S)$ in Chomsky normal form, such that $L(G) = L(\overline{G})$.

Method

```
introduce $W = \{\overline{a} | a \in T\}$ and bijection $\beta$ from $V$ to $W \cup V - T$ defined as
    $\beta(a) = \overline{a}$ for all $a \in T$ and $\beta(A) = A$ for all $A \in V - T$
set $\overline{V} = W \cup V$ and $\overline{P} = \emptyset$
for all $a \in T$ do
    add $\overline{a} \rightarrow a$ to $\overline{P}$
for all $A \rightarrow a \in P, A \in V - T, a \in T$ do
    add $A \rightarrow a \in P$ to $\overline{P}$
for all $A \rightarrow X_1X_2 \in P, A \in V - T, X_1, X_2 \in V$ do
    add $A \rightarrow \beta(X_1)\beta(X_2)P$ to $\overline{P}$
for all $A \rightarrow X_1X_2...X_n \in P, A \in V - T, X_i \in V^+, 1 \geq i \geq n$ such that $n \geq 3$ do
    add new nonterminals $\langle X_2...X_n \rangle, \langle X_3...X_n \rangle, ...\langle X_{n-1}X_n \rangle$ to $\overline{V}$
    add $A \rightarrow \beta(X_1)\langle X_2...X_n \rangle, \langle X_2...X_n \rangle \rightarrow \beta(X_2)\langle X_3...X_n \rangle, ...,$
    $\langle X_{n-1}X_n \rangle \rightarrow \beta(X_{n-1})\beta(X_n)$ to $\overline{P}$
```

2.6 Language families

Each formal model has limits to the languages it can describe. For example, there is no context-free grammar that can generate language $\{a^nb^nc^n | n \geq 1\}$. The proof is omitted but the reader can try to construct such CFG. Refer back to Definitions 2.19 and 2.24 of unrestricted and context-free grammars, respectively. Intuitively, unrestricted grammars provide more powerful rules than CFGs. Therefore, unrestricted grammars have the ability to generate more complex languages than CFGs.

A *language family* represents a set of all the languages that can be described by some formal model. The relation between the language families is the same as the relation between the expressive powers of their formal models.

Hierarchy of language families

The language families of some formal models defined in this thesis can be arranged into a hierarchy by their expressive power. Now, let us define these language families.

Definition 2.42. $\mathbf{REG}, \mathbf{PAL}, \mathbf{ML}, \mathbf{LIN}, \mathbf{CF}, \mathbf{CS}$ and $\mathbf{RE}$ denote the language families that are generated by the regular, palindromial, minimal linear, linear, context-free, and unrestricted grammars, respectively.

These families are so-called families of regular, palindromial, minimal linear, linear, context-free, and recursively enumerable languages, respectively. Languages from **RE** are also described by the turing machines. **RE** represent all of the problems, that we can compute.

Claim 2.1. Following properties hold

(a) $\mathbf{REG} \subset \mathbf{LIN} \subset \mathbf{CF} \subset \mathbf{CS} \subset \mathbf{RE}$

(b) $\mathbf{REG} \cup \mathbf{PAL} \subset \mathbf{LIN}$

(c) $\mathbf{PAL} \subset \mathbf{ML}$

Proof. Proof of these properties can be found in [4] and [9]. $\square$

The strict inclusions of the language families from (a) is also called Chomsky hierarchy and can be seen on Figure 2.2.

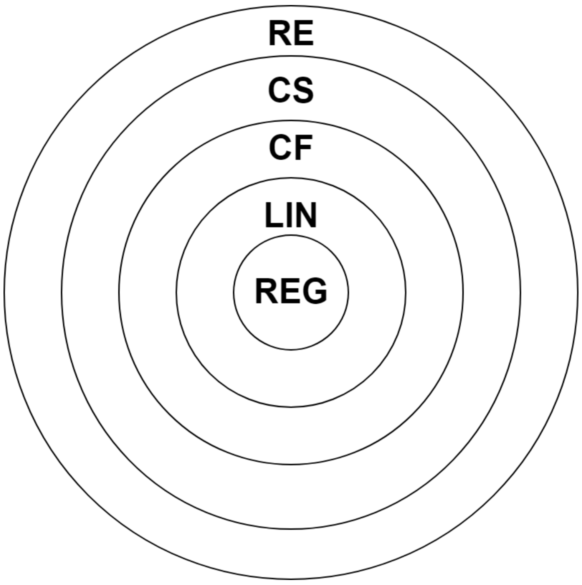


Figure 2.2: Chomsky hierarchy.

Claim 2.2. $\mathbf{REG} \not\subseteq \mathbf{ML}$ and $\mathbf{ML} \not\subseteq \mathbf{REG}$.

Proof. To prove $\mathbf{ML} \not\subseteq \mathbf{REG}$, consider a minimal linear grammar

$$P = (\{S, 0, 1, \#\}, \{0, 1, \#\}, \{S \to 0S1, S \to \#\}).$$

It is apparent that $L(P) = \{0^n\#1^n \mid n \geq 0\}$. There exists no regular grammar that could generate such language. Since $L(P) \in \mathbf{ML}$ and $L(P) \notin \mathbf{REG}$, $\mathbf{ML} \not\subseteq \mathbf{REG}$ holds true.

To prove $\mathbf{REG} \not\subseteq \mathbf{ML}$, consider a regular grammar $R = (\{S, B, C, a, b, c\}, \{a, b, c\}, \{S \to aS, S \to aB, B \to bB, B \to bC, C \to cC, C \to c\}, S)$. A language generated by R is $L(R) = \{a^x b^y c^z \mid x, y, z \geq 1\}$. Intuitively, since the minimal linear grammars contain only single nonterminal, they have no way to ensure transition between a, b, c in the words of the resulting language. Therefore, $L(P) \in \mathbf{REG}$ and $L(P) \notin \mathbf{ML}$, so $\mathbf{REG} \not\subseteq \mathbf{ML}$. $\square$

When there can be no subset relation between the two sets made, we call them mutually incomparable. Since the languages are the sets, we say that **REG** and **ML** are mutually incomparable.

Claim 2.3. $\mathbf{REG} \cap \mathbf{PAL} = \emptyset$.

Proof. Recall Definition 2.28 of the palindromial grammars. Let $P = (V, T, P, S)$ be any palindromial grammar. The language of P must be $\{\alpha\#\gamma \mid \alpha, \gamma \in T^* \text{ and } \alpha = \gamma^R\}$. The regular grammars cannot generate such languages. Therefore, this Claim holds. $\square$

Chapter 3

Final Sentential Forms

Context-free grammars represent one of the most used model in the formal language theory since they are simple and allow us to define useful languages, such as programming languages. However, for many applications, CFGs are not sufficient. Over the years, many efforts were made to restrict the way CFGs generate words to improve their generative power.

In this chapter, we introduce the notion of a *final sentential forms* of CFGs. These forms represent the subset of sentential forms of CFGs, where parts of individual sentential forms belong to specified *finalizing language L*. Since only parts of individual sentential forms are controlled, a less powerful language family can be used.

The main topic of this thesis are the final sentential forms, where sentential forms of CFGs are finalized by either minimal linear languages or regular languages. First, we start with the definition of a weak identity which extracts parts of the sentential forms.

Definition 3.1. Let $G = (V, T, P, S)$ be a context-free grammar, W be a set and ω be a homomorphism from V^* to W^*; ω is a **weak identity** if $\omega(a) \in \{a, \varepsilon\}$ for all $a \in V$. Define the weak identity ${}_W\omega$ from V^* to W^* as ${}_W\omega(X) = X$ for all $X \in W$, and ${}_W\omega(X) = \varepsilon$ for all $X \in V - W$.

The purpose of the weak identity is to erase selected symbols and keep the rest. In our case, we define a set of symbols W, which we want to preserve in the word. Symbols that do not belong to W are erased.

Definition 3.2. Let $G = (V, T, P, S)$ be a context-free grammar and $W \subseteq V$. Let $F \subseteq W^*$. Set

$$\phi(G, F) = \{x \mid x \in \phi(G),\ {}_W\omega(x) \in F\}$$

F is called a **final language** or **finalizing language** Members of $\phi(G, F)$ are called the **final sentential forms of G**.

The set $\phi(G, F)$ represents all the sentential forms of G, that belong to the final language F after the application of weak identity ${}_W\omega$. This mechanism allows us to impose further constraints on the underlying CFG. In turn, we may increase the generative power of said CFG. The resulting power depends on the selection of final language F. Notice that the final sentential forms may contain nonterminals. However, these nonterminals cannot be present in the resulting language.

Definition 3.3. $G = (V, T, P, S)$ be a CFG and $W \subseteq V$. Let $F \subseteq W^*$. Set

$$L(G, F) = \{\, {}_T\omega(y) \mid y \in \phi(G, F),\ {}_{(V-(W\cup T))}\omega(y) = \varepsilon\}.$$

The language $L(G, F)$ is called **language of G finalized by F**.

By the definition, we remove all nonterminals from final sentential forms $\phi(G, F)$ to obtain $L(G, F)$. Notice that by the definition of $L(G, F)$, final sentential forms that cannot contain nonterminals from $V - (W \cup T)$. We could omit this condition without changing any of the results proven in this chapter. However, by application of this restriction, we are able to present a stronger result.

In the first chapter, we defined queue grammars (see Definition 2.29). These grammars work in a different way compared to unrestricted grammars. In a way, queue grammars combine the derivation process of grammars with the state transitions of automata. We use the queue grammars in our proofs presented below. The reason we use the queue grammars is that they are Turing-equivalent. In other words, the language family generated by queue grammars is recursively enumerable (**RE** for short).

Lemma 3.1. *Let $L \in$ **RE**. Then, there exists a left-extended queue grammar Q satisfying $L(Q) = L$.*

Proof. See *Lemma 1* in [2]. □

To be more precise, we don't use queue grammars directly, but we use left-extended queue grammars (see Definition 2.32). The left-extended queue grammars work in the same way as queue grammars, but they also record the derivation history on the left side of the derived word. The derivation history is separated from the sentential form by #. From the definition of the left-extended queue grammar, it is apparent that they are equivalent to the ordinary queue grammars.

We need to constrain the way left-extended queue grammars derive the words. Similarly to the normal forms of CFGs. Since queue grammars always rewrite the leftmost symbol, we constrain the left-extended queue grammar Q in such a way, that derivation can be split into these two phases:

1. rewrite the nonterminals only to the sequences of nonterminals;

2. rewrite the nonterminals only to the sequences of terminals.

At the beginning of the derivation, Q uses rules that rewrite nonterminal to the sequence of nonterminals. Once the rule that rewrites nonterminal to the sequence of terminals is used, Q can only use rules that rewrite nonterminal to the sequence of terminals. The sequences mentioned can also be empty, meaning that the nonterminal is rewritten to ε.

The proof that we can achieve such constraint of the left-extended queue grammar without changing its generative power can be seen in [3].

Lemma 3.2. *Let H be a left-extended queue grammar. Then, there exists a left-extended queue grammar, $Q = (V, T, U, D, s, R)$, such that $L(H) = L(Q)$ and every $(a, b, z, c) \in R$ satisfies $a \in V - T, b \in U - D, z \in ((V - T)^* \cup T^*)$ and $c \in U$.*

Proof. See Lemma 2 in [3]. □

3.1 Palindromial finalizing language

In Definition 3.2 of the final sentential forms, no particular finalizing language F is specified. Based on our selection of F, we study the resulting language $L(G, F)$. In this section, we show, that using a palindromial (see Definition 2.28) final language $F = \{w\#w^R \mid w \in \{0,1\}^*\}$, any language $L \in \mathbf{RE}$ can be represented by $L(G, F)$. Notice, that the language $F = \{w\#w^R \mid w \in \{0,1\}^*\}$ is indeed palindromial. It can be generated by a palindromial grammar $G = (\{S, 0, 1, \#\}, \{0, 1, \#\}, \{S \to 0S0, S \to 1S1, S \to \#\}, S)$.

Lemma 3.3. Let $Q = (V, T, U, D, s, R)$ be a left-extended queue grammar. Then, $L(Q) = L(G, \{w\#w^R \mid w \in \{0,1\}^*\})$, where G is a CFG.

Idea. In proof of this lemma, we simulate the derivations Q by some CFG G. Recall that left-extended queue grammars have greater generative power than CFGs. Therefore, G cannot simulate Q itself. Hence we use the final language F that is able to recognize proper derivations of Q simulated by G.

Proof. Without any loss of generality, assume that Q satisfies the properties described in Lemma 3.2 and that $\{0,1\} \cap (V \cup U) = \emptyset$. For some positive integer, n, define an injection, ι, from Ψ^* to $(\{0,1\}^n - 1^n)$, where $\Psi = \{ab \mid (a, b, x, c) \in R, a \in V - T, b \in U - D, x \in (V - T)^* \cup T^*, c \in U\}$ so that ι is an injective homomorphism when its domain is extended to Ψ^*; after this extension, ι thus represents an injective homomorphism from Ψ^* to $(\{0,1\}^n - 1^n)^*$(a proof that such an injection necessarily exists is simple and left to the reader). Based on ι, define the substitution, ν from V to $(\{0,1\}^n - 1^n)$ as $\nu(a) = \{\iota(aq) \mid q \in U\}$ for every $a \in V$. Extend domain of ν to V^*. Furthermore, define the substitution, μ, from U to $(\{0,1\}^n - 1^n)$ as $\mu(q) = \{\iota(aq)^R \mid a \in V\}$ for every $q \in U$. Extend the domain of μ to U^*.

The set Ψ represents pairs of the symbol and state, that can be rewritten in the derivation step. Injective homomorphism ι encodes these pairs into binary form. The substitution ν creates a set for some nonterminal A, that contains all of the possible pairs for A from Ψ encoded by ι. The substitution μ creates a set for some state B, that contains all of the possible pairs for B from Ψ encoded by ι that are reversed.

Consider $ab \in \Psi$, where $a \in V - T$ and $U - D$. From the definitions of ν and μ, it is apparent that $\nu(a) \cap \mu(b)^R = \{\iota(ab)\}$.

Construction. Next, we introduce a context-free grammar G so that

$$L(Q) = L(G, \{w\#w^R \mid w \in \{0,1\}^*\}).$$

Set $J = \{\langle p, i\rangle \mid p \in U - D \text{ and } i \in \{1, 2\}\}$. Let $G = (\overline{V}, T, P, S)$, where $\overline{V} = J \cup \{0, 1, \#\} \cup T$. Construct P in the following way. Initially, set $P = \emptyset$; then, perform the following steps 1 through 5.

1. if $(a, q, y, p) \in R$, where $a \in V - T, p, q \in U - D, y \in (V - T)^*$ and $aq = s$, then add $S \to u\langle p, 1\rangle v$ to P, for all $u \in \nu(y)$ and $v \in \mu(p)$;

2. if $(a, q, y, p) \in R$, where $a \in V - T, p, q \in U - D$ and $y \in (V - T)^*$, then add $\langle q, 1\rangle \to u\langle p, 1\rangle v$ to P, for all $u \in \nu(y)$ and $v \in \mu(p)$;

3. for every $q \in U - D$, add $\langle q, 1\rangle \to \langle q, 2\rangle$ to P;

4. if $(a, q, y, p) \in R$, where $a \in V - T, p, q \in U - D, y \in T^*$, then add $\langle q, 2\rangle \to y\langle p, 2\rangle v$ to P, for all $v \in \mu(p)$;

5. if $(a, q, y, p) \in R$, where $a \in V - T, q \in U - D, y \in T^*$, and $p \in D$, then add $\langle q, 2\rangle \to y\#$ to P.

Each derivation of Q goes through these phases in the following order

(a) Q uses rules $(a, b, z, c) \in R$ such that $a \in V - T, b, c \in U - D$, and $z \in (V - T)^*$;

(b) Q uses rules $(a, b, z, c) \in R$ such that $a \in V - T, b, c \in U - D$, and $z \in T^*$;

(c) Q uses final rule $(a, b, z, c) \in R$ such that $a \in V - T, b \in U - D, c \in D$, and $z \in T^*$.

By Lemma 3.2, this order must be fulfilled. Examine the constructed rules. Rules constructed in (1) simulate the start of the derivation by Q. Rules from (1), together with rules from (2) simulate part (a) of the derivation of Q. Rules from (3) simulate the transition from (a) to (b). Part of the derivation described by (b) is simulated by rules from (4). And finally, rules from (5) are used to finish the derivation in Q, denoted by (c).

In rules constructed in (1), (2), and (4) we can see the use of substitutions ν and μ. Context-free grammars have no mechanism to ensure that simulation of derivation in Q would not perform invalid steps. Meaning that the simulation would not generate $L(Q)$. Therefore, mentioned substitutions, that are later controlled by F are added.

The purpose of the substitution $\nu(y)$ for some $y \in (V - T)^*$ is to record nonterminals that must be rewritten in the future on the left side of the derived word. This mechanism effectively acts as a queue. For example let $A, B, C \in (V - T)$, then $\nu(ABC) = \iota(AD)\iota(BE)\iota(CF)$ for some $D, E, F \in U$. Remember, that from the definition of ν, that $|\nu(ABC)|$ can be greater than 1. Conversely, the substitution μ represents the nonterminal and the current state of Q that are being rewritten. At the time of application of the rules containing substitution μ, there is no way for G to know the leftmost symbol that should be rewritten. At the end of the derivation, the final language F verifies, that G selected correct rules.

Notice that in the definition of μ, a reversal of ι is used. In order for G to make proper derivation according to Q, each of the nonterminals encoded by the substitution ν must be fulfilled by an actual derivation by μ. This property is fulfilled only by members of $\{xy\#z \in \phi(G) \,|\, x \in W^+, y \in T^*, z = x^R\}$.

Set $W = \{0, 1, \#\}$ and $\Omega = \{xy\#z \in \phi(G) \,|\, x \in \{0, 1\}^+, y \in T^*, z = x^R\}$.

Claim 3.1. Every $h \in \Omega$ is generated by G in this way

$$\begin{aligned}
& S \\
\Rightarrow\ & g_1\langle q_1, 1\rangle t_1 \Rightarrow g_2\langle q_2, 1\rangle t_2 \Rightarrow \dots \Rightarrow g_k\langle q_k, 1\rangle t_k \Rightarrow g_k\langle q_k, 2\rangle t_k \\
\Rightarrow\ & g_k y_1\langle q_{k+1}, 2\rangle t_{k+1} \Rightarrow g_k y_1 y_2\langle q_{k+2}, 2\rangle t_{k+2} \Rightarrow \dots \Rightarrow g_k y_1 y_2 \dots y_{m-1}\langle q_{k+m-1}, 2\rangle t_{k+m-1} \\
\Rightarrow\ & g_k y_1 y_2 \dots y_{m-1} y_m \# t_{k+m}
\end{aligned}$$

in G, where $k, m \geq 1; q_1, \dots, q_{k+m-1} \in U - D; y_1, \dots, y_m \in T^*; t_i \in \mu(q_i \dots q_1)$ for $i = 1, \dots, k+m$; $g_j \in \nu(d_1 \dots d_j)$ with $d_1, \dots, d_j \in (V - T)^*$ for $j = 1, \dots, k$; $d_1 \dots d_k = a_1 \dots a_{k+m}$ with $a_1, \dots, a_{k+m} \in V - T$ (that is, $g_k \in \nu(a_1 \dots a_{k+m})$ with $g_k = (t_{k+m})^R$); $h = y_1 y_2 \dots y_{m-1} y_m$.

Proof. Examine the construction of P. Observe that every derivation begins with an application of a production having S on its left-hand side. Set 1-$J = \{\langle p, 1\rangle \,|\, p \in U\}$, 2-$J = \{\langle p, 2\rangle \,|\, p \in U\}$, 1-$P = \{p \,|\, p \in P$ and $lhs(p) \in$ 1-$J\}$, 2-$P = \{p \,|\, p \in P$ and $lhs(p) \in$ 2-$J\}$. Observe that in every successful derivation of h, all applications of productions from 1-P

precede the applications of productions from 2-P. Thus, the generation of h can be expressed as

$$\begin{array}{ll} & S \\ \Rightarrow & g_1\langle q_1, 1\rangle t_1 \Rightarrow g_2\langle q_2, 1\rangle t_2 \Rightarrow ... \Rightarrow g_k\langle q_k, 1\rangle t_k \Rightarrow g_k\langle q_k, 2\rangle t_k \\ \Rightarrow & g_k y_1\langle q_{k+1}, 2\rangle t_{k+1} \Rightarrow g_k y_1 y_2\langle q_{k+2}, 2\rangle t_{k+2} \Rightarrow ... \Rightarrow g_k y_1 y_2 ... y_{m-1}\langle q_{k+m-1}, 2\rangle t_{k+m-1} \\ \Rightarrow & g_k y_1 y_2 ... y_{m-1} y_m \# t_{k+m} \end{array}$$

where all the involved symbols have the meaning stated in Claim 3.1.

In the first $k, k \geq 1$ steps of the derivation, only rules in form of $(a, b, z, c) \in R$, where $a \in V - T, b, c \in U - D$, and $z \in V^*$ are simulated by G. The first derivation step is made according to rules from (1), and the following $k-1$ steps are done according to rules constructed in (2). Remember that in rules from (1) and (2), the $y \in (V-T)^*$. Therefore, in the derivation above, the $|g_k|$ can be larger than $|t_k|$. Then, G needs to make a single derivation step according to rules from (3). In next $m-1, m \geq 1$ derivation steps, only rules in form of $(a, b, z, c) \in R$, where $a \in V - T, b, c \in U - D$, and $z \in T^*$ are simulated by G according to rules from (4). The derivation is then finished by the rule from (5). □

Claim 3.2. Every $h \in L(Q)$ is generated by Q in this way

$$\begin{array}{lll} & \# a_0 q_0 & \\ \Rightarrow & a_0 \# x_0 q_1 & [(a_0, q_0, z_0, q_1)] \\ \Rightarrow & a_0 a_1 \# x_1 q_2 & [(a_1, q_1, z_1, q_2)] \\ ... & & \\ \Rightarrow & a_0 a_1 ... a_k \# x_k q_{k+1} & [(a_k, q_k, z_k, q_{k+1})] \\ \Rightarrow & a_0 a_1 ... a_k a_{k+1} \# x_{k+1} q_{k+2} & [(a_{k+1}, q_{k+1}, y_1, q_{k+2})] \\ ... & & \\ \Rightarrow & a_0 a_1 ... a_k a_{k+1} ... a_{k+m-1} \# x_{k+m-1} y_1 ... y_{m-1} q_{k+m} & [(a_{k+m-1}, q_{k+m-1}, y_{m-1}, q_{k+m})] \\ \Rightarrow & a_0 a_1 ... a_k a_{k+1} ... a_{k+m} \# y_1 ... y_m q_{k+m+1} & [(a_{k+m}, q_{k+m}, y_m, q_{k+m+1})] \end{array}$$

where $k, m \geq 1$, $a_i \in V - T$ for $i = 0, ..., k+m$, $x_j \in (V-T)^*$ for $j = 1, ..., k+m$, $s = a_0 q_0$, $a_j x_j = x_{j-1} z_j$ for $j = 1, ..., k$, $a_1 ... a_k x_{k+1} = z_0 ... z_k$, $a_{k+1} ... a_{k+m} = x_k$, $q_0, q_1, ..., q_{k+m} \in U - D$ and $q_{k+m+1} \in D$, $z_1, ..., z_k \in (V-T)^*$, $y_1, ... y_m \in T^*$, $h = y_1 y_2 ... y_{m-1} y_m$.

Proof. Recall that Q satisfies the properties given in Lemma 3.2. These properties imply that Claim 3.2 holds true. □

Claim 3.3. $L(G, \{w\#w^R \mid w \in (W - \{\#\}^*\}) = L(Q)$.

Proof. To prove that $L(G, F) \subseteq L(Q)$, take any $h \in \Omega$ generated in the way described in Claim 3.1. From ${}_W\omega(h) \in \{w\#w^R \mid w \in \{0,1\}^*\}$, it follows that $xy\#z$ with $z = x^R$ where $x = g_k, y = y_1 ... y_m, z = t_{k+m}$. At this point R contains (a_0, q_0, z_0, q_1), ...,(a_k, q_k, z_k, q_{k+1}), $(a_{k+1}, q_{k+1}, y_1, q_{k+2})$, ..., $(a_{k+m-1}, q_{k+m-1}, y_{m-1}, q_{k+m})$, $(a_{k+m}, q_{k+m}, y_m, q_{k+m+1})$, where z_1, ..., $z_k \in (V-T)^*$, and y_1, ..., $y_m \in T^*$. Then, Q makes the generation of ${}_T\omega(h)$ in the way described in Claim 3.2. Thus ${}_T\omega(h) \in L(Q)$.

To prove $L(Q) \subseteq L(G, \{w\#w^R \mid w \in \{0,1\}^*\})$, take any $h \in L(Q)$. Recall that h is generated in the way described in Claim 3.2. Consider the rules used in this generation. Furthermore, consider the definition of ν and μ. Based on this consideration, observe that from the construction of P, it follows that $S \Rightarrow^* oh\#\bar{o}$ in G for some $o, \bar{o} \in \{0,1\}^+$ with $\bar{o} = o^R$. Thus, ${}_W\omega(oh\#\bar{o}) \in \{w\#w^R \mid w \in \{0,1\}^*\}$, so consequently, $h \in L(G, \{w\#w^R \mid w \in \{0,1\}^*\})$.

□

Claims 3.1 through 3.3 imply that Lemma 3.3 holds true.

Theorem 3.1. A language $L \in \mathbf{RE}$ if and only if $L = L(G, \{w\#w^R \mid w \in \{0,1\}^*\})$, where G is a propagating CFG.

Proof. This theorem follows from Lemmas 3.1 through 3.3. □

Corollary 3.1. $\mathbf{RE} = \mathbf{CF_{PAL}}$.

3.2 Regular finalizing language

In the previous chapter, we chose the finalizing language F as a palindromial language leads to the recursively enumerable language family. Furthermore, only a single palindromial language is needed for this result. A natural question arises. What happens when we use a language from different language family as the finalizing language? In this section, we study the case of a regular finalizing language. Recall, that palindromial and regular languages do not contain any common languages. Meaning that $\mathbf{PAL} \cap \mathbf{REG} = \emptyset$ (see Claim 2.3). We proved, that by using regular finalizing language, $L(G, F)$ remains context-free language. Meaning that such finalizing language is useless.

The regular languages can be represented by either regular grammars, or by finite automata. In our proofs, we use finite automata for representation of the regular finalizing language.

Lemma 3.4. Let $G = (V, T, P, S)$ be any CFG and $F \in \mathbf{REG}$. Then, $L(G, F) \in \mathbf{CF}$.

The proof of this lemma is done by simulation of $L(G, F)$ by newly constructed CFG H. H simulates both the derivation in G and the run of a deterministic finite automaton M, such that $F = L(M)$. Remember that the finite automata accept the family of regular languages $\mathbf{REG}$. Furthermore, for every finite automaton, there exists an equivalent deterministic finite automaton. It is then shown, that $L(H) = L(G, F)$.

Proof. Let $G = (V, T, P, S)$ be any CFG and $F \in \mathbf{REG}$. Let $F = L(M)$, where $M = (Q, W, R, q_s, Q_F)$ is a deterministic finite automaton.

Construction. Introduce $U = \{\langle paq \rangle \mid p, q \in Q, a \in V\} \cup \{\langle q_s S Q_F \rangle\}$. Members of U have following meaning. Let $\langle paq \rangle \in U, p, q \in Q$, and $a \in V$. The $\langle paq \rangle$ represents a sequence of moves in M, where p denotes a beginning state, and q denotes an ending state of the sequence. U contains special member $\langle q_s S Q_F \rangle$. Since the finite automaton can have multiple final states, $\langle q_s S Q_F \rangle$ represents any successful run of the finite automaton from start state to any final state.

From G and M, construct a new CFG H such that $L(H) = L(G, F)$ in the following way. Set

$$H = (\overline{V}, T, \overline{P}, \langle q_s S Q_F \rangle)$$

The components of H are constructed as follows. Set $\overline{V} = V \cup U$. Construct $\overline{P}$ as follows:

(0) Add $\langle q_s S Q_F \rangle \to \langle q_s S q_f \rangle$ for all $q_f \in Q_F$.

(1) Let $A \to y_0 X_1 y_1 X_2 ... X_n y_n \in P$, where $A \in V - T, y_i \in (V - W)^*$ and $X_j \in V, 0 \leq i \leq n, 1 \leq j \leq n$, for some $n \geq 1$;
then, add $\langle q_1 A q_{n+1} \rangle \to y_0 \langle q_1 X_1 q_2 \rangle y_1 \langle q_2 X_2 q_3 \rangle ... \langle q_n X_n q_{n+1} \rangle y_n$ to $\overline{P}$, for all $q_1, q_2, ..., q_{n+1} \in Q$.

(2) Let $A \to \alpha \in P$, where $A \in V - (T \cup W), \alpha \in (V - W)^*$;
then, add $A \to \alpha$ to $\overline{P}$.

(3) Let $\langle paq \rangle \in U$, where $a \in W \cap T, pa \to q \in R$;
then, add $\langle paq \rangle \to a$ to $\overline{P}$.

(4) Let $\langle pBq \rangle \in U$, where $pB \to q \in R, B \in W \cap (V - T)$;
then, add $\langle pBq \rangle \to \varepsilon$ to $\overline{P}$.

Rules from (0) are used to select a concrete run in M. Rules from (1) are used to divide a sequence of moves $\langle q_1 A q_{n+1} \rangle \in U$ of M. Notice that for every two consequential sequences on the right-side of these rules, the first sequence ends with the same state as the second one begins with. Furthermore, notice that the beginning state of the sequence on the left-side of the rules is the same as the beginning state of the first subsequence on the right-side. The ending state of the sequence on the left-side is the same as the ending state of the last subsequence on the right-side. This means, that the original sequence as a whole remains the same. The $y_i \in (V - W)^*$ for $0 \leq i \leq n$ must not contain the symbols from W. All the symbols from W must be encapsulated by some symbol from U. Otherwise, we could not simulate a run of M over all of the symbols from W in the derived word. Notice that the symbol from U does not have to contain the member of W, since there could exist such nonterminal $X \in (V - W)$, that $X \Rightarrow^+ Y$, where $Y \in W$. The rules from (2) are used to simulate the derivation of G for the parts that are not covered by M. These rules cannot contain the symbols from W on either side of the rule.

When the simulation of M inside of H gets to the point that $\langle q_1 A q_2 \rangle \in U$ corresponds to a single step of M such that $q_1 A \to q_2 \in R$, rules from (3) and (4) are used. Remember from the definition of $L(G, F)$, that nonterminals from W are erased from $\phi(G, F)$. So if A is nonterminal from W, we need to erase it. On the contrary, if A is terminal from W, we need to keep it. Rules from (3) are used to keep described terminals and rules from (4) erase described nonterminals.

To prove $L(G, F) = L(H)$, we first prove $L(H) \subseteq L(G, F)$; then, we establish $L(G, F) \subseteq L(H)$. To demonstrate $L(H) \subseteq L(G, F)$, we first make three observations—(i) through (iii).

(i) By using rules constructed in (1) and (2), H makes a derivation of the form

$$\langle q_s S q_f \rangle \Rightarrow^* x_0 \langle q_1 Z_1 q_2 \rangle x_1 ... \langle q_n Z_n q_{n+1} \rangle x_n$$

where $x_i \in (T - W)^*, 0 \leq i \leq n, \langle q_j Z_j q_{j+1} \rangle \in U, Z_j \in W, 1 \leq j \leq n, q_1 = q_s, q_{n+1} = q_f, q_1 ... q_{n+1} \in Q, q_f \in Q_F$.

(ii) If

$$\langle q_s S q_f \rangle \Rightarrow^* x_0 \langle q_1 Z_1 q_2 \rangle x_1 ... \langle q_n Z_n q_{n+1} \rangle x_n$$

in H, then

$$S \Rightarrow^* x_0 Z_1 x_1 ... Z_n x_n$$

in G, where all the symbols have the same meaning as in (i).

(iii) Let H make

$$x_0\langle q_1 Z_1 q_2\rangle x_1 ... \langle q_n Z_n q_{n+1}\rangle x_n \Rightarrow^* y$$

by using rules constructed in (3) and (4), where $y \in T^*$, and all the other symbols have the same meaning as in (i). Then, for all $1 \leq j \leq n, q_j Z_j \to q_{j+1} \in R, y = x_0 U_1 x_1 ... U_n x_n$, where $U_j = {}_T\omega(Z_j)$. As $q_j Z_j \to q_{j+1} \in R, 1 \leq j \leq n, q_1 = q_s$ and $q_{n+1} = q_f, q_f \in Q_F$, we have $Z_1 ... Z_n \in L(M)$.

Based on (i) through (iii), we are now ready to prove $L(H) \subseteq L(G, F)$. Let $y \in L(H)$. Thus $\langle q_s S Q_F\rangle \Rightarrow^* y, y \in T^*$ in H. As H is an ordinary CFG, we can always rearrange the applications of rules during $\langle q_s S Q_F\rangle \Rightarrow^* y$ in such a way that

$$\begin{array}{lll} \langle q_s S Q_F\rangle & \Rightarrow & \langle q_s S q_f\rangle & (\alpha) \\ & \Rightarrow^* & x_0\langle q_1 Z_1 q_2\rangle x_1 ... \langle q_m Z_m q_{m+1}\rangle x_m & (\beta) \\ & \Rightarrow^* & y & (\gamma) \end{array}$$

so that during (α), only a rule from (0) is used, during β only rules from (1) and (2) are used, and during (γ) only rules from (3) and (4) are used. Recall that $Z_1 Z_2 ... Z_n \in F$ (see (iii)). Consequently, ${}_W\omega(x_0 Z_1 x_1 ... Z_n x_n) \in F$. From (3), (4), (ii), and (iii), it follows that

$$S \Rightarrow^* x_0 Z_1 x_1 ... x_{n-1} Z_n x_n \text{ in } G$$

Thus, as $L(M) = F$, we have $y \in L(G, F)$, so $L(H) \subseteq L(G, F)$.
To prove $L(G, F) \subseteq L(H)$, take any $y \in L(G, F)$. Thus,

$$\begin{array}{l} S \Rightarrow^* x_0 Z_1 x_1 ... x_{n-1} Z_n x_n \text{ in G, and} \\ y = {}_T\omega(x_0 Z_1 x_1 ... x_{n-1} Z_n x_n) \text{ with } Z_1 ... Z_n \in F, \end{array}$$

where $x_i \in (T - W)^*, 0 \leq i \leq n, Z_j \in W, 1 \leq j \leq n$. As $Z_1 ... Z_n \in F$, we have $q_1 Z_1 \to q_1, ..., q_n Z_n \to q_{n+1} \in R, q_1 ... q_{n+1} \in Q, q_1 = q_s, q_{n+1} = q_f, q_f \in Q_F$. Consequently, from (0) through (4) of the Construction, we see that

$$\begin{array}{l} \langle q_s S Q_f\rangle \Rightarrow \langle q_s S q_f\rangle \\ \quad \Rightarrow^* x_0 Z_1 x_1 ... Z_n x_n \\ \quad \Rightarrow^* x_0 U_1 x_1 ... U_n x_n \end{array}$$

where $U_j = {}_T\omega(Z_j), 1 \leq j \leq n$. Hence, $y \in L(H)$, so $L(G, F) \subseteq L(H)$.
Thus, $L(G, F) = L(H)$. □

Theorem 3.2. $\mathbf{CF_{REG}} = \mathbf{CF}$.

Proof. Clearly, $\mathbf{CF} \subseteq \mathbf{CF_{REG}}$. From Lemma 3.4, $\mathbf{CF_{REG}} \subseteq \mathbf{CF}$. Thus, Theorem 3.2 holds true. □

3.3 Results

In this chapter, two types of the final sentential forms were studied. The sentential forms of the context-free grammars finalized by the palindromial languages and the sentential forms of the context-free grammars finalized by the regular languages. Notice that these final sentential forms differ only by the type of finalizing language. Refer back to the Claim 2.1 to see, that the families of palindromial and regular languages are mutually incomparable. Furthermore, both of these families are small compared to the context-free languages. However, when palindromial languages are used for finalization, the recursively enumerable language family is achieved. In contrast, when the regular languages are used as the finalizing language, the resulting language family is still context-free.

The most interesting result of this chapter is, that by using a single palindromial language $\{w\#w^R \mid w \in \{0,1\}^*\}$ as the finalizing language, the recursively enumerable language family is achieved.

The biggest advantage of the final sentential forms is, that by using the palindromial languages as the finalizing language, we achieve recursively enumerable languages. The palindromial languages have a simple structure, therefore it is easy to check, whether some word belongs to some palindromial language.

The disadvantage of the final sentential forms is, that during the last part of a derivation process, all the nonterminals from W are erased from them. By this mechanism, the resulting language is produced. We could reproduce this mechanism in the underlying context-free grammar G simply by adding rule $X \to \varepsilon$ for each nonterminal from W. The problem occurs when we want to decide, whether some word belongs to the generated language $L(G, F)$. We need to reconstruct every possible sentential form of G and decide, whether at least one of them is also the final sentential form.

Chapter 4

Syntax analysis

In the formal language theory, a membership problem is a problem with the most practical use. It asks whether or not a given word is a member of a given language. However, since the languages are potentially infinite and we are mostly interested in the infinite languages, we describe them by formal models. For example, we ask whether the given word belongs to the language generated by a given context-free grammar. Solution of membership problem has different complexity based on the formal model used to describe the language. Intuitively, it is easier to decide the membership problem for the regular grammars than for the unrestricted grammars.

The membership problem has many practical uses. Its biggest use is in programming languages. Source codes of programming languages are processed by finite automata as part of the lexical analysis of a compiler, and it is determined whether the structure of lexemes is fulfilled. As part of the syntactical analysis in compilers, it is checked whether or not the source code fulfills the specification of said programming language, described by some grammar (usually context-free grammar). In other words, the membership problem is decided for the source code and the programming language specified by some grammar. The syntactical analysis is also often called *parsing* and is used for the membership problem, where the language is described by some grammar.

Syntax analysis can be divided into two main categories, bottom-up, and top-down syntax analysis. The bottom-up analysis starts with a word, which represents the bottom of a derivation tree. Then, by use of reduction rules, bottom-up syntax analysis tries to reduce a given word to some representation of a start symbol. Conversely, top-down analysis usually starts with a representation of a start symbol, which represents the top of a derivation tree. Then by use of expansions than represent a derivation rule, it tries to expand this start symbol to the given word.

In this chapter, we describe the fundamentals of syntax analysis first. Syntax analysis is an extensive topic tightly connected to the compilers. Since the syntax analysis is not the main focus of this thesis, only the basics needed for comprehension of our use are described. More information about parsing and compilers themselves can be found in [5].

The main part of this chapter is the algorithm Cocke-Younger-Kasami (CYK for short). Algorithm CYK is a method of the syntax analysis of the context-free grammars. It can be modified to work as syntax analysis for other grammar-based formal models.

4.1 Top-down syntax analysis

The top in the top-down syntax analysis refers to the top of the derivation tree, meaning the starting symbol. Conversely, bottom refers to the bottom of the derivation tree, meaning the derived word. Top-down syntax analysis is more straightforward than bottom-up syntax analysis. The most naive way of top-down syntax analysis is to start with the start symbol, and generate all of the possible words by the given grammar, until we either generate same word as an input word or we deplete all the words of the same size as an input word. Obviously, this method is inefficient. Another example of the method of top-down syntax analysis is the recursive descent parser which is widely used in compilers.

4.2 Bottom-up syntax analysis

Bottom-up syntax analysis represents a fundamentally opposite way of syntax analysis to top-down syntax analysis. It with a given word and it tries to work its way to the starting symbol by application of reductions. In the compilers, LR parsing is a method for bottom-up syntax analysis. In this chapter, we are using the algorithm Cocke-Younger-Kasami, which is the bottom-up syntax analysis. However, while it is a method for syntax analysis, it is not the preferred method for compilers, since it requires a CFG in Chomsky normal form.

4.3 Algorithm Cocke-Younger-Kasami

An algorithm Cocke-Younger-Kasami (CYK for short) represents a general way to solve the membership problem for CFGs. It takes some word and the CFG in Chomsky normal form (see Definition 2.41) as an input. Recall that grammar in Chomsky normal form if and only if for all the rules in grammar, the right-side contains either two nonterminals or one terminal. The output of the algorithm is, whether or not an input grammar can generate an input word. The main part of this algorithm is the so-called CYK table. It is a two-dimensional array with both dimensions of the size of the input word.

Basic idea. Let $G = (V, T, P, S)$ be an input CFG and $w = w_1 ... w_n$, with $w_i \in T, 1 \leq i \leq n$ be an input word. The base of the algorithm is two dimensional array of sets of nonterminals $CYK[i, j]$, for $1 \leq i \leq j \leq n$. From definition of Chomsky normal form, we know that in order to $w \in L(G)$ hold true, there must exist at least one nonterminal $A \in V - T$, such that $A \rightarrow w_i \in R$, for each $1 \leq i \leq n$. Therefore, the algorithm starts by placing these nonterminals into corresponding positions in CYK. For each $1 \leq i \leq n$, add each nonterminal A such that $A \rightarrow w_i$ to $CYK[i, i]$. After that, pairs of nonterminals on right-side of the rules can be reduced to single nonterminal. Whenever $B \in CYK[i, j], C \in CYK[j + 1, k]$ and $X \rightarrow BC \in P$, add X to $CYK[i, k]$. The end of algorithm occurs when no further reduction can be applied. At the end of the run of the algorithm, if $S \in CYK[1, n]$, $w \in L(G)$. Otherwise, $w \notin L(G)$.

A successful run of the algorithm can be seen in Figure 4.1. The algorithm CYK is taken from [5]. This algorithm is also described in the form of pseudocode in Algorithm 4.1.

Algorithm CYK is also suitable for other types of grammars, assuming that it is properly modified.

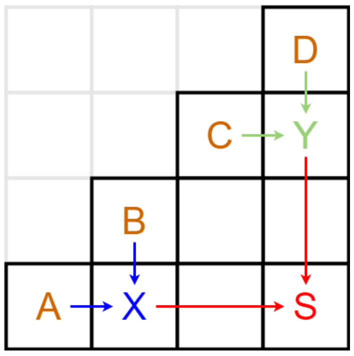


Figure 4.1: Example of the successful run of the algorithm CYK with a visualization of the CYK table for grammar $G = (\{A, B, C, D, X, Y, S\}, \{a, b, c, d\}, \{S \rightarrow XY, X \rightarrow AB, Y \rightarrow CD, A \rightarrow a, B \rightarrow b, C \rightarrow c, D \rightarrow d\}, S)$ for an input word *abcd*. Bottom left corner represents position [1,1] and bottom right corner represents position [1,4] in the table.

Algorithm 4.1 An algorithm Cocke-Younger-Kasami for syntax analysis of context-free grammars

Input

- a context-free grammar, $G = (V, T, P, S)$ in Chomsky normal form
- $w = a_1 a_2 ... a_n$ where $a_i \in T, 1 \leq i \leq n$, for some $n \geq 1$.

Output

- **ACCEPT** if $w \in L(G)$
- **REJECT** if $w \notin L(G)$

Method

```
$CYK[i,j] = \varnothing$ for $1 \leq i \leq j \leq n$
for i = i to $n$ do
    if $A \rightarrow a_i \in R$ then
        add $A$ to $CYK[i,i]$
repeat
    if $B \in CYK[i,j], C \in CYK[j+1,k], A \rightarrow BC \in R$ for some $A, B, C \in (V - T)$ then
        add $A$ to $CYK[i,k]$
until no changes
if $S \in CYK[1,n]$ then
    ACCEPT
else
    REJECT
```

4.4 Syntax analysis of final sentential forms

In Chapter 3, we described the properties of the final sentential forms with two different types of finalizing languages. When we use the palindromial languages to finalize sentential forms of the context-free grammars, we get generative power equivalent to turing machines. When regular languages are used for finalizing, the resulting languages are context-free. Therefore, we are only interested in finalizing by palindromial grammars.

The algorithm CYK for syntax analysis of CFGs was described in Section 4.3. Since the basis of the final sentential forms are sentential forms of CFGs, we modify the CYK algorithm for our purposes. The modifications of the CYK algorithm must solve the increased complexity of the final sentential forms.

Problems of syntax analysis of final sentential forms

Let $G = (V, T, P, S)$ be a CFG and F a palindromial language. In Definition 3.2 we defined the final sentential forms $\phi(G, F)$ and the language of G finalized by $F, L(G, F)$. For the syntax analysis, it is needed to somehow represent this palindromial language F. For this purpose, we use the set W since the palindromial language have trivial form. Remember, that palindromial grammars have only a single nonterminal and rules in the form $S \to aSa$ or $\to \#$, where S is the only nonterminal, a is a terminal, and $\#$ is a terminal that only occurs on the right-side of a single rule. Therefore, every palindromial language over W is defined as $\{w\#w^R \,|\, w \in W^*\}$.

It may seem, that syntax analysis of the final sentential forms is straightforward. We already described the algorithm for syntax analysis of context-free grammars, and syntax grammar of palindromial grammars is trivial. However, there is one more problem. The biggest challenge during syntax analysis of the final sentential forms is a reconstruction of the deleted symbols. Remember that $L(G, F)$ is defined as $L(G, F) = \{{}_T\omega(y) \,|\, y \in \phi(G, F), {}_{(N-W)}\omega(y) = \varepsilon\}$. By this definition, all of the occurrences of $(W - T)$ are deleted. Parsing of final sentential forms consists of these three steps

(1) reconstruction of y, such that ${}_T\omega(y) \in L(G, F)$;

(2) syntax analysis of y by the CFG G;

(3) syntax analysis of ${}_W\omega(y)$ by the palindromial grammar M, such that $L(M) = F$.

Let us start with the description of step (3), since it is the easiest.

Syntax analysis of the palindromial languages

In Lemma 3.3, we have proven, that by using a palindromial finalizing language $F = \{w\#w^R \,|\, w \in \{0, 1\}^*\}$, the resulting language family is recursively enumerable.

The usage of the concrete palindromial language F in Lemma 3.3 results in a stronger theoretical result. However, the use of this language is impractical in practice. Instead of this language, we allow usage of any palindromial language. To represent the palindromial language, we could use palindromial grammar (see Definition 2.28). However, notice that the forms of the rules of palindromial grammars are very constricted, allowing only rules in the form $S \to \#$, or $S \to aSa$, where S is the only nonterminal, $\#$ is the central symbol and the a is any terminal except for $\#$. In final sentential forms, the palindromial language F is defined over the set W. Therefore, we represent F indirectly by the set W. Let

$G = (V, T, P, S)$ be a palindromial grammar such that $L(G) = F$. We assume that $T = W$ and $V = W \cup \{S\}, P = \{S \to xSx \mid x \in W - \{\#\}\} \cup \{S \to \#\}$ and S is the start symbol. As result, every final language F can be expressed as $\{w\#w^R \mid w \in (W - \{\#\})^*\}$. This allows us to perform the syntax analysis of the palindromial languages easily without working with the palindromial grammars.

Palindromial languages have a simple structure. We must verify that the input word x is of the form $x = y\#z$, where $y, z \in (W - \{\#\})^*$, and $y = z^R$. The central symbol $\#$ is always member of W. The algorithm for syntax analysis palindromial languages is described in the form of the pseudocode in Algorithm 4.2.

Algorithm 4.2 A algorithm for syntax analysis of the palindromial languages

Input

- a set W, such that $F = \{w\#w^R \mid w \in (W - \#)^*\}$
- an input word w

Output

- **ACCEPT** if $w \in F$
- **REJECT** if $w \notin F$

Method

```
if $w = x\#y, x, y \in W - \{\#\}, |x| = |y|$, and $x = y^R$ then
    ACCEPT
else
    REJECT
```

Syntax analysis of the context-free final sentential forms

Let $L(G, F)$ be a language of G finalized by F, where G is a CFG and F is a palindromial language. The sentential forms of G are the basis of the final sentential forms, which then form $L(G, F)$. As an input of syntax analysis, we take some word w which consists of terminals. Recall Definition 3.3 of $L(G, F)$. Notice, that by this definition, $w \in L(G) \iff w \in L(G, F)$ may not hold. And it most likely does not hold. This property comes from the fact, that we construct $L(G, F)$ by erasion of symbols from $W - T$ from sentential forms of G.

Constraints of the context-free final sentential forms

In Section 3.1, we have shown and proven, that by using a palindromial language to finalize sentential forms of CFGs, we achieve recursively enumerable language family. When we look at the proof of Lemma 3.3, we can see that the context-free grammar $G = (V, T, P, S)$ that is constructed in this proof has certain properties. G is propagating, meaning that it contains no ε-rules. Also notice, that symbols from W are never rewritten, even though they are nonterminals. This means that for G, symbols from W are useless. However, in final sentential forms, these symbols play a crucial role.

In the context-free grammars, we defined the notion of ambiguity (see Definition 2.39). In final sentential forms, there may exist multiple sentential forms that can be transformed into the same word. Remember, that this transformation is done by application of ${}_T\omega$ to the sentential form u. However, u does not have to be the final sentential form. If we allow symbols from W to be rewritten, there may exist an infinite number of sentential forms that can be transformed into the same word. We would potentially need to check every one of them, whether at least one of them belong to the final language. We could not reliably decide, whether there exists some sentential form that also belongs to the final language since there could be an infinite number of these sentential forms. Therefore, we cannot allow nonterminals from W to be rewritten, or right-side to consist only of nonterminals from W. The rules in these forms could potentially keep expanding the sentential form without adding any terminals.

In conclusion, the context-free grammar $G = (V, T, P, S)$ used to generate the final sentential forms must fulfill following properties:

- G is propagating;
- $P \subseteq V - (W \cup T) \times V^*((V - W) \cup T))V^*$.

4.4.1 Modification of algorithm Cocke-Younger-Kasami

In Section 4.3, we described the Cocke-Younger-Kasami syntax analysis algorithm for context-free grammars. Let $L(G, F)$ be a language of G finalized by F, where $G = (V, T, P, S)$ is a CFG and F is a palindromial language. Since we use CFG to generate sentential forms that are then finalized by F, we decided to use the algorithm CYK for syntax analysis of final sentential forms. Algorithm CYK requires the input CFG to be in Chomsky normal form (see Definition 2.41). Earlier in this chapter, we constrained CFGs for final sentential forms to propagating CFGs. Meaning that they contain no ε-rules. To transform G to the Chomsky normal form, we need to remove unit rules and apply Algorithm 2.1 to G. Notice, that by Definition 2.41, useless symbols are prohibited in CNF. However, we require that nonterminal symbols from W are never rewritten, therefore they are useless by Definition 2.35. Hence, we ignore this requirement. From this point, we assume that G is already in Chomsky normal form.

The algorithm CYK takes a word z consisting only of terminals as input. The language $L(G, F)$ may contain the word z, but G does not have to be able to generate it. G generates sentential forms, that are then finalized and transformed by the application of ${}_T\omega$. So we need algorithm CYK to find such sentential form s of G, that s can be finalized by F and transformed to z input word by application of ${}_T\omega$. Meaning that CYK needs to find s, such that ${}_W\omega(s) \in F$, and ${}_T\omega(s) = z$.

The CYK table in the CYK algorithm has a size according to the length of the input word z, $|z|$. The sentential form s of G, such that ${}_W\omega(s) \in F$, and ${}_T\omega(s) = z$ may be longer than z. There is no way to compute the length of s at the beginning of the algorithm, since we obtain s at the end of the algorithm. Therefore, we have to adapt the algorithm for this fact. By the definition of final sentential forms, they can only contain nonterminals from W. This means, that the extra length of s, $|s|$ over $|z|$ comes from nonterminal symbols from W. Since G is in Chomsky normal form, nonterminals from W can only occur on the right-side of the rules in three cases:

(1) $A \to BC$, where $A \in V - (W \cup T), B \in W - T, C \in V - (W \cup T)$;

(2) $A \to BC$, where $A \in V - (W \cup T), B \in V - (W \cup T), C \in W - T$;

(3) $A \to BC$, where $A \in V - (W \cup T), B \in W - T, C \in W - T$.

The cases (1) and (2) mean, that the nonterminal symbol from W is either on the right or left position of pair of nonterminals. Remember that the case (3) is forbidden by the restrictions that we put on the input CFG. The nonterminals from W are simply removed to acquire the resulting word in $L(G, F)$. Therefore, we check each nonterminal X in the CYK table, whether it could be part of some rule described in (1) and (2). If such rule exists, make reduction according to this rule and place the left-side nonterminal on the same position as X. With the addition of handling of these rules, we can continue the algorithm CYK as usual.

By mentioned modification, we are able to decide, whether there exists a sentential form s generated by G, such that ${}_T\omega(s) = z$, where z is the input word. However, this does not mean that $z \in L(G, F)$. We also need to verify that ${}_W\omega(s) \in F$. In the original algorithm CYK, once we get the start symbol $S, S \in CYK[1, n]$, where n is the length of z, in the CYK table, we declare syntax analysis as successful. For our modification, once we reach $S \in CYK[1, n]$, we need to check, whether ${}_W\omega(s) \in F$. For this purpose, we need to extract ${}_W\omega(s)$ and pass it to syntax analysis for the palindromial languages. In CYK for CFGs, it doesn't matter what sequence of reductions was used to reach the start symbol. In other words, it doesn't matter which derivation tree we discovered. Once we know that some derivation tree exists for the input word x and input grammar H, we know that $x \in L(H)$. Even if H is ambiguous. On the contrary, for the syntax analysis of final sentential forms, we also need to check that ${}_W\omega(s) \in F$.

To extract ${}_W\omega(s)$ from sentential form s, we need to keep the history of reductions for each symbol. That means, that we are effectively constructing the derivation tree t. The root of t is start symbol $S \in CYK[1, n]$. The leaves of t are terminals and nonterminal from W, and nodes are nonterminals. The child nodes of each node represent the right side of the rules that were used to reduce this node. We construct t in a bottom-up way. It is important that we construct every possible derivation tree. Therefore, the CYK table might contain duplicate nonterminals in the same position, that represent different derivation subtrees. Of course, when we perform reduction with rules that contain nonterminal from W, we must put that nonterminal into the derivation tree.

Once we construct the full derivation tree t, we can perform extraction of ${}_W\omega(s)$. Notice that because of the use of Chomsky normal form, each node of t has exactly two child nodes, except for nodes whose child node is a leaf. This means, that we can think of t as a binary tree. We traverse t in an inorder fashion, and when we encounter the symbol from W, we put it as the next symbol in a word u that reconstructs ${}_W\omega(s)$. When the traversal of t is finished, ${}_W\omega(s) = u$.

Once we obtain u, we can perform the syntax analysis of $u \in F$. If the result is a success, $z \in L(G, F)$, otherwise, we continue the CYK algorithm to get the next derivation tree. Once we cannot find any new derivation tree, and no syntax analysis of $u \in F$ was successful, the result of $z \in L(G, F)$ is unsuccessful.

The whole modification of the CYK algorithm for final sentential forms can be seen as pseudocode in Algorithm 4.3.

Algorithm 4.3 A modification of algorithm Cocke-Younger-Kasami for syntax analysis of final sentential forms

```
Input
  • a context-free grammar, $G = (V, T, P, S)$ in Chomsky normal form
  • a set $W$, such that $F = \{w\#w^R \mid w \in (W - \#)^*\}$
  • $w = a_1 a_2 ... a_n$ where $a_i \in T, 1 \leq i \leq n$, for some $n \geq 1$.
Output
  • ACCEPT if $w \in L(G, F)$
  • REJECT if $w \notin L(G, F)$
Data structures
structure $Item$ $\{symbol, leftParent, rightParent\}$
Method
  $CYK[i, j] = \varnothing$ for $1 \leq i \leq j \leq n$
  for i = i to $n$ do
      if $A \to a_i \in R$ then
          add $Item\{symbol : A, leftParent : a_i, rightParent : null\}$ to $CYK[i, i]$
  repeat
      for every unique $X_{Item} \in CYK[i, j], Y \in W$ such that, $Z \to X_{symbol}Y \in R$ or $Z \to YX_{symbol} \in R$ do
          add $Item\{symbol : Z, leftParent : X_{Item}, rightItem : Y\}$ or
          $Item\{symbol : Z, leftParent : Y, rightItem : X_{Item}\}$
          to $CYK[i, j]$, respectively
      for every unique $B_{Item} \in CYK[i, j], C_{Item} \in CYK[j + 1, k]$ such that, $A \to B_{symbol}C_{symbol} \in R$ do
          add $Item\{symbol : A, leftParent : B_{Item}, rightItem : C_{Item}\}$ to $CYK[i, k]$
  until no changes
  if there exists any $S \in CYK[1, n]$ and $TraverseItem(S_{Item}) \in F$ then
      ACCEPT
  else
      REJECT

  function TraverseItem(Item X)
      $word = \varepsilon$
      if $X_{leftParent}$ is not null then
          $word = word +$ TraverseItem($X_{leftItem}$)
      if $X_{symbol} \in W$ then
          $word = word +$ TraverseItem($X_{leftItem}$)
      if $X_{rightParent}$ is not null then
          $word = word +$ TraverseItem($X_{leftItem}$)
      return $word$
```

Chapter 5

Implementation and applications

In Chapter 3, we have shown that language $L(G, F)$ can represent any recursively enumerable language, in case that G is a propagating context-free grammar and F is a palindromial language. This means that the final sentential forms of context-free grammar finalized by a palindromial language are equivalent to turing machines.

In this chapter, we firstly discuss the implementation of the introduced syntax analysis for the final sentential forms. Then we discuss the applications of the final sentential forms that can be put into the practice using the implementation of the syntax analysis.

When we mention the final sentential forms in this chapter, we always refer to the context-free sentential forms finalized by a palindromial language.

5.1 Implementation

The problems of syntax analysis of final sentential forms are described in Chapter 4. The point of the syntax analysis is to determine whether the given word is a member of a language described by some final sentential forms. This implementation is supposed to act as a demonstration of the final sentential forms in practice.

One of the first stages of the implementation is a selection of a programming language. For purposes of the implementation part of this thesis, I chose the programming language **Python**, version **3.10**. Python is interpreted high-level programming with rich abstractions and standard library. Therefore, it is suitable for prototyping and small-scale implementations, where its slower execution speed does not cause an issue.

The implementation is written as a console application. The main purpose of this implementation is to demonstrate the capabilities of the final sentential forms.

File structure

The implementation of the syntax analysis of the final sentential forms is divided into multiple parts—files. There are 5 files that contain source code. Those are

(a) `parse.py`;

(b) `fileReader.py`;

(c) `rules.py`

(d) `chomskyNormalForm.py`;

(e) `cyk.py`.

The file (a) represent the console interface between the user and the program. It reads the command-line parameters and performs the syntax analysis of the final sentential forms based on these parameters. The input file that defines the input context-free grammar and the palindromial language is read and processed by file (b). It provides the internal representation of the mentioned CFG and the palindromial language to the other parts of the program. The file (c) contains the data structures used for the internal representation of the symbols and the rules of context-free grammars.

The algorithm Cocke-Younger-Kasami requires the input CFG to be in Chomsky normal form. For the user, it would be impractical and difficult to provide the CFG in Chomsky normal form directly. Therefore, the file (d) performs the transformation of the input CFG G to the CFG $\overline{G}$ in Chomsky normal form, such that $L(G) = L(\overline{G})$. The syntax analysis itself is done entirely in the file (d). It performs both the modified Cocke-Younger-Kasami algorithm and the syntax analysis of the palindromial language.

5.1.1 Command-line interface

Interaction with the user is done using the command-line. Meaning that there is no graphical interface. The interaction with the implemented program is simple. The user needs to provide the input file and the word to parse. Both can be easily done through the command-line.

The file that implements the command-line interaction with the user is `parse.py`. The program accepts only 3 command-line parameters and if syntax analysis should be executed, it passes the parameters to the proper objects. The parameters that `parse.py` accept are:

(1) `-h`

(2) `-i <input file>`

(3) `-w <word to parse>`

The parameter (1) is used to print the help text to the standard output. When this parameter is used, only the help text is printed. No syntax analysis is performed. The input file is specified by the parameter (2). The input file can be specified by either relative or absolute path. This parameter is mandatory when the parameter (1) is not used. The last parameter (3) is used to specify the input word for syntax analysis. This parameter is also mandatory when the parameter (1) is not used.

We mentioned, that we must specify the input CFG and the palindromial grammar. However, so far we have not specified the format of this input file.

Data structures for the context-free grammar

In the implementation of the final sentential forms, we need to represent the context-free grammars internally by proper data structures. Specifically, the symbols and the rules of the context-free grammar. The file `rules.py` contains 6 classes, which represent the data structures.

Symbols

Let $G = (V, T, P, S)$ be an input context-free grammar. The symbols of the total alphabet V are represented by the classes `Symbol`, `Terminal`, and `NonTerminal`. These three classes use the inheritance. The `Symbol` class is a parent class and both `Terminal` and `NonTerminal` classes are its subclasses (they inherit from `Symbol` class). The only instance variable that these classes contain is `value` which the character representing the symbol. The main purpose of the `Terminal` and `NonTerminal` is to differentiate between terminals and nonterminals. These two classes have their equivalence defined based on their type and their value, meaning that nonterminal with value A is not equal to the terminal with the value A.

The `SymbolSequence` represents the sequences of the terminals and nonterminals. These sequences are mainly used as the right-sides of the rules. This class also contains methods that are able to classify a type of the sequence. For example `isCNF` method decides, whether the sequence satisfies the Chomsky normal form.

Rules

We have already described, how are the symbols and the symbol sequences of the G represented by the data structures. The rules use both the symbols and the symbol sequences. The nonterminal symbols are the left-side of the rules, and the symbol sequences are the right-sides of the rules. The class that represent the rules is `Rule`. It contains two instance variables—`leftSides` and `rightSide`. Each rule object keeps the list of the left-sides, that can be rewritten to some single right-side.

To manage all the rules of the input CFG, class `RuleHandler` is used. It contains three instance variables—`terminals`, `nonterminals`, and `rules`. The instance variables `terminals` and `nonterminals` are lists, that are used mainly during the creation of the instances of the `SymbolSequence` class. The `rules` instance variable is the data structure hash table (`dictionary` in Python) used to store all the rules of the input CFG. Hash tables are used because of their native built-in support in Python and their fast value access. Hash tables store key-value pairs, similarly to the JSON files. The key is used to access the value in hash map. Any data type can be used as the key as long as the hash function if provided for that data type. For the `rules` hash table, the keys are `SymbolSequence` objects and the values are `Rule` objects. The `Rule` objects contain the list of the left-sides that can be rewritten to the same right-side. Therefore, the keys are the right-sides of the rules and the lists of the left-sides are the values. It may seem counter-intuitive to use the right-sides as the keys, but it simplifies the implementation of the syntax analysis.

5.1.2 Input file

The input file contains the specification of the input CFG and the palindromial grammar in **JSON** format. The abbreviation JSON stands for **J**avaScript **o**bject **n**otation. It is a standardized text-based syntax used for storing structured data. JSONs syntax is light-weight which makes it easily readable for the user. Even though JSON is derived from the programming language JavaScript, it is language-independent. Python programming language allows us to easily work with the JSON files by providing the module `json`. JSON is also widely used in real-world applications.

We mentioned that JSON has light-weight syntax. However, the full specification of the JSON syntax has complicated details. The specification of the JSON is specified in both

ECMA standard[1] and IETF RFC[2]. We describe only the basics needed for the user to write a proper input file.

The main structure of the JSON is the `object`. The `object` definition starts with a left curly bracket `{` and ends with a right curly bracket `}`. Inside the curly brackets, zero or more name-value pairs are defined. The pairs are delimited by comma `,`. The name and the value of the pair are delimited by a colon `:`. In the name-value pair, the name is string literal and the value can be an object, array, string, number, or `true/false/null` literal.

In JSON, array structure starts with the left square bracket `[` and ends with the right square bracket `]`. Inside the square brackets, zero or more values delimited by the comma can be specified. The string and number values are in the usual format. The `true` and `false` values represent boolean values. The `null` value is the special value for unspecified value. The white-space in JSON is ignored.

The input file must contain the following name-value pairs

(a) `nonterminals`

(b) `terminals`

(c) `rules`

(d) `W`

The values of the (a), (b), (c) and (d) are an arrays of a strings. Let $G = (V, T, P, S_G)$ be a context-free grammar, $F \subseteq W^* \in \mathbf{PAL}$. The (a) represents $V - T$, the (b) represents T, the (c) represents P and the (d) represents W, that is used as a representation of the final language F. The strings of (a), (b) and (d) must be a single character.

The rule strings of (c) start with a single symbol representing a nonterminal, followed by the dash and „greater-than“ symbols '`->`', and a right-sides. The right-side is sequence of terminal and nonterminal symbols. Each rule string can contain single or multiple right-sides. When we want to declare multiple right-sides at once, we delimit them vertical bar symbol '`|`'. The example of the rule string is following:

```
R -> 1RA|0RB|0WC|0C
```

The start symbol is implicitly `S`.

Notice that the only information about the finalizing palindromial language is the set W. The palindromial languages can be represented by the palindromial grammars (see Definition 2.28). The rules of the palindromial languages are trivial. For some palindromial grammar $H = (\{S\} \cup W, W, R, S_H)$, we assume, that for every $x \in W - \{\#\}$, there exists rule $xSx \in R$. Furthermore, we assume that the set of the rules R contains only one other rule—$S \to \#$. As mentioned earlier, the final language $F \subseteq W^*$ is defined as $F = L(H)$, and only W is used for the syntax analysis.

5.1.3 Input file reader

Earlier in this section, we described the format and the contents of the input file. The input file is in JSON format and contains the symbols and the rules of the input context-free grammar and the set W. The file `fileReader.py` is responsible for the reading of

[1]ECMA-404 The JSON data interchange syntax
https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

[2]IETF RFC 8259 The JavaScript Object Notation (JSON) Data Interchange Format
https://datatracker.ietf.org/doc/html/rfc8259#section-2

the input file, the verification of the JSON contents, and the creation of the proper data structures from the input file.

The input file is read using the Python built-in package `json`. It allows us to load the JSON file directly and provides the basic verification of the JSON format. The function `json.load` loads the input file into the hash table (dictionary in Python). The JSON does not provide any feature to define and enforce the schema of the data. However, `fileReader.py` contains functionality to verify that the JSON contains all required data in proper format and noting more. The `fileReader.py` contains the class `InputData` whose instance it returns as an internal representation of the input file. The only purpose of this class is to store the contents of the loaded input file.

5.1.4 Transformation of context-free grammar to Chomsky normal form

In Section 4.4.1, we introduced a modification of the algorithm Cocke-Younger-Kasami that is suitable for syntax analysis of the final sentential forms. Remember, that the algorithm CYK requires the input CFG to be in Chomsky normal form. Chomsky normal form is described in Section 2.5.2. It also describes the algorithm for conversion of any CFG to Chomsky normal form. In Algorithm 2.1 for the transformation of the context-free grammar to the context-free grammar in Chomsky normal form, we assume that the input CFG is in proper form. Remember, that the CFG is proper (see Definition 2.40, if it contains no useless symbols, no ε-rules, and no unit rules. For the purposes of the syntax analysis of the final sentential forms, we require the nonterminals from W to be, by Definition 2.35, useless. Therefore, we ignore this property. The CFG of the final sentential forms is propagating, therefore it contains no ε-rules. The input CFG for the transformation may contain the unit rules, so we must be able to replace them with rules that satisfy Chomsky normal form.

The file `chomskyNormalForm.py` contains the class `CNF`. `CNF` loads the rules of the input context-free grammar G and transforms them into the Chomsky normal form. The main method of the `CNF` is the method `getRulesInCNF`. It takes the instance of the class `InputData` as its parameter and returns the instance of the class `RuleHandler`. At first, the method `getRulesInCNF` verifies the format of the rules and puts rules already in CNF to the new instance of the `RuleHandler` r. Then, the method `transformToCNF` transforms the remaining rules of G into the Chomsky normal form. These rules are added to r and r is then returned.

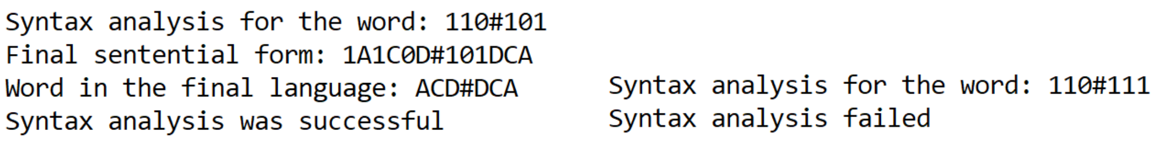
5.1.5 Syntax analysis

The file `cyk.py` implements the syntax analysis of the final sentential forms. Algorithm 4.3 describes the implemented syntax analysis. It is a modification of the CYK algorithm for the syntax analysis of the context-free grammars. By the described modifications of the CYK algorithm, we are able to perform the syntax analysis of the final sentential forms.

The file `cyk.py` contains two classes used for the syntax analysis of the final sentential forms. The class `CYKItem` is used for the representation of the items in the CYK table. This class contains three instance variables. The `value` represents the nonterminal itself. The `leftParent` and the `rightParent` represent left and the right nonterminals that were reduced to the nonterminal in `value`. Remember that every rule in Chomsky normal form satisfies the form $1 : A \to BC$ or $2 : A \to a$, where A, B, C are some nonterminals and a is some terminal. If A in CYK table is reduced according to the rule 1, the `leftParent` is the nonterminal B and the `rightParent` is the nonterminal B. If the rule 2 is used for the reduction, a is assigned to the `leftParent` and `rightParent` remains empty. When a rule with a single terminal on the right-side is used, the `leftParent` is used. The purpose of the left and right parents is to construct the parse tree, hence the name.

The second class of the file `cyk.py` is the `CYK`. Its purpose is to load the rules from the input file, transform them to CNF, and perform the syntax analysis on the given input word. The constructor of the `CYK` class takes two parameters—file name of the input file and the input word. The file `fileReader.py` is used to get the instance of `InputData` that is passed to the instance of the `CNF` class, which returns the rules in the Chomsky normal form.

The method `parse` performs the syntax analysis of the final sentential forms. The method `parsePalindromial` is used for the syntax analysis of the palindromial language described in Algorithm 4.2. As mentioned earlier, the syntax analysis of the palindromial language is done in regard to the set W. Meaning that the word must satisfy the form $x\#x^R, x \in (W - \{\#\})^*$. If the syntax analysis of the final sentential forms is successful, the input word, the final sentential form, the word in the final language, and the success message are printed to the standard output. In the case of the failure, only the input word and the failure message are printed to the standard output. The examples of the successful and unsuccessful runs can be seen in Figure 5.1.



```
Syntax analysis for the word: 110#101
Final sentential form: 1A1C0D#101DCA
Word in the final language: ACD#DCA
Syntax analysis was successful
```

```
Syntax analysis for the word: 110#111
Syntax analysis failed
```

Figure 5.1: Example of the outputs from the implemented program. The language $L(G, F)$ and the set W used is taken from Application 5.3. On the left side, an input word $110\#101 \in L(G, F)$. The ${}_W\omega(1A1C0D\#101DCA) \in F$ holds true. On the right side, an input word $110\#111 \notin L(G, F)$.

5.1.6 Comparison to other software

The main result of this thesis are the final sentential forms. They are newly introduced concept that allow us to filter the sentential forms of the context-free grammars and select only some of them, based on the final language. In Chapter 4, we described the algorithm for syntax analysis of the final sentential forms. The implementation in this thesis is supposed to act as demonstration of feasibility of syntax analysis of the final sentential forms. We have also shown some applications of the final sentential forms. Since the final sentential forms are newly introduced concept, we cannot compare this implementation to the other software fairly.

5.2 Applications

For many applications, the CFGs simply do not have sufficient generative power. By the finalization of context-free sentential forms, we achieve far greater generative power. We focus only on applications that cannot be represented by the context-free grammars. It would be unnecessary to represent context-free languages by final sentential forms since we are already using a CFG to generate sentential forms that are then finalized. One of the areas, where we applied the final sentential forms is linguistics.

Application 5.1. Let Σ represent an English alphabet. Set $L = \{w\#\sigma(w) \mid w \in \Sigma^+\}$, where σ is the homomorphism from Σ^* to $\{0,1\}^*$ defined as $\sigma(x) = 1$ and $\sigma(y) = 0$ for every consonant x in Σ and every vowel y in Σ, respectively. For instance, considering Σ as the English alphabet, $the\#110 \in L$ while $the\#100 \notin L$. Define the context-free grammar G with following rules.

- $S \to A\#B, B \to 0YB, B \to 0Y, B \to 1XB, B \to 1X,$
- $A \to aAY, A \to aY$ for all vowels a in Σ,
- $A \to bAX, A \to bX$ for all consonants b in Σ,

where the uppercases are nonterminals with S being the start nonterminal, and the other symbols are terminals. Set $W = \{X, Y, \#\}$ and $F = \{w\#w^R \mid w \in \{X,Y\}^*\}$. For instance, take this step-by-step derivation

$$\begin{aligned} S &\Rightarrow A\#B \Rightarrow tAX\#B \Rightarrow thAXX\#B \Rightarrow theYXX\#B \\ &\Rightarrow theYXX\#1XB \Rightarrow theYXX\#1X1XB \Rightarrow theYXX\#1X1X0Y \end{aligned}$$

In $theYXX\#1X1X0Y$, ${}_W\omega(theYXX\#1X1X0Y) = YXX\#XXY \in F$, and apart from $X, Y, \# \in W$, $theYXX\#1X1X0Y$ contains only terminals. The removal of all Xs and Ys in $theYXX\#1X1X0Y$ results into $the\#110$, which thus belongs to $L(G,F)$. Clearly, $L(G,F) = L$.

Application 5.1 indicates whether the letters are the vowels or the consonants. Notice, that the language generated in this example is a modification of the mentioned non context-free language $L_2 = \{ww \mid w \in \Sigma^*\}$. This means, that we are able to effectively represent the non-context-free language by the final sentential forms.

The following example is taken from the [6]. Consider two sentences. „Your great-grandparents are all your great-grandfathers and all your great-grandmothers.“ and „Your great-grandparents are all your grandfathers and all your grandmothers.“. The former sentence is both grammatically correct and truthful. The latter sentence is also grammatically correct but it is not truthful. For such sentence to be both grammatically correct and truthful, it must belong to the language

$$\begin{aligned} L = \{&\text{Your } \{\text{great-}\}^i\text{grandparents are all your} \\ &\{\text{great-}\}^i\text{grandfathers and all your } \{\text{great-}\}^i\text{grandmothers.} \mid i \geq 0\}. \end{aligned}$$

To create such truthful sentence, we need to know the context of the sentence. Notice that the language L corresponds to the language $L_x = \{a^n b^n c^n \mid n \geq 0\}$, which is known to be non-context-free language.

Application 5.2. Consider a language $L = \{a^n b^n c^n \mid n \geq 1\}$.
Let $G = (\{S, X, Y, 0, 1, a, b, c, \#\}, \{a, b, c\}, P, S)$ be a context-free grammar. Add a following rules to P

$$S \to X\#Y, X \to a0Xb, Y \to 0Yc, X \to a0b, Y \to 0c.$$

Set $W = \{0, \#\}$ and $F = \{0^n\#0^n \mid n \geq 1\}$. Let $H = (\{S, 0, \#\}, \{0, \#\}, \{S \to 0S0, S \to \#\}, S)$ be a palindromial grammar, such that $F = L(H)$. Then, $L(G,F) = L$. Notice that by $\#$ being a nonterminal in G, it gets erased and does not occur in words of the $L(G, F)$. In other applications, we set $\#$ as a terminal, therefore it does not get erased from the words in $L(G, F)$.

Application 5.3. Set $I = \{i(x) \mid x \in \{0,1\}^+\}$, where $i(x)$ denotes the integer represented by x in the standard way; for instance, $i(011) = 3$. Consider

$$L = \{u\#v \mid u, v \in \{0,1\}^+, i(u) > i(v) \text{ and } |u| = |v|\}.$$

Next, we define a CFG G and $F \in \mathbf{PAL}$ such that $L = L(G, F)$. Let $G = (V, T, P, S)$ be a context-free grammar. Set $V = \{S, X, \overline{X}, Y, \overline{Y}, C, D, 0, 1, \#\}, T = \{0, 1, \#\}$ and add following rules to P

- $S \to X\#\overline{X}$
- $X \to 1AX, X \to 0BX, X \to 1CY, X \to 1C$
- $\overline{X} \to 1\overline{X}A, \overline{X} \to 0\overline{X}B, \overline{X} \to 0\overline{Y}C, \overline{X} \to 0C$
- $Y \to \alpha DY, Y \to \alpha D, \overline{Y} \to \alpha\overline{Y}D, \overline{Y} \to \alpha D$ for all $\alpha \in \{0, 1\}$.

Set $W = \{A, B, C, D, \#\}$ and $F = \{w\#w^R \mid w \in \{A, B, C, D\}^+ \text{ and } n \geq 1\}$. Observe that $F = L(H)$, where $H = (\{S, A, B, C, D, \#\}, \{A, B, C, D, \#\}, \{S \to ASA, S \to BSB, S \to CSC, S \to DSD, S \to \#\}, S)$ is a palindromial grammar. Therefore, $F \in \mathbf{PAL}$. For instance, take this step-by-step derivation

$$\begin{aligned} S &\Rightarrow X\#\overline{X} \Rightarrow 1AX\#\overline{X} \Rightarrow 1A0BX\#\overline{X} \Rightarrow 1A0B1CY\#\overline{X} \Rightarrow 1A0B1C0D\#\overline{X} \\ &\Rightarrow 1A0B1C0D\#1\overline{X}A \Rightarrow 1A0B1C0D\#10\overline{X}BA \Rightarrow 1A0B1C0D\#100\overline{X}CBA \\ &\Rightarrow 1A0B1C0D\#1001\overline{Y}DCBA \Rightarrow 1A0B1C0D\#1001DCBA \end{aligned}$$

in G. Notice that

$$\begin{aligned} &{}_{W}\omega(1A0B1C0D\#1001DCBA) \in F, \text{therefore} \\ &{}_{T}\omega(1A0B1C0D\#1001DCBA) \in L(G, F). \end{aligned}$$

It is apparent that $L(G, F) = L$.

In Application 5.3, the language L is not context-free language. It represents the relation of operation of $>$ on two numbers represented in binary form.

We can also apply the final sentential forms in bioinformatics. We can use them to represent the secondary structures of RNA—pseudoknots.

Application 5.4. Set $\Sigma = \{a, g, c, u\}$. Define the bijection α from Σ to Σ as $\{a \longmapsto u, u \longmapsto a, c \longmapsto g, g \longmapsto c\}$ and extend it to Σ^*. Let $L = \{xy\alpha(x)\alpha(y) \mid x, y \in \Sigma^*\}$.

The Σ represents possible bases of the RNA and α represents their complements. Set $W = \{M, N, O, P, \#\}$ and $F = \{w\#w^R \mid w \in W^*\}$. Next, we construct CFG $G = (V, \Sigma, P, S)$, such that $L(G) = L$. Set $V = \Sigma \cup \{S, A, B, C, D\}$. Add following rules to P

$$S \to A\#C$$
$$A \to MaA, A \to NgA, A \to OcA, A \to PuA, A \to B$$
$$B \to MaB, B \to NgB, B \to OcB, B \to PuB, B \to Ma, B \to Ng, B \to Oc, B \to Pu$$
$$C \to uCM, C \to cCN, C \to gCO, C \to aCP, C \to D$$
$$D \to uDM, D \to cDN, D \to gDO, D \to aDP, D \to uM, D \to cN, D \to gO, D \to aP$$

The resulting $L(G, F) = L$. For example, $agcuucga \in L(G, F)$, since $uc = \alpha(ag)$ and $ga = \alpha(cu)$.

The specifics of the problem of representation of the pseudoknots in RNA are explaned in [7]. This example simplifies representation of pseudoknots slightly. In real world, the pseudoknots may contain unpaired bases between the paired sequences. In terms of the defined language, there could be sequences of Σ^* randomly between $x, y, \alpha(x), \alpha(y)$. We could easily add this fact to the grammar G, but it would make the example much bigger.

In Applications 5.1, 5.2, 5.3, and 5.4, we have shown some selected application of final sentential forms. Notice, that throughout these examples, only the propagating CFGs and the palindromial languages were used. Also, all of the languages that these examples presented are not context-free languages.

Input files for the applications

For each example presented in Section 5.2, we created an input file for the implemented program. The symbols used in the input files may differ, since we used special characters in the examples. The input files are `application1.json`, `application2.json`, `application3.json`, and `application4.json` for Application 5.1, Application 5.2, Application 5.3, and Application 5.4, respectively.

Chapter 6

Conclusion

The main purpose of this thesis is to introduce the notion of the final sentential forms and study their generative power. This is done by defining the notion of a weak identity ω, a finalizing language F, the final sentential forms of the context-free grammars, and a language of G, finalized by $F, L(G, F)$, where G is a context-free grammar. We prove that based on the selection of F, the resulting language family is either recursively enumerable or context-free. In addition to the introduction to the final sentential forms, this thesis studies their syntax analysis. This is achieved by the modification of the algorithm Cocke-Younger-Kasami for the context-free grammars. Finally, we study the applications of the final sentential forms for the non-context-free languages.

As the prerequisites for the final sentential forms, we defined the basic notions of the formal language theory. The symbols represent the most basic elements of the formal languages, the words are sequences of the symbols and the languages are sets of the words. The context-free grammars are the basis of the final sentential forms. They start the generation of the words with a start nonterminal, and they repeatedly apply their rewriting rules to generate sentential forms. Context-free grammars can only rewrite single nonterminals to any sequence of the nonterminals and terminals. The sentential forms are any sequences that can be generated by the grammar. When the sentential form contains only terminals, it becomes a member of a generated language. For the final sentential forms, we modify this mechanism to obtain a much more powerful language family.

By restriction of the rules of the context-free grammars, we obtain less powerful, but more specific grammars. The minimal-linear grammars contain only a single nonterminal—start symbol. This nonterminal can occur only once on the right side of the rule. In this thesis, we also use a notion of palindromial grammar, that can generate only palindromials delimited by the central symbol #. Palindromial grammars generate palindromial languages that are much weaker compared to the context-free languages.

To obtain final sentential forms from the context-free sentential forms, we use the weak identity ω, the set W, and the finalizing language F. The purpose of the weak identity is to remove all symbols that do not belong to the specified set from some word. We take a sentential form and use weak identity ω to erase all the symbols that do not belong to W. After the application of ω, we check whether or not the result belongs to the F. If the result belongs to the F, the original context-free sentential form is also a final sentential form. To obtain $L(G, F)$, we simply erase all nonterminals from the individual final sentential forms.

In formal language theory, we take great interest in the language families of the formal models. The language family of a formal model is a set of all the languages that can be represented by the said model. Possibly the most important language family is the recursively enumerable language family. This language family represents all the computable languages and is represented by the Turing machines. In this thesis, we prove that by using a context-free grammar G to generate the sentential forms and a minimal-linear language as the finalizing language F, the $L(G, F)$ represents the recursively enumerable language family. However, we need only one minimal-linear language to represent the recursively enumerable language family—the language $\{w\#w^R \mid w \in \{0,1\}^*\}$. In comparison, when we use a regular language as the finalizing language F, the resulting language family is the same as the language family of context-free grammars. In other words, we do not change the power of the context-free grammars.

To prove that $L(G, F)$, where G is context-free grammar and $F = \{w\#w^R \mid w \in \{0,1\}^*\}$, can represent any recursively enumerable language, we use queue grammars. Queue grammars also represent the recursively enumerable language family. We prove that we can simulate any queue grammar by $L(G, F)$, therefore $L(G, F)$ can represent any recursively enumerable language. The queue grammars have more complex derivation process than context-free grammars. Therefore we cannot simulate it directly. The idea is simple. We try to simulate any queue grammar with context-free grammar. The constructed context-free grammar for simulation also simulates derivations that are not possible. It cannot decide whether the simulation is correct or not. For this purpose, it suitably places the symbols from W, and the finalizing language F is used to distinguish the correct simulations from incorrect ones.

To prove that $L(G, F)$, where G is context-free grammar and F is a regular language, can represent only context-free languages, we use finite automata. We construct new context-free grammar for any context-free grammar and any regular language represented by a finite automaton. This newly constructed grammar simulates both the original context-free grammar and the run of the finite automaton over the symbols from W. Since we have proven that we in fact can construct such grammar, in this case, $L(G, F)$ can only represent context-free languages.

In formal language theory, the most important property of a formal model is the language it represents. The most important problem for the formal model is the membership problem. It asks whether a given word belongs to the language that the given formal model represents. Syntax analysis is used to decide the membership problem for the grammars. There are many algorithms for the syntax analysis. We decided to use the algorithm Cocke-Younger-Kasami (CYK for short). It is a general syntax analysis algorithm that requires the input grammar to be in Chomsky normal form. Chomsky normal form of the context-free grammars requires each right-side of the rule to consist of either two nonterminals or a single terminal. We use this algorithm as a basis for the syntax analysis of the final sentential forms since it can be modified for this purpose. The CYK algorithm works in a bottom-up way and it used the CYK table for the reductions. It starts with a word and it reduces the right-sides of the rules to the nonterminal on the left-side. When a start symbol is reached in a proper position of the CYK table, the syntax analysis is successful.

The biggest challenge for the syntax analysis of the final sentential forms is the erasion of the nonterminals from W. Since these symbols are not present in the input word, we need to reconstruct them. We introduce special reductions that handle the erasion of such nonterminals. Another issue is, that the result of an application of the weak identity ω must belong to the finalizing language L. To solve this, we record the history of reductions for each symbol in the CYK table. This means that a single position in the CYK table can contain multiple matching symbols, but they need to differ in their history of reductions. Once the start symbol occurs in the proper position, we need to check that the result of the weak identity ω belongs to the resulting language F. Furthermore, we need to restrict the underlying context-free grammar. We do not allow it to rewrite the nonterminals from W, or the right-side of any rule to consist purely of the nonterminals from W.

To demonstrate the described algorithm for the syntax analysis of the final sentential forms, we implemented said syntax analysis as a program. This program uses a command-line interface to specify the input context-free grammar, the finalizing language F, and the input word. The input context-free grammar and the finalizing language F are specified using an input file which is provided to the program using the command-line. The input file is in JSON format since it is simple and easily readable for the user. For the algorithm CYK, the input context-free grammar must be in Chomsky normal form. For more complex context-free grammar, it would be hard for the user to provide the input context-free grammar directly in Chomsky normal form. Therefore, the user does not have to specify it in Chomsky normal form. As part of the implementation, the input grammar is converted to the Chomsky normal form. Another purpose of this program is, that it allows us to show the applications of the final sentential forms. We show 4 applications of the final sentential forms. We are only interested in the applications, which cannot be represented using the context-free grammars. The first application corresponds to the language $L_1 = \{ww \mid w \in \Sigma^*\}$, which is known non-context-free language. The second application constructs grammatically correct and truthful sentences in English. It corresponds to the formal language $L_2 = \{a^n b^n c^n \mid n \geq 0\}$. The third application constructs pairs of the binary numbers of the same size, such that the value of the first binary number is greater than the value of the second binary number. The last application is from the field of bioinformatics. It is a simplified representation of a secondary structure of an RNA—a pseudoknot.

The main result of this thesis is, that by using a single palindromial language, $\{w\#w^R \mid w \in \{0,1\}^*\}$, as a finalizing language, we can represent any recursively enumerable language. However, by using a regular language as a finalizing language, we do not increase the generative power of context-free grammars. These results are also interesting since palindromial languages and regular languages are both weak language families in comparison to the context-free languages. It is worth pointing out, that palindromial and regular language families do not have any common language.

As part of this thesis, we modified the algorithm CYK for syntax analysis of the final sentential forms and implemented it. This implementation serves the purpose of demonstration of the mentioned modification of the algorithm CYK on introduced applications. It is therefore not compared with any other software. We do not study the complexity of the proposed algorithm. As a future development of this topic, the complexity of the proposed algorithm could be studied. How much did the complexity increase in comparison to the original algorithm CYK?

We used the modification of the algorithm CYK for syntax analysis of the final sentential forms. Algorithm CYK uses context-free grammar in Chomsky normal form and its speed suffers from the size of the grammars in Chomsky normal form. Are we able to modify other algorithms for syntax analysis in order to achieve lower complexity of the resulting algorithm?

We put restrictions on the underlying context-free grammar of $L(G, F)$. Are we able to restrict it further without lowering its generative power? Can we restrict the number of the nonterminals that G contains? We could also study whether we can swap the underlying context-free grammar for other grammar.

Bibliography

[1] Jiří Techet, T. M. and Meduna, A. *Other Grammars*. 2007. Available at: http://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:tid:frvs:13-othergrampres.pdf.

[2] Kleijn, H. C. M. and Rozenberg, G. On the Generative Power of Regular Pattern Grammars. *Acta Informatica*. 1983, vol. 20, p. 391–411.

[3] Meduna, A. Generative Power of Three-Nonterminal Scattered Context Grammars. *Theoretical Computer Science*. 2000, vol. 2000, no. 246, p. 279–284. ISSN 0304-3975. Available at: https://www.fit.vut.cz/research/publication/6182.

[4] Meduna, A. *Automata and Languages: Theory and Applications [Springer, 2000]*. Springer Verlag, 2005. 892 p. ISBN 1-85233-074-0.

[5] Meduna, A. *Elements of Compiler Design*. Taylor & Francis Informa plc, 2008. 304 p. Taylor and Francis. ISBN 978-1-4200-6323-3. Available at: https://www.fit.vut.cz/research/publication/8538.

[6] Meduna, A. and Techet, J. *Scattered Context Grammars and their Applications*. WIT Press, 2010. 224 p. WIT Press, UK. ISBN 978-1-84564-426-0. Available at: https://www.fit.vut.cz/research/publication/8997.

[7] Rivas, E. and Eddy, S. The language of RNA: A formal grammar that includes pseudoknots. *Bioinformatics (Oxford, England)*. may 2000, vol. 16, p. 334–40. DOI: 10.1093/bioinformatics/16.4.334.

[8] Rozenberg, G. and Salomaa, A., ed. *Handbook of Formal Languages, Vol. 1: Word, Language, Grammar*. Berlin, Heidelberg: Springer-Verlag, 1997. ISBN 3540604200.

[9] Salomaa, A. *Formal Languages*. Academic Press, 1973. ACM monograph series. ISBN 9780126157505.