BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

ALTERNATIVE TRANSFORMATIONS OF GRAMMARS

ALTERNATIVNÍ GRAMATICKÉ TRANSFORMACE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MARTIN HAVEL**
AUTOR PRÁCE

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

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

146204

Institut:	Department of Information Systems (UIFS)
Student:	**Havel Martin, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Software Engineering
Title:	**Alternative Transformations of Grammars**
Category:	Theoretical Computer Science
Academic year:	2022/23

Assignment:

1. Study grammatical transformations in formal language theory. Consult this study with your supervisor.
2. Based upon instructions given by your supervisor, design alternative versions of these transformations.
3. Study the properties of transformations constructed in 2. Compare them with their original counterparts.
4. Implement selected grammatical transformations from 2. Consult this selection with your supervisor. Experiment with it to validate properties studied in 3.
5. Summarize the results. Discuss the future investigation concerning this project.

Literature:

- Meduna, A.: Automata and Languages, Springer, London, 2000, ISBN 978-1-4471-0501-5
- Rozenberg, G. and Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1 through 3, Springer, 1997, ISBN 3-540-60649-1

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

Supervisor:	**Meduna Alexandr, prof. RNDr., CSc.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	22.10.2022

Abstract

This thesis provides an alternative algorithm for the removal of erasing rules from E0S grammars. As opposed to the standard way of eliminating erasing rules in most E0S-like grammars such as context-free grammars, this method does not require predetermination of symbols that generate the empty string. The proposed algorithm is proved. In the application chapter of the thesis, the proposed algorithm is implemented, and the applicability of the algorithm to E0S grammars that work in a semi-parallel way is demonstrated. In the conclusion of the thesis, the algorithm is evaluated and two open problems are formulated.

Abstrakt

Tato práce poskytuje alternativní algoritmus pro odstranění epsilon-pravidel z E0S gramatik. Oproti standardnímu způsobu odstranění epsilon-pravidel ve většině gramatik podobných E0S gramatikám, jako bezkontextové gramatiky, tato metoda nevyžaduje zjištění neterminálů, které generují prázdné řetězce. Navržený algoritmus je dokázán. V implementační kapitole práce, navržený algoritmus je implementován a je demonstrována použitelnost algoritmu na E0S gramatikách. Na závěr je algoritmus vyhodnocen a jsou nastíněny dvě otevřené oblasti.

Keywords

Formal languages, context-free grammars, E0S grammars, elimination of erasing rules.

Klíčová slova

Formální jazyky, bezkontextové gramatiky, E0S gramatiky, eliminace epsilon-pravidel.

Reference

HAVEL, Martin. *Alternative Transformations of Grammars*. Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. RNDr. Alexander Meduna, CSc.

Rozšířený abstrakt

V dnešním moderním světě je využívání jazykových modelů v informatice stále více a více běžné. Tyto modely mají širokou škálu aplikací, jako například strojové překlady, textové zpracování a generování, překladače a mnoho dalších. Abychom však tyto modely mohli využít co nejefektivněji, je nezbytné studovat, jak je upravovat, optimalizovat a provádět převody mezi různými modely.

Alternativní transformace jazykových modelů jsou jednou z nejvíce inovativních oblastí, které se soustředí na vylepšení jazykových modelů. Studování těchto alternativních transformací je důležité, protože nám umožňuje dosáhnout větší efektivity a přesnosti v našich jazykových modelech.

Studium alternativních transformací jazykových modelů v IT může mít mnoho výhod. Za prvé, tato oblast umožňuje inovaci a vylepšení stávajících jazykových modelů, což může vést k větší přesnosti a rychlosti při práci s texty. Za druhé, studium těchto transformací může vést ke vzniku nových aplikací a technologií, které by mohly být využity v mnoha různých odvětvích IT.

Tématem této práce jsou alternativní transformace gramatik. Moderní jazykové modely jako například: E0L nebo E0S gramatiky jsou objektem nejnovějších výzkumů v oboru teorie formálních jazyků. Práce se primárně zaměřuje na E0S gramatiky, u kterých posouvá hranice dosud neobjeveného a nabízí novou transformaci E0S gramatik.

Hlavním účelem této práce je představit alternativní transformace gramatik. Tato práce poskytuje alternativní algoritmus pro odstranění epsilon-pravidel z E0S gramatik. Oproti standardnímu způsobu odstranění epsilon-pravidel ve většině gramatik podobných E0S gramatikám, jako bezkontextové gramatiky, tato metoda nevyžaduje předběžné zjištění symbolů, které generují prázdné řetězce.

Nejprve jsme zahrnuli všechny potřebné úvodní pojmy a definice. Počínaje základními pojmy jako jsou formální jazyky, množiny a postupně skrz pojmy a definice se práce posunuje k detailnějšímu zaměření, které blíže souvisejí s algoritmem. Jmenovitě E0S gramatiky a algoritmy pro eliminaci epsilon-pravidel. Práce zahrnuje teorii, která není nutná, ale stojí za zmínku pro přehled kontextu a okrajových souvislostí, které pomáhají porozumět potřebě studovat tento problém a širšímu vyhodnocení a přínosu v kontextu formálních jazyků.

Práce se věnuje důkladnému porozumění a studování problému. Detailní znalost v oblasti formálních modelů a jejich transformací umožňuje zhodnotit situaci a najít příležitosti k doplnění mezer v tématu alternativních transformací gramatik. Doplnění formálních modelů a jejich transformací je provedeno návrhem nového algoritmu pro eliminaci epsilon pravidel. Algoritmus se vyznačuje tím, že nepoužívá množinu neterminálů, které lze vymazat. Tato množina musí být předem stanovena, a proto zvyšuje dobu, kterou algoritmus pro eliminaci epsilon pravidel potřebuje k vytvoření gramatiky bez epsilon pravidel, která přijímá ekvivalentní jazyk jako vstupní gramatika s epsilon pravidly.

Aby měl algoritmus přínos k oblasti jazykových modelů a transformací, je nutné prokázat, že algoritmus pracuje správně a generuje novou gramatiku z vstupní gramatiky, která skutečně neobsahuje epsilon pravidla a také přijímá stejný jazyk jako vstupní gramatika. Práce formálně verifikuje, že nová E0S gramatika generovaná algoritmem bez epsilon pravidel přijímá stejný jazyk jako vstupní E0S gramatika formálním ověřením pomocí matematické indukce. Je zbytečné demonstrovat, že E0S gramatika vytvořená navrhovaným algoritmem je bez epsilon pravidel, protože tato vlastnost E0S gramatiky generované navrhovaným algoritmem je očividná.

Část této práce je implementace založená na navrženém algoritmu. Hlavním účelem implementace algoritmu je ukázat jeho praktické využití a umožnit snadný způsob demonstrace jeho fungování. Dalším důvodem implementace je generovat a vytvářet nové příklady výstupních E0S gramatik bez epsilon-pravidel ze vstupních E0S gramatik. Implementace algoritmu také slouží k demonstraci způsobu odstraňování epsilon pravidel. To nám umožňuje vypozorovat některé vlastnosti algoritmu, které již mohou být zřetelné při analýze pseudokódu algoritmu, ale při generování příkladů jsou triviálně viditelné a prakticky demonstrované.

Z algoritmu lze vypozorovat, že funguje správně v režimu sekvenční derivace a také správně funguje v režimu částečně-paralelní derivace. Práce algoritmu v režimu paralelní derivace nezaručuje, že algoritmus bude vracet E0S gramatiku takovou, že bude přijímat stejný jazyk jako vstupní E0S gramatika, proto algoritmus nelze použít v paralelní derivaci. Paralelní derivace je zajímavé téma, které by mohlo být dále studováno, a je podrobněji diskutováno v sekci o otevřených problémech. V sekci otevřených problémů je dále diskutována vlastnost algoritmu, která vyměňuje časovou složitost za prostorovou složitost.

Stojí za zmínku, že navržený algoritmus lze velmi snadno implementovat paralelně, protože všechny cykly procházejí pravidla a neterminály právě jednou. Tento fakt činí z algoritmu jednu z nejjednodušších úloh při rozdělování výpočtů mezi procesory v paralelním programování.

Nakonec bychom rádi zdůraznili, že součástí této práce je publikace [10], kde jsme zveřejnili hlavní výsledek a formální ověření, což poskytlo hodnocení od anonymního recenzenta, který přinesl některé hodnotné připomínky a zvýšil hodnotu této práce. Tato práce je uzavřena návrhem dvou otevřených problémů.

Alternative Transformations of Grammars

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. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Martin Havel
May 10, 2023

Acknowledgements

I wish to express my thanks to my supervisor prof. RNDr. Alexander Meduna, CSc. I am also grateful for the comments on the paper that has been published as part of this thesis made by Ing. Petr Zemek, Ph.D. and the anonymous referee.

Contents

Chapter 1

Introduction

In essence, the theory of formal languages is the basic foundation of any computer-based effort. The history of the theory of formal languages reaches as far as the first computing units, but it is being used in the same amount in modern times. Even if this problem has such deep roots in history, many unexplored areas and issues remain to be solved. We can also find many places to improve the solution of the problems that have been solved, or at least come up with alternative solutions that can be more efficient in optimal situations. We can find such alternatives even in the basic representation of formal languages, such as grammars or other equivalent systems.

In the theory of formal languages, E0S grammars represent, in essence, ordinary context-free grammars generalized so that they can rewrite both terminal and nonterminal symbols. Recall that these grammars underlie some important general frameworks of context-free rewriting systems, such as selective substitution grammars. The following chapter will discuss those context-free rewriting systems in more detail. The chapter is based on sources [12] and [13]).

The standard method of eliminating erasing rules from E0S grammars and their particular cases, such as context-free grammars, requires a predetermination of symbols that derive an empty string. For instance, see Subchapter 2.10, where there are algorithms included from internationally accepted publications such as [16] and [1]. Of course, the theory of formal languages would benefit from obtaining an alternative method that performs this elimination without any predetermination like this. This thesis attempts to achieve such a method. It will try to present and verify an alternative algorithm that performs the elimination of erasing rules from E0S grammars without this predetermination. In addition, it will be demonstrated that this algorithm is straightforwardly applicable to E0S grammars working in a semi-parallel way. It will also be demonstrated by implementation in the practical part of the thesis. If possible, we will discuss several other derivation modes at the end of the thesis.

The paper will be structured as follows. This chapter is the first of this thesis labeled Introduction 1. It should familiarize the reader with the researched problem discussed in this thesis and introduce the structural organization of the thesis for better orientation. At the end of this chapter, all crucial decisions, that have been made, will be mentioned at the beginning of the work so that any misunderstandings will be left out of the equation.

Next, it will be necessary to establish essential preliminaries and definitions to make the reader understand the main result of this thesis. It will start from basic definitions such as formal languages and sets, etc., and it will introduce every necessary terminology

to understand the main result of this thesis with gradually rising difficulty. Everything needed will be provided in Chapter 2. It should include a definition for everything needed.

We will also try to provide a description for better understanding and examples of the possibility of fitting the discussed part of the theory of formal languages. In some parts, topics not implicitly connected to the main result will be mentioned. Still, they are essential for seeing a wider perspective, or they will be mentioned in the Chapter 5 for the following research. If possible, every preliminary and definition will include citations to the source of introduction from a worldwide accepted mention or the first source, where the discussed topic was introduced. These citations will also allow the reader to learn more about the part if the preliminaries and definitions included in this chapter will not be satisfactory for the reader.

Then, Chapter 3 will try to present the main result of the thesis. It will describe an alternative algorithm for the elimination of erasing rules from E0S grammars. Of course, formal verification of an alternative algorithm must be included; otherwise, it would devalue all the work done in this thesis. As said, this work will focus on the algorithm for eliminating the erasing rules of E0S grammar without pre-determining symbols that derive an empty string.

The next Chapter 4 of the thesis will implement the algorithm of the preceding Chapter 3 and allow the reader to try, or at least showcase, the practical usage of the algorithm to eliminate erasing rules from E0S grammars. The chapter will include detailed information about implementation and how to run the implementation on any device. It should help the reader to understand the algorithm and show the benefits or downsides of the alternative algorithm. Furthermore, it will generate examples, which will also be shown in Chapter 4, and provide helpful information for further evaluation.

The final Chapter 5 should evaluate the algorithm offered in the main result and the advantages and disadvantages compared to other algorithms, which require predetermination of symbols that derive an empty string. Then we will foreshadow our algorithm's applicability to different derivation modes used in E0S grammars. Lastly, we will try to formulate open problem areas, where there could be the possibility of advanced research from this thesis.

After consultation with the supervisor, we decided that the algorithm in the main result of the Chapter 3 is significant enough to be worth publishing in an international science journal as part of the work on this thesis. With the supervisor, we chose the computer science journal called Computer Science Journal of Moldova (CSJM), and we wrote a paper based on the work in this thesis. After review by the technical editor and an anonymous referee of this paper, it was accepted and can be found in the issue published in 2022, which the reader can see here [10].

Chapter 2

Preliminaries and Definitions

This chapter will define and introduce the necessary terms to understand the main result of this thesis. Firstly, we will introduce the basics and explain everything with gradually increasing difficulty to fully understand the problem and the solution offered in this thesis. The following sections explain mathematical terms (such as set) in detail. Then we cover everything necessary, from theoretical informatics and formal language models to include mainstream models, which are not required but bring a broader perspective and make them worthy of mentioning. We will begin with languages, their categorizations, and models. In the next step, the thesis will focus on grammars, and there will be special sections for E0S grammars and for tightly connected models, which are crucial to this work. Lastly, we will explain algorithms for erasing rules, because understanding their limits is the core of this thesis and is obligatory for creating alternative transformations. Most of the sections of this chapter will include references which can be helpful.

2.1 Formal language and regular set

The preliminary introduction to the formal language is knowledge of *alphabet*, *string*, and *set*.

Let *set* be any collection of elements. A set is represented throughout this thesis as a list of all its members enclosed in braces and will be explained in detail later in this chapter. For example:

$$S = \{a, b, cd, 0\}$$

An alphabet is a nonempty set of elements called *symbols*. In some cases, we can even use infinite alphabets, but in the following text, it is unnecessary, so we will use only finite alphabets. For convenience, we will denote the alphabet as Σ if we can. For example:

$$\Sigma = \{a, b, c\}$$

String

A string (sometimes referred to as a word) is any finite sequence of symbols. A string over a given alphabet is any finite sequence of alphabet symbols. An empty string will correspond to an empty sequence of symbols. This string will be denoted as ε. Let Σ be an alphabet. Next, we will formally define the strings over Σ as:

1. The empty string ε is a string over Σ.

2. If w is a string over Σ and $a \in \Sigma$, then wa is a string over Σ

3. Let w be a string over Σ if and only if it can be created using Rules 1 and 2.

Furthermore, we will need the length of the string. Let w be a string over the alphabet Σ. The length of the string w, denoted as $|w|$, is defined as:

1. If $w = \varepsilon$, then $|w| = 0$.

2. If $w = a_1...a_n$, then $|w| = n$, for $n \geq 1$ and $a_i \in \Sigma$ for all $i = 1, ..., n$.

For example we can introduce language $L = \{a, b, c\}$ and 2 strings $w_1 = aabc$ and $w_2 = 0aac$, where $w_1 \in L$ and $w_2 \notin L$ and $|w_1| = 4$ and $|w_2| = 4$.
We will be using the following notation:

- Let w and x be strings of the alphabet Σ. The concatenation of w and x is the string wx.

- Let w be a string of the alphabet Σ. For $i \geq 0$, the string power i of w, w^i is defined:

  1. $w^0 = \varepsilon$
  2. for $i \geq 1$, where $w^i = ww^{i-1}$

- Let w be a string of the alphabet Σ. The reverse of string w, w^R, is defined:

  1. if $w = \varepsilon$, then $w^R = \varepsilon$
  2. if $w = a_1...a_n$, then $w^R = a_n...a_1$, for $n \geq 1$ and $a_i \in \Sigma$ for $i = 1, ..., n$

- Let w and x be strings of the alphabet Σ; w is the prefix x, if there exists a string y of the alphabet Σ, while valid $wy = x$.

- Let w and x be strings of the alphabet Σ; w is the suffix x, if there exists a string y of the alphabet Σ, while valid $yw = x$.

- Let w and x be strings of the alphabet Σ. w is substring x, if there exist strings y and z of the alphabet Σ, while valid $ywz = x$.

Regular set

Mathematicians utilize sets, a fundamental notion, to group together components, or members, that share a characteristic. Sets can be expressed in various ways, most commonly by listing the elements inside curly brackets, and for purposes of this thesis, we strictly use curly brackets. Sets are defined by their properties and elements. In various fields of mathematics, such as algebra, geometry, topology and computer science, sets are commonly utilized to denote distinct entities.

Sets offer a variety of advantageous characteristic usage, including the ability to describe other mathematical objects like relations and functions or computer language models such as alphabets, nonterminals, etc. We also utilize the property of being closed under some operations. For example, union and intersection. Also, they are used to define many kinds of numbers, including real and natural.

The notion of set membership, which decides if an element belongs to a set or not, is one of the most crucial things to determine in sets, and it is essential for this thesis. Several

set operations are defined, like union, intersection, and difference. These are there for sets to merge and change in various ways, based on the idea of membership, and their math notation is defined.

Let Σ be a finite alphabet. The class of regular sets is the smallest class of languages that contains the sets $\emptyset$, $\{\varepsilon\}$, $\{a\}$ for all symbols a. More precisely, we can define regular sets (similarly defined in [22]) over the alphabet Σ as follows:

- $\emptyset$ (the empty set) is a regular set over Σ
- $\{\varepsilon\}$ (the set consisting only of the empty string) is a regular set over Σ
- $\{a\}$ for all $a \in \Sigma$ is a regular set over Σ

The regular set is closed under the following operations:

- $P \cup Q$ (union), where P and Q are regular sets over Σ
- $P \cap Q$ (intersection), where P and Q are regular sets over Σ
- P^* (iteration), where P is a regular set over Σ
- $co - P$ (complement), where P is a regular set over Σ

Regular sets are only the sets that arise from the application of the definitions and operations. A more in-depth look can be found in Chapter 0 in [1]. In this thesis, sometimes denoting an empty set $\emptyset$ as $\{\}$ for implementation purposes will be mandatory.

Formal language

In many applications, formal languages serve as a universal tool for encoding data and object descriptions. Their universality brings many advantages that computer programs can use for representation and processing. The algebraic character of formal language operations allows us to use an algebraic basis of the operations to generate complicated solutions by using the decomposition principle, a formal description of the decomposed parts, and a composition of the final solution from those parts. Often used methods for splitting application problems into groups, and a mechanism to describe the modeling and decision-making capabilities of a particular class of various formal models, is the Chomsky hierarchy of grammars and formal languages, described in detail later.

Furthermore, formal languages are divided into various groups according to the kind of grammar they use. They include context-free, context-sensitive, and regular languages, where grammars correspond with the Chomsky hierarchy. The complexity of the grammar is used to define the language. The kinds of operations that can be carried out on it are the basis for these classifications, which decide the computation power of grammar. For instance, context-free languages have more complex grammar. Pushdown automata can model them but cannot be modeled by finite automata.

In addition, we can use formal languages for a representation of a variety of systems and procedures, such as computer programming, the definition of hardware and software, and modeling in several different fields, including biology, physics, and finance, where biology is a very modern and exciting field for formal languages. More on this connection can be found in [17] and the section about L-systems. Understanding how a formal language can be used to simulate the system or process of interest and how to reason about its attributes and behavior requires a solid grasp of the language's grammar.

Definition

Let Σ^* be a set of all strings in Σ. Every subset $L \subseteq \Sigma^*$ is a language over Σ.

We introduce these operations:

- Let L_1 and L_2 be languages of the alphabet Σ. The union of languages L_1 and L_2, $L_1 \cup L_2$, is defined as: $L_1 \cup L_2 = \{x : x \in L_1 \vee x \in L_2\}$.
- Let L_1 and L_2 be languages of the alphabet Σ. The intersection of languages L_1 and L_2, $L_1 \cap L_2$, is defined as: $L_1 \cap L_2 = \{x : x \in L_1 \wedge x \in L_2\}$.
- Let L_1 and L_2 be languages of the alphabet Σ. The difference of languages L_1 and L_2, $L_1 - L_2$, is defined as: $L_1 - L_2 = \{x : x \in L_1 \wedge x \notin L_2\}$.
- Let L_1 and L_2 be languages of the alphabet Σ. The concatenation of the languages L_1 and L_2, L_1L_2, is defined as: $L_1L_2 = \{xy : x \in L_1 \wedge y \in L_2\}$.
- Let L be a language of the alphabet Σ. The complement of language L, $co - L$, is defined as: $co - L = \Sigma^* - L$.
- Let L be a language of the alphabet Σ. The reverse of language L, L^R, is defined as: $L^R = \{w^R : w \in L\}$.
- Let L be language of the alphabet Σ. The power of language L, L^i, is defined as:
  1. $L^0 = \{\varepsilon\}$
  2. for $i \geq 1 : L^i = LL^{i-1}$
- Let L be a language of the alphabet Σ. The iteration of language L, L^*, is defined as:
$$L^* = \bigcup_{i=0}^{\infty} L^i$$
- Let L be a language of the alphabet Σ. The positive iteration of language L, L^+, is defined as:
$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

2.2 Language models

Our definition of a language L is a fixed length string set over a finite alphabet Σ. This first important question is how to represent the L when the L is infinite. In the circumstance that L contains a restricted quantity of character sequences, a simple method is to enumerate all such strings in L. Nevertheless, imposing restrictions on the maximum length of a string in numerous languages is implausible and even unappealing. It is reasonable to consider languages that contain many strings.

Languages of this type, of course, cannot be defined by an exhaustive listing of strings belonging to language, and it is necessary to find some other representation. We constantly want our specification of a language to be finite, even if the specified language cannot be finite. Several methods provide this requirement. One way is to use a generative system called grammar. Each and every sentence found within a given language can be formed by

utilizing grammar regulations that are clearly defined. One of the advantages of defining a language using grammar is that parsing and translations are often made easier due to the structure that is prescribed to a language sentence by its grammar. In detail, we shall introduce types of grammar, particularly context-free ones, and E0S ones, because they are needed for the main result. But for simple cases, we use regular expressions.

Regular expression

Regular expressions are practical and simple-to-use tools for identifying patterns in text strings. They find usage in many informatics technology fields, like text processing, search and replace operations, and input validation. They are a condensed and adaptable method of specifying a list of valid strings. The most intriguing application for this theory is another usage, which is to explain formal language, and it is essential for this thesis. We will provide a formal definition and example. Still, we will not go deep into this topic since it is expected that reader is already familiar with such a fundamental construct. More about regular expressions in Chapter 1 in [23] or in [11] with definitions that follow.

Definition

Let Σ be an alphabet. Regular expressions over the alphabet Σ and the languages which they represent are defined as:

- $\emptyset$ is a regular expression denoting the regular set $\emptyset$
- ε is a regular expression denoting the regular set $\{\varepsilon\}$
- a is a regular expression denoting the regular set $\{a\}$ for all $a \in \Sigma$

Operations

Let r a s be regular expressions denoting the languages L_r a L_s, then we define operations as:

- $(r.s)$ is a regular expression denoting language $L = L_r L_s$
- $(r + s)$ is a regular expression denoting language $L = L_r \cup L_s$
- (r^*) is a regular expression denoting language $L = L_r^*$

Example

For example following languages are all over the alphabet $\Sigma = \{a, b\}$. We can have the following regular expressions describing languages:

- $L_{e1} = \{\varepsilon\}$
- $L_{e2} = \{ab, ba, aa, bb\}$
- $L_{e3} = \{a^p | p \text{ is prime number}\}$
- $L_{e4} = \{a^n b^n | n \geq 0\}$

2.3 Grammar

For words that belong to a given language over a grammar to be considered valid in formal languages, a set of grammar rules must be followed. A collection of grammar rules is the most complicated part and typically has the most significant role in establishing formal language grammar. These rules lead to how various grammar components like nonterminals and terminals can be joined to create strings belonging to the language described by a grammar. Compilers and interpreters employ these rules to examine the syntax of language-written statements and serve as a practical example of using formal models as grammar.

Syntax and semantics are the two main components of using formal models such as a grammar in compilers. A language's syntax purpose is to decide how words, symbols, and phrases can be combined to produce correct sentences as well as the structure of the sentences themselves. A language's semantics determines the meaning of sentences and how they apply to the logic.

Definition

A general grammar is a quadruple (same as in [23]):

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

where:

- N is a finite alphabet of nonterminals
- T is a finite alphabet of terminals, $N \cap T = \emptyset$
- P is a finite set of rules of the form:

$$\alpha \to \beta, \text{ where } \alpha \in (N \cup T)^* N (N \cup T)^*,\ \beta \in (N \cup T)^*$$

- $S \in N$ is a start symbol

In some publications, for example, in [16], there is used notation for a left side of the rule and a right side of the rule $p \in P$ of the form $\alpha \to \beta$, as follows:

- $lhs(p) = \alpha$ - means the left-hand side of a rule
- $rhs(p) = \beta$ - the right-hand side of a rule

Derivation steps

In a grammar and other formal models, a derivation step is a relation between two sentential forms that use a rule to create a sentential form of nonterminals and terminals from another. Formal grammar rules are made of terminals and nonterminals on the rhs() and nonterminals on the lhs(), but this does not have to be true in some modern language models.

A parse tree, which represents the derivation process that can look like a tree if graphically represented appropriately, can be used to represent the derivation steps, which can help evaluate a grammar. The children of each node in the parse tree correspond to the child string in the rule, whereas the node itself represents a nonterminal symbol. The leaves

of the parse tree represent the terminal symbols in the grammar. As an illustration, see Figure fig:dertree.

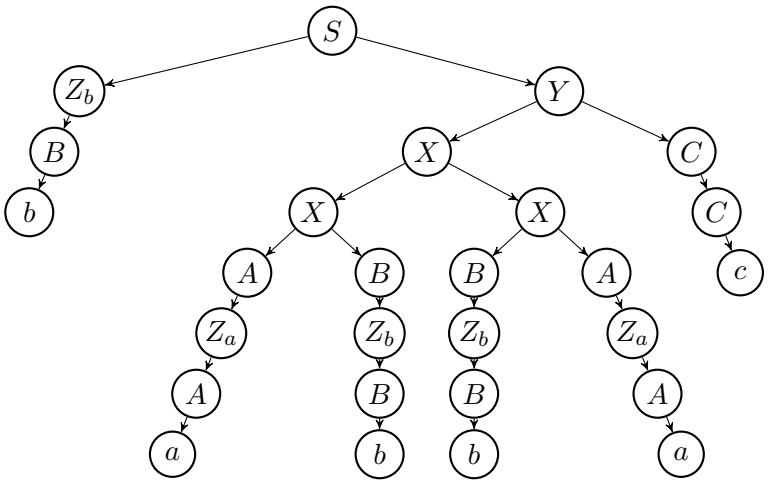

Figure 2.1: The illustration example of a derivation tree

A parse tree can be constructed in two ways: top-down (also known as leftmost derivation) and bottom-up (also known as rightmost derivation). In top-down, the parse tree is constructed by expanding the start symbol using grammar rules and working towards the tree's leaves. In the bottom-up, the parse tree is constructed by starting with the leaves of the derivation tree and working towards the start symbol.

Definition

Let $G = (N, T, P, S)$ be grammar, and let λ, μ be a string from $(N \cup T)^*$. Between λ, μ there is the binary relation $\Rightarrow_G$, called the derivation step, if we can formulate strings λ and μ as:

$$\lambda = \gamma\alpha\delta,$$
$$\mu = \gamma\beta\delta,$$

where $\lambda, \mu \in (N \cup T)^*$ and $\alpha \to \beta$ is the rule of P. Then we write:

$$\lambda \Rightarrow_G \mu \text{ or}$$
$$\lambda \Rightarrow \mu$$

We can split the derivation steps into three categories:

Sequential

A sequential derivation step is a process of applying rules of a grammar to a sentential form of terminals or nonterminals for the purpose of generating a new sentential form in sequential order. In a sequential derivation, a sentential form of terminals and nonterminals is rewritten with only one rule at a time. Choosing a rule to apply is selected by the set of grammar rules and the actual sentential form being rewritten. The process starts with the grammar's start symbol, which is always rewritten first because it is the starting symbol, and according to rules and sentential forms that replace it, until the sentential form only contains terminal symbols, then the generation process is over.

Sequential derivation can also be specified as the leftmost derivation or leftmost derivation step, and it always applies a rule to the leftmost nonterminal symbol of the sentential form. It is defined as a sequence of rules applied to the start symbol. The methodology employed facilitates the generation of an exclusive parse tree that represents the sentence's structure.

We note a sequential step as follows:

$$\Rightarrow_G \text{ or } {}_{\text{seq}}\Rightarrow_G, \text{where it denotes sequential step with grammar } G$$

Semi-parallel

A semi-parallel derivation step is applying rules simultaneously to multiple nonterminals within a sentential form of terminals or nonterminals to generate a new sentential form of terminals or nonterminals.

In a semi-parallel derivation, multiple nonterminals in a sentential form are rewritten simultaneously. The process starts with the start symbol of the grammar, which is again rewritten first based on available rules in grammar. Then it follows rewriting until only terminals are left, which ends the derivation process. The order of the nonterminals being rewritten is also again determined by the grammar's rules and the given sentential form being rewritten.

When undertaking a comparison of the semi-parallel derivation process to that of sequential derivation, it becomes evident that this one allows multiple nonterminals to be replaced in unison, but from the previous section, we know that in sequential derivation, only one nonterminal can be replaced in one step.

The semi-parallel derivation is less common than the other two types of derivation mentioned in this thesis. Furthermore, it is not widely used in practice because the advantages of semi-parallel derivation are very niche. This kind of derivation finds its purpose mainly in the research field. Its usage is also limited to specific types of grammars and particular types of parsing algorithms.

We note semi-parallel steps as follows:

$${}_{\text{s-par}}\Rightarrow_G, \text{where it denotes semi-parallel step with grammar } G$$

Parallel

A parallel derivation step is a process of applying rules simultaneously to all nonterminals within a sentential form of terminals and nonterminals in order to generate a new sentential form.

In a parallel derivation, all nonterminals in a current sentential form are rewritten simultaneously. The process starts as always with the start symbol of the grammar, which is rewritten according to the grammar rule available that replaces it. And derivation ends when the sentential form only contains only terminal symbols. The order of the nonterminals being rewritten is determined by the grammar's rules and the current sentential form being rewritten. It is also worth noting that if one of the nonterminals in a current sentential form cannot be rewritten, the derivation process ends unsuccessfully.

Comparing parallel derivation with sequential derivation, we can also spot differences because it forces for simultaneous replacement of all the nonterminals in the sentential form. In contrast, only one nonterminal can be replaced simultaneously in sequential derivation. It also differs from semi-parallel derivation, which is a middle between those two.

Parallel derivation is less common than sequential derivation. It is mainly used in the research field. Its usage is limited to specific types of grammars and a particular type of parsing algorithms, which can be found in the biology of the future, such as E0L grammars, which are mentioned later.

We note parallel steps as follows:

$$_{\text{par}}\Rightarrow_G, \text{where it denotes parallel step with grammar } G$$

2.4 The Chomsky Hierarchy

As already mentioned, it is mandatory to classify formal languages. We are able to classify languages on many different attributes. Dependent on grammar's restrictions and complexity, languages are placed in the formal language hierarchy. Noam Chomsky defined this hierarchy in 1956. To this day, it is called the Chomsky hierarchy. The Chomsky hierarchy consists of 4 levels: Type-3, Type-2, Type-1, and Type-0. With higher numbers, language is more restrictive. The graphical representation is included as Figure 2.2 for better indication.

Although his concept is pretty old, there is renewed interest because of its relevance to natural language processing. Chomsky hierarchy helps us answer questions like: Can a natural language like English be described, parsed, and compiled with the same methods as used for formal languages in computer science?

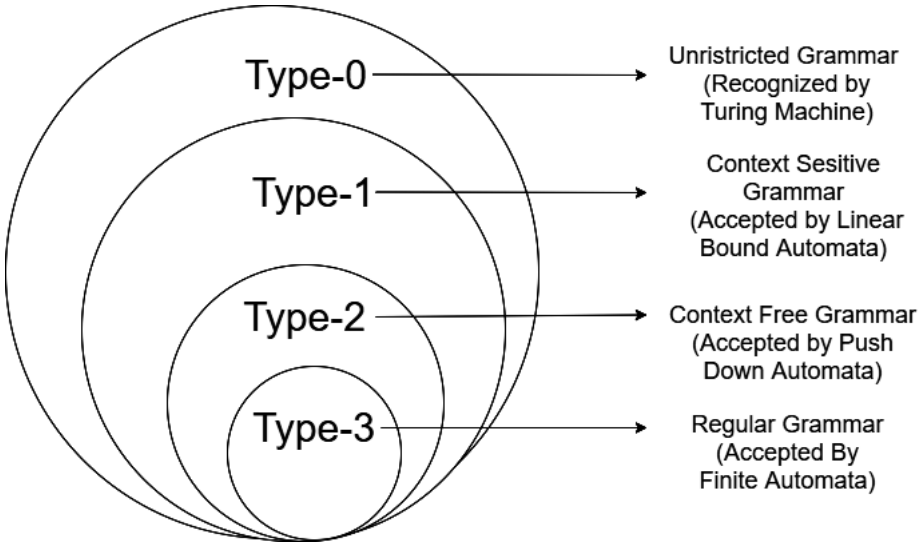

Figure 2.2: Graph showing the Chomsky hierarchy of language families from [5]

Now, we follow the necessary definition of language models for every level of the Chomsky hierarchy.

2.4.1 Type-3 language models

Regular grammar (Type-3 grammars)

A regular grammar (sometimes also called a type-3 grammar) is the first of the language models introduced in the Chomsky hierarchy. It generates and analyzes regular languages.

They are considered the least complex in the Chomsky hierarchy. As mentioned before regular languages are a subset of formal languages that are generated by regular grammars, but there exist, other models, such as finite automata, which are simple machines that read input symbols and move between states to form a starting state toward one of the final states, where movement is decided by a set of rules of a given finite automaton, but more about them later.

One of the essential properties of a regular grammar is its so-called regularity. Regularity means that the rules for replacing nonterminals in a string are based on only regular rules in regular grammar. For example, we can consider a regular grammar that all nonterminal symbol replacement rules to be based only on the preceding operator and the present symbol. This so-called regularity is more evident in a definition.

Because regular grammars are predictable and very simple, it makes parsing quick and easy. The parser needs to look up the current symbol and it is going to use one regular rule, which compared to other models does not require taking context into account. Regular languages are also closed under most operations like union, concatenation, and iteration, which is less occurring in complicated models.

Definition

A regular grammar is a quadruple (from [23]):

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

where:

- N is a finite alphabet of nonterminals
- T is a finite alphabet of terminals, $N \cap T = \emptyset$
- P is a finite set of rules in form:

$$\begin{gathered} A \to xB \quad \text{or} \\ A \to x|\varepsilon, \text{ where } A, B \in N,\ x \in T \end{gathered}$$

  or respectively:

$$\begin{gathered} A \to Bx \quad \text{or} \\ A \to x|\varepsilon, \text{ where } A, B \in N,\ x \in T \end{gathered}$$

  is called the right-regular, resp. the left-regular grammar.

- $S \in N$ is a start symbol

Note that regular grammar $G = (N, T, P, S)$ with rules of form $A \to xBy$ and $A \to x$, where $A, B \in N, x \in T$.

We also can denote languages:

- L_{RR} - language generated by a right-regular grammar
- L_{LR} - language generated by a left-regular grammar
- L_L - language generated by a linear grammar

Note that $L_{RR} = L_{LR} \subset L_L$

Example

We show regular grammar on example G_{REG} describing language $L_3 = \{a^*b^*\}$ over $\Sigma = \{a, b\}$.

$G_{REG} = (N, T, P, S)$

- $N = \{S\}$
- $T = \{a, b\}$
- P as follows:
  1. $S \to aS$
  2. $S \to D$
  3. $D \to bD$
  4. $S \to \varepsilon$
  5. $D \to \varepsilon$

Derivation

Now we provide an example of generation of $w = aaaabbb$, where $w \in L_3$

$$
\begin{array}{ll}
S & \\
\Rightarrow aS & [1] \\
\Rightarrow aaS & [1] \\
\Rightarrow aaaS & [1] \\
\Rightarrow aaaaS & [1] \\
\Rightarrow aaaaD & [2] \\
\Rightarrow aaaabD & [3] \\
\Rightarrow aaaabbD & [3] \\
\Rightarrow aaaabbbD & [3] \\
\Rightarrow aaaabbb & [5]
\end{array}
$$

Finite automata

The simplest way to introduce finite automata is as a fundamental concept in the field of theoretical computer science. They have a wide range of applications in various areas, such as always mentioned compilers, but also network protocol design and pattern matching in text. The automata are able and therefore used to recognize patterns within the input and can also be used to determine whether a given input is a valid word in the input language.

When an automaton starts reading an input, the automaton begins from the start state and reads each symbol one at a time, or it can read none if it is non-deterministic. An automaton utilizes a set of rules to determine its next state based on the current input symbol and its present state. If it is not any of the rules from the set available, the automaton rejects the input. Otherwise, the automaton is considered to have accepted the input if it enters the accepting (final) state, which there can be several of those in an automaton. As mentioned above, finite automata can be classified into two types:

- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)

In a DFA, there is only one possible next state for any given input symbol and current state. In an NFA, there may be multiple possible next states for a given input symbol and current state, as the decision to choose a state is made nondeterministicaly. Finite automata can also be expanded with some addition like a stack or algorithms to solve more complex problems. For example, adding a stack is called a pushdown automaton and allows it to recognize context-free languages but more about them in a type-2 language model.

Definition finite automata

A finite automaton is a quintuple (a similar definition can be found in [11] or [22]):

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

where:

- Q is a finite nonempty set of states
- Σ is the finite input alphabet
- R is a finite set of transition rules denoted as: $pa \to q$, where $p, q \in Q, a \in \Sigma \cup \{\varepsilon\}$
- $s \in Q$ is a starting state
- $F \subseteq Q$ is a set of accepting states

Definition configuration

$M = (Q, \Sigma, R, s, F)$ is a finite automaton. We denote by a configuration of the automaton M a couple $C = (q, w)$, where $q \in Q$ and $w \in \Sigma^*$ The configuration (s, w) is the initial configuration, a configuration (f, ε), where $f \in F$ is the final configuration.

Definition transition

A transition of automaton M is represented by binary relation $\Rightarrow$ on the set of configurations C. Let the set $\lambda(q, a)$ contains the state q' (realize $\lambda(q, a) = Q_j$, where$Q_j \in 2^Q, q' \in Q \in Q_j$), then $(q, aw) \Rightarrow M(q', w)$ for all $w \in \Sigma^*$. Denote by the symbol $\Rightarrow^k$, where $k \geq 0$ the k-power ($C \Rightarrow^0 C'$ if and only if $C = C'$), by the symbol $\Rightarrow^+$ the transitive closure and by the symbol $\Rightarrow^*$ the transitive and reflexive closure of relation.

Definition accepting language

The input string w is accepted by the finite automaton M, if and only if $(s, w) \Rightarrow^* (q, \varepsilon)$, where $q \in F$. We denote by $L(M$ the language accepted by finite automaton M. The language $L(M)$ is a set of all strings accepted by the finite automaton M: $L(M) = \{w|(s, w) \Rightarrow^* (q, \varepsilon)$ and $q \in F\}$

Theorem 1. *Let L_3 be a language of type 3. Then there exists a finite automaton M, for which holds $L = L(M)$, it means $L_3 \subseteq L_M$*

Proof. Proof can be found in [16]. □

Theorem 2.

Example

We show finite automata on example M_1 describing language $L_{FA} = \{(a|b)^*\}$ over $\Sigma = \{a, b\}$.

$M_1 = (Q, \Sigma, R, s, F)$

- $Q = \{s_1, s_2, s_3, s_4, f_1, f_2, f_3, f_4\}$
- $\Sigma = \{a, b\}$
- R as follows:
  1. $s_4 \to s_3$
  2. $s_4 \to f_4$
  3. $s_3 \to s_1$
  4. $s_3 \to s_2$
  5. $s_1 a \to f_1$
  6. $s_2 b \to f_2$
  7. $f_1 \to f_3$
  8. $f_2 \to f_3$
  9. $f_3 \to s_3$
  10. $f_3 \to f_4$
- $s = s_4$
- $F = \{f_4\}$

Graphical representation

A finite automaton's behavior and structure can be seen through a graphic representation for a better overview and a more natural way of visualizing. A finite automaton is often represented graphically by a collection of states, usually as circles, bubbles, or squares. The set of rules is represented as directed edges between the states. The symbol or input that causes the transition is written close next to the edges. The automaton also has one or more designated accept or final states, most of the time marked with a double lining and a selected start state, marked as a directed edge leading from out of an automaton.

As mentioned before, a graphical representation is a beneficial tool for comprehending and creating automata. It makes it simple to see the behavior and structure of the automaton and their properties, such as determinism or unreachable states that humans cannot see so easily in nongraphical representation. Furthermore, it aids in developing automata-related algorithms and theorems, such as those for building automata, detecting if an automaton would take a given string of input, and many more. It is widely utilized in the fields of formal languages and computing theory when more human representation is needed. We showcase it in figure 2.3.

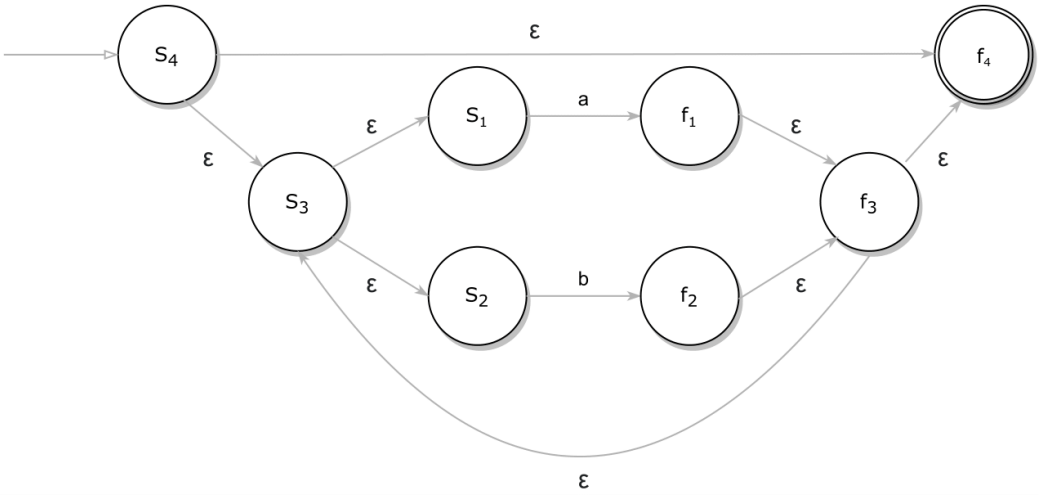

Figure 2.3: Finite automata M_1 representing the language described with the regular expression $(a|b)^*$ from [9]

Example acceptation

Now we provide an example of acceptance of $w = abb$, where $w \in L_{FA}$

$$
\begin{array}{ll}
 & s_4abba \\
\Rightarrow & s_3abb \quad [1] \\
\Rightarrow & s_1abb \quad [3] \\
\Rightarrow & f_1bb \quad [5] \\
\Rightarrow & f_3bb \quad [7] \\
\Rightarrow & s_3bb \quad [9] \\
\Rightarrow & s_2bb \quad [4] \\
\Rightarrow & f_2b \quad [6] \\
\Rightarrow & f_3b \quad [8] \\
\Rightarrow & s_3b \quad [9] \\
\Rightarrow & s_2b \quad [4] \\
\Rightarrow & f_2 \quad [6] \\
\Rightarrow & f_3 \quad [8] \\
\Rightarrow & f_4 \quad [10]
\end{array}
$$

Practical Example

As mentioned before, finite automata have many practical usages, which we would like to demonstrate in figure 2.4.

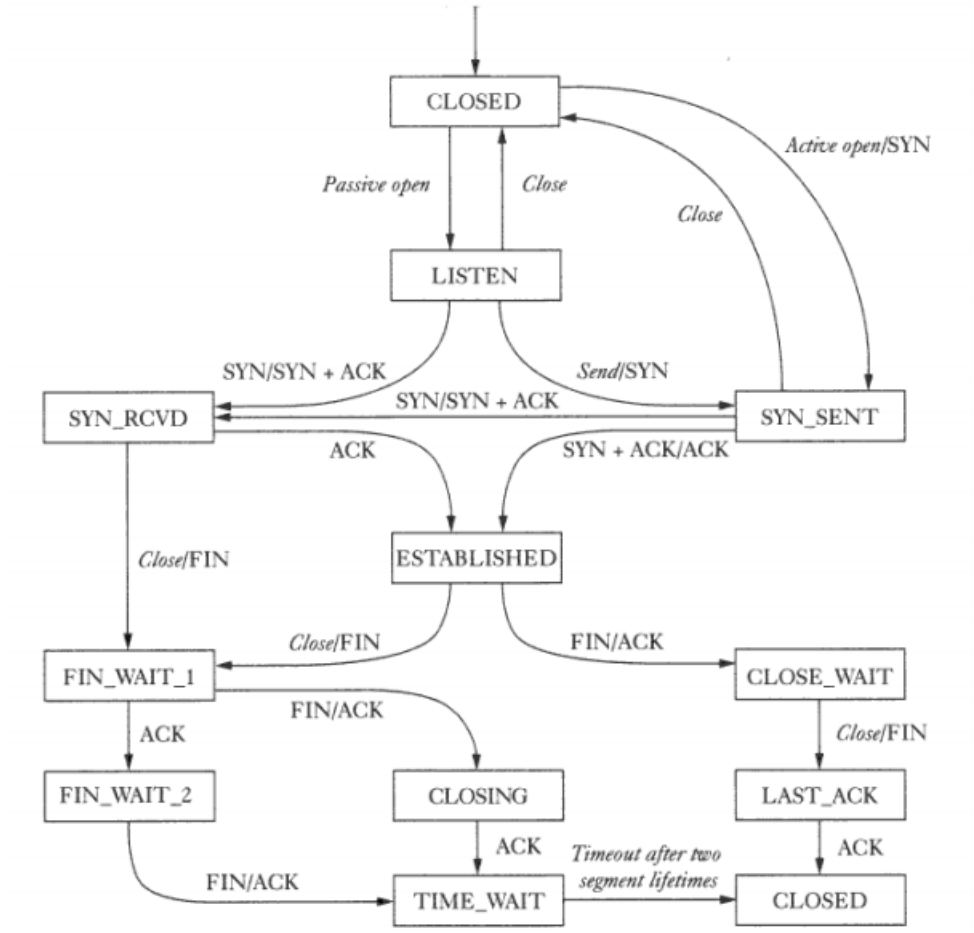

Figure 2.4: The illustration picture of a finite automaton describing the TCP connection from [28].

2.4.2 Type-2 language models

Context-free grammar (Type-2 grammars)

Context-freeness, or the fact that the rules for replacing symbols in a string are independent of the context in which the symbols appear, is one of the critical characteristics of Context-free grammar (CFG). In other words, the rules of a CFG are independent of the symbols surrounding the nonterminal symbol that is being replaced.

We define context-freeness to make parsing easier and more effective because the parser just needs to focus on the present symbol and its associated rule rather than considering its context. Another important property worth noting is that a context-free grammar can generate an infinite number of words, which makes context-free grammar proficient for programming languages and natural language processing. It is of great interest to recognize context-freeness.

Definition

A context-free grammar is a quadruple (from [23] or [11]):

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

where:

- N is a finite alphabet of nonterminals
- T is a finite alphabet of terminals, $N \cap T = \emptyset$
- P is a finite set of rules of the form:

$$A \to \alpha, \text{ where } A \in N,\ \alpha \in (N \cup T)^*$$

- $S \in N$ is a start symbol

Alternative Definition

We also show this alternative definition, which makes it easier to compare it with an E0S grammar and will be only used for this purpose. A context-free grammar is also a quadruple:

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

where:

- V is a finite alphabet
- T is a finite alphabet of terminals, $T \subset V$
- P is a finite set of rules of the form:

$$A \to \alpha, \text{ where } A \in V \setminus T,\ \alpha \in V^*$$

- $S \in N$ is a start symbol, $S \in V \setminus T$

Example

We show a context-free grammar on example G_{CFG} describing language: $L_2 = \{a^n(b + c)^n | n \geq 0\}$ over $\Sigma = \{a, b, c\}$.

$G_{CFG} = (N, T, P, S)$

- $N = \{S, D\}$
- $T = \{a, b, c\}$
- P as follows:
  1. $S \to aSD$
  2. $D \to b$
  3. $D \to c$
  4. $S \to \varepsilon$

Derivation

We provide an example of a generation of $w = aaaaccbc$, where $w \in L_2$

$$
\begin{aligned}
& S \\
\Rightarrow\ & aSD && [1] \\
\Rightarrow\ & aSc && [3] \\
\Rightarrow\ & aaSDc && [1] \\
\Rightarrow\ & aaSbc && [2] \\
\Rightarrow\ & aaaSDbc && [1] \\
\Rightarrow\ & aaaaScbc && [3] \\
\Rightarrow\ & aaaaSDcbc && [1] \\
\Rightarrow\ & aaaaSccbc && [3] \\
\Rightarrow\ & aaaaccbc && [4]
\end{aligned}
$$

Pushdown automata

A pushdown automaton (PDA) is a type of automaton that extends a finite automaton by expanding the basic notation with a stack. The stack is the main reason it is able to recognize context-free languages. For a simple introduction of a PDA, it consists of a finite set of states, a finite set of input symbols called an input alphabet, a set of rules, a start state, accepting states, which are identical to the finite automata, and a stack which is represented by a starting symbol on a stack and a stack alphabet. The stack is used to store stack symbols from a stack alphabet, and the automaton reads and writes symbols to the stack during transitioning from the starting state to one of the final states.

The automaton gets from the starting state to the final state by reading input symbols one at a time and switches between states based on the input symbol read from the input, the state it is currently in, and the symbol at the top of the stack. Furthermore, instructions to push and pop symbols from the stack may be included in the transition function, or we push or pop multiple symbols in a single transition step, and then we call it an extended pushdown automaton. The input is accepted if the automaton enters the accepting states and the stack empties.

Next to extended pushdown automata, we can recognize two types of PDA similar to FA:

- deterministic pushdown automata (DPDA)
- non-deterministic pushdown automata (PDA)

As you can see, we do not use the special shortcut for non-deterministic PDA for one important reason. We are unable to recognize all languages belonging to the family of context-free languages. Therefore non-deterministic PDA and a DPDA are not equivalent, which makes it unnecessary to refer to a non-deterministic PDA as a non-deterministic. Pushdown automata similar to FA have found an application in many fields, such as compilers, parsing, and formal language theory. They are particularly useful for recognizing nested structures and can be used to recognize languages such as programming languages, markup languages, and natural languages.

Definition

A pushdown automaton is a sep-tuple (from [11]):

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

where:

- Q is a finite nonempty set of states
- Σ is the finite set of symbols called the input alphabet
- Γ is a finite set of symbols called the stack alphabet
- R is a finite set of transition rules denoted as: $Apa \to wq$, where $A \in \Gamma, p, q \in Q, a \in \Sigma \cup \{\varepsilon\}, w \in \Gamma^*$
- $s \in Q$ is a starting state
- $S \in \Gamma$ is a starting symbol on the stack
- $F \subseteq Q$ is a set of accepting states

Example

We show PDA on example M_{PDA} describing language $L_{PDA} = \{a^{3n}b^{4n} \mid n \geq 0\}$ over $\Sigma = \{a, b\}$.

$M_{PDA} = (Q, \Sigma, \Gamma, R, s, S, F)$,

- $Q = \{Q_1, Q_2, Q_3, Q_4, F_1\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{Z, a\}$
- R as follows:
  1. $ZQ_1a \to aaZQ_2$
  2. $aQ_1a \to aaa2$
  3. $aQ_1 \to aQ_4$
  4. $ZQ_1 \to ZQ_4$
  5. $aQ_2a \to aQ_3$
  6. $aQ_3a \to aQ_1$
  7. $bQ_4a \to Q_4$
  8. $ZQ_4 \to ZF_1$
- $s = Q_1$
- $S = Z$
- $F = \{F_1\}$

Graphical representation

Similarly to FA, we can visualize PDA in graphical form, as you can see in an example 2.5.

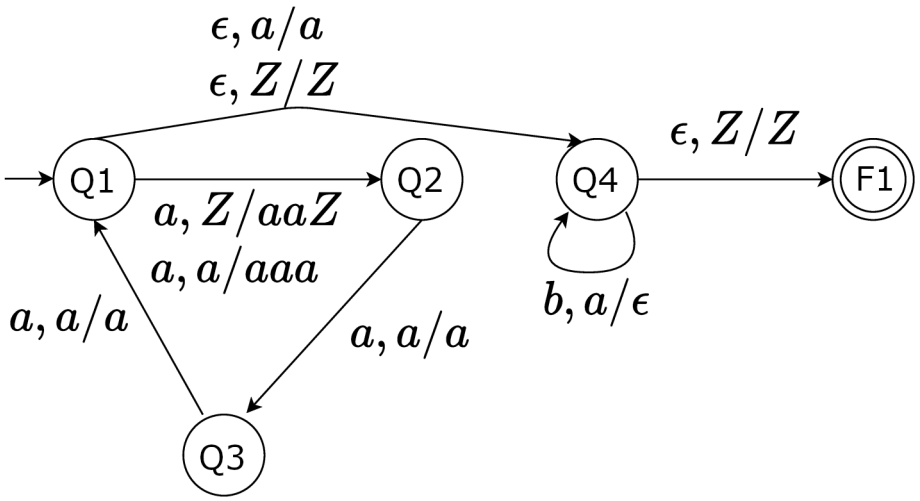

Figure 2.5: The example of the pushdown automata from [4]

Derivation

Now we provide an example of a generation of $w = aaabbbb$, where $w \in L_{PDA}$

$$
\begin{array}{ll}
 & ZQ_1aaabbbb \\
\Rightarrow & ZaaQ_2aabbbb \quad [1] \\
\Rightarrow & ZaaaQ_3abbbb \quad [5] \\
\Rightarrow & ZaaaaQ_13bbbb \quad [6] \\
\Rightarrow & ZaaaaQ_4bbbb \quad [3] \\
\Rightarrow & ZaaaQ_4bbb \quad [7] \\
\Rightarrow & ZaaQ_4bb \quad [7] \\
\Rightarrow & ZaQ_4b \quad [7] \\
\Rightarrow & ZQ_4 \quad [7] \\
\Rightarrow & ZF_1 \quad [8]
\end{array}
$$

2.4.3 Type-1 language models

Context-sensitive grammar (Type-1 grammars)

A context-sensitive grammar is a quadruple (from [23]):

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

where:

- N is a finite alphabet of nonterminals

- T is a finite alphabet of terminals, $N \cap T = \emptyset$
- P is a finite set of rules of the form:

$$\alpha A \beta \to \alpha \gamma \beta, \text{ where } A \in N,\ \alpha, \beta \in (N \cup T)^*,\ \gamma \in (N \cup T)^+$$

- $S \in N$ is a start symbol

Example

We show context-sensitive grammar on example G_{CSG} describing language $L_1 = \{a^n b^n c^n | n \geq 1\}$ over $\Sigma = \{a, b, c\}$.

$G_{CSG} = (N, T, P, S)$

- $N = \{S, B, C\}$
- $T = \{a, b, c\}$
- P as follows:
  1. $S \to aBC$
  2. $S \to aSBC$
  3. $CB \to BC$
  4. $aB \to ab$
  5. $bB \to bb$
  6. $bC \to bc$
  7. $cC \to cc$

Derivation

Now, we provide an example of generation of $w = aaabbbccc$, where $w \in L_1$

$$\begin{aligned}
& S \\
\Rightarrow\ & aSBC && [2] \\
\Rightarrow\ & aaSBCBC && [2] \\
\Rightarrow\ & aaaBCBCBC && [1] \\
\Rightarrow\ & aaaBCBBCC && [3] \\
\Rightarrow\ & aaaBBCBCC && [3] \\
\Rightarrow\ & aaaBBBCCC && [3] \\
\Rightarrow\ & aaabBBCCC && [4] \\
\Rightarrow\ & aaabbBCCC && [5] \\
\Rightarrow\ & aaabbbCCC && [5] \\
\Rightarrow\ & aaabbbcCC && [6] \\
\Rightarrow\ & aaabbbccC && [7] \\
\Rightarrow\ & aaabbbccc && [7]
\end{aligned}$$

Linear bounded automata

A linear bounded automaton (LBA) is an automaton that is an advanced version of a finite automaton or a pushdown automaton, which brings the ability to check context-sensitiveness. We can describe it in a very oversimplified way as a finite automaton augmented with a stack memory. Stack memory helps the LBA remember a certain amount of information with even more information to keep than a stack, which enables it to recognize more complex languages, such as context-sensitive ones.

LBAs do not have a deep connection with this thesis's main result. Therefore, the mention is sufficient to cover the topic of LBAs. There is still a place for studying LBA and deriving similar models and their transformations. But for this thesis, it was chosen to stay at the level of context-free languages because studying context-free languages and models describing these languages are more important. Therefore there will be no definition, and we only show an example of LBA in Figure 2.6.

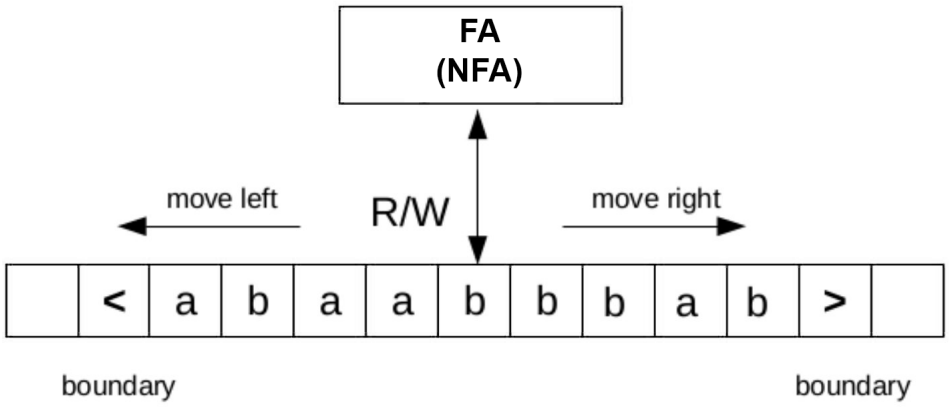

Figure 2.6: The illustration example of an LBA from [6]

2.4.4 Type-0 language models

Unrestricted grammar (Type-0 grammar)

Definition

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

where:

- N is a finite alphabet of nonterminals
- T is a finite alphabet of terminals, $N \cap T = \emptyset$
- P is a finite set of rules of the form:

$$\alpha \to \beta, \text{ where } \alpha \in (N \cup T)^* N (N \cup T)^*,\ \beta \in (N \cup T)^*$$

- $S \in N$ is a start symbol

As we can see, the definition is identical to a general grammar.

Turing machine

When the Turing machine is mentioned, we should always mention the Church thesis: Turing machines define, by their computational power, what we consider effectively computable. Church's thesis is not a theorem. We cannot prove formally that something corresponds to our intuitive ideas. Nevertheless, it is supported by several arguments:

- Turing machines are very robust. We will see that their various modifications do not change their computational power (determinism x nondeterminism, number of input tapes, ...).
- Various different models of computation (λ-calculus, partially recursive function, etc.) have been proposed, whose power is equal to Turing machines.
- No computation process, which we can call effectively computable and we cannot execute on a Turing machine, is known.

Definition

Turing machine is a sex-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_f),$$

where:

- Q is a finite nonempty set of control states
- Σ is the finite set of symbols called input alphabet, $\Delta \notin \Sigma$
- Γ is a finite set of symbols, $\Sigma \subset \Gamma, \Delta \in \Sigma$, called as tape alphabet
- partial function $\delta : (Q \setminus \{q_f\}) \times \Gamma \to Q \times (\Gamma \cup \{L, R\}$, where $L, R \notin \Gamma$, is called a transitions function
- $q_0 \in Q$ is a starting state
- $q_f \in Q$ is an accepting state

Example

The example describes a TM M_2 that recognizes the language $L = \{0^{2^n} | n \geq 0\}$ and also includes rejecting the state, which automatically rejects input. The example can be found in [7].

$M_2 = (Q, \Sigma, \Gamma, \delta, q1, q_{accept}, q_{reject})$, where:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}$
- $\Sigma = \{0\}$
- $\Gamma = \{0, x, \sqcup\}$
- δ is described in Figure 2.7

- q_1 is a starting state
- q_{accept} is an accepting state
- q_{reject} is a special rejecting state

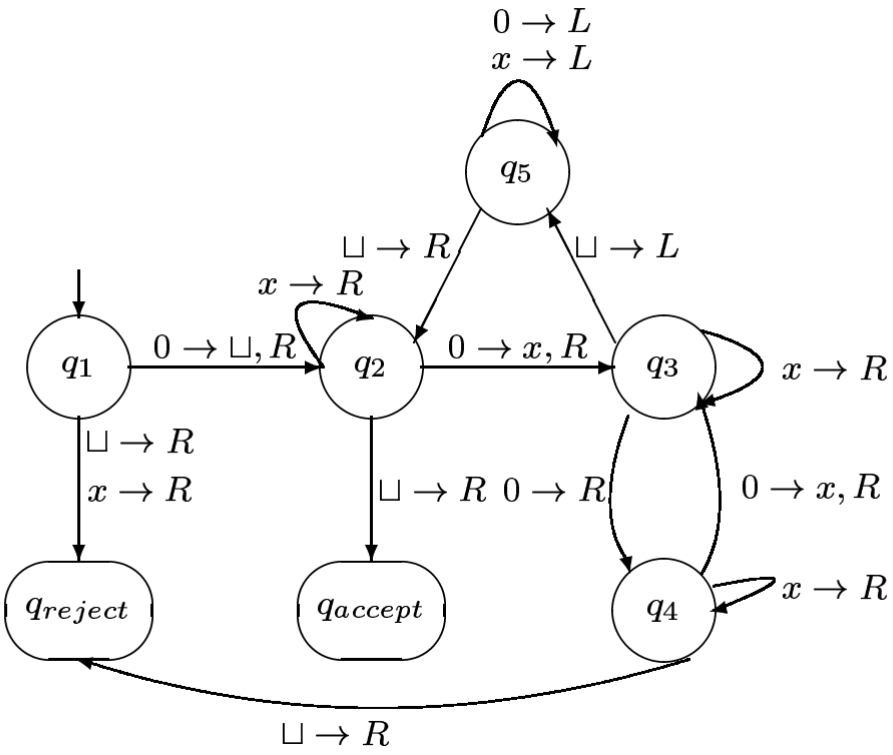

Figure 2.7: Example of a Turing machine M_2 from [7]

2.5 L-grammars

The theory of L systems (grammars) originated from the work of Lindenmayer [15]. The original aim of this theory was to provide mathematical models for the development of simple filamentous organisms. At first, L grammars were defined as linear arrays of finite automata, later reformulated into the more suitable framework of grammar-like constructs. From then on, the theory of L systems was developed essentially as a branch of formal language theory. It constitutes today one of the most vigorously investigated areas of formal language theory. For more you can see already mentioned [15] and also [3], [20] or see pages 63–65 in [21]. We introduce two L grammars, E0L and EP0L. E0L stands for:

- E-„extended“ (meaning that nonterminals are allowed)
- 0-„zero“ (meaning that the rewriting of occurrences of a letter does not depend on its context, or, in more technical terms, zero-sided context is allowed)
- L-„parallel“ (meaning that all letter occurrence in a string is rewritten in a direct derivation step).

EP0L stands for:

- E, 0, and L have the same meaning as in E0L
- P-„propagating“ (meaning that no erasing rules are allowed)

Definition of E0L

An E0L grammar is quadruple:

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

where:

- V is a finite alphabet
- T is a finite alphabet of terminals, $T \subset V$
- P is a finite set of rules of the form:

$$A \to \alpha, \text{ where } A \in N,\ \alpha \in (N \cup T)^*$$

- $S \in N$ is a start symbol

Note the language of G consists of all words that can be derived from w using rules of P in a parallel way.

Definition of EP0L

An EP0L grammar is quadruple:

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

where:

- V is a finite alphabet
- T is a finite alphabet of terminals, $T \subset V$
- P is a finite set of rules of the form:

$$A \to \alpha, \text{ where } A \in N,\ \alpha \in (N \cup T)^+$$

- $S \in N$ is a start symbol

Note the language of G consists of all words that can be derived from w using rules of P in a parallel way.

Practical example 1

The example illustrates the idea of plants simulated by parametric 0L grammars with permitting conditions. We demonstrate a plant development with a resource flow controlled by the number of apexes, so it erases the apex and generates two new internodes terminated by apexes. Figure 2.8 from [17] developmental stages of a plant simulation based on this model. From the presented example, we see that with permitting conditions, parametric 0L grammars can describe sophisticated models of plants in a very natural way. Notably, compared to the context-sensitive L grammars, they allow one to refer to modules not adjacent to the rewritten module. This property makes them more adequate, succinct, and elegant.

Practical example 2

For another example, let us assume that the red alga occurs in some unhealthy conditions in which only some of its parts survive, while the rest dies. This dying process starts from the newly born marginal parts of branches, which are too young and weak to survive, and proceeds toward the older parts, which are strong enough to live under these conditions. To be quite specific, all the red alga parts become gradually dead except for the parts denoted by 2s and 8s. This process can be specified by 0L grammar with forbidding conditions, and Figure 2.9 shows the dying process corresponding to the next derivation, whose last eight strings correspond to stages (a) through (h) in the figure.

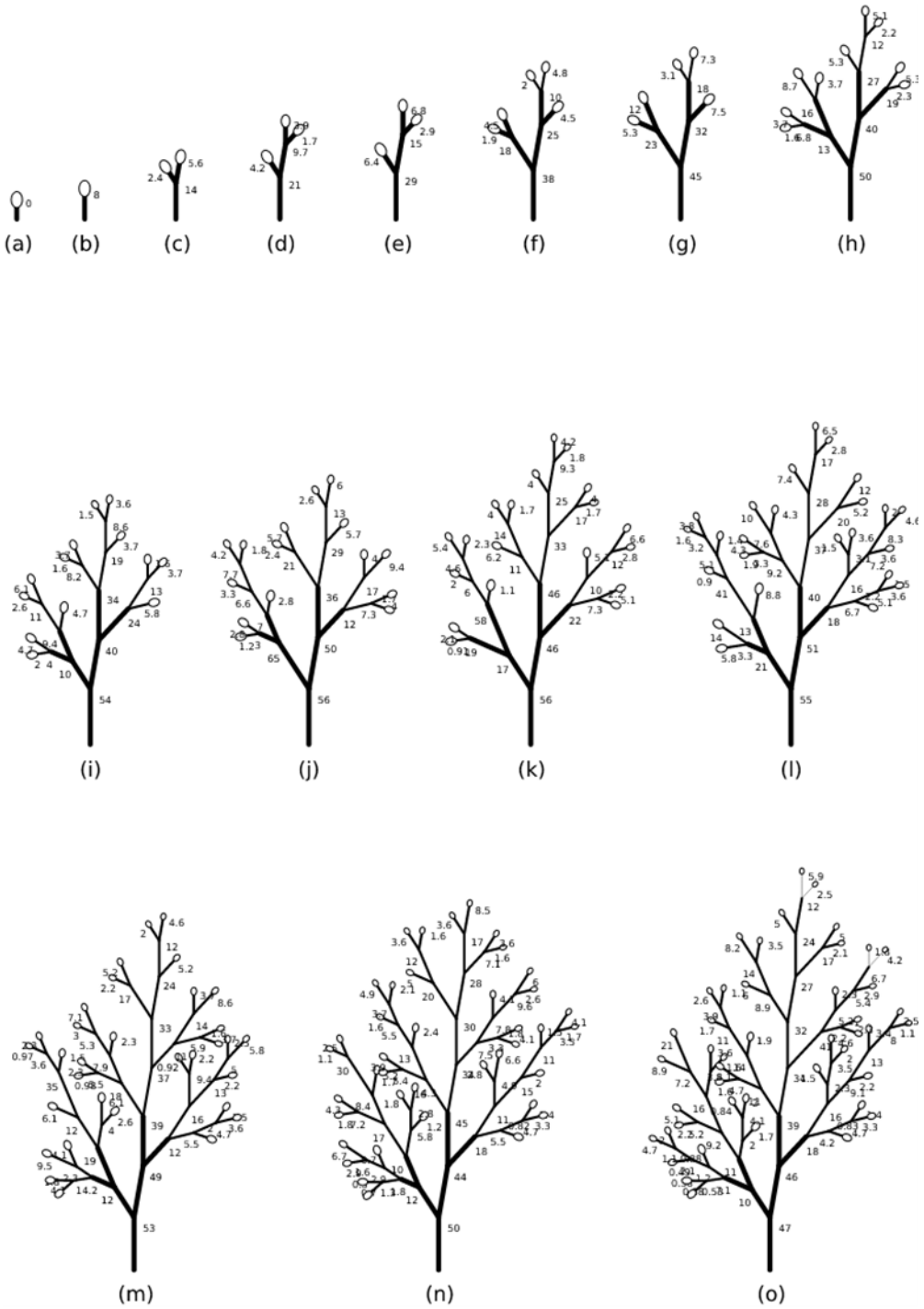

Figure 2.8: The example of the usage of the E0L grammar in the development of a plant from [17].

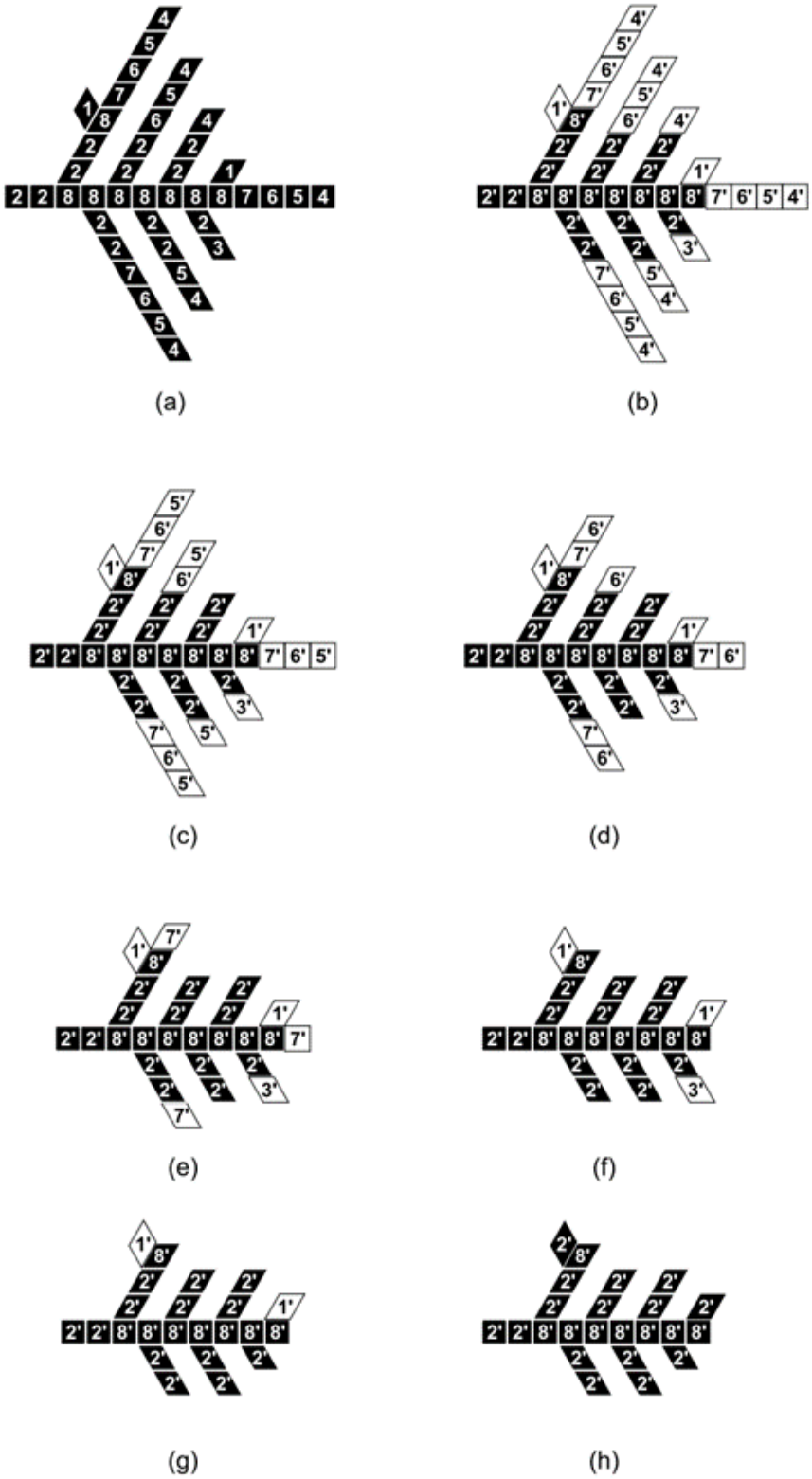

Figure 2.9: The example of the usage of the 0L grammar in death of marginal branch parts from [17].

2.6 Indian parallel grammar

We introduce an indian parallel grammar because it is a topic of interest in Chapter 5.

An indian parallel grammar is quadruple:

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

where:

- N is a finite alphabet of nonterminals
- T is a finite alphabet of terminals, $N \cap T = \emptyset$
- P is a finite set of rules of the form:

$$A \to \alpha, \text{ where } A \in N,\ \alpha \in (N \cup T)^*$$

- $S \in N$ is a start symbol

We can see that the components are defined the same as in a context-free grammar. We say that $x \overset{i}{\Rightarrow} y$ holds in $G, x \in (N \cup T)^+, y \in (N \cup T)^*$ iff the following conditions are satisfied:

- $x = x_1 A x_2 A ... x_n A x_{n+1}$, where $A \in N, x_i \in ((N \cup T) \setminus \{A\})^*$ for $1 \leq i \leq n+1$
- $y = x_1 w x_2 w ... x_n w x_{n+1}$, where $w \in (N \cup T)^*, x_i \in ((N \cup T) \setminus \{A\})^*$ for $1 \leq i \leq n+1$
- $A \to w \in P$

Again, the language $L(G)$ consists of all words $y \in T^*$ with $S \overset{i}{\Rightarrow}^* y$ where $\overset{i}{\Rightarrow}^*$ is the the reflexive and transitive closure of the relation $\overset{i}{\Rightarrow}$. Note that this type of parallelism lies between the sequential context-free derivations and the purely parallel derivations in L systems. In a context-free derivation, one occurrence of one letter is replaced. In an L system, all occurrences of all letters are rewritten, whereas, in an indian parallel grammar, all occurrences of one letter are replaced according to one rule. More about an indian parallel grammar in [25], [26], or Section 2.4 in [2].

2.7 Russian parallel grammar

Similarly, as an indian parallel grammar, we introduce A russian parallel grammar for Chapter 5.

A russian parallel grammar is quadruple:

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

where:

- N is a finite alphabet of nonterminals
- T is a finite alphabet of terminals, $N \cap T = \emptyset$
- P is a finite set of rules of the form:

$$A \to \alpha, \text{ where } A \in N,\ \alpha \in (N \cup T)^*$$

- $S \in N$ is a start symbol

We can again see that the components are defined as in a context-free grammar, and the set P is divided into two disjoint parts, $P = P_1 \cup P_2$. The yield relation $x \overset{r}{\Rightarrow} y$ is defined as follows: either

$$x = x_1 A x_2, y = x_1 w x_2, x_1, x_2, w \in (N \cup T)*, A \in N \text{ and } A \to w \in P_1.$$

or

$$x = x_1 A x_2 A ... x_n A x_{n+1}, y = x_1 w x_2 w ... x_n w x_{n+1}, \text{ where } A \in N, w \in (N \cup T)^*,$$

$$x_i \in ((N \cup T) \setminus \{A\})^* \text{ for some } 1 \leq i \leq n+1, \text{ and } A \to w \in P_2.$$

The language $L(G)$ is defined by

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

where $\overset{r}{\Rightarrow}^*$ is the reflexive and transitive closure of $\overset{r}{\Rightarrow}$. If in a russian grammar as above, we choose $P_1 = \emptyset$ or $P_2 = \emptyset$, we obtain an indian parallel or a context-free grammar, respectively. Thus each context-free and each indian parallel language is a russian parallel language, too. More about a russian parallel grammar can be found in [14] or Section 2.4 in [2].

2.8 K-grammars

For the same reason as an indian parallel grammar and a russian parallel grammar, we introduce a k-grammar. A k-grammar is quadruple:

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

where:

- N is a finite alphabet of nonterminals
- T is a finite alphabet of terminals, $N \cap T = \emptyset$
- P is a finite set of rules of the form:

$$A \to \alpha, \text{ where } A \in N,\ \alpha \in (N \cup T)^*$$

- $S \in N$ is a start symbol

We can see that the components are defined as in a context-free grammar and $k \geq 1$. For $x, y \in (N \cup T)^*$, we say that x directly derives y iff

$$x = S \text{ and } S \to y \in P$$

or

$$x = x_1 A_1 x_2 A_2 ... x_k A_k x_{k+1}, \text{ for certain } x_1, x_2, ..., x_{k+1} \in (N \cup T)*,$$

$$y = x_1 w_1 x_2 w_2 ... x_k w_k x_{k+1},$$
$$A_1 \to w_1, A_2 \to w_2, ..., A_k \to w_k \in P.$$

The language $L(G)$ is defined by

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

Note that by definition, besides the first step of the derivation, we have to replace in parallel k occurrences of nonterminals according to the rules of the grammar. Therefore derivation is blocked if the sentential form contains less than k nonterminals. More in [24] and page 126 in the second volume of [22].

2.9 E0S grammars

The most important model of language for this thesis is E0S grammar. We have chosen this model because it can be viewed as a bridge between a context-free grammar and L grammar. We can decipher shortcut E0S and EP0S as follows. E0S stands for:

- E-„extended“ (meaning that nonterminals are allowed)
- 0-„zero“ (meaning that the rewriting of occurrences of a letter does not depend on its context, or, in more technical terms, zero-sided context is allowed)
- S-„sequential“ (meaning that one occurrence of a letter in a string is rewritten in a direct derivation step).

EP0S stands for:

- E, 0, and L have the same meaning as in E0L
- P-„propagating“ (meaning that no erasing rules are allowed)

If we compare E0S and E0L and EP0S and EP0L respectively, we can see that their only difference is in the derivation process where E0S and EP0S work in sequential derivation and E0L and EP0L work in parallel derivation mode. Therefore connection and theoretical conversion of the main results algorithm from E0S to E0L could be possible.

The relation between E0S grammars and context-free grammars is not as obvious as for E0L grammars, but they are still very connected, as proven in the section Comparison to CFG. More information about E0S grammar can be found in [2].

Definition E0S

A E0S grammar is quadruple:

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

where:

- V is a finite alphabet
- T is a finite alphabet of terminals, $T \subset V$
- P is a finite set of rules of the form:

$$\alpha \to \beta, \text{ where} \alpha \in V,\ \beta \in V^*$$

- $S \in V$ is a start symbol, $S \in V \setminus T$

Definition EP0S

EP0S grammar is quadruple:

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

where:

- V is a finite alphabet
- T is a finite alphabet of terminals, $T \subset V$
- P is a finite set of rules of the form:

$$\alpha \to \beta, \text{ where } \alpha \in V,\ \beta \in V^+$$

- $S \in V$ is a start symbol, $S \in V \setminus T$

Derivation step definition

Let $G = (V, T, P, S)$ be a grammar and let λ, μ be strings from V^*. Between λ, μ is binary relation $\Rightarrow_G$, called the derivation step, if we can formulate strings λ and μ as:

$$\lambda = \gamma\alpha\delta,$$
$$\mu = \gamma\beta\delta,$$

where $\lambda, \mu \in V^*$, and $\alpha \to \beta$ is rule from P. Then we write:

$$\lambda \Rightarrow_G \mu \text{ or}$$
$$\lambda \Rightarrow \mu$$

Example

We demonstrate E0S grammar on example G_{e0s} describing language: $L_2 = \{a^n(b+c)^n | n \geq 0\}$ over $\Sigma = \{a, b, c\}$.

$G_{e0s} = (V, T, P, S)$

- $V = \{S, a, b, c\}$
- $T = \{a, b, c\}$
- P as follows:
  1. $S \to aSb$
  2. $b \to c$
  3. $S \to \varepsilon$

Derivation

Now we provide an example of generation of $w = aaaaccbc$, where $w \in L_2$

$$
\begin{aligned}
& S \\
\Rightarrow\ & aSb && [1] \\
\Rightarrow\ & aSc && [2] \\
\Rightarrow\ & aaSbc && [1] \\
\Rightarrow\ & aaaSbbc && [1] \\
\Rightarrow\ & aaaScbc && [2] \\
\Rightarrow\ & aaaaSbcbc && [1] \\
\Rightarrow\ & aaaaSccbc && [2] \\
\Rightarrow\ & aaaaccbc && [3]
\end{aligned}
$$

Comparison to CFG

Let G be a context-free grammar and G_{E0S} be an E0S grammar.

Theorem 3. $L(G) \subseteq L(G_{E0S})$

Proof. Obvious. Since every context-free grammar is E0S. □

Theorem 4. $L(G_{E0S}) \subseteq L(G)$

Proof. We can convert every E0S $G_{E0S} = (V, T, P, S)$ into equivalent context-free grammar $G = (V', T', P', S')$ by following the algorithm.

Algorithm 1: Convert the $E0S$ grammar to CFG

Input: E0S $G_{E0S} = (V, T, P, S)$.
Output: An equivalent context-free grammar $G = (V', T', P', S')$ by alternative definition.

1 $V' := V$
2 $T' := T$
3 $P' := P$
4 $S' := S$
5 **for** *each* $p \in P$ *such that* $a \to X$, *where* $a \in T$ *and* $X \in V$ **do**
    1. create new terminal N_a, where $N_a \notin V$
    2. $V' := V' \cup \{N_a\}$
    3. every occurrence of a in P' replace with N_a
    4. $P' := P' \cup \{N_a \to a\}$
6 **end**
7 return G

With this simple substitution, we eliminate all noncontext-free rules without changing the accepted language at the cost of extending the set of complete alphabet rules.

□

Theorem 5. $L(G) = L(G_{EP0S})$

Proof. Based on theorems 4 and 3 is obvious proof. □

2.10 Elimination of erasing rules

We want to introduce an algorithm as a set of instructions. These instructions can be written in a specific programming language such as Python or C or can be written in a pseudo-programming language, which is also a case for this thesis. We use these instructions to solve complex problems by simple progress, which each instruction provides.

The elimination of erasing rules from a grammar has been debated among linguists and grammarians in recent years. Erasing rules, also known as erasing rules or ε-rules refer to the process by which certain parts of the sentential form, for example, in context-free grammars nonterminals, are removed from a sentential form in order to progress a derivation with a derivation step.

Supporters of eliminating erasing rules argue that they are unnecessary and can actually make the language more confusing. They argue that eliminating erasing rules would lead to more consistent and predictable language, making it easier for people to communicate effectively. It also makes programming simpler since erasing is not a programmer-friendly operation.

Those who oppose their elimination say erasing rules is a crucial grammar component. It helps make sentences clear and concise, making grammar more compact and sometimes even more predictable. They argue that by eliminating rules, sentences become shorter and simpler to grasp by eliminating superfluous words and phrases.

The argument has merit on both sides, and the problem is not quickly resolved. Although it is evident that doing without erasing rules would have a tremendous impact on language and communication, additional research is required to comprehend the potential repercussions of such a move fully, so we should continue researching this topic and be open-minded to use as the elimination of erasing rules as a tool to help with a problem or in the same way uses the elimination rules.

Now we provide an example of an algorithm to build N_ε, which we show, in the next step, it is often mandatory to calculate.

Algorithm 2: Construction of N_ε

Input: A grammar $G = (N, \Sigma, P, S)$.
Output: The set N_ε that contains all ε-nonterminals in G

We construct sets $N_0, N_1, ...$ recursively this way:
$N_0 := \emptyset$
$i := 1$
repeat
    $N_i := \{A | A \to \alpha \in P \wedge \alpha \in (N_{i-1} \cup \varepsilon)\} \cup N_{i-1}$
    $i := i + 1$
until $N_i = N_{i-1}$
$N_\varepsilon := N_i$
return N_ε

Well-known algorithm 1

First, the well-known algorithm can be found in [18] and in [1]. As expected, the algorithm needs N_ε created like in algorithm 2. We can see that requirement is on line 1 in 3. The algorithm then proceeds with generating new rules, where some conditions use N_ε.

Algorithm 3: Transformation to grammar without ε-rules

Input: A grammar $G = (N, \Sigma, P, S)$.

Output: A equivalent grammar, $G' = (N', \Sigma', P', S')$ without ε-rules

1 Construct $N_\varepsilon = \{A | A \in N \wedge A \Rightarrow^* \varepsilon\}$. A construction of the set N_ε is in Algorithm 2

2 Let P' be a set of rules, which we construct in this way:

3 **if** $A \to \alpha_0 B_1 \alpha_1 B_2 ... B_k \alpha_k$ *is in* P, $k \geq 0$ *and each* B_i *is in* N_ε *for* $1 \leq i \leq k$, *but any of the symbols of the strings* α_j *is not in* N_ε, $0 \leq j \leq k$ **then**

4 add to P' the new subset of rules of form

$$A \to \alpha_0 X_1 \alpha_1 X_2 ... X_k \alpha_k$$

where X_i is either B_i or ε. Do not add the ε-rule $A \to \varepsilon$, which appears if all $\alpha_i = \varepsilon$.

5 **end**

6 **if** $S \in N_\varepsilon$ **then**

7 add to P' set of rules

$$S' \to \varepsilon | S,$$

S' is the new start symbol. Put $N' := N \cup \{S'\}$

8 **end**

9 **if** $S \notin N_\varepsilon$ **then**

10 $N' := N$ and $S' := S$.

11 **end**

12 The resulting grammar is $G' = (N', \Sigma', P', S')$

Well-known algorithm 2

The second chosen well-known algorithm is from [16]. This publication process uses more steps, which have their own advantages and disadvantages.

As a first step in the algorithm 6 on line 1, we are required to calculate N_ε with an algorithm such as the algorithm 2. Then we generate the new set of rules with the help of the calculated set N_ε.

In the next step, we are required to use the algorithm 5 to converse a context-free grammar with an equivalent context-free grammar without nonterminating symbols. The first-line algorithm for conversion of a context-free grammar to an equivalent context-free grammar without nonterminating symbols uses another algorithm 4 to determine terminating symbols.

Algorithm 4: Determination of terminating symbols

Input: A context-free grammar $G = (N, T, P, S)$
Output: The set V that consists of all terminating symbols in G.

```
1 V := T ∪ {lhs(p) : p ∈ P and rhs(p) ∈ T*}
2 repeat
3 |   U := V
4 |   for all p ∈ P such thal lhs(p) ∉ U do
5 |   |   if rhs(p) ∈ U+ then
6 |   |   |   V := V ∪ {lhs(p)}
7 |   |   end
8 |   end
9 until U = V
```

Algorithm 5: Conversion of a context-free grammar to an equivalent context-free grammar without nonterminating symbols

Input: A context-free grammar $G = (N, T, P, S)$
Output: A context-free grammar $G_{term} = (N_{term}, T, P_{term}, S)$, such that $L(G_{term}) = L(G)$ and $N_{term} \cup T$ contains only terminating symbols.

```
1 use Algorithm 4 to determine the set V of all terminating symbols in G
2 N_term := {A : A ∈ N ∩ V}
3 P_term := {p : p ∈ P, lhs(p) ∈ N_term, and rhs(p) ∈ V*}
4 produce G_term = (N_term, T, P_term, S)
```

Algorithm 6: Conversion of a context-free grammar to an equivalent ε-free context-free grammar

Input: A context-free grammar $G = (N, T, P, S)$ such that $(N \cup T)$ contains only useful symbols.

Output: An ε-free context-free grammar $G_{\varepsilon-free} = (N_{\varepsilon-free}, T, P_{\varepsilon-free}, S)$), satisfying these two properties:

1. $L(G_{\varepsilon-free}) = L(G) - \{\varepsilon\}$
2. $(N_{\varepsilon-free} \cup T)$ contains only useful symbols

1 Use Algorithm 2 to determine the set N_ε that contains all ε-nonterminals in G
2 **for** *each* $p \in P$ *such that* $rhs(p) = x_0A_1x_1...A_nx_n$ *and* $x_0x_1...x_n \neq \varepsilon$, *where* $A_i \in N_\varepsilon$, *and* $x_i \in (N \cup T)^*$, *for all* $i = 1, .., n$, *and where* $n \in \{1, ..., |rhs(p)| - 1\}$ **do**
3 $\quad P' := P' \cup \{lhs(p) \rightarrow x_0x_1x_2...x_n\}$
4 **end**
5 use Algorithm 5 to convert $G' = (N, T, P', S)$ to the context-free grammar, $G_{term} = (N_{term}, T, P_{term}, S)$, such that $L(G) = L(G')$ and $N_{term} \cup T$ contains only terminating symbols
6 $N_{\varepsilon-free} := N_{term}$
7 $P_{\varepsilon-free} := P_{term}$
8 produce $G_{\varepsilon-free} = (N_{\varepsilon-free}, T, P_{\varepsilon-free}, S)$

Chapter 3

Main Result

In this section, we present and verify an alternative algorithm that performs the elimination of erasing rules from E0S grammars. To give a more detailed understanding of this algorithm, consider an arbitrary E0S grammar, $G = (V, T, P, S)$, and $Y \in V$. If Y derives ε in G, then a derivation like this can be expressed step by step:

$$Y \Rightarrow Gy_1 \Rightarrow Gy_2 \Rightarrow G \cdots \Rightarrow Gy_n \Rightarrow G\varepsilon$$

where $y_i \in V^*$ for all $i = 1, \ldots, n$, for some $n \geq 1$. If a sentential form contains several occurrences of Y, each of them can be erased in this way, although there may exist many alternative ways of erasing Y. Based upon these observations, the next algorithm introduces compound nonterminals of the form $\langle X, U \rangle$, in which X is a symbol that is not erased during the derivation, and U is a set of symbols that are erased. Within the compound nonterminal, the algorithm simulates the erasure of symbols in U in the way sketched above. Observe that since U is a set, U contains no more than one occurrence of any symbol because there is no need to record several occurrences of the same symbol; indeed, as already pointed out, all these occurrences can be erased in the same way.

3.1 Algorithm

Algorithm 7: An alternative elimination of erasing rules in E0S grammars without any predetermination of symbols that derive the empty string.

Input: An E0S grammar, $G = (V, T, P, S)$.
Output: A propagating E0S grammar, $H = (V', T, P', S')$, such that $L(G) = L(H)$

1 $V' := T \cup \{\langle X, U\rangle | X \in V, U \subseteq V\}$.
2 $S' := \langle S, \emptyset\rangle$.
3 $P' := \{\langle a, \emptyset\rangle \to a | a \in T\}$.
4 **repeat**
5 **if** $Y \to y_0Y_1y_1Y_2y_2 \cdots Y_ny_n \in P$, *where* $y_i \in V^*$, $Y_j \in V$, *for all* i *and* j, $0 \leq i \leq n$, $1 \leq j \leq n$, *for some* $n \geq 1$ **then**
6 **for** *every* $U \subseteq V$ **do**
7 extend P' by adding $\langle Y, U\rangle \to \langle Y_1, U \cup alph(y_0y_1 \cdots y_n)\rangle\langle Y_2, \emptyset\rangle \cdots \langle Y_n, \emptyset\rangle$.
8 **end**
9 **end**
10 **if** $\langle X, U\rangle \in V'$ *and* $Y \to y \in P$, *where* $Y \in U$ *and* $y \in V^*$ **then**
11 extend P' by adding $\langle X, U\rangle \to \langle X, (U - \{Y\}) \cup alph(y)\rangle$.
12 **end**
13 **until** P' *cannot be extended*

3.2 Formal verification

Theorem 6. *Let G be an E0S grammar. Algorithm 7 halts and correctly converts G to a propagating E0S grammar H satisfying $L(G) = L(H)$.*

Proof. Clearly, the algorithm always halts. Since P' does not contain any erasing rules, H is propagating. To establish $L(H) = L(G)$, we prove three claims.

The first claim shows how derivations of G are simulated by H. It is used to prove $L(G) \subseteq L(H)$ later in the proof. Recall that the meaning of ${}^{\varepsilon}x$ is defined in Section 2.

Claim 1. *If $S \to_G^m {}^{\varepsilon}x_0X_1{}^{\varepsilon}x_1X_2{}^{\varepsilon}x_2 \cdots X_h{}^{\varepsilon}x_h \Rightarrow_G^* z$, where $z \in L(G)$, $|z| \geq 1$, $x_i \in V^*$, $X_j \in V$, for all i and j, $0 \leq i \leq h$, $1 \leq j \leq h$, for some $m \geq 0$ and $h \geq 1$, then $\langle S, \emptyset\rangle \Rightarrow_H^* \langle X_1, U_1\rangle\langle X_2, U_2\rangle \cdots \langle X_h, U_h\rangle$, where $\bigcup_{1 \leq i \leq h} U_i \subseteq \bigcup_{0 \leq i \leq h} alph(x_i)$.*

Proof. This claim is established by induction on $m \geq 0$.

Basis. The basis for $m = 0$ is clear.

Induction Hypothesis. Suppose that the claim holds for all derivations of length ℓ, where $0 \leq \ell \leq m$, for some $m \geq 0$.

Induction Step. Consider any derivation of the form

$$S \Rightarrow_G^{m+1} w \Rightarrow_G^* z,$$

where $w \in V^+$, $z \in L(G)$, and $|z| \geq 1$. Since $m + 1 \geq 1$, this derivation can be expressed as

$$S \Rightarrow_G^m x \Rightarrow_G w \Rightarrow_G^* z,$$

where $x \in V^+$. Let $x = {}^{\varepsilon}x_0X_1{}^{\varepsilon}x_1X_2{}^{\varepsilon}x_2 \cdots X_h{}^{\varepsilon}x_h$, where $x_i \in V^*$, $X_j \in V$, for all i and j, $0 \leq i \leq h$, $1 \leq j \leq h$, for some $h \geq 1$. Then, by the induction hypothesis,

$$\langle S, \emptyset \rangle \Rightarrow_H^* \langle X_1, U_1 \rangle \langle X_2, U_2 \rangle \cdots \langle X_h, U_h \rangle,$$

where $\bigcup_{1 \leq i \leq h} U_i \subseteq \bigcup_{0 \leq i \leq h} alph(x_i)$, for some $n \geq 0$.

Next, we consider all possible forms of $x \Rightarrow_G w$, covered by the next two cases—(i) and (ii).

(i) Let $X_j \to y_0Y_1y_1 \cdots Y_qy_q \in P$, where $y_i \in V^*$, for all i, $0 \leq i \leq q$, $Y_i \in V$, for all i, $1 \leq i \leq q$, for some j, $1 \leq j \leq h$, and $q \geq 1$, so

$$\begin{gathered} {}^{\varepsilon}x_0X_1{}^{\varepsilon}x_1 \cdots X_j{}^{\varepsilon}x_j \cdots X_h{}^{\varepsilon}x_h \Rightarrow_G \\ {}^{\varepsilon}x_0X_1{}^{\varepsilon}x_1 \cdots X_{j-1}{}^{\varepsilon}x_{j-1}{}^{\varepsilon}y_0Y_1{}^{\varepsilon}y_1 \cdots Y_q{}^{\varepsilon}y_q{}^{\varepsilon}x_jX_{j+1}{}^{\varepsilon}x_{j+1} \cdots X_h{}^{\varepsilon}x_h. \end{gathered}$$

By (1) in the algorithm,

$$\langle X_j, U_j \rangle \to \langle Y_1, U_j \cup alph(y_0y_1 \cdots y_q) \rangle \langle Y_2, \emptyset \rangle \cdots \langle Y_q, \emptyset \rangle \in P'$$

so

$$\begin{gathered} \langle X_1, U_1 \rangle \langle X_2, U_2 \rangle \cdots \langle X_j, U_j \rangle \cdots \langle X_h, U_h \rangle \Rightarrow_H \\ \langle X_1, U_1 \rangle \langle X_2, U_2 \rangle \cdots \langle X_{j-1}, U_{j-1} \rangle \langle Y_1, U_j \cup alph(y_0y_1 \cdots y_q) \rangle \\ \langle Y_2, \emptyset \rangle \cdots \langle Y_q, \emptyset \rangle \langle X_{j+1}, U_{j+1} \rangle \cdots \langle X_h, U_h \rangle. \end{gathered}$$

Clearly,

$$\left(\bigcup_{1 \leq i \leq h} U_i\right) \cup \left(\bigcup_{0 \leq i \leq q} alph(y_i)\right) \subseteq \left(\bigcup_{0 \leq i \leq h} alph(x_i)\right) \cup \left(\bigcup_{0 \leq i \leq q} alph(y_i)\right)$$

which completes the induction step for (i).

(ii) Let $x_j = x'_jYx''_j$ and $Y \to y \in P$, where $y \in V^*$ and $x'_j, x''_j \in V^*$, so

$${}^{\varepsilon}x_0X_1{}^{\varepsilon}x_1 \cdots X_j{}^{\varepsilon}x_j \cdots X_h{}^{\varepsilon}x_h \Rightarrow_G {}^{\varepsilon}x_0X_1{}^{\varepsilon}x_1 \cdots X_j{}^{\varepsilon}x'_j{}^{\varepsilon}y{}^{\varepsilon}x''_j \cdots X_h{}^{\varepsilon}x_h.$$

If $Y \notin \bigcup_{1 \leq i \leq h} U_i$, then

$$\langle X_1, U_1 \rangle \langle X_2, U_2 \rangle \cdots \langle X_h, U_h \rangle \Rightarrow_H^0 \langle X_1, U_1 \rangle \langle X_2, U_2 \rangle \cdots \langle X_h, U_h \rangle$$

and clearly and clearly

$$\bigcup_{1 \leq i \leq h} U_i \subseteq \left(\bigcup_{0 \leq i \leq h, i \neq j} alph(x_i)\right) \cup alph\left(x'_jyx''_j\right)$$

So, assume that $Y \in \bigcup_{1 \leq i \leq h} U_i$. By (2) in the algorithm,

$$\langle X_k, U_k \rangle \to \langle X_k, (U_k - \{Y\}) \cup alph(y) \rangle \in P',$$

where $U_k = U'_k \cup \{Y\}$, $U'_k \subseteq V$, for some k, $1 \leq k \leq h$, so

$$\begin{gathered} \langle X_1, U_1 \rangle \langle X_2, U_2 \rangle \cdots \langle X_k, U_k \rangle \cdots \langle X_h, U_h \rangle \Rightarrow_H \\ \langle X_1, U_1 \rangle \langle X_2, U_2 \rangle \cdots \langle X_k, (U_k - \{Y\}) \cup alph(y) \rangle \cdots \langle X_h, U_h \rangle. \end{gathered}$$

Clearly,

$$\left(\bigcup_{1\leq i\leq h, i\neq k} U_i\right) \cup \left(U'_k \cup alph(y)\right) \subseteq \left(\bigcup_{0\leq i\leq h, i\neq j} alph(x_i)\right) \cup alph\left(x'_j y x''_j\right)$$

which completes the induction step for (ii).

Observe that these two cases cover all possible derivations of the form $x \Rightarrow_G w$. Thus, the claim is valid. □

The second claim shows that in H, every derivation of any $z \in L(H)$ can be expressed as a two-part derivation. In the first part, every occurring symbol is a two-component nonterminal. In the second part, only the rules of the form $\langle a, \emptyset\rangle \to a$, where $a \in T$, is used.

Claim 2. *For every $z \in L(H)$, there exists a derivation $\langle S, \emptyset\rangle \Rightarrow_H^* x \Rightarrow_H^* z$, where $x \in V'^+$, and during $x \Rightarrow_H^* z$, only rules of the form $\langle a, \emptyset\rangle \to a$, where $a \in T$ are used.*

Proof. Since H is an E0S grammar, we can always rearrange all applications of rules, so the claim holds. □

The third claim shows how derivations of H are simulated by G. It is used to prove that $L(H) \subseteq L(G)$ later in the proof.

Claim 3. *If $\langle S, \emptyset\rangle \Rightarrow_H^n \langle X_1, U_1\rangle\langle X_2, U_2\rangle \cdots \langle X_h, U_h\rangle$, where $X_i \in V$ and $U_i \subseteq V$, for all i, $1 \leq i \leq h$, for some $n \geq 0$ and $h \geq 1$, then $S \Rightarrow_G^* x_0 X_1 x_1 X_2 x_2 \cdots X_h x_h$, where $x_i \in V^*$, for all i, $0 \leq i \leq h$, and $\bigcup_{1\leq i\leq h} U_i \subseteq \bigcup_{0\leq i\leq h} alph(x_i)$.*

Proof. This claim is established by induction on $n \geq 0$.

Basis. The basis for $n = 0$ is clear.

Induction Hypothesis. Suppose that the claim holds for all derivations of length ℓ, where $0 \leq \ell \leq n$, for some $n \geq 0$.

Induction Step. Consider any derivation of the form

$$\langle S, \emptyset\rangle \Rightarrow_H^{n+1} w,$$

where $w \in V'^+$. Since $n + 1 \geq 1$, this derivation can be expressed as

$$\langle S, \emptyset\rangle \Rightarrow_H^n x \Rightarrow_H w,$$

where $x \in V'^+$. Let $x = \langle X_1, U_1\rangle\langle X_2, U_2\rangle \cdots \langle X_h, U_h\rangle$, where $X_i \in V$, $U_i \in V^*$, for all i, $1 \leq i \leq h$, for some $h \geq 1$. By the induction hypothesis,

$$S \Rightarrow_G^* x_0 X_1 x_1 X_2 x_2 \cdots X_h x_h,$$

where $x_i \in V^*$, for all i, $1 \leq i \leq h$, such that $\bigcup_{1\leq i\leq h} U_i \subseteq \bigcup_{0\leq i\leq h} alph(x_i)$.

Next, we consider all possible forms of $x \Rightarrow_H w$, covered by the next two cases—(i) and (ii).

(i) Let $\langle X_j, U_j\rangle \to \langle Y_1, W\rangle\langle Y_2, \emptyset\rangle \cdots \langle Y_q, \emptyset\rangle \in P'$ be a rule introduced in (1) in the algorithm, where $W \subseteq V$ and $Y_i \in V$, for all i, $1 \leq i \leq q$, for some $q \geq 1$, so

$$\langle X_1, U_1\rangle\langle X_2, U_2\rangle \cdots \langle X_j, U_j\rangle \cdots \langle X_h, U_h\rangle \Rightarrow_H$$
$$\langle X_1, U_1\rangle \cdots \langle X_{j-1}, U_{j-1}\rangle\langle Y_1, W\rangle\langle Y_2, \emptyset\rangle \cdots \langle Y_q, \emptyset\rangle\langle X_{j+1}, U_{j+1}\rangle \cdots \langle X_h, U_h\rangle.$$

By (1) in the algorithm, W is of the form $W = U_j \cup alph(y_0 y_1 \cdots y_q)$, where $y_i \in V^*$, for all i, $1 \leq i \leq q$, and $X_j \to y_0 Y_1 y_1 \cdots Y_q y_q \in P$. Therefore,

$$x_0 X_1 x_1 \cdots X_j x_j \cdots X_h x_h \Rightarrow_G$$
$$x_0 X_1 x_1 \cdots X_{j-1} x_{j-1} y_0 Y_1 y_1 \cdots Y_q y_q x_j X_{j+1} x_{j+1} \cdots X_h x_h.$$

Clearly,

$$\left(\bigcup_{1 \leq i \leq h} U_i\right) \cup \left(\bigcup_{0 \leq i \leq q} alph(y_i)\right) \subseteq \left(\bigcup_{0 \leq i \leq h} alph(x_i)\right) \cup \left(\bigcup_{0 \leq i \leq q} alph(y_i)\right).$$

(ii) Let $\langle X_j, U_j \rangle \to \langle X_j, W \rangle \in P'$ be a rule introduced in (2) in the algorithm, for some j, $1 \leq j \leq h$, where $W \subseteq V$, so

$$\langle X_1, U_1 \rangle \langle X_2, U_2 \rangle \cdots \langle X_j, U_j \rangle \cdots \langle X_h, U_h \rangle \Rightarrow_H$$
$$\langle X_1, U_1 \rangle \langle X_2, U_2 \rangle \cdots \langle X_j, W \rangle \cdots \langle X_h, U_h \rangle.$$

By (2) in the algorithm, W is of the form $W = \left(U_j - \{Y\}\right) \cup alph(y)$, where $Y \in V$, $y \in V^*$, and $Y \to y \in P$. Recall that $\bigcup_{1 \leq i \leq h} U_i \subseteq \bigcup_{0 \leq i \leq h} alph(x_i)$ by the induction hypothesis. Since $Y \in \bigcup_{1 \leq i \leq h} U_i$, some x_k must be of the form $x_k = x'_k Y x''_k$, where $x'_k, x''_k \in V^*$, so

$$x_0 X_1 x_1 \cdots X_k x_k \cdots X_h x_h \Rightarrow_G x_0 X_1 x_1 \cdots X_k x'_k y x''_k \cdots X_h x_h.$$

Clearly,

$$\left(\bigcup_{1 \leq i \leq h, i \neq j} U_i\right) \cup \left(U_j - \{Y\}\right) \cup alph(y) \subseteq$$
$$\left(\bigcup_{1 \leq i \leq h, i \neq k} alph(x_i)\right) \cup \left(alph(x_k) - \{Y\}\right) \cup alph(y).$$

Observe that these two cases cover all possible derivations of the form $x \Rightarrow_H w$. Therefore, the claim is valid. □

Next, we establish the identity $L(H) = L(G)$. Consider a special case of claim 1 when $x_i = \varepsilon$, $X_j \in T$, for all i and j, $0 \leq i \leq h$, $1 \leq j \leq h$, for some $h \geq 1$. Then, $S \Rightarrow_G^* X_1 X_2 \cdots X_h$ implies that $\langle S, \emptyset \rangle \Rightarrow_H^* \langle X_1, \emptyset \rangle \langle X_2, \emptyset \rangle \cdots \langle X_h, \emptyset \rangle$. By the initialization part of the algorithm, $\langle X_j, \emptyset \rangle \to X_j \in P'$, for all j, $1 \leq j \leq h$, so

$$\begin{aligned} \langle S, \emptyset \rangle &\Rightarrow_H X_1 \langle X_2, \emptyset \rangle \cdots \langle X_h, \emptyset \rangle \\ &\Rightarrow_H X_1 X_2 \cdots \langle X_h, \emptyset \rangle \\ &\vdots \\ &\Rightarrow_H X_1 X_2 \cdots X_h. \end{aligned}$$

Therefore, $L(G) \subseteq L(H)$. Let $z \in L(H)$. By claim 2, $\langle S, \emptyset \rangle \Rightarrow_H^* x \Rightarrow_H^* z$, where $x \in V'^+$. By Claim 3, $S \Rightarrow_G^* z$. Therefore, $L(H) \subseteq L(G)$, and the theorem holds. □

3.3 Semi-Parallel and Parallel Derivation Modes

During every derivation step in an E0S grammar, a single symbol is rewritten. Therefore, these grammars work under *sequential derivation mode*. We next define a generalization of this mode in which some symbols are simultaneously rewritten while others remain unrewritten (like in scattered context grammars, see [8]).

Let $G = (V, T, P, S)$ be an E0S grammar. G makes a *semi-parallel derivation step* from $u_0v_1u_1 \cdots v_nu_n$ to $u_0w_1u_1 \cdots w_nu_n$, denoted by

$$u_0v_1u_1 \cdots v_nu_n \,{}_{\text{s-par}}\Rightarrow_G u_0w_1u_1 \cdots w_nu_n$$

if and only if $u_i \in V^*$ for all $i = 1, \ldots, n$, $v_j, w_j \in V^*$, and $v_j \Rightarrow_G w_j$ for all $j = 1, \ldots, n$, for some $n \geq 1$. Let ${}_{\text{s-par}}\Rightarrow^*_G$ denote the reflexive-transitive closure of ${}_{\text{s-par}}\Rightarrow_G$. The *language generated by G under the semi-parallel mode* is denoted by $L(G, {}_{\text{s-par}}\Rightarrow_G)$ and defined as

$$L\big(G, {}_{\text{s-par}}\Rightarrow_G\big) = \big\{w \in T^* \mid S \,{}_{\text{s-par}}\Rightarrow^*_G w\big\}.$$

The following theorem says that Algorithm 7 applies to E0S grammars working under this mode. Let us note that the standard algorithm also works for this mode.

Theorem 7. *Let G be an E0S grammar. Algorithm 7 halts and correctly converts G to a propagating E0S grammar H satisfying $L(G, {}_{\text{s-par}}\Rightarrow_G) = L(H, {}_{\text{s-par}}\Rightarrow_H)$.*

Proof. This theorem can be established by analogy with the proof of Theorem 6, so we leave its proof to the reader. □

By analogy with the definition of the semi-parallel mode, we may define a *parallel mode* where during a single derivation step, all occurrences of symbols in the current sentential form have to be rewritten. However, observe that neither Algorithm 7 nor the standard algorithm are applicable to E0S grammars working under this mode. To remove erasing rules from E0S grammars working under the parallel mode, we may use the algorithm for the elimination of erasing rules in E0L grammars.

Chapter 4

Application

This chapter will describe the implementation and a detailed specification of the technology used. Then it will offer a condensed manual on how to use the implemented application. At the end of the chapter, we will showcase some examples of usage and produced E0S grammars by the algorithm 7 in the implemented application.

4.1 Technology

Since the goal of the implementation is to present a newly created algorithm 7, the best possible way is to achieve its simplicity. Therefore, a form of implementation is a console application. A console application, sometimes also called a command-line application or cli in shortcut, is an application that strictly communicates via the command line, as the name suggests. This means it is typically without a graphical user interface. The missing user interface can be helpful or problematic, depending on the user. These applications are typically run on a command prompt or terminal window and take input and provide output through the command-line interface as it is in an application of the main result. Console applications are usually built using a programming language like Haskell or Python. Both of the given examples are cases of this thesis. Cli has many practical usages. For example, a console application could be used to automate system tasks, perform data processing, control other software, or simulate algorithms.

We use cli, because they are lightweight, faster, and simpler to use than graphical applications. They also do not require additional libraries or frameworks for GUI. Therefore, the developer does not need to allocate resources such as time and money to spend on developing a graphical user interface. Additionally, console applications can run on various operating systems with just minor, if even any, changes. Another advantage of console applications is that they are often used for more advanced tasks and are preferred by experienced users and developers, which are more used to using commands and find clicking buttons unpractical.

For implementation, there were selected two programming languages: Python and Haskell. They are both very different and provide different views on the given algorithm.

Python

Python is a commonly utilized top-level programming language for its simplicity and readability. Produced by Guido van Rossum in 1989, it's called after the British comedic team Monty Python. Among the essential functions of Python is its use of indentation to sug-

gest code obstructs instead of curly dental braces or keywords. This makes the code more aesthetically tidy and straightforward to check out, eliminating the need for end-of-line semicolons. The objective of the application is to show the application of the algorithm. Aesthetic clearness is essential.

Python is likewise understood for its comprehensive basic library, which includes components for various jobs such as linking to internet web servers, reviewing and composing data, and dealing with information. Furthermore, there are countless third-party libraries offered. Among Python's most prominent utilized situations are information scientific research and artificial intelligence. The Pandas library is commonly used for information control and evaluation, and libraries such as TensorFlow and Scikit-learn make it simple to develop and educate artificial intelligence designs. All this makes it an outstanding option. Python has a solid and supportive community and is regularly updated with new versions and features. All of this makes it an excellent choice. More about Python in [29].

Venv

For good programming practice, implementation will include Venv to make running it more straightforward and less intrusive for the user.

In Python, virtual environments, likewise called Venv, offer a method to separate the dependencies of various projects. This enables programmers to deal with several projects with multiple dependencies and variations of bundles without disrupting each other. Producing a virtual environment in Python is done utilizing the integrated Venv module. Python -m Venv <path/to/new/virtual/environment> is the fundamental command to produce a virtual environment. This will create a brand-new directory site in the defined path, which contains the required data for the virtual environment, but more about using Venv in the Section Usage.

Furthermore, It makes it easy to share projects with others. That is because the dependencies are all contained within the virtual environment. Therefore, installing them globally on the user's machine is unnecessary. A specific command sequence to create all dependencies within the virtual environment will be included later in this chapter.

Library Itertools

The only library we will discuss for Python implementation is Itertools. As the name suggests, the Python itertools library is a collection of tools for working with iterators (an object that can be iterated upon). The library mainly provides functions that allow an efficient way to work with iterators. We use this library for its one of our main features. It can provide a variety of iterators; for this code, we need iterators for working with combinations and permutations. In the implementation, we use the following functions (cited from [19]):

- **combinations(iterable, r)** - Return r length subsequences of elements from the input iterable.

  The combination tuples are emitted in lexicographic ordering according to the order of the input iterable. So, if the input iterable is sorted, the output tuples will be produced in sorted order.

  Elements are treated as unique based on their position, not on their value. So if the input elements are unique, each combination will have no repeated values.

- **permutations(iterable, r=None)** - Return successive r length permutations of elements in the iterable.

  If r is not specified or is None, then r defaults to the length of the iterable and all possible full-length permutations are generated.

  The permutation tuples are emitted in lexicographic order according to the order of the input iterable. So, if the input iterable is sorted, the output tuples will be produced in sorted order.

  Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeated values within a permutation.

- **groupby(iterable, key=None)** - Make an iterator that returns consecutive keys and groups from the iterable. The key is a function computing a key value for each element. If not specified or is None, key defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function.

  The operation of groupby() is similar to the uniq filter in Unix. It generates a break or new group every time the value of the key function changes (which is why it is usually necessary to have sorted the data using the same key function). That behavior differs from SQL's GROUP BY which aggregates common elements regardless of their input order.

Implementation details

Every part of the code is commented on in detail and references the specific line to algorithm 7.

Haskell

Functional programming is an existing option for programmers to use. Even though there are many disadvantages, we can find benefits, which follow:

- Simple: Functional programming aims to write compact, manageable functions that may be coupled to build more extensive programs. This method simplifies writing and comprehending some codes because each function is defined and tested independently.
- Immutability: Functional programming promotes the usage of immutable data structures, which prevent changes to assigned values after they have been made. This method simplifies building concurrent and parallel programming by doing away with complicated locking techniques.
- Pure functions: Functions are considered „pure“ because they always generate the same output for a given input and have no side effects. It is now simpler to reason about the code behavior and to create reliable programs as a result.
- Easy to parallel: As mentioned in immutability, functional programming focuses on pure functions and immutable data structures, making programs easier to parallelize. This implies that functional programs can benefit from distributed computing systems and multicore processors, resulting in faster and more effective code.

The functional programming language Haskell was created by and named after logician Haskell Curry. Haskell's specific feature is that type-checking happens at compilation rather than at runtime. Haskell's purity, which we already mentioned before, is one of its distinguishing characteristics. Haskell also offers a robust type system with support for algebraic data types, type classes, and type inference. These properties make it possible to write expressive and condensed code while still ensuring that it is accurate, which is an excellent property for implementing algorithms.

Haskell also contains a variety of sophisticated features, like lazyness, which makes it possible to handle infinite data structures quickly. It also contains monads, which give pure functional programmers means to control effects performed in a predictable and composable manner. There are numerous open-source tools and libraries available for Haskell development. The Glasgow Haskell compiler (GHC), the most extensively used Haskell compiler, is one of the most used tools in development and is used to compile our implementation. Next, we follow with used libraries.

Data.List

Contained within Haskell's Data.List module are many functions dedicated to manipulating and navigating lists - one indispensable data structure utilized prolifically throughout the Haskell environment. Practical, streamlined list management is efficient and effective with this collection of unique tools.

Aside from the previously mentioned capabilities, Data.List also encompasses many other advantageous features for manipulating lists, such as identifying extremities within a list or slicing up more extensive lists into more manageable sub-lists. This particular library serves exceptionally well in situations that demand these functionalities.

We use the following function (cited from [27]):

- **intersperse** - $O(n)$. The intersperse function takes an element and a list and 'intersperses' that element between the elements of the list.
- **intercalate** - It inserts the list xs in between the lists in xss and concatenates the result.
- **subsequences** - The subsequences function returns the list of all subsequences of the argument.
- **delete** - $O(n)$. delete x removes the first occurrence of x from its list argument.
- **nub** - $O(n^2)$. The nub function removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element. (The name nub means 'essence'.) It is a special case of nubBy, which allows the programmer to supply their own equality test.
- **union** - The union function returns the list union of the two lists. It is a special case of unionBy, which allows the programmer to supply their own equality test.
- **sort** - The sort function implements a stable sorting algorithm. It is a special case of sortBy, which allows the programmer to supply their own comparison function. Elements are arranged from lowest to highest, keeping duplicates in the order they appeared in the input.

Data.List.Split

In Haskell, the Data.List.SplitOn module is an expansion to the Data.List module that offers more methods for separating lists based on a specified separator. It is simple to divide lists into sub-lists based on a specific element or sequence of items using the functions in this module. The Data.List.SplitOn module has the benefit of enabling effective list splitting without the need for intermediary data structures. This may be extremely helpful when working with huge lists or conducting several splitting procedures.

We use this function:

- **splitOn** - Split on the given sublist.

Implementation details

Every part of the code is commented on in detail and references specific lines to algorithm 7.

4.2 Usage

This chapter will provide basic information on how to use the program and can be used as a simple manual to run implementation. Note that this chapter focuses on Linux-based distributions and that the steps are similar for other operating systems.

Python

Installation

1. **Makefile**

   Use the command: **make py-install**

   This should solve every dependency and necessity for running a program.

2. **Manual**

   It is highly recommended to use a virtual environment, so following commands go as follows:

   (a) **virtualenv elim_e-venv**
   (b) **source elim_e-venv/bin/activate**
   (c) **pip install -r requirements.txt**

Run

1. **Makefile**

   Makefile includes some examples of runs, which are named here with a simple description:

   - **make py-run** - will run over *test1.in* without output
   - **make py-run1** - will run over *test1.in* with output as output grammar
   - **make py-run2** - will run over *test2.in* with output as output grammar
   - **make py-run3** - will run over *test3.in* with output as output grammar

2. **Manual**

   The manual works via Python, so it is needed to run with Python. It is recommended to run it with arguments. There are two possible arguments:

   - **i** - to print out input grammar
   - **v** - to print out output grammar

   It is also possible to use both arguments to print input and output grammar in the same execution.

Format of the input grammar

This subsection provides simple rules to create input grammar, which is recommended to follow.

1. An input grammar has to be E0S grammar with all formalities satisfied.
2. ε is replaced with #, so example of erasing rule is $A \rightarrow \#$.
3. The first line in an input grammar is a complete alphabet, where every symbol has to be split by „,“ and it is recommended to use one character for a symbol of the complete alphabet
4. The second line in an input grammar is terminals, where every terminal has to be split by „,“ and it is recommended to use one character for terminal
5. The third line in an input grammar is starting symbol, and it is mandatory to use one character for starting symbol
6. Forth and following lines represent rules, where the line is one rule, where the left and right sides are divided by „->“.

Haskell

Installation

1. **Makefile**

   Use the command: **make hs-build** This should compile source code into a running program.

2. **Manual**

   It is necessary to have some distribution of Haskell and a compiler installed. Then it is possible to compile a program with the following:

   (a) **ghc -Wall –make -o hs-elim_e**

Run

1. **Makefile**

   Makefile includes some examples of runs, which are named here with a simple description:

- **make hs-run** - will run over *test1.in* with returning the input grammar
- **make hs-run1** - will run over *test1.in* with output as an output grammar
- **make hs-run2** - will run over *test2.in* with output as an output grammar
- **make hs-run3** - will run over *test3.in* with output as an output grammar

2. **Manual**

   The manual works via an executable built-in **hs-build**. It is recommended to run it with arguments. There are two possible arguments:

   - **-i** - to print out input grammar
   - **-v** - to print out output grammar

Format of the input grammar

This is the same as in Python code.

Format of an output grammar

There was a need to substitute $\emptyset$ as {}.

4.3 Examples

This section provides descriptions of generated examples of usage of implementation. The examples generated by the implementation can be found in Appendices.

Example 1

The first example refers to the examples in Appendices A and B for Python and Haskell examples, respectively. Since there is none, the first example does not remove any erasing rules. But this example is valuable since it is great for demonstrating how the algorithm works. Since input grammar has a relatively small alphabet and set of rules, it also produces small output grammar, which makes it visually readable. As we can see, the language of this grammar is $L_{ex1} = \{a\}$.

As we can see, many rules in output grammar cannot be reached. From the starting symbol, we cannot find any nonterminal, which is different from the empty set in the second position.

Example 2

The second example is also referring to the examples in Appendices A and B. As we can see, it has erasing rules in input grammar. So in this example, we can see how the algorithm works when it deals with the erasing rule. This example is not very interesting in sequential derivation. Still, it is more interesting in parallel because sequential derivations describe language $L_{ex2} = \{a^*\}$ more about it in the Chapter 5 of this thesis.

Example 3

The final example is again referring to the examples in Appendices A and B. The final example is one of the most used context-free grammar as an example with language $L_{ex3} = \{a^n b^n | n \geq 0\}$.

As we can see, the size of the alphabet and set of rules have an exponential increase, which could be solved since many rules and nonterminals cannot be reached, therefore, can be eliminated by some well-known algorithm for minimization.

Chapter 5

Conclusion

In this final section, we evaluate the completion of the goals set in the assignment. Then we proceed with the evaluation of the resulting algorithm. At the end of the chapter, we discuss the algorithm's applicability to other derivation modes in E0S grammars. Lastly, we propose two open problems to consider in future investigations related to the subject of this thesis.

The primary purpose of this thesis is to introduce the alternative transformations of grammar. This is done by creating a new algorithm for the elimination of erasing rules of E0S grammar, where we do not utilize a set of erasing nonterminals, which itself brings many advantages and disadvantages.

Firstly, we included the necessary preliminaries and definitions. We started from basic notions for example formal languages and slowly built to E0S grammars and algorithms for eliminating erasing rules. We included some theory, which is not necessarily needed but is worth mentioning for the knowledge overview and the borderline connection. This helps to understand, the need to study this problem and the need for wider evaluation in the context of formal languages.

After acquiring deep knowledge and a good understanding of the problem, it was possible to evaluate the situation and find missing possibilities to shrink a gap in the topic of the alternative transformation of grammars. This was done by designing a new algorithm for the elimination of erasing rules. The algorithm stands out from existing solutions thanks to not using the set of nonterminals, which can be erased. Such a set needs to be predetermined and therefore extends the time, which an algorithm for the elimination of erasing rules requires to create a grammar without erasing rules, which accepts equivalent language as an input grammar with erasing rules.

To consider the algorithm useful, it is necessary to prove that it works correctly and generates a new grammar from input grammar, which does not include erasing rules and accepts the same language as input grammar. We show that the new E0S grammar generated by the algorithm without erasing rules accepts the same language as the input E0S grammar by formal verification using mathematical induction. Demonstration that an E0S grammar created by the proposed algorithm is without the erasing rules is unnecessary because this property of an E0S grammar generated by the proposed algorithm is trivial.

Part of this thesis is an implementation based on algorithm 7. The primary purpose of implementing the algorithm is to show its practical usage, give the reader an easy way to see how it works, and generate their own examples of output E0S grammar. We also implemented the algorithm to demonstrate how it removes erasing rules. It also helps to

see some details, which can already be noticed in evaluating the algorithm itself, but by generating examples, it is trivial to see them.

As the reader probably already spotted, the algorithm works properly under sequential derivation mode and under semi-parallel derivation mode. The algorithm's correctness under parallel derivation mode is not guaranteed. Therefore, it cannot be used in parallel derivation mode. It is an exciting topic that could be studied further and is discussed in more detail in open problems. Another issue discussed in the open problems is based on an example where we clearly see that the algorithm trades the time complexity for the space complexity.

It is also worth mentioning that the proposed algorithm can be implemented in parallel since all loops are through all rules and all nonterminals exactly once, which makes splitting calculations between processors the most trivial task in parallel programming.

Lastly, we cannot stress enough that part of the thesis is the publication [10], where we published the main result and formal verification. We got an evaluation from an anonymous referee, which brought some valid points and solidified the value of this thesis.

We close this thesis by proposing two open problems.

5.1 Open problems

I. Observe that there exist other derivation modes under which the algorithm achieved in the previous section does not work properly. For instance, the algorithm is inapplicable to the indian derivation mode (see Section 2.6). Recall that an indian derivation step is performed so a rule is selected, and all symbols coinciding with the left-hand side of this rule are rewritten by it in the current sentential form. Consider the indian parallel grammar having the three rules $S \to SS$, $S \to a$, and $S \to \varepsilon$, where S is a nonterminal and a is a terminal. Obviously, its language equals

$$\{a^{2^n} | n \geq 0\} \cup \{\varepsilon\}.$$

If we convert this grammar to another indian parallel grammar by Algorithm 7, the resulting indian grammar generates $\{a^n | n \geq 1\}$, which differs from $\{a^{2^n} | n \geq 0\}$. As a result, the algorithm is inapplicable to derivation modes that involve the indian derivation mode, such as the mode of russian parallel grammars (see Section 2.7). Also, note that our algorithm is not applicable to the mode of k-grammars, where at every derivation step, exactly k occurrences of symbols have to be simultaneously rewritten (see Section 2.8). Can we modify Algorithm 7 so it works under these derivation modes as well? Recall that it is an open problem whether we can always eliminate all erasing rules from any russian parallel grammar or from any k-grammar.

II. Compared to its input grammar, the output grammar produced by Algorithm 7 has many more symbols and rules. To be precise, the cardinality of symbols and rules of output grammar has an exponential increase from the cardinality of symbols and rules of input grammar. Can we improve this algorithm so it works in a more economical way?

Bibliography

[1] Aho, A. V. and Ullman, J. D. *The Theory of Parsing, Translation, and Compiling.* USA: Prentice-Hall, Inc., 1972. ISBN 0139145567.

[2] Dassow, J. and Paun, G. *Regulated Rewriting in Formal Language Theory.* 1st ed. New York: Springer, 1989. ISBN 978-3-642-74934-6.

[3] G. Rozenberg, A. L. Developmental systems with locally catenative formulas. In: *Acta Informatica.* 1973, p. 214–248. 2 (3).

[4] GeeksforGeeks. *Construct Pushdown Automata for given languages* [online]. GeeksforGeeks, november 2021 [cit. 2022-10-16]. Available at: https://www.geeksforgeeks.org/construct-pushdown-automata-given-languages/.

[5] GeeksforGeeks. *Chomsky Hierarchy in Theory of Computation* [online]. GeeksforGeeks, august 2022 [cit. 2022-10-16]. Available at: https://www.geeksforgeeks.org/chomsky-hierarchy-in-theory-of-computation/.

[6] George, A. and .P, H. *Linear Bounded Automata* [online]. 2018 [cit. 2023-03-01]. Available at: https://www.csa.iisc.ac.in/~deepakd/atc-2018/seminar-slides/LBA.pdf.

[7] Goodrich, M. T. *Course Notes - CS 162 - Formal Languages and Automata Theory* [online]. [cit. 2023-03-01]. Available at: https://www.ics.uci.edu/~goodrich/teach/cs162/notes/turing2.pdf.

[8] Greibach, S. A. and Hopcroft, J. E. Scattered context grammars. *Journal of Computer and System Sciences.* 1st ed. 1969, vol. 3, no. 3, p. 233–247. ISSN 0022-0000.

[9] Havel, M. *Alternativní transformace jazykových modelů.* Brno, CZ, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/24417/.

[10] Havel, M. and Meduna, A. On Elimination of Erasing Rules from E0S Grammars. *Computer Science Journal of Moldova.* 2022, vol. 30, no. 2, p. 135–147. DOI: 10.56415/csjm.v30.08. ISSN 1561-4042. Available at: https://www.fit.vut.cz/research/publication/12800.

[11] Hopcroft, J. E., Motwani, R. and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation.* Boston, MA: Pearson Addison Wesley, 2006. ISBN 9780321455369.

[12] Kleijn, H. C. M. Basic ideas of selective substitution grammars. In: Kelemenová, A. and Kelemen, J., ed. *Trends, Techniques, and Problems in Theoretical Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, p. 75–95. ISBN 978-3-540-48008-2.

[13] Kleijn, H. C. M. and Rozenberg, G. Context-free like restrictions on selective rewriting. *Theoretical Computer Science*. 1st ed. 1981, vol. 16, no. 3, p. 237–269. ISSN 0304-3975.

[14] Levitina, M. K. On some grammars with global productions (in Russian). *NTI*. 1st ed. 1972, vol. 2, no. 3, p. 32–36.

[15] Lindenmayer, A. Mathematical Models for Cellular Interactions in Development. I and II. In: *Journal of Theoretical Biology*. 1968, p. 280–315. 18.

[16] Meduna, A. *Automata and Languages: Theory and Applications*. 1st ed. London: Springer, 2000. ISBN 978-1-4471-0501-5.

[17] Meduna, A. and Švec, M. *Grammars with Context Conditions and Their Applications*. USA: Wiley-Interscience, 2005. ISBN 0471718319.

[18] Milan Češka, T. V. *Theoretical Computer Science TCS Textbook*. 2009.

[19] PythonSoftwareFoundation. *Python Itertools Module Documentation* [online]. 2021 [cit. 2023-03-01]. Available at: https://docs.python.org/3/library/itertools.html.

[20] Rozenberg, G. and Salomaa, A. The Mathematical Theory of L Systems. In: Tou, J. T., ed. *Advances in Information Systems Science: Volume 6*. Boston, MA: Springer US, 1976, p. 161–206. ISBN 978-1-4615-8249-6.

[21] Rozenberg, G. and Salomaa, A. *Mathematical Theory of L Systems*. 1st ed. Orlando: Academic Press, 1980. ISBN 0-12-597140-0.

[22] Rozenberg, G. and Salomaa, A. *Handbook of Formal Languages, Volumes I through III*. 1st ed. New York: Springer, 2004. ISBN 3540614869.

[23] Salomaa, A. *Formal languages*. Academic Press, 1973. Computer science classics. ISBN 0126157502; 9780126157505.

[24] Salomaa, K. Hierarchy of k-context-free languages – parts 1 and 2. In: *International Journal of Computer Mathematics*. 1989, p. 69–90, 193–205. 26(2-3).

[25] Siromoney, R. and Krithivasan, K. Parallel context-free languages. *Information and Control*. 1st ed. 1974, vol. 24, no. 2, p. 155–162. ISSN 0019-9958.

[26] Skyum, S. Parallel context-free languages. *Information and Control*. 1st ed. 1974, vol. 26, no. 3, p. 280–285. 26(3). ISSN 0019-9958.

[27] TheHaskellTeam. *Data.List - Haskell Base Package Documentation* [online]. 2022 [cit. 2023-03-01]. Available at: https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-List.html.

[28] Vais, V. *Teoretická informatika 1. část* [online]. 2020 [cit. 2022-10-16]. Available at: http://home.zcu.cz/~vais/Vais%20-%20KA%20a%20RG.pdf.

[29] Van Rossum, G. and Drake, F. L. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN 1441412697.

Appendix A

Examples generated in Python

Example 1

Example 1 describes language $L_{ex2} = \{a^*\}$.

Input

```
S,a
a
S
S->a
```

Output - Python

```
⟨S,∅⟩,⟨S,{S}⟩,⟨S,{a}⟩,⟨S,{S,a}⟩,⟨a,∅⟩,⟨a,{S}⟩,⟨a,{a}⟩,⟨a,{S,a}⟩,a
a
⟨S,∅⟩
⟨a,∅⟩->a
⟨S,∅⟩->⟨a,∅⟩
⟨S,{S}⟩->⟨a,{S}⟩
⟨S,{a}⟩->⟨a,{a}⟩
⟨S,{S,a}⟩->⟨a,{S,a}⟩
⟨S,{S}⟩->⟨S,{a}⟩
⟨S,{S,a}⟩->⟨S,{a}⟩
⟨a,{S}⟩->⟨a,{a}⟩
⟨a,{S,a}⟩->⟨a,{a}⟩
```

Example 2

Example 2 describes language $L_{ex2} = \{a^*\}$.

Input

```
S,a
a
S
```

```
S->SS
S->a
S->#
```

Output - Python

```
⟨S,∅⟩,⟨S,{S}⟩,⟨S,{a}⟩,⟨S,{a,S}⟩,⟨a,∅⟩,⟨a,{S}⟩,⟨a,{a}⟩,⟨a,{a,S}⟩,a
a
⟨S,∅⟩
⟨a,∅⟩->a
⟨S,∅⟩->⟨S,{S}⟩
⟨S,∅⟩->⟨S,∅⟩⟨S,∅⟩
⟨S,{S}⟩->⟨S,{S}⟩
⟨S,{S}⟩->⟨S,{S}⟩⟨S,∅⟩
⟨S,{a}⟩->⟨S,{a,S}⟩
⟨S,{a}⟩->⟨S,{a}⟩⟨S,∅⟩
⟨S,{a,S}⟩->⟨S,{a,S}⟩
⟨S,{a,S}⟩->⟨S,{a,S}⟩⟨S,∅⟩
⟨a,{S}⟩->⟨a,{S}⟩
⟨a,{a,S}⟩->⟨a,{a,S}⟩
⟨S,∅⟩->⟨a,∅⟩
⟨S,{S}⟩->⟨a,{S}⟩
⟨S,{a}⟩->⟨a,{a}⟩
⟨S,{a,S}⟩->⟨a,{a,S}⟩
⟨S,{S}⟩->⟨S,{a}⟩
⟨S,{a,S}⟩->⟨S,{a}⟩
⟨a,{S}⟩->⟨a,{a}⟩
⟨a,{a,S}⟩->⟨a,{a}⟩
⟨S,{S}⟩->⟨S,∅⟩
⟨a,{S}⟩->⟨a,∅⟩
```

Example 3

Example 3 shows language $L_{ex3} = \{a^n b^n | n \geq 0\}$.

Input

```
S,a,b
a,b
S
S->aSb
S->#
```

Output - Python

```
⟨S,∅⟩,⟨S,{S}⟩,⟨S,{a}⟩,⟨S,{b}⟩,⟨S,{S,a}⟩,⟨S,{S,b}⟩,⟨S,{a,b}⟩,⟨S,{S,a,b}⟩,⟨a
↪  ,∅⟩,⟨a,{S}⟩,⟨a,{a}⟩,⟨a,{b}⟩,⟨a,{S,a}⟩,⟨a,{S,b}⟩,⟨a,{a,b}⟩,⟨a,{S,a,b}⟩,
↪  ⟨b,∅⟩,⟨b,{S}⟩,⟨b,{a}⟩,⟨b,{b}⟩,⟨b,{S,a}⟩,⟨b,{S,b}⟩,⟨b,{a,b}⟩,⟨b,{S,a,b}
↪  ⟩,a,b
a,b
⟨S,∅⟩
⟨a,∅⟩->a
⟨b,∅⟩->b
⟨S,∅⟩->⟨a,{S,b}⟩
⟨S,∅⟩->⟨S,{a,b}⟩
⟨S,∅⟩->⟨b,{S,a}⟩
⟨S,∅⟩->⟨a,{b}⟩⟨S,∅⟩
⟨S,∅⟩->⟨a,{S}⟩⟨b,∅⟩
⟨S,∅⟩->⟨S,{a}⟩⟨b,∅⟩
⟨S,∅⟩->⟨a,∅⟩⟨S,∅⟩⟨b,∅⟩
⟨S,{S}⟩->⟨a,{S,b}⟩
⟨S,{S}⟩->⟨S,{S,a,b}⟩
⟨S,{S}⟩->⟨b,{S,a}⟩
⟨S,{S}⟩->⟨a,{S,b}⟩⟨S,∅⟩
⟨S,{S}⟩->⟨a,{S}⟩⟨b,∅⟩
⟨S,{S}⟩->⟨S,{S,a}⟩⟨b,∅⟩
⟨S,{S}⟩->⟨a,{S}⟩⟨S,∅⟩⟨b,∅⟩
⟨S,{a}⟩->⟨a,{S,a,b}⟩
⟨S,{a}⟩->⟨S,{a,b}⟩
⟨S,{a}⟩->⟨b,{S,a}⟩
⟨S,{a}⟩->⟨a,{a,b}⟩⟨S,∅⟩
⟨S,{a}⟩->⟨a,{S,a}⟩⟨b,∅⟩
⟨S,{a}⟩->⟨S,{a}⟩⟨b,∅⟩
⟨S,{a}⟩->⟨a,{a}⟩⟨S,∅⟩⟨b,∅⟩
⟨S,{b}⟩->⟨a,{S,b}⟩
⟨S,{b}⟩->⟨S,{a,b}⟩
⟨S,{b}⟩->⟨b,{S,a,b}⟩
⟨S,{b}⟩->⟨a,{b}⟩⟨S,∅⟩
⟨S,{b}⟩->⟨a,{S,b}⟩⟨b,∅⟩
⟨S,{b}⟩->⟨S,{a,b}⟩⟨b,∅⟩
⟨S,{b}⟩->⟨a,{b}⟩⟨S,∅⟩⟨b,∅⟩
⟨S,{S,a}⟩->⟨a,{S,a,b}⟩
⟨S,{S,a}⟩->⟨S,{S,a,b}⟩
⟨S,{S,a}⟩->⟨b,{S,a}⟩
⟨S,{S,a}⟩->⟨a,{S,a,b}⟩⟨S,∅⟩
⟨S,{S,a}⟩->⟨a,{S,a}⟩⟨b,∅⟩
⟨S,{S,a}⟩->⟨S,{S,a}⟩⟨b,∅⟩
⟨S,{S,a}⟩->⟨a,{S,a}⟩⟨S,∅⟩⟨b,∅⟩
⟨S,{S,b}⟩->⟨a,{S,b}⟩
⟨S,{S,b}⟩->⟨S,{S,a,b}⟩
⟨S,{S,b}⟩->⟨b,{S,a,b}⟩
```

⟨S,{S,b}⟩->⟨a,{S,b}⟩⟨S,∅⟩
⟨S,{S,b}⟩->⟨a,{S,b}⟩⟨b,∅⟩
⟨S,{S,b}⟩->⟨S,{S,a,b}⟩⟨b,∅⟩
⟨S,{S,b}⟩->⟨a,{S,b}⟩⟨S,∅⟩⟨b,∅⟩
⟨S,{a,b}⟩->⟨a,{S,a,b}⟩
⟨S,{a,b}⟩->⟨S,{a,b}⟩
⟨S,{a,b}⟩->⟨b,{S,a,b}⟩
⟨S,{a,b}⟩->⟨a,{a,b}⟩⟨S,∅⟩
⟨S,{a,b}⟩->⟨a,{S,a,b}⟩⟨b,∅⟩
⟨S,{a,b}⟩->⟨S,{a,b}⟩⟨b,∅⟩
⟨S,{a,b}⟩->⟨a,{a,b}⟩⟨S,∅⟩⟨b,∅⟩
⟨S,{S,a,b}⟩->⟨a,{S,a,b}⟩
⟨S,{S,a,b}⟩->⟨S,{S,a,b}⟩
⟨S,{S,a,b}⟩->⟨b,{S,a,b}⟩
⟨S,{S,a,b}⟩->⟨a,{S,a,b}⟩⟨S,∅⟩
⟨S,{S,a,b}⟩->⟨a,{S,a,b}⟩⟨b,∅⟩
⟨S,{S,a,b}⟩->⟨S,{S,a,b}⟩⟨b,∅⟩
⟨S,{S,a,b}⟩->⟨a,{S,a,b}⟩⟨S,∅⟩⟨b,∅⟩
⟨a,{S}⟩->⟨a,{S,a,b}⟩
⟨a,{S,a}⟩->⟨a,{S,a,b}⟩
⟨a,{S,b}⟩->⟨a,{S,a,b}⟩
⟨a,{S,a,b}⟩->⟨a,{S,a,b}⟩
⟨b,{S}⟩->⟨b,{S,a,b}⟩
⟨b,{S,a}⟩->⟨b,{S,a,b}⟩
⟨b,{S,b}⟩->⟨b,{S,a,b}⟩
⟨b,{S,a,b}⟩->⟨b,{S,a,b}⟩
⟨S,{S}⟩->⟨S,∅⟩
⟨S,{S,a}⟩->⟨S,{a}⟩
⟨S,{S,b}⟩->⟨S,{b}⟩
⟨S,{S,a,b}⟩->⟨S,{a,b}⟩
⟨a,{S}⟩->⟨a,∅⟩
⟨a,{S,a}⟩->⟨a,{a}⟩
⟨a,{S,b}⟩->⟨a,{b}⟩
⟨a,{S,a,b}⟩->⟨a,{a,b}⟩
⟨b,{S}⟩->⟨b,∅⟩
⟨b,{S,a}⟩->⟨b,{a}⟩
⟨b,{S,b}⟩->⟨b,{b}⟩
⟨b,{S,a,b}⟩->⟨b,{a,b}⟩

Appendix B

Examples generated in Haskell

Example 1

Example 1 describes language $L_{ex2} = \{a^*\}$.

Input

```
S,a
a
S
S->a
```

Output - Haskell

```
⟨S,{}⟩,⟨S,{S}⟩,⟨S,{a}⟩,⟨S,{S,a}⟩,⟨a,{}⟩,⟨a,{S}⟩,⟨a,{a}⟩,⟨a,{S,a}⟩,a
a
⟨S,{}⟩
⟨"S",{}⟩->⟨"a",{}⟩
⟨"S",{S}⟩->⟨"a",{S}⟩
⟨"S",{a}⟩->⟨"a",{a}⟩
⟨"S",{S,a}⟩->⟨"a",{S,a}⟩
⟨"S",{S}⟩->⟨"S",{a}⟩
⟨"S",{S,a}⟩->⟨"S",{a}⟩
⟨"a",{S}⟩->⟨"a",{a}⟩
⟨"a",{S,a}⟩->⟨"a",{a}⟩
⟨"a",{}⟩ -> "a"
```

Example 2

Example 2 describes language $L_{ex2} = \{a^*\}$.

Input

```
S,a
a
S
```

```
S->SS
S->a
S->#
```

Output - Haskell

```
⟨S,{}⟩,⟨S,{S}⟩,⟨S,{a}⟩,⟨S,{S,a}⟩,⟨a,{}⟩,⟨a,{S}⟩,⟨a,{a}⟩,⟨a,{S,a}⟩,a
a
⟨S,{}⟩
⟨"S",{}⟩->⟨"S",{S}⟩
⟨"S",{}⟩->⟨"S",{}⟩⟨"S",{}⟩
⟨"S",{S}⟩->⟨"S",{S}⟩
⟨"S",{S}⟩->⟨"S",{S}⟩⟨"S",{}⟩
⟨"S",{a}⟩->⟨"S",{S,a}⟩
⟨"S",{a}⟩->⟨"S",{a}⟩⟨"S",{}⟩
⟨"S",{S,a}⟩->⟨"S",{S,a}⟩
⟨"S",{S,a}⟩->⟨"S",{S,a}⟩⟨"S",{}⟩
⟨"S",{}⟩->⟨"a",{}⟩
⟨"S",{S}⟩->⟨"a",{S}⟩
⟨"S",{a}⟩->⟨"a",{a}⟩
⟨"S",{S,a}⟩->⟨"a",{S,a}⟩
⟨"S",{S}⟩->⟨"S",{a}⟩
⟨"S",{S}⟩->⟨"S",{}⟩
⟨"S",{S,a}⟩->⟨"S",{a}⟩
⟨"a",{S}⟩->⟨"a",{S}⟩
⟨"a",{S}⟩->⟨"a",{a}⟩
⟨"a",{S}⟩->⟨"a",{}⟩
⟨"a",{S,a}⟩->⟨"a",{S,a}⟩
⟨"a",{S,a}⟩->⟨"a",{a}⟩
⟨"a",{}⟩ -> "a"
```

Example 3

Example 3 shows language $L_{ex3} = \{a^n b^n | n \geq 0\}$.

Input

```
S,a,b
a,b
S
S->aSb
S->#
```

Output - Haskell

```
⟨S,{}⟩,⟨S,{S}⟩,⟨S,{a}⟩,⟨S,{S,a}⟩,⟨S,{b}⟩,⟨S,{S,b}⟩,⟨S,{a,b}⟩,⟨S,{S,a,b}⟩,⟨
↪  a,{}⟩,⟨a,{S}⟩,⟨a,{a}⟩,⟨a,{S,a}⟩,⟨a,{b}⟩,⟨a,{S,b}⟩,⟨a,{a,b}⟩,⟨a,{S,a,b}
↪  ⟩,⟨b,{}⟩,⟨b,{S}⟩,⟨b,{a}⟩,⟨b,{S,a}⟩,⟨b,{b}⟩,⟨b,{S,b}⟩,⟨b,{a,b}⟩,⟨b,{S,a
↪  ,b}⟩,a,b
a,b
⟨S,{}⟩
⟨"S",{}⟩->⟨"a",{S,b}⟩
⟨"S",{}⟩->⟨"S",{a,b}⟩
⟨"S",{}⟩->⟨"a",{b}⟩⟨"S",{}⟩
⟨"S",{}⟩->⟨"b",{S,a}⟩
⟨"S",{}⟩->⟨"a",{S}⟩⟨"b",{}⟩
⟨"S",{}⟩->⟨"S",{a}⟩⟨"b",{}⟩
⟨"S",{}⟩->⟨"a",{}⟩⟨"S",{}⟩⟨"b",{}⟩
⟨"S",{S}⟩->⟨"a",{S,b}⟩
⟨"S",{S}⟩->⟨"S",{S,a,b}⟩
⟨"S",{S}⟩->⟨"a",{S,b}⟩⟨"S",{}⟩
⟨"S",{S}⟩->⟨"b",{S,a}⟩
⟨"S",{S}⟩->⟨"a",{S}⟩⟨"b",{}⟩
⟨"S",{S}⟩->⟨"S",{S,a}⟩⟨"b",{}⟩
⟨"S",{S}⟩->⟨"a",{S}⟩⟨"S",{}⟩⟨"b",{}⟩
⟨"S",{a}⟩->⟨"a",{S,a,b}⟩
⟨"S",{a}⟩->⟨"S",{a,b}⟩
⟨"S",{a}⟩->⟨"a",{a,b}⟩⟨"S",{}⟩
⟨"S",{a}⟩->⟨"b",{S,a}⟩
⟨"S",{a}⟩->⟨"a",{S,a}⟩⟨"b",{}⟩
⟨"S",{a}⟩->⟨"S",{a}⟩⟨"b",{}⟩
⟨"S",{a}⟩->⟨"a",{a}⟩⟨"S",{}⟩⟨"b",{}⟩
⟨"S",{S,a}⟩->⟨"a",{S,a,b}⟩
⟨"S",{S,a}⟩->⟨"S",{S,a,b}⟩
⟨"S",{S,a}⟩->⟨"a",{S,a,b}⟩⟨"S",{}⟩
⟨"S",{S,a}⟩->⟨"b",{S,a}⟩
⟨"S",{S,a}⟩->⟨"a",{S,a}⟩⟨"b",{}⟩
⟨"S",{S,a}⟩->⟨"S",{S,a}⟩⟨"b",{}⟩
⟨"S",{S,a}⟩->⟨"a",{S,a}⟩⟨"S",{}⟩⟨"b",{}⟩
⟨"S",{b}⟩->⟨"a",{S,b}⟩
⟨"S",{b}⟩->⟨"S",{a,b}⟩
⟨"S",{b}⟩->⟨"a",{b}⟩⟨"S",{}⟩
⟨"S",{b}⟩->⟨"b",{S,a,b}⟩
⟨"S",{b}⟩->⟨"a",{S,b}⟩⟨"b",{}⟩
⟨"S",{b}⟩->⟨"S",{a,b}⟩⟨"b",{}⟩
⟨"S",{b}⟩->⟨"a",{b}⟩⟨"S",{}⟩⟨"b",{}⟩
⟨"S",{S,b}⟩->⟨"a",{S,b}⟩
⟨"S",{S,b}⟩->⟨"S",{S,a,b}⟩
⟨"S",{S,b}⟩->⟨"a",{S,b}⟩⟨"S",{}⟩
⟨"S",{S,b}⟩->⟨"b",{S,a,b}⟩
⟨"S",{S,b}⟩->⟨"a",{S,b}⟩⟨"b",{}⟩
```

```
⟨"S",{S,b}⟩->⟨"S",{S,a,b}⟩⟨"b",{}⟩
⟨"S",{S,b}⟩->⟨"a",{S,b}⟩⟨"S",{}⟩⟨"b",{}⟩
⟨"S",{a,b}⟩->⟨"a",{S,a,b}⟩
⟨"S",{a,b}⟩->⟨"S",{a,b}⟩
⟨"S",{a,b}⟩->⟨"a",{a,b}⟩⟨"S",{}⟩
⟨"S",{a,b}⟩->⟨"b",{S,a,b}⟩
⟨"S",{a,b}⟩->⟨"a",{S,a,b}⟩⟨"b",{}⟩
⟨"S",{a,b}⟩->⟨"S",{a,b}⟩⟨"b",{}⟩
⟨"S",{a,b}⟩->⟨"a",{a,b}⟩⟨"S",{}⟩⟨"b",{}⟩
⟨"S",{S,a,b}⟩->⟨"a",{S,a,b}⟩
⟨"S",{S,a,b}⟩->⟨"S",{S,a,b}⟩
⟨"S",{S,a,b}⟩->⟨"a",{S,a,b}⟩⟨"S",{}⟩
⟨"S",{S,a,b}⟩->⟨"b",{S,a,b}⟩
⟨"S",{S,a,b}⟩->⟨"a",{S,a,b}⟩⟨"b",{}⟩
⟨"S",{S,a,b}⟩->⟨"S",{S,a,b}⟩⟨"b",{}⟩
⟨"S",{S,a,b}⟩->⟨"a",{S,a,b}⟩⟨"S",{}⟩⟨"b",{}⟩
⟨"S",{S}⟩->⟨"S",{}⟩
⟨"S",{S,a}⟩->⟨"S",{a}⟩
⟨"S",{S,b}⟩->⟨"S",{b}⟩
⟨"S",{S,a,b}⟩->⟨"S",{a,b}⟩
⟨"a",{S}⟩->⟨"a",{S,a,b}⟩
⟨"a",{S}⟩->⟨"a",{}⟩
⟨"a",{S,a}⟩->⟨"a",{S,a,b}⟩
⟨"a",{S,a}⟩->⟨"a",{a}⟩
⟨"a",{S,b}⟩->⟨"a",{S,a,b}⟩
⟨"a",{S,b}⟩->⟨"a",{b}⟩
⟨"a",{S,a,b}⟩->⟨"a",{S,a,b}⟩
⟨"a",{S,a,b}⟩->⟨"a",{a,b}⟩
⟨"b",{S}⟩->⟨"b",{S,a,b}⟩
⟨"b",{S}⟩->⟨"b",{}⟩
⟨"b",{S,a}⟩->⟨"b",{S,a,b}⟩
⟨"b",{S,a}⟩->⟨"b",{a}⟩
⟨"b",{S,b}⟩->⟨"b",{S,a,b}⟩
⟨"b",{S,b}⟩->⟨"b",{b}⟩
⟨"b",{S,a,b}⟩->⟨"b",{S,a,b}⟩
⟨"b",{S,a,b}⟩->⟨"b",{a,b}⟩
⟨"a",{}⟩ -> "a"
⟨"b",{}⟩ -> "b"
```