POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Characterizing Non Counting Operator Precedence Languages in a Locally Testable manner

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Giorgio Corbetta**

Student ID: 964531
Advisor: Prof. Matteo Pradella
Academic Year: 2022-23

Abstract

The aim of this thesis is to expand the knowledge about Operator Precedence Languages (OPL) by finding one of its subclasses such that it is Non-Counting. In order to achieve such a goal, there has been taken inspiration from the LTO family of languages, which corresponds to the Non-Counting Regular Language family. The found class (LTOP) is shown to coincide with the family of Non-Counting Operator Precedence Languages, and its relation with other representations from the literature is studied. In the end, the problem of deciding whether a language is LTOP or not is proven to be decidable.

Keywords: Languages, Operator Precedence, Locally Testable, Non-Counting, Structured Language, Decidability

Abstract in lingua italiana

Lo scopo della tesi è di trovare una sottoclasse dei linguaggi a precedenza ad operatori che sia non counting. Per raggiungere tale obiettivo ci si è ispirati alla classe dei linguaggi LTO, che corrisponde ai linguaggi Non Counting Regolari. La classe cosi trovata (LTOP) si dimostra coincidere con la classe dei linguaggi Non Counting OP, e viene studiata la sua relazione con le altre rappresentazioni presenti in letteratura. In conclusione, viene dimostrato che il problema di decidere se un linguaggio è o meno LTOP è decidibile.

Parole chiave: Linguaggi, Precedenza ad Operatori, Loalmente Testabili, Non-Counting, Linguaggi Strutturati, Decidibilità

Contents

Introduction

In the field of theoretical computer science, there is a field devoted to the study of languages, from their classification to their expression, through the algorithms recognizing them. In particular, the main goal of studies in this field has been the research of classes of languages satisfying a tradeoff between expressiveness and simplicity, thus the optimum would have been finding a language that perfectly models a problem and is simple enough to be approached with the tools available at the time. As there will be introduced in section 1.1.3, in 1956 Noam Chomsky introduced a classification of languages, their expressiveness, and representation. A great result of Chomsky's classification is that there is a direct mapping between the expressive power of a class of languages and the automata recognizing it. The smallest family in Chomsky's classification, thus the one with lots of studies and results, is the so-called "Regular Language" class of languages. It is in this class that can be found the family of Non-Counting/Aperiodic languages, a family that has been studied a lot for the similarities that bind it to lots of real-world applications. This class will be defined and discussed in section 1.2.3, however, as the name suggests, it contains the languages containing strings not depending on the number of times something is repeated. A simple example of a Counting language could be the language recognizing books with an even number of words [1]. Although there may seem that Non-Countingness does not provide significant restrictions to Regular Languages, it turned out to have lots of properties, and there exist lots of representations corresponding to the NC family of languages.

Another family of languages from Chomsky's classification, extending the Regular Languages one, is the so-called Context-Free class of Languages (CFL), discussed in chapter 3. A notorious example of a language that is CFL but it's not RL is the language of well-parenthesized strings. Together with expanded expressiveness, with CFL the concept of structure takes more importance, and with that also the concept of ambiguity starts to present itself. Ambiguity can be found, for example, in associating a unique structure to a string, or even also in recognizing if a string belongs or not to a given language, slowing

[1]This example, which is inaccurate, has been given only to hint what Counting and Non-Counting means, and to justify later results, not to provide an actual or significant example of Non-Counting languages.

the algorithms solving those problems. In order to solve the problem of ambiguity, there have been presented different solutions, among which there can be found:

- Floyd's idea of 1963 to use Operator Precedence (like done in arithmetic);
- The idea of McNaughton in 1967 was to embed explicitly the structure in the strings via parenthesization.

The idea of removing ambiguity brought out some results that permit the definition of some classes of languages closed under boolean operations. Although Floyd's idea was quite natural, being derived from arithmetic, and was exposed before the McNaughton one, parenthesization brought out firstly some major results, obscuring the studies on OP languages. However, already from 1978, in [8], the results of parenthesized languages have been transposed into OPL class, demonstrating that they were valid for a more general variety of languages, comprehending OPL and parenthesized ones: the so-called, Structured Context-Free Languages, disserted in section 3.2.
With those results, the growth of parallelized computational power, and the local parsability property of OPLs, the study of OPL can assume an important role nowadays.

Expanding the knowledge about OPL family can offer the possibility to develop programming languages that enable compilation to rely on distributed architectures and be more scalable than what is now available, to better model some phenomena or to model them in such a way that relies on the growth direction of the technology: parallelization. For example, there should be the possibility to change some of the methodologies used nowadays from working with some generic Context Free Language, to using an Operation Precedence Language and leveraging some of the specific OPLs' properties. However, in order to make this transition more captivating, there can be found more properties for OPL or for some of its subfamilies. It is with this goal that this work has been done: it aims to find a subclass of OPL that is Non-Counting and characterize it.

Non-Countingness is a property that is shared among lots of practical cases, and its class, in RL, has been shown to coincide with lots of other formalisms' simplifications. The definition of NC has already been transposed from the RL domain to the CFL one, and also in OPL, but relying only on one definition is not a practical way to represent a class of languages, as there may be many other equivalent representations, making it simpler to identify or describe languages belonging to that class.

The idea driving this work is to find a way to relate the Locally Testable family of languages (LTO) with the family of Operator Precedence Non-Counting Languages, or NC_{OP}, to see if there is a way to transfer some of LTO's properties from RL to NC_{OP}.

The LTO family, described in chapter 2, has been chosen not only for how it is defined but also because of its features, in particular, the fact that it provides another equivalent way to define the NC family in RL.

In the literature, as in [15], there have been presented different characterizations for the NC_{OP} family, and most of them are inspired by the characterizations of Non-Counting Regular Languages. However, among those characterizations, there lacks one inspired by the Locally Testable Languages. For those reason, to add a piece in the puzzle of characterizations for NC_{OP}, and to study the relation between Regular Non-Counting Languages and Operator Precedence Non-Counting Languages, this thesis has the goal to find a Locally Testable characterization of a family of languages that are Operator Precedence Non-Counting.

1 | Background

This chapter will provide an overview of the main known results of the theoretical language field, which can be found in many books, like [21]. The goal is to introduce all the concepts that are used in the following and describe the perimeter of this work. Here is provided the first characterization of languages' families with their properties and peculiarities.

In particular, this chapter covers the concepts needed to represent languages in section 1.1 with particular attention to grammar, and the definition and properties of Regular Languages in section 1.2.
In the first section, the goal is to present grammars and their properties, and introduce with the use of grammars, Chomsky's Classification of languages, which is done in section 1.1.3.
In section 1.2 the focus is to present the class of Regular Languages, with its characterizations, definition, and properties. Lastly, the concept of Non-Countingness is presented in section 1.2.3.

1.1. Language Characterization

This section presents the main concepts that enable to define the language. The concepts here presented start from the ground and arrive at the characterization of languages via grammars or logical formulae, passing by the Chomskyŝ Classification. There is no need for prior knowledge because all the needed definitions are provided step by step, followed by the major results regarding them.

1.1.1. From Alphabet to Languages

Character and alphabet are the first concepts to be introduced in order to talk of languages

Definition 1.1 (Alphabet/character). *An alphabet is a set of simple elements, called characters or symbols.*

In the literature, Σ is commonly used to refer to a generic alphabet.

Example 1.1 (Alphabets)**.** *Here are some examples of common alphabets:*

- *The set of the English letters is the alphabet* $\{a, b, c, \dots, z\}$*;*
- *The set of decimal digits is the alphabet* $\{0, 1, 2, \dots, 9\}$*;*
- *The set of ASCII characters is an alphabet;*
- *Using words as symbols, programming languages' instructions constitute an alphabet, like* $\{if, else, for, \dots\}$*;*
- *Using words as symbols, a dictionary is an alphabet, like* $\{fox, hello$ $, world, \dots\}$[1]*.*

Definition 1.2 (string)**.** *A string s is a sequence of characters belonging to a certain alphabet.*
Given a string $s = \{c_1, c_2, \dots, c_n\}$, its length, $|s|$, is defined as the number of characters it contains, so $|s| = n$.
There is a special string ε that is the only string with $|\varepsilon| = 0$, and is called the empty string.
Inside a string, the order (or equivalently the position) of the characters is essential to define it. A string $s = \{c_1, c_2, \dots, c_n\}$ will be represented by the easier notation $s = c_1c_2 \dots c_n$

Given the definition of strings, the equality relation is a trivial concept that is worth to be defined as all the future concepts will rely on it.

Definition 1.3 (equality)**.** *Given two strings $s_1 = a_1 \dots a_n$ and $s_2 = b_1 \dots b_m$, they are said to be equal iff $|s_1| = |s_2|$ and for each $0 < i \leq n$, a_i is the same character ad b_i.*

It is trivial to see that this relation satisfies the symmetric, transitive, and reflexive properties, so it can be classified as an equality relation.

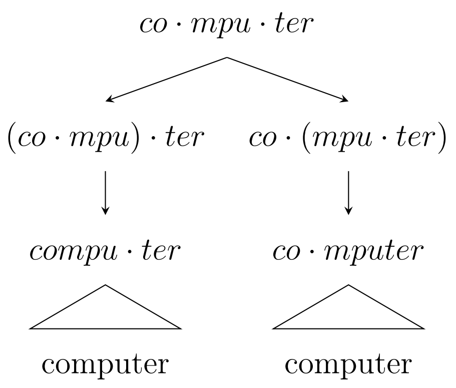
After the presentation of the definition of string, there can be introduced the operations over strings.

Definition 1.4 (Character Concatenation)**.** *Given a string $s = c_1c_2 \dots c_n$ and a character a, the concatenation operation $s \cdot a$ defines the string $s \cdot a = c_1c_2 \dots c_na$, the concatenation operation $a \cdot s$ defines the string $a \cdot s = ac_1c_2 \dots c_n$.*

[1]This idea is the one on which the stemming algorithms for Natural Language relies.

Definition 1.5 (String Concatenation). *Given two strings $s_1 = a_1 \dots a_n$ and $s_2 = b_1 \dots b_m$, the concatenation operation $s_1 \cdot s_2$ defines the string $s_1 \cdot s_2 = s_1 s_2 = a_1 \dots a_n b_1 \dots b_m$.*

It is trivial to see that any string can be described as the concatenation of characters. Please note that the concatenation operation satisfies the associative property. For example, given the strings $s_1 = co$, $s_2 = mpu$, $s_3 = ter$, we have that $s_1 \cdot s_2 \cdot s_3 = computer$.



As it happens with the multiplication operation in algebra, also the concatenation has the identity element, which is ε, because, as it can be trivially seen, for any string x, $\varepsilon \cdot x = x \cdot \varepsilon = x$.
Moreover, also with concatenation, there could be considered the case of a string concatenating with itself. This operation is called *power*, as it happens with numbers and multiplication.

Definition 1.6 (concatenation power). *Given a string s and an integer i, the string s^i is defined as:*

- $s^0 = \varepsilon$;
- $s^i = s^{i-1} \cdot s$

So, given any string s, $s^0 =$ ””, $s^1 = s$, $s^2 = s \cdot s$, ...

Example 1.2. *Gven the string $s = ab$, there is:*
$s^0 = = \varepsilon$
$s^1 = ab$
$s^2 = abab$
$s^3 = ababab$
$s^5 = ababababab$

Example 1.3. *The (meaningful) English sentence "buffalo buffalo buffalo buffalo buffalo buffalo buffalo buffalo buffalo buffalo " could be written as "buffalo "^{10}.*

Now that the concept of alphabet and string have been introduced, there can be stated also the definition of language.

Definition 1.7 (language). *A language L is defined as a set of strings over a certain alphabet. A string s is said to belong to a language L, or, equivalently, to be recognized by a language L, iff $s \in L$.*

Thus the concatenation operation can be extended also to languages:

Definition 1.8 (language concatenation). *Given any two languages*
$L_1 = \{x_1, x_2, \dots, x_n\},\ L_2 = \{y_1, y_2, \dots, y_m\}$
the concatenation of those languages $L_1 \cdot L_2$ is defined as the set of the concatenation of all the possible couples of strings in L_1 and L_2, so
$L_1 \cdot L_2 = \{x_1 \cdot y_1, x_1, \cdot y_2, \dots, x_1 \cdot y_m, x_2 \cdot y_1,$
$x_2 \cdot y_2, \dots, x_2 \cdot y_m, \dots, x_n \cdot y_1, x_n \cdot y_2, \dots x_n \cdot y_m\}$

Definition 1.9 (language and characters concatenation). *Given any language*
$L = \{x_1, x_2, \dots, x_n\}$ *and a set of charactersa* $= \{c_1, c_2, \dots, c_m\}$, *the concatenation of the language and the set of character $L \cdot L$ is defined as the set of the concatenation of all the possible couples of a string in L and a character in a, so* $L \cdot a = \{x_1 \cdot c_1, x_1, \cdot c_2, \dots, x_1 \cdot c_m, x_2 \cdot c_1,$
$x_2 \cdot c_2, \dots, x_2 \cdot c_m, \dots, x_n \cdot c_1, x_n \cdot c_2, \dots x_n \cdot c_m\}$

Example 1.4. *Given $L_1 = \{a, b\}$ and $L_2 = \{c, d\}$, there is: $L_1 \cdot L_2 = \{ac, ad, bc, bd\}$.*

Also in this case there can be defined the operation of concatenating a language with itself.

Definition 1.10 (language concatenation power). *Given a language L and an integer i, the language L^i is defined as:*

- $L^0 = \emptyset$;
- $L^i = L^{i-1} \cdot L$.

Example 1.5. *Given a language $L = \{ab, cd\}$, there is: $L^0 = \{\}$;*
$L^1 = \{ab, cd\}$;
$L^2 = \{abab, abcd, cdab, cdcd\}$;
$L^3 = \{ababab, ababcd, abcdab, abcdcd, cdabab, cdabcd, cdcdab, cdcdcd\}$.

Definition 1.11 (character concatenation power). *Given a character a and an integer i,* *the language a^i is defined as:*

- $a^0 = \emptyset$;
- $a^i = a^{i-1} \cdot a$.

Given a set of characters V and an integer i, the language V^i is defined as:

- $V^0 = \{\varepsilon\}$;
- $V^i = V^{i-1} \cdot V$

An important operation present in the literature and that can be introduced is the Kleene Star. Presented by the computer scientist Stephen Kleene, it is a unary operator that works on sets of characters or strings.

Definition 1.12 (Kleene Star)**.** *Given a set of strings (characters) V, its Kleene star is the set of strings defined as $V^* = \bigcup_{i \in \mathbb{N}} V^i$.*

So, if V is a set of characters, then V^* is the set of all possible strings containing only characters of V; else if V is a language, V^* contains all the strings obtained by concatenating any number of times any element of V. It is trivial to extend the Kleene Star to be applied over a string. For example, given the string s, the notation s^* is considered to be a short-cut for $\{s\}^*$.

Example 1.6. *Consider the set of character $\{a, b\}$, then:*
$\{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, \ldots\}$
Consider a string ab, then:
$ab^* = \{\varepsilon, ab, abab, ababab, abababab, ababababab, \ldots\}$
Consider a set of strings $\{ab, cd\}$, then:
$\{ab, cd\}^* = \{\varepsilon, ab, cd, abab, abcd, cdab, cdcd, ababab, ababcd, abcdab, abcdcd, \ldots\}$

The Kleene star operator relates to any set V a language containing the empty string. To exclude the empty string from the language generated from the Kleene star, another operator has been introduced: $V^+ = V^* \cdot V$, which is equivalent to $V^+ = V^* \setminus \{\varepsilon\}$.

Given an alphabet Σ, Σ^* is the same set as the free monoid over Σ with respect to the concatenation. A language L is said to be defined on Σ if it is a subset of Σ^*: $L \subseteq \Sigma^*$.

Being languages defined as sets of strings, there can be extended also to them the operations proper of languages.

Definition 1.13 (Set/Boolean Operations)**.** *The set/boolean operation are defined as:*

Intersection *Given two languages L_1 and L_2, we define $L_1 \cap L_2$ as the set containing all the strings belonging to both languages.*

Union *Given two languages L_1 and L_2, we define $L_1 \cup L_2$ as the set containing all the strings belonging either to L_1 or L_2.*

Negation *Given a language L we define as $\neg L$ the set of strings not belonging to the language L.* [2]

Later on, we will refer to set operations also as boolean operations. Please note that it is valid the De Morgan theorem also for the languages:

Theorem 1.1 (De Morgan)**.** *Given any two languages $L_1, L_2 \subseteq \Sigma^*$, the following is true:* $L_1 \cap L_2 = \neg_\Sigma(\neg_\Sigma L_1) \cup (\neg_\Sigma L_2))$

1.1.2. Grammar

Grammars are a way to represent languages in a simpler manner with respect to enumerating all the strings belonging to them. he idea behind grammar is to provide rules guiding the construction of strings.

Definition 1.14 (Grammar)**.** *A (non restricted) Grammar G is a quadruple $G = \langle V_T, V_N, P, S\rangle$ where:*

- *V_T is a finite set of terminal characters, called terminal alphabet;*
- *V_N is a finite set of non terminal characters such that $V_T \cap V_N = \emptyset$, and it is called non terminal alphabet. We will indicate with V the set $V_T \cup V_N$;*
- *P is a finite subset of $V_N^+ \times V^*$, called production set of G. An element $p = \langle \alpha, \beta \rangle \in P$ will be represented as $\alpha \to \beta$. The string α is the left part (lhs) of p while the string β is the right part (rhs) of p. p is also called "rule";*
- *S is a particular character of V_N, and is called axiom or initial character.* [3]

Grammars, thus, are composed of two alphabets, a language, and a character. Let us present some examples of what is and what is not a grammar:

Example 1.7. *Let us define two alphabets: $V_N = \{S, A, B, C\}$ $V_T = \{a, b, c\}$. As stated in definition 1.14 to define a grammar an initial character is also needed, and S will be*

[2] The negation operation is usually used specifying also the alphabet on which the resulting language will be defined, so, if we want the final language to be defined on Σ, $\neg L = (\neg L) \cap \Sigma^*$

[3] S can also be a set of characters, however, there is no substantial difference, as we will see.

used, as well as a production set. Consider the following production sets:
$P_1 = \{S \to ASB,\ S \to C,\ A \to a,\ B \to b,\ C \to c\}$
$P_2 = \{S \to ASB,\ c \to C,\ A \to a,\ B \to b,\ C \to c\}$
$P_3 = \{S \to ASB,\ ASB \to C,\ A \to a,\ B \to b,\ AB \to C,\ ACB \to c\}$
$P_4 = \{S \to ASB,\ aSB \to C,\ A \to a,\ B \to b,\ AB \to C,\ ACB \to c\}$
and then the four quadruples:
$Q_1 = \langle V_T, V_N, P_1, S\rangle$
$Q_2 = \langle V_T, V_N, P_2, S\rangle$
$Q_3 = \langle V_T, V_N, P_3, S\rangle$
$Q_4 = \langle V_T, V_N, P_4, S\rangle$
do they define grammars?
Consider them one by one.
Q_1 *is a grammar, because* $P_1 \subset V_N \times V^*$*. Its grammar will be called also* G_1*.*
Q_2 *is not a grammar, because there is the tuple* $\langle c, C\rangle \notin V_N^+ \times V^*$*.*
$P_3 \subset V_N^+ \times V^*$*, thus* Q_3 *is a grammar. Its grammar will be called also* G_3*.*
It is easy to see that $\langle aSB, C\rangle \notin V_N^+ \times V^*$*, so neither* Q_4 *is a grammar.*

Please note that rules can have as a left part both a single non-terminal or a sequence of non-terminals, and there can be more rules with the same left (or right) part.

In order to allow grammars to represent languages, there must be presented the way by which rules can be combined together to produce the strings.

Definition 1.15 (Derivation)**.** *Given a grammar* $G = \langle V_T, V_N, P, S\rangle$*, it is defined as immediate derivation a binary relation on* V^* *noted as* $\underset{G}{\Longrightarrow}$: $\alpha \underset{G}{\Longrightarrow} \beta$ *being* $\alpha = \alpha_1\gamma\alpha_2$*, iff* $\beta = \alpha_1\delta\alpha_2$*, with* α_1*,* α_2 *and* $\delta \in V^*$*,* $\gamma \in V_N^+$*,* $\gamma \to \delta \in P$*. As with the concatenation, there is used the notation* $\overset{k}{\underset{G}{\Longrightarrow}}$, $\overset{+}{\underset{G}{\Longrightarrow}}$, $\overset{*}{\underset{G}{\Longrightarrow}}$*. When there is no risk of misunderstanding, there is dropped the* G *inside the* $\underset{G}{\Longrightarrow}$*.*

With the concept of derivation there is a way to shape the strings; let us do an example:

Example 1.8. *Let us see how to apply the derivation of the strings with the grammar* G_1 *of example 1.7. There will be written over the arrow the rule used in the passage for the sake of understandability:*
$SA \overset{S\to ASB}{\underset{G_1}{\Longrightarrow}} ASBA \overset{S\to ASB}{\underset{G_1}{\Longrightarrow}} AASBBA \overset{ASB\to C}{\underset{G_1}{\Longrightarrow}} ACBA \overset{A\to a}{\underset{G_1}{\Longrightarrow}} aCBA \overset{C\to c}{\underset{G_1}{\Longrightarrow}} acBA \overset{B\to b}{\underset{G_1}{\Longrightarrow}}$
$acbA \overset{A\to a}{\underset{G_1}{\Longrightarrow}} acba$
This same sequence can be rewritten as $SA \overset{7}{\underset{G_1}{\Longrightarrow}} acba$*,* $SA \overset{+}{\underset{G_1}{\Longrightarrow}} acba$ *or* $SA \overset{*}{\underset{G_1}{\Longrightarrow}} acba$*.*
It is trivial to see that $CBACBAAB \overset{*}{\underset{G_1}{\Longrightarrow}} cbacbaab$*.*

Definition 1.16. *Given a grammar* $G = \langle V_T, V_N, P, S\rangle$*, it is said that it generates/recognizes the language* $L(G)$ *defined as:*
$L(G) = \{x \mid S \underset{G}{\overset{*}{\Longrightarrow}} x,\ x \in V_T^*\}$. [4]

As there can be seen, definition 1.16 restricts the language defined by grammars to be defined only on the Terminal alphabet: $L(G) \subseteq V_T^*$. Here are some examples on grammar working:

Example 1.9. *Consider the grammar*
$G_5 = \langle\{0,1\}, \{S\}, P_5, S\rangle$
where
$P_5 = \{S \to 0S,\ S \to 1S,\ S \to 1\}$.
And here are some possible derivations of G_5 *starting from* S*:*
$S \Rightarrow 1$
$S \Rightarrow 0S \Rightarrow 01$
$S \Rightarrow 1S \Rightarrow 11$
$S \Rightarrow 0S \Rightarrow 00S \Rightarrow 001$
$S \Rightarrow 0S \Rightarrow 01S \Rightarrow 011$
$S \Rightarrow 1S \Rightarrow 10S \Rightarrow 101$
$S \Rightarrow 1S \Rightarrow 11S \Rightarrow 111$
It is trivial to see that the language $L(G_5) = \{0,1\}^* \cdot 1$*, so the set of all the odd binary numbers.*
Please, note that $S \underset{G_5}{\overset{*}{\Longrightarrow}} 101$ *and* $101 \in L(G_5)$*, and, even if* $S \underset{G_5}{\overset{*}{\Longrightarrow}} 10S$ *there is* $10S \notin L(G_5)$ *because* $10S \notin V_T^*$.

Example 1.10. *Consider the grammar:*
$G_6 = \langle\{a,b\}, \{S,T\}, P_6, S\rangle$
where
$P_6 = \{S \to aTb,\ S \to ab,\ T \to aSb\}$
Some examples of derivations are:
$S \Rightarrow ab$
$T \Rightarrow aSb \Rightarrow aabb$
$S \Rightarrow aTb \Rightarrow aaSbb \Rightarrow aaabbb$
$ST \Rightarrow abT \Rightarrow abaSb \Rightarrow abaabb$
The language defined by G_6 *is* $L(G_6) = a^{2n+1}b^{2n+1}$*, for example* $aaabbb \in L(G_6)$ *because*

[4] If S is a set, then $L(G) = \{x \mid X \underset{G}{\overset{*}{\Longrightarrow}} x,\ x \in V_T^*, X \in S\}$. Please note that there can be added rules in P $S \longrightarrow X$ fr each element $X \in S$, having a grammar recognizing the same language and having S as a character and not as set.

of $S \xRightarrow[G_6]{*} aaabbb$*, but* $aTb \notin L(G_6)$ *because, even if* $S \xRightarrow[G_6]{*} aTb$ $aTb \notin V_T^*$*, and even* $abaabb \notin L(G_6)$*, because, even if* $abaabb \in V_T^*$ *and* $ST \xRightarrow[G_6]{*} abaabb$*, but there is no way to have* $S \xRightarrow[G_6]{*} abaabb$*.*

Example 1.11. *Consider the grammar* G_1 *of example 1.8: the language* $L(G_1)$ *is equivalent to* $a^n cb^n$*,* $n \geq 0$*.*

To conclude the discourse about grammars, there are a few concepts left to be introduced. The new concepts and results, taken from the literature ([15], [21], [16]), will be used later, but are reported here because they concern grammars.

Definition 1.17 (backward deterministic grammar)**.** *A grammar* $G = \langle V_T, V_N, P, S \rangle$ *is defined backward deterministic (or BD-grammar) if for each* $B \to \alpha, C \to \beta \in P$ *with* $\alpha = \beta$ *implies* $B = C$*.*

Definition 1.18 (blank and context)**.** *A context over an alphabet* Σ *is a string in* $\Sigma^* \cdot \{-\} \cdot \Sigma^*$*, where the character* $'-' \notin \Sigma$ *and is called a blank. The context* α*, with its blank replaced by the string* x*, is denoted by* $\alpha[x]$*.*

Definition 1.19 (equivalent non-terminal)**.** *Two nonterminals* B *and* C *of a grammar* G *are defined equivalent if, for every context,* α*,* $\alpha[B]$ *is derivable exactly in case* $\alpha[c]$ *is derivable (not necessarily from the same axiom).*

Definition 1.20 (useless nonterminal)**.** *A nonterminal* B *is useless if there is no context* α *such that* $\alpha[B]$ *is derivable or* B *generates no terminal string.*

Definition 1.21 (useless terminal)**.** *A terminal* b *is useless if does not occur in any string of* $L(G)$*.*

Definition 1.22 (grammar clean, reduced, BDNF/BDR)**.** *A grammar is clean if has no useless terminal or nonterminal.*
A grammar is reduced if is clean and there are not two equivalent nonterminals.
A BDR grammar, o a grammar in BDNF (Backward Deterministic Normal Form), is a grammar both backward deterministic and reduced.

Theorem 1.2. *Any grammar can be rewritten as a BRD-grammar, and so in BDNF.*

1.1.3. Chomsky classification

In 1956, Noam Chomsky described in [4] a way to categorize languages. This classification relies on 4 families of languages where the more expressive include the simplers. This classification has been studied a lot and there are lots of results linking each family to the equivalent representation via grammar, automata, and logic. As in the literature, also in this work, there will be used Chomsky's classification to relate languages between them.

The family of all non-restricted grammars (definition 1.14) corresponds to Chomsky's *type-0*, which includes all the possible languages; the family of languages recognized by such grammars is called *recursively enumerable*.
By restricting the rules of the production set to be in the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ with $\alpha, \beta \in V_N^*, \gamma \in V^+$, $A \in V_N$, then there can be obtained the *type-1* family of grammars, for which the corresponding family of languages is called *context-sensitive*.
By further restricting the Production set to be contained in $V_N \rightarrow (V_T \cup V_N)^*$, it is described the *type-2* class of grammars.

Definition 1.23 (type-2 and Context-Free languages)**.** *Given a grammar $G = \langle V_T, V_N, P, S\rangle$, if for each rule $\alpha \rightarrow \beta \in P$ we have that $|\alpha| = 1$ (so $\alpha \in V_N$), then G will be a type-2 grammar, and the language that it recognizes will be classified as context-free.*

A notable result is that this class of languages corresponds to the one recognizable from non-deterministic pushdown automata.
There can be one last restriction of expressiveness of grammar in order to define the so-called *type-3*.

Definition 1.24 (type-3 and Regular languages)**.** *Given a grammar $G = \langle V_T, V_N, P, S\rangle$, if for each rule $\alpha \rightarrow \beta \in P$ we have that $|\alpha| = 1$ (so $\alpha \in V_N$), and $\beta = aB$ or $\beta = a$, with $a \in V_T$, and $B \in V_N$, then G will be a type-3 grammar, and its language will be called Regular Languages.*

A trivial point is that the classes of languages are contained one in the other, so the *type-0* family contains the *type-1*, that contains the *type-2*, that contains *type-3*, as illustrated in fig. 1.1.
The relation between the family of grammars and the corresponding family of languages is bidirectional, so there can be interchangeably referred to both of them, and the results for one will be valid also for the other.

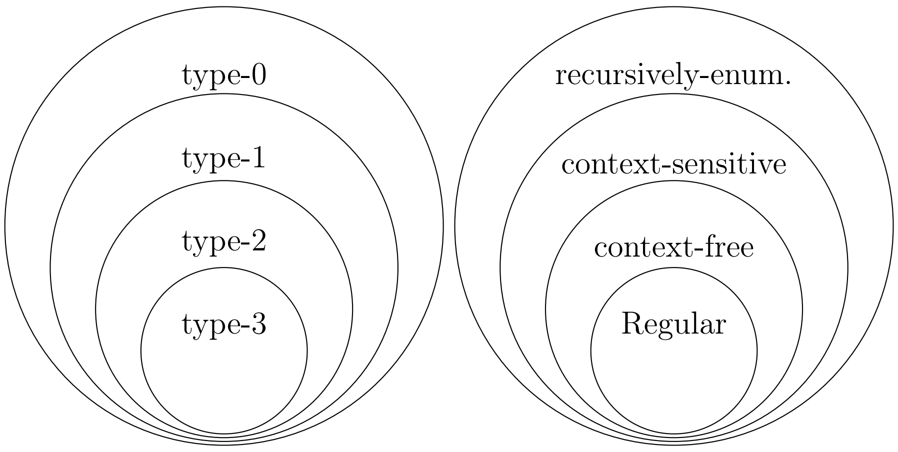


Figure 1.1: Venn diagram of Chomsky's grammar and languages families

1.1.4. Logic characterization

Among the different characterizations that have been used for languages, the symbolic logic description is both one of the first discovered and studied, and less explicit. The Symbolic Logic Characterization is not exposing directly the structure of the strings recognized by the language, however, it expresses some of the features the recognized strings must have. The main idea behind this type of characterization relies on providing a frame over which to evaluate logical formulae by binding the predicates to the string in the study. Given this frame, the evaluation of the logic formulae and their expressiveness follows the known results of logic reported, for example, in [18]. Although the symbolic logic characterization of languages may, at a first seen, seem a secondary appendix of logic, thanks to the Church-Turing thesis, it plays a central role in defining what is and what is not computable. In fact, if something is computable, it can be computed by a Turing Machine (or via $\lambda - calculus$), which has the expressive power of Chomsky's type-0 family of languages; thus not only there is some interest in binding sets of logic expressions and languages, but there exists a homomorphism mapping each computable formalism into a language, and the problem solved by the formalism into the problem of deciding whether or not a string belongs to a certain language.

Being symbolic logic a too vast field and being the goal of this section to give an overview of the literature-studied tools to describe languages, there will be presented only how First Order Logic can be used to represent languages, giving a hint on how the binding from strings to logic formulae can be done.

In order to characterize languages using *First-Order Logic* (FOL), there must be defined the concept of formula:

Definition 1.25 (FOL formulae). *Being P a unary predicate taken from a finite set of atomic propositions and $x, y, \dots$ variables representing positions in strings, are defined as atomic formulae $P(x)$ and $x < y$. Atomic formulae are formulae.*

Given two FOL formulae ϕ, ψ, then:

- *$\neg\phi$ is a valid FOL formula, which is true iff ϕ is not ture;*
- *$\phi \vee \psi$ is a valid FOL formula, which is true iff ϕ or ψ (or both) is true;*
- *$\exists x \phi$ is a valid FOL formula, which is true iff there exists a value for x such that ϕ is true.*

Nothing else is a valid FOL formula.

It is trivial from this definition to add the derived formulae:
$x > y = y < x$,
$\phi \wedge \psi = \neg((\neg\phi) \vee (\neg\psi))$,
$\forall x \phi = \neg\exists x \neg\phi$ and $x = y = \neg((x < y) \vee (y < x))$[5].

In the given definition, the only binding between a formula and a string is with the variables indicating the string's positions. To add the expressiveness needed to differentiate between characters present in a given position, there should be used the predicates, in particular:

Definition 1.26 (unary predicates). *Let Σ be a finite alphabet, then for each $a \in \Sigma$ and for each predicate P, then the truth-value $P(a)$ must be well-defined. There will be indicated as P_a a particular predicate such that $P_a(b)$ is true iff $a = b$.*
Given a string where at the x position there is the character $a \in \Sigma$, then $P(x)$ in the formula has the value of $P(a)$.

Now there can be introduced the way by which there is binded to a Logic Representation the corresponding Language:

Definition 1.27 (sentence). *A sentence is a logic formula without free variables.*

Definition 1.28 (language). *A (First Order) sentence defines a subset of Σ^*, thus a language.*

Example 1.12 ($a \cdot \Sigma^*$). *Let us define the language of strings starting with a:*
$\exists x \forall y P_a(x) \wedge x \leq y$

[5]this is valid because $x, y \in \mathbb{N}$, which is an ordered set

Example 1.13 (a followed by b). *Let us define a language on which each occurrence of a is immediately followed by an occurrence of b:*
$\forall x \neg P_a(x) \vee \exists y x < y \wedge \forall z \neg (x < z \wedge z < y)$

Example 1.14 ($\Sigma^* \cdot b$). *Let us define the language of strings ending with b:*
$\exists x \forall y P_b(x) \wedge x \geq y$

Example 1.15 ($(ab)^+$). *Let us define the language $(ab)^+$:*
$(\exists x \forall y P_a(x) \wedge x \leq y) \wedge$
$(\forall x \neg P_a(x) \vee \exists y x < y \wedge \forall z \neg (x < z \wedge z < y)) \wedge$
$(\exists x (\neg P_b(x) \vee \neg (\exists z x < z) \vee \exists y (P_a(y) \wedge y > x \wedge \neg \exists z (x < z \wedge z < y)))) \wedge$
$(\exists x \forall y P_b(x) \wedge x \geq y)$

More pieces of information about symbolic logic characterization can be found in most theoretical computer science like [21], or, with particular attention to FO logic, on [9].

1.2. Regular Languages

In this section is presented the family of *Regular Languages* with its definition, properties, characterizations, and major results. This family corresponds to Chomsky's *type-3*, which is the less expressive class of the classification. Because of the simplicity of the representation of this family and expressiveness sufficient to describe lots of real-world problems, there have been lots of studies about this class, and thus there are lots of results.

In this section is also presented the class of Non-Counting (or Aperiodic) languages.

1.2.1. Regular Languages and Regular Expression

There have been many results in the literature about Regular Language expressiveness and about ways o represent its languages. Previously the concept of grammar has been defined, and via the definition 1.24 has been presented how to relate grammars to Regular Languages. It is known that the family of Regular Languages corresponds to the class of languages recognized by *Finite State Automata*. The representation of RL via *Finite State Automata*, or FSA, is used for example to prove the *Pumping Lemma* and the results on decidability that are presented later.

Among the different representations roposed, however, let us now introduce the *Regular Expression* one.

Definition 1.29 (Regular Expression). *Given an alphabet Σ of terminal characters, the*

Regular Expressions (RE) built on it are defined with the following rules:

1. *$\emptyset$ is a Regular Expression with as recognized language the empty set $\emptyset = \{\}$;*
2. *$\forall c \in \Sigma$, c is a Regular Expression. The language denoted by the RE c is $\{c\}$;*
3. *Being R_1 and R_2 two Regular Expressions, also $R_1 \cup R_2$ is a Regular Expression, indicated also as $R_1 + R_2$; its language is the union of the two languages denoted by R_1 and R_2;*
4. *Being R_1 and R_2 two Regular Expressions, also $R_1 \cdot R_2$ is a regular expression, indicated also as R_1R_2; its language is the concatenation of the two languages denoted by R_1 and R_2;*
5. *Being R a Regular Expression, also R^* is a regular expression; the language denoted by R^* is the result of the Kleene star operator applied to the language R (so $R^0 = \emptyset$ and $R^* = \bigcup_{i\in\mathbb{N}} R^{i-1} \cdot R$);*
6. *Nothing else is a Regular Expression.*

Example 1.16. *Given the alphabet $\Sigma = \{B, C, P, S\}$, let us define some Regular Expressions and their languages:*

1. *$R_b = B$, with its language that is: $\{B\}$;*
2. *$R_c = C$, with its language that is: $\{C\}$;*
3. *$R_p = P$, with its language that is: $\{P\}$;*
4. *$R_s = S$, with its language that is: $\{S\}$;*
5. *$R_{b^*} = R_b^* = B^*$, with its language that is: $\{\varepsilon, B, BB, BBB, \ldots\}$;*
6. *$R_{b^+} = R_b \cdot R_b^* = B \cdot B^*$, with its language that is: $\{B, BB, BBB, BBBB, \ldots\}$;*
7. *$R_{b^+c} = R_{b^+} \cdot R_c = B \cdot B^* \cdot C$, with its language that is: $\{BC, BBC, BBBC, BBBBC, \ldots\}$;*
8. *$R_{(b^+c)^*} = R_{b^+c}^* = (B \cdot B^* \cdot C)^*$, with its language that is: $\{\varepsilon, BC, BCBC, BCBCBC, BBC, BBCBC, BCBBC, \ldots\}$;*
9. *$R_{p(b^+c)^*} = R_p \cdot R_{(b^+c)^*} = P \cdot (B \cdot B^* \cdot C)^*$, with its language that is: $\{P, PBC, PBCBC, PBCBCBC, PBBC, PBBCBC, PBCBBC, \ldots\}$;*

10. *$R_{p(b+c)^*s} = R_{p(b+c)^*} \cdot R_s = P \cdot (B \cdot B^* \cdot C)^* \cdot S$, with its language that is:* $\{PS, PBCS, PBCBCS, PBCBCBCS, PBBCS, PBBCBCS, PBCBBCS, \dots\}$*;*
11. *$R_{(p(b+c)^*s)^*} = R^*_{p(b+c)^*s} = (P \cdot (B \cdot B^* \cdot C)^* \cdot S)^*$, with its language that is:* $\{\varepsilon, PS, PSPS, PSPSPS, PBCS, PSPBCS, PSPBCSPS, PBCBCS, PBCBCSPS, PSPBCBCS, PSPBCBCSPS, \dots\}$*;*

Given this definition, it is easy to extend it, without adding expressiveness to it, with the use of the following operations: being R a Regular Expression, $R^+ = R \cdot R^*$ and R^n such that $R^0 = \emptyset$ and $R^i = R \cdot R^{i-1}$.

Example 1.17. *Please note that in example 1.16 the Regular Expression R_{b^+} can be redefined as $R_b^+ = B^+$ without impacting any language of the example.*

definition 1.29 is the one presented by Kleene in 1956 [1], where he presented the Kleenee's star operator (definition 1.24), and proposed the definition for the class of Regular Languages. This class of languages, however, has been proven to coincide with the one defined via the Chomsky's type-3 family in [17]. In particular, one major result from the equality of definition 1.29 and definition 1.24 is that the regular expressions are closed also with respect to negation over the universe Σ^*. So there can be extended the definition with the use of the addition:

- Being R a regular expression over Σ, also $\neg_\Sigma R$ is a regular expression. The language denoted by $\neg_\Sigma R$ is the set of all strings $s \in \Sigma^*$ such that $s \notin R$.

Please note that with addition, there can also be expressed inside the Regular Languages the operation $R_1 \cap R_2 = \neg(\neg R_1 \cup \neg R_2)$.

A peculiarity of complementation is that it is relative to the alphabet, so if R is a regular expression over either alphabet Σ_1 and Σ_2 (so $R \subseteq (\Sigma_1 \cap \Sigma_2)^*$), it is not always true that $\neg_{\Sigma_1} R \subseteq \Sigma_2^*$, in particular, it is true if $\Sigma_1 = \Sigma_2$.

Definition 1.30 (Regular Language)**.** *The class of Regular Languages (RL) is the smallest class of languages containing all finite sets of strings and closed under the concatenation, the star operator, and the boolean operations* [6]*over its alphabet Σ.*

This definition matches the one proposed in [20].

Example 1.18. *Consider the alphabet $\Sigma = \{a, b\}$ and the regular expressions:*
$R_1 = a = \{a\}$

[6]The closure under boolean operations is the part missing in definition 1.29 but is sufficient to add the closure under the negation operation to have the closure under all boolean operations.

$R_2 = b = \{b\}$
$R_3 = R_1 \cup R_2 = a \cup b = \{a, b\}$
$R_4 = R_3^* = (a \cup b)^* = (a + b)^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \ldots\}$
$R_5 = R_1 \cdot R_4 = a \cdot (a \cup b)^* = a(a+b)^* = \{a, aa, ab, aaa, aab, aba, abb, \ldots\} = \textit{Strings starting with a}$
$R_6 = R_4 \cdot R_1 = (a \cup b)^* \cdot a = (a+b)^* a = \{a, aa, ba, aaa, aba, baa, bba, \ldots\} = \textit{Strings ending with a}$
$R_7 = R_5 \cup R_6 = ((a \cdot (a \cup b)^*) \cup ((a \cup b)^* \cdot a)) = a(a+b)^* + (a+b)^* a =$
$\{a, aa, ab, ba, aaa, aab, aba, abb, baa, bba, \ldots\} = \textit{Strings starting or ending with a}$
$R_8 = R_5 \cap R_6 = ((a \cdot (a \cup b)^*) \cap ((a \cup b)^* \cdot a)) = a(a+b)^* \cap (a+b)^* a =$
$\{a, aa, aaa, aba, aaaa, aaba, abaa, abba, \ldots\} = \textit{Strings starting and ending with a}$

There is a particular subclass of RL: the Star Free. This family, presented here, will be later recalled to prove its properties. It can be defined as a reshaping of the definition 1.30 to limit its expressive power.

Definition 1.31 (Star Free)**.** *The class of Star Free languages (SF) is the smallest class of languages containing all finite sets of strings and closed under the concatenation and the boolean operations.*

The difference between definition 1.30 and definition 1.31 is that in the second case, as the name suggests, there is no closure with respect to the Kleene's operator $*$, and thus neither$^+$. Even if it may seem a small change with respect to the original RL definition, there will be presented how this change in language closures impacts a lot also the "structure" [7] of languages it will recognize. Let us do a few examples of SF languages over the alphabet $\Sigma = \{a, b, c\}$.

Example 1.19 ($\emptyset$)**.** *The language $\emptyset$ can be expressed is SF:*
$A = \{a\} \in SF$
$B = \neg A \in SF;$
$\emptyset = A \cap B \in SF.$

Example 1.20 (Σ^*)**.** *Given the previous example, then:*
$A = \emptyset \in SF;$
$B = \neg A \in SF;$
And thus, being $B = \neg\emptyset$, it is trivial to see that $B = \Sigma^*$.

Example 1.21 (Not containing substring)**.** *Given a string s, there can be defined the SF language of all the strings on Σ not having s as substring:*

[7] Not to be intended in the strict sense because there isn't the concept of structure.

$A = \Sigma^* \in SF$;
$B = \{s\} \in SF$;
$C = A \cdot B \cdot A \in SF$;
$D = \neg C \in SF$.
And it is trivial to see that D is the language of all the strings not containing s as a substring.

Example 1.22 (Not containing language)**.** *Given an SF language η, there can be defined an SF language of all the strings on Σ not having as a substring an element of η :$A = \Sigma^* \in SF$;*
$B = A \cdot \eta \cdot A \in SF$;
$C = \neg B \in SF$.
It is trivial to see that the language C is the language we were searching for.

There are now reported different examples of languages that can be expressed using Kleene's Star and there will be discussed whether those languages are SF or not. The languages presenter will be: a^*, $(ab)^*$, $(aba)^*$, $(aa)^*$ and $(abab)^*$.

Example 1.23 (a^*)**.** *As can be seen, a^* corresponds to the language of all the strings not containing b or c, so there can be expressed $a^* = neg(\Sigma^*\{b, c\}\Sigma^*)$, and, being $\{b, c\} \in SF$, then also a^* is SF.*
To uniform this example with the next ones, let us define $\alpha = \{b, c\}$, which is an SF language, and then:
$a^* = \neg(\Sigma^*\alpha\Sigma^*) \in SF$.

Example 1.24 ($(ab)^*$)**.** *As done in example 1.23, let us consider $(ab)^*$ as the set of all the strings not containing any substring not accepted as a substring of $(ab)^*$. To do so, the language will not recognize any string containing:*
$\alpha = \Sigma^2 \setminus \{ab, ba\} = \{aa, ac, bb, bc, ca, cb, cc\}$ which is SF.
Moreover, there is the needing to have the accepted string to start with ab and to end with ab, not accepting, for example, baba, so:
$(ab)^* = \neg(\Sigma^*\alpha\Sigma^*) \cap (\{\varepsilon\} \cup (\{ab\}\Sigma^* \cap \Sigma^*\{ab\}))$.

Example 1.25 ($(aba)^*$)**.** *Let us expand the idea of example 1.24 to also cover $(aba)^*$: the language must accept only strings having all substrings in $\{aba, baa, aab\}(\in SF)$, thus it will not accept any string with as substring any of*
$\alpha = \Sigma^3 \setminus \{aba, baa, aab\} =$
$\{aaa, aac, abb, abc, aca, acb, acc, bab, bac, bba.bbb, bbc,$

$bca, bcb, bcc, caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc\}$ which is SF.
Moreover it has to accept the strings starting with aba, and ending with aba, so:
$(aba)^* = \neg(\Sigma^*\alpha\Sigma^*) \cap (\{\varepsilon\} \cup (aba\Sigma^* \cap \Sigma^* aba))$.

Example 1.26 ($(aa)^*$ and $(abab)^*$)**.** *In the previous examples, there has been seen how to express the languages $a^*, (ab)^*, (aba)^*$ in a star free manner, and the way it has been done, was quite always the same, however, there cannot be expressed $(aa)^*$ or $(abab)^*$ in the same way, because there is no way to "count" the number of repetitions of a or ab. For example, consider the language:*
$\eta = \neg(\Sigma^*\{ab, ac, ba, bb, bc, ca, cb, cc\}\Sigma^*) \cap (\{\varepsilon\} \cup (\{aa\} \cdot \Sigma^* \cap \Sigma^* \cdot \{aa\}))$
It is trivial to see that η is a language that recognizes $a \cdot (a^+) \cup \varepsilon$, or, equivalently, $a^* \setminus \{a\}$; however it is the language built in the same way of the previous ones but over the language "$(aa)^*$. There is no provided here a proof of the fact that this language is not Star Free, however, there has been proved that the same method previously adopted is not suitable for this case. Later in the work, there are reported results that will make the proof of this language not to be Star Free trivial.

1.2.2. Decidability

Here are reported the main results regarding Regular Languages with respect to Decidability. So there are reported problems that are known to be decidable. As the first thing, however, the concept of decidable must be introduced.

Definition 1.32 (decidable)**.** *A problem is said to be decidable if it is computable.*

Theorem 1.3. *Given a regular language $\eta \subseteq \Sigma^*$ and a string $s \in \Sigma^*$, it's decidable whether or not s belongs to the language.*

This result, however, can be further improved by another important result: the *Pumping Lemma*

Theorem 1.4 (Pumping Lemma)**.** *Given a Regular Language η, there exists a number $k \in \mathbb{N}$ such that for each $x \in \eta$ and $|x| \geq k$, then there exists $y, w, z \in \Sigma^*$ such that $1 \leq |w| \leq k$ and $yw^iz \in \eta$ for each $i \geq 0$.*

Corollary 1.1. *Given an RL language η, there exists a finite set of strings $S \subseteq \bigcup_{0 \leq i \leq k} \Sigma^i$ such that $S \subseteq \eta$. In particular, if $S = \emptyset$, then also $\eta = \emptyset$.*

Proof. This is a corollary of theorem 1.4, and k has the same value it has in it.

The fact that S is finite derives from the fact that also $\bigcup_{0\leq i\leq k}\Sigma^i$ is finite.

Assume, for absurdum, that S is empty (so that $\eta\cap\bigcup_{0\leq i\leq k}\Sigma^i)=\emptyset)$ and that $\eta\neq\emptyset$. In that case, there exists $s\in\eta$ with $|s|\geq k$ and $s=ywz$ with $|w|>1$. For the Pumping Lemma, also $yz\in\eta$, and $|yz|<s$. From this it can be seen that the hypothesis of having $S=\emptyset$ and $\eta\neq\emptyset$ is absurdum because either $|yz|<k$ or there can be iteratively applied the Pumping Lemma to yz. □

Corollary 1.2. *It's decidable if a regular language is empty or not.*

Corollary 1.3. *Given two regular languages η_1,η_2, then it is decidable whether or not $\eta_1\cap\eta_2=\emptyset$.*

The last two results are trivial consequences of the previous ones.

All the information about decidability can be found in [21].

1.2.3. Non Counting

The family of Non-Counting languages naturally belongs to the class of Regular Languages. However, given its importance and the theoretical results that it has, the concept of Non-Countingness (or Aperiodicity) has been expanded also to the other Chomsky's classification families. The feature of Non-Countingness can be found, for example, in natural languages, and for this reason, this family has been studied and used in the natural and programming languages fields.

The concept of non-countability informally refers to the fact of counting the occurrences of substrings, as hinted in example 1.26, thus there can be defined 2 ideas for "countability":

threshold A language accepts a string iff it contains more than n repetitions of a certain substring s;

modulo A language accepts a string iff it contains a number i of repetitions of a certain substring s such that $i \mod n = K$, where n, K are fixed for the language.

Given the ideas of the 2 types of "countability", the class of Non-Counting languages only refers to the second idea (the "modulo" one), ignoring the threshold ones. In particular, the definition of this class is:

Definition 1.33 (Non Counting). *Given a language $\eta\subseteq\Sigma^*$, η is called Non Counting if there exists a number n such that for all $x,y,z\in\Sigma^*$, $m\in\mathbb{N}$ we have that $x\cdot y^n\cdot z\in\eta\Leftrightarrow$*

$x \cdot y^{n+m} \cdot z \in \eta$.
If η is not Non-Counting, then it is Counting.

Here are reported some examples of either counting or non-counting languages:

Example 1.27. *Consider the languages $(aa)^*$ and $(abab)^*$ discussed in example 1.26: those languages are counting because they accept an even number of repetitions of a or ab. Consider $\eta = (aa)^*$ and the string $(aa)^n \in \eta$; we can easily see that $(aa)^n = a^n a^n$, and, setting $x = \varepsilon,\ y = a,\ z = a^n$, we can see that $xy^n z \in \eta$, but, for example, $xy^{n+1}z = a^{n+1}a^n = a^{2n+1} \notin \eta$, so η is not NC, thus it is Counting. Similar reasoning can be done for the language $(abab)^*$.*

Example 1.28. *Here is an example based on real life: Consider a coffee machine that can be powered up and switched off, that has a button that can be pressed to require coffee, and, most important, can deliver coffee.*
We want to find a language describing the series of actions that bring the machine from being switched off to being switched off again, assuming that it works properly, thus:

- *it is possible to press the button and deliver coffee only when it is powered up;*
- *the machine will deliver coffee after one pressure of the button and will ignore all the other pressures after the first one and before the corresponding delivery of coffee;*
- *the machine will not be switched off between the pressure of the button and the delivery of coffee.*

There will be used as alphabet $\Sigma = \{P, S, B, C\}$ where

- *P stands for power up;*
- *S stands for shut down;*
- *B stands for button pressed;*
- *C stands for coffee delivered.*

This language will recognize all the strings that can be recognized by $\eta = (P((B)^+C)^*S)^*$. For now, the fact that η is NC is left with no demonstration, and the reader can do some tests with different strings to check it, however, in a few pages, there will be provided some tools to make this proof in at least 2 different ways.

Consider $x = P,\ y = BBBC,\ z = S$, there can be seen that $xy^n z \in \eta$ for each value of n.

Please note that this language is the same one described by the final Regular Expression of example 1.16.

Example 1.29. *We want a language on alphabet $\Sigma = \{0,1\}$, that accepts all the strings that contain at least 3 occurrences of* 1. *It can be exemplified by $\eta = \Sigma^* \cdot 1 \cdot \Sigma^* \cdot 1 \cdot \Sigma^* \cdot 1 \cdot \Sigma^*$. As can be trivially seen, this language is SF, even if the language is "counting" in the threshold manner. It is trivial to see that this language is Non-Counting with the use of* 3 *as the value of n inside the definition 1.33.*

The following result will expand the definition 1.33 making it easier to demonstrate whether a language is counting or non-counting

Lemma 1.1. *Given a language $\eta \in \wp(\Sigma^*)$ and an integer n, the following statements are equivalents:*

A $\forall_{m\in\mathbb{N},\, x,y,z\in\Sigma^*} xy^nz \in \eta \Leftrightarrow xy^{n+m}z \in \eta$

B $\forall_{x,y,z\in\Sigma^*} xy^nz \in \eta \Leftrightarrow xy^{n+1}z \in \eta$

Proof. To prove the lemma, there will be firstly proven that A implies B.
It is trivial to see that if A is valid, then also B is valid as a special case of A.
Let us now prove that B implies A.
Suppose B is valid and consider 2 strings xy^nz, $xy^{n+1}z\Box\eta$, where $\Box$ is either $\in$ or $\notin$. Let now $p = xy$, we have that $py^nz = xy^{n+1}z\Box\eta$, and, having B valid, it is also $py^{n+1}z\Box\eta$, but $py^{n+1}z = x^{n+2}z$, and with this same process, there can be proven that also $xy^{n+3}z\Box\eta$, and so on, till any m. Thus B implies A. $\Box$

This lemma is helpful to limit the number of iterations needed to demonstrate whether a language is Counting or not.

Another particular feature is that there are lots of results linking the family of NC languages to other families.

Theorem 1.5. *The family of NC languages corresponds to the family of Star Free languages.*

The previous result explains why there has been reported the family of Star Free languages.

Theorem 1.6. *The family of NC languages corresponds to the family of languages recognized by an automaton with finite and aperiodic transition monoid.*

For further information about the topic, refer to [20].

There can be now provided the proof that the language expressed in example 1.28 is NC by leveraging theorem 1.5:

Example 1.30. *The language to analyze is:* $\eta = (P((B)^+C)^*S)^*$.

To prove that η *is SF, we will proceed in a similar way as done in example 1.25, so enumerating all the tuples of 4 characters we will accept:*
$\alpha = \Sigma^4 \setminus \{$
$BBBB, BBBC, BBCB, BBCS, BCBB, BCBC, BCSP, CBBB, CBBC,$
$CBCB, CBCS, CSPB, CSPS, PBBB, PBBC, PBCB, PBCS, PSPB,$
$PSPS, SPBB, SPBC, SPSP\}$
Now let us define the language in a SF manner: $\eta = \neg(\Sigma^* \cdot \alpha \cdot \Sigma^*) \cap (\{SP\} \cup \{S\} \cdot \Sigma \cdot \Sigma^* \cdot \Sigma \cdot \{P\})$.

Please note that the examples of SF languages are also examples of NC languages.

From those examples, it can be noticed that there is somehow a correlation between Star Freeness and the enumeration of substrings accepted or not. This concept is going to be reinforced later in chapter 2 with the family of locally testable languages.

Another remarkable result is that the class of languages definable via First Order Logic corresponds to the class of NC languages, so:

Theorem 1.7. *The family of NC (without considering the* ε *string) languages corresponds to the family of languages described by the First Oder Logic.*

For more information on the relation between FO and NC refer to [9], or for more pieces of information on NC in general, refer to [20].

2 | Locally Testable Languages

In this chapter the Locally Testable family of languages is presented. This family has been proven to be equivalent to the Non-Counting one, and for this reason it is presented here. This class is predent in the literature, and more information can be found in [20], however, this class of languages is not as widely known as the concepts presented so far.

The Locally Testable family of languages has been chosen as reference for the research of a Non-ounting subclass of Operator PRecedence Languages, not only because of its roperties, but also for the semplicity of its definition. There have been studies relating this family with the Star Free one, the Non-Counting, . . . and can be found in [20].

The aim of this section is to introduce the LTO family of languages and to provide as much knowledge about it as possible, in order to make it easier to enrich this family to build the LTOP one. In order to achieve such a goal, there are presented some concepts used in the definition of the class itself before presenting the main results.

2.1. Some Tools

Here are presented different ideas that will be used later to represent languages.

As the first thing, there are defined some special characters not belonging to a general alphabet Σ. Each one of this characters has a specific role, that will be presented when it will be used.

Definition 2.1 $(\Sigma, \#, \Phi, -)$. *Given any alphabet Σ, let $\#$ be a character such that $\# \notin \Sigma$. With $\Sigma_\#$ will be referred the set $\Sigma \cup \{\#\}$.*
Given any alphabet Σ, let $\lfloor$, $\rfloor$ be two characters such that
$\{\lfloor, \rfloor\} \cap \Sigma_\# = \emptyset$.
There will be used $\Sigma_{par} = \Sigma \cup \{\lfloor, \rfloor\}$.
There will be used $\Sigma_{\#,par} = \Sigma_\# \cup \{\lfloor, \rfloor\}$
Given any alphabet Σ, let $\Phi = \{-_1, -_2, \dots, -_n\}$, Φ is defined as set of blanks for Σ if

$\Sigma \cap \Phi = \emptyset$, and the elements of Φ are called blanks [1].
There will be used $\Sigma_{\Phi} = \Sigma \cup \Phi$.
There will be used $\Sigma_{\#,\Phi} = \Sigma_{\#} \cup \Phi$.
There will be used $\Sigma_{\#,\Phi,par} = \Sigma_{\#,par} \cup \Phi$.
There will be used $\Sigma_{any} \supseteq \Sigma_{\Phi}$ to indicate any set of characters in order to not limitate the definitions.
There will be used $\Sigma_{anyT} = \Sigma_{any} \backslash \{\#, \lfloor, \rfloor\}$ to indicate the set of Σ_{any} without the characters used to build the string.

With definition 2.1 there has been set a common agreement on which there are built, step by step, all the models that are presented in this work from now on.
In this section are used only the # character, and, as done in lots of works, it is used to represent the beginning and the ending of a string. So a string $s \in \Sigma^*$ will be represented, using also #, as $\# \cdot s \cdot \# \in \Sigma^*_{\#}$.

In order to introduce the family of Locally Testable languages, it is useful to previously define two other families:

Definition 2.2 (Finite Language). *A language $\alpha \in \wp(\Sigma^*_{any})$* [2] *is finite if it consists of a finite set of strings (of finite length).*

Definition 2.3 (Definite Language). *A language $\eta \in \wp(\Sigma^*_{any})$ is said to be definite if there exist two finite languages $\alpha, \beta \in \wp(\Sigma^*_{any})$ such that $\eta = \Sigma^*_{any} \cdot \alpha \cup \beta$.*

In other words, a language η is definite if there exists a number k such that for any string it is sufficient to check the last k characters of it to check whether the string belongs or not to the language. In particular, k depends on the length of the strings in the finite languages (α and β in the definition 2.3) defining η. Let us do a few examples to clarify those classes of languages:

Example 2.1. *Nick and Raul play table football, knowing that at any time Valentina will stop their game, they choose to apply the rule of "queen ball": the last to score a goal will win the match. Knowing the sequence of goals, a definite language will be able to recognize if the winner is Raul or Nick just by looking at the last goal.*

Example 2.2. *We want to define a language that recognizes strings that represent binary numbers for which the modulo 4_{dec} operation gives 3_{dec} as result, or the modulo 64_{dec}*

[1] Please note that this definition can be seen somehow as an extension of definition 1.18.
[2] The $\wp(X)$ notation is used to identify the "set of the parts of X": The set of all the possible subsets of X^*.

operation gives 30_{dec} *as result, or that are exactly one between* $1_{dec}, 5_{dec}$ *and* $_{dec}$*.*
Translating the requests from the decimal world into the binary one, those are equivalent to ask that:

1. *The modulo* 4_{dec} *operation gives* 3_{dec} *as result* $\Rightarrow$ *The binary representation of number ends with* 11*;*
2. *The modulo* 64_{dec} *operation gives* 30_{dec} *as result* $\Rightarrow$ *The binary representation of number ends with* 11110*;*
3. *The number is* 1_{dec} $\Rightarrow$ *The binary representation of number is* 1*;*
4. *The number is* 5_{dec} $\Rightarrow$ *The binary representation of number is* 101*;*
5. *The number is* 32_{dec} $\Rightarrow$ *The binary representation of number is* 1000*.*

So, over an alphabet $\Sigma = \{0, 1\}$ *we can define the 2 finite languages* $\alpha = \{11, 11110\}$ *and* $\beta = \{1, 101, 1000\}$ *to finally build our language* $\eta = \Sigma^* \cdot \alpha \cup \beta$*.*
Let us analyze some strings and see if they belong to the language η *or not:*

- $101010111 \Rightarrow$*it ends with* $11 \in \alpha \Rightarrow 101010111 \in \eta$
- $0011011110 \Rightarrow$*it ends with* $11110 \in \alpha \Rightarrow 0011011110 \in \eta$
- $1101101101 \Rightarrow$*it does not belong to* β *and does not end with any element of* $\alpha \Rightarrow 1101101101 \notin \eta$
- $101 \Rightarrow$*it belongs to* $\beta \Rightarrow 101 \in \eta$
- $100 \Rightarrow$*it does not belong to* β *and does not terminate with any element of* $\alpha \Rightarrow 100 \notin \eta$

2.2. Locally Testable Language and its Closure

Let us talk about the Locally Testable family of languages. The idea behind its languages, is to expand the check of definite languages not only to the latest k characters of a string, but to split the string into all possible substrings of length k and check directly them. Formally, here is the definition of Locally Testable in the strict sense (LT) languages:

Definition 2.4 (Locally Testable in strict sense)**.** *A language* $\eta \in \wp(\Sigma^*_{any})$ *is defined k-testable if there exists 4 sets*
$\alpha,\ \beta,\ \gamma \in \wp(\Sigma^k_{any})$*,*$\delta = \{x \in \wp(\Sigma^*) \mid |x| < k\}$ *such that, for any string* $s \in \Sigma^*$*,* $s \in \eta$ *iff:*

- $|s| \geq k \rightarrow L_k(s) \in \alpha,\ I_k(s) \subseteq \beta,$ *and* $R_k(s) \in \gamma$*;*

- $|s| < k \rightarrow s \in \delta$.

Where $L_k(s)$ is defined as the left-end segment of s of lenght k, $I_k(s)$ as the set of interior segments of s of lenght k, $R_k(s)$ as the right-end segment of s of lenght k.
If η is k-testable, then also η is defined as Locally Testable in the strict sense (LT).

Example 2.3. *Consider the alphabet $\Sigma = \{a, b, c, d\}$, $k = 2$, and the sets:*
$\alpha = \{ab, ad\}$
$\beta = \{cd, ad\}$
$\gamma = \{bc, cb\}$
$\delta = \{c, b\}$
With these 4 sets, there can be defined the LT_2 language η such that:

- *$c \in \eta$, because $|c| < 2$ and $c \in \delta$;*
- *$ad \in \eta$, because $|ad| \geq 2$ and $ad \in \alpha$ and $ad \in \beta$;*
- *$abcd \in \eta$, because $|abcd| \geq 2$ and $ab \in \alpha$ and $cd \in \beta$ and $\{bc\} \subseteq \gamma$;*
- *$abcbcd \in \eta$, because $|abcbcd| \geq 2$ and $ab \in \alpha$ and $cd \in \beta$ and $\{bc, cb\} \subseteq \gamma$;*
- *$abd \notin \eta$, because $|abd| \geq 2$ and $ab \in \alpha$ but $bd \notin \beta$;*
- *$acd \notin \eta$, because $|acd| \geq 2$ and $ac \in \beta$ but $ac \notin \alpha$;*
- *$abbcd \in \eta$, because $|abbcd| \geq 2$ and $ab \in \alpha$ and $cd \in \beta$ but $\{bb, bc\} \not\subseteq \gamma$.*

Although definition 2.4 is taken from the literature, for our purposes it is better to reshape it in a way a bit more similar to the notation introduced in definition 2.1 to better relate LT family with the OP on:

Definition 2.5. *A language $\eta \subseteq \Sigma^*_{any}$ is said to be k-testable if there exist a number k and a set $Y \subseteq ((\bigcup_{i \in \mathbb{N},\, i<k-1} \# \cdot \Sigma^i \cdot \#) \cup (\# \cdot \Sigma^{k-1} \cup \Sigma^{k-1} \cdot \#) \cup \Sigma^k)$ such that the language η recognises the strings $s \in \Sigma^*$ iff:*

- $|s| \leq k - 2 \rightarrow \# \cdot s \cdot \# \in Y$;
- $|s| > k - 2 \rightarrow$ *each substring of length k of $\# \cdot s \cdot \#$ is in Y.*

If the language η is k-testable, then it's said also to be Locally Testable(LT) in the strict sense.

It is trivial to see that the definition definition 2.5 and the definition definition 2.4 are equivalent, and the sets of the beginning, ending, intermediate or complete substrings are differentiated by the presence (or the position) of the # character.

Example 2.4. *Take the language defined in example 2.3 and transform it into an equivalent form, following definition 2.5, using $k = 3$ and as set defining it Y.*
To "include" δ in Y, then it must be: $\{\#c\#, \#b\#\} \subseteq Y$.
To "include" α in Y, then it must be: $\{\#ab, \#ad\} \subseteq Y$.
To "include" β in Y, then it must be: $\{cd\#, ad\#\} \subseteq Y$.
To "include" γ in Y, then it must be: $\{bcb, cbc\} \subseteq Y$.
Finally, to include the mix of α, β, γ, then it must be $\{abc, bcd\} \subseteq Y$.

The language $\eta = LT_3(Y)$ is such that:

- $c \in \eta$, *because* $|c| \leq 1$ *and* $\#c\# \in Y$;
- $ad \in \eta$, *because* $|ad| > 1$ *and* $\{\#ad, ad\#\} \subseteq Y$;
- $abcd \in \eta$, *because* $|abcd| > 1$ *and* $\{\#ab, abc, bcd, cd\#\} \subseteq Y$;
- $abcbcd \in \eta$, *because* $|abcbcd| > 1$ *and* $\{\#ab, abc, bcb, cbc, bcd, cd\#\} \subseteq Y$;
- $abd \notin \eta$, *because* $|abd| > 1$ *and* $\{\#ab, abd, bd\#\} \nsubseteq Y$;
- $acd \notin \eta$, *because* $|acd| > 1$ *and* $\{\#ac, acd, cd\#\} \nsubseteq Y$;
- $abbcd \in \eta$, *because* $|abbcd| > 1$ *and* $\{\#ab, bbc, bcd, cd\#\} \nsubseteq Y$.

There has been already presented a language that can be transformed in a LT shape: the one of example 1.30.

Example 2.5. *The language of example 1.30 is $\eta = (P((B)^+C)^*S)^*$. It can be defined in an LT manner with $k = 2$ where the set Y is such that:*

- $\varepsilon \in \eta$ *so* $\#\# \in Y$;
- *The strings of η will start with P, so $\#P \in Y$;*
- *The strings of η will start with S, so $S\# \in Y$;*
- *In the strings of η, the character P can be followed by S or B, so $PS, PB \in Y$;*
- *In the strings of η, the character B can be followed by B or C, so $BB, BC \in Y$;*
- *In the strings of η, the character C can be followed by B or S, so $CB, CS \in Y$;*
- *In the strings of η, the character S can be followed by P so $SP \in Y$.*

Said that, the set $Y = \{\#\#, \#P, S\#, PS, PB, BB, BC, CS, CB, SP\}$ will define a language such that $LT_2(Y) = \eta$.

From the given definitions of LT languages, it is trivial to see that there can be multiple sets describing the same language. A question may then arise: is there any way to guarantee that the a language has always the same set representing it?

Example 2.6. *Consider the alphabet $\Sigma = \{a, b, c\}$, $k = 3$ and the 2 sets:*
$A = \{\#ab, abb, bbb, bbc, abc, cbb, bc\#\}$
$B = \{\#ab, abb, bbb, bbc, abc, acb, bc\#\}$
It is simple to verify that $LT_3(A) = LT_3(B)$ and that it is the language recognized by the Regular Expression ab^+c.

Please note that the elements that differ in the two sets are cbb, acb and that they are not substrings of any string recognized by $LT_3(A)$ or equivalently $LT_3(B)$.

To answer the question, the concept of clean$_{\text{LT}}$ is introduced.

Definition 2.6 (clean$_{\text{LT}}$)**.** *Given a set $S \subseteq \bigcup_{i\in\mathbb{N},\, i\le k} \Sigma^i_{any}$, it is said to be clean$_{LT}$ iff:*

- *Removing any element from S, the LT language generated by the new set is different from the one generated by S;*
- *each element of S that contains $\#$, contains it at the beginning and/or at the ending of it.*

Example 2.7. *Consider the two sets of example 2.6; as stated also at the end of the example, they are not clean$_{LT}$. Their cleaned version is:*
$A_{cleaned} = A \setminus \{cbb, acb\} = \{\#ab, abb, bbb, bbc, abc, bc\#\}$
$B_{cleaned} = B \setminus \{cbb, acb\} = \{\#ab, abb, bbb, bbc, abc, bc\#\}$
It is trivial, from the clean$_{LT}$ definition, that given a set Y and its cleaned version $Y_{cleaned}$, then $LT_k(Y) = LT_k(Y_{cleaned})$, and so $LT_3(A_{cleaned}) = LT_3(A) = LT_3(B) = LT_3(B_{cleaned})$.

Note that the two sets $A_{cleaned}$ and $B_{cleaned}$ are equivalent.

Lemma 2.1. *If two sets $S_1, S_2 \subseteq \bigcup_{i\in\mathbb{N},\, i\le k} \Sigma^i_{any}$ are such that $S_1 \cap \Sigma^k_{any} \neq \emptyset$, $S_2 \cap \Sigma^k_{any} \neq \emptyset$, $LT(S_1) = LT(S_2)$ and are both clean$_{LT}$, then $S_1 = S_2$.*

From the previous result, the following is trivial:

Corollary 2.1. *Given any language $\eta \subseteq \Sigma^*$ such that there exists some k for which η is LT_k, there is only one set $S \subseteq \bigcup_{i\in\mathbb{N},\, i\le k} \Sigma^i_{any}$ such that S is clean$_{LT}$.*

The proof of this corollary is trivial.

The concept that enables to relate two elements belonging to a set defining an *LT* language, is the following:

Definition 2.7 (connectable)**.** *In a set S defining an LT language, an element $t \in S, t = t_1t_2 \dots t_m$ where $t_m \neq \#$ is said to be connectable to an element $r \in S, r = r_1 \dots r_m$ if $\forall 1 \leq i < m \; r_i = t_{i+1}$.*

Based on the concept of *connectable*, there can be stated the following reguarding clean_{LT}.

Corollary 2.2. *In a set S that is $clean_{LT}$ there does not exist an element $t \in S$ such that its first and last characters are not $\#$ and that is not connectable to any other element in S.*

For the sake of simplicity, from now on whenever the contrary is not stated, when referencing to a set defining an LT language, it is implicitly considered to be clean_{LT}.

Let us introduce the LTO family of languages: an extension of LT that reserves lots of interesting results.

Definition 2.8 (LTO)**.** *The class of LTO languages is defined as the smallest set of languages containing all the LT languages and such that it is closed under the boolean operations, and the concatenation.*

Example 2.8. *Here there are some examples of LTO languages over the alphabet $\Sigma = \{a, b, c\}$:*

- *The language $L_a = \{a\}$ is LT_2 (with $\{\#a\#\}$as set defining it), so it is also LTO.*
- *The language $\Sigma^* \setminus L_a$, being definable as $\neg L_a$, is LTO.*
- *The language Σ^*, being $L_a \cup \neg L_a$, is LTO.*
- *The language $\emptyset$, being $L_a \cap \neg L_a$, is LTO.*
- *The language L_{a^*} is LT_2 (with $\{\#\#, \#a, aa, a\#\}$ defining it), so it is also LTO.*

These examples, however, fall all into the LT family too. Let us analyze something more interesting:

- *The language L_{ab^+c} is LT_2 with $\{\#a, ab, bb, bc, c\#\}$ defining it, thus it is also LTO.*
- *The language $L_{ab^+cab^+c}$, definable via the Regular Expression ab^+cab^+c, is not LT, however it is LTO because it is the concatenation of the two LTO languages $L_{ab^+c} \cdot L_{ab^+c}$.*

- *The language $L_{(ab^+c)+a^*}$, definable via the Regular Expression $(ab^+c)+a^*$, is not LT also if it is LTO being it definable as $L_{ab^+c} \cup L_{a^*}$.*

2.3. Results of Locally Testable Languages

The LTO family of languages, is one really interesting class, not only for its definition but also because of its relation with the NC family which is in this section.

Theorem 2.1 ($LTO \subseteq NC$). *The class of LTO languages is a subclass of NC languages.*

The previous result implies that each language η that is a LTO language, is also a NC language.

Proof. The proof consists of proving the following steps:

1. If α is LT, then $\alpha \in NC$;
2. If α, β are NC, then $\alpha \cup \beta \in NC$
3. If α is NC, then $\neg\alpha \in NC$
4. If α, β are NC, then $\alpha \cdot \beta \in NC$

Let us prove item 1. Suppose that there exists k such that α is k-testable, we want to find the value of n such that $\forall_{x,y,z\in\Sigma^*} xy^nz \in \alpha \Leftrightarrow xy^{n+1}z \in \alpha$. It is trivial to see that if $y = \varepsilon$ then it is valid for any value of n; else we have that $|y| \geq 1$, and choosing $n = k+1$ is trivial to see that the sets of substrings of length k of $xy^{k+1}z$ and $xy^{k+2}z$ are the same, so also in this case it is valid. Thus the point item 1 is valid.

The item 3 is trivial.

For the point item 2, being $\alpha, \beta \in NC$, we have that $\forall_{x,y,z\in\Sigma^*} xy^pz \in \alpha \Leftrightarrow xy^{p+1}z \in \alpha$ and $xy^qz \in \beta \Leftrightarrow xy^{q+1}z \in \beta$. Let us choose $n = max(p, q)$. We want to prove that there exists n such that $xy^nz \in \alpha \cup \beta$, which is so if and only if xy^nz is in α or in β, which is so if and only if $xy^{n+1}z$ is in α or in β, which is so if and only if $xy^{n+1}z \in \alpha \cup \beta$.

For the item 4, being $\alpha, \beta \in NC$, we have that $\forall_{x,y,z\in\Sigma^*} xy^pz \in \alpha \Leftrightarrow xy^{p+1}z \in \alpha$ and $xy^qz \in \beta \Leftrightarrow xy^{q+1}z \in \beta$. Let us choose $n = p+q+1$. We want to prove that $xy^nz \in \alpha \cdot \beta \Leftrightarrow xy^{n+1}z \in \alpha \cdot \beta$. Let's prove $xy^nz \in \alpha \cdot \beta \Rightarrow xy^{n+1}z \in \alpha \cdot \beta$ the cases:

- $x = x_1 \cdot x_2,\ x_1 \in \alpha,\ x_2y^nz \in \beta$, in this case $x_2y^{n+1}z \in \beta$, thus $xy^{n+1}z \in \alpha \cdot \beta$;
- $z = z_1 \cdot z_2,\ z_2 \in \beta,\ xy^nz_1 \in \alpha$, in this case $xy^{n+1}z_1 \in \alpha$, thus $xy^{n+1}z \in \alpha \cdot \beta$;

- $y = y_1 \cdot y_2,\ xy^s y_1 \in \alpha,\ y_2 y^t z \in \beta$ where $s + t = n = p + q + 1$, thus either $t \geq q$ or $s \geq p$, thus either $xy^{s+1} y_1 \in \alpha$ or $y_2 y^{t+1} z \in \beta$, thus $xy^{n+1} z \in \alpha \cdot \beta$.

Let us now prove the other way: $xy^{n+1} z \in \alpha \cdot \beta \Rightarrow xy^n z \in \alpha \cdot \beta$

- $x = x_1 \cdot x_2,\ x_1 \in \alpha,\ x_2 y^{n+1} z \in \beta$, in this case $x_2 y^n z \in \beta$, thus $xy^n z \in \alpha \cdot \beta$;
- $z = z_1 \cdot z_2,\ z_2 \in \beta,\ xy^{n+1} z_1 \in \alpha$, in this case $xy^n z_1 \in \alpha$, thus $xy^n z \in \alpha \cdot \beta$;
- $y = y_1 \cdot y_2,\ xy^s y_1 \in \alpha,\ y_2 y^t z \in \beta$ where $s + t = n = p + q + 1$, thus either $t \geq q$ or $s \geq p$, thus either $xy^{s+1} y_1 \in \alpha$ or $y_2 y^{t+1} z \in \beta$, thus $xy^n z \in \alpha \cdot \beta$.

In this way, we proved the theorem as will be done for theorem 6.3. □

Leveraging this theorem, there can be proven that the language of example 1.28 is NC by proving it is LTO, as it has already been done in example 2.4.

Example 2.9. *Here is an example that apparently is trivial, but that will reserve not a few problems. Consider the example 1.28, and extend its language recognized by:*
$RE_{once} = (B)^*(P((B)^+C)^*S)^*$
It is trivial to see that the new language is LTO:
being B^* *and* $(P((B)^+C)^*S)^*$ *both LT, then* RE_{once} *characterizes a language that is the concatenation of two LT ones.*

Let us now complicate it a little more: consider the language recognized by:
$RE_{more} = ((B)^*(P((B)^+C)^*S)^*)^*$
For this new language, which is the application of the Kleene operator on the preceding one, is not trivial to say whether it is Counting or not. Try to prove it is not, and in order to do so, let us do a step behind and think about the real-world phenomenon it is describing.
The language RE_{more} *recognizes only the sequences of events that bring the coffee machine from being switched off to being switched off again, but in addition to* $RE_{orig} = (P((B)^+C)^*S)^*$ *it also accepts that the button can be pressed while the machine is switched off, and in that case, the machine doesn't provide the coffee.*

Let us reformulate this description in the opposite way: stating what the language does not recognize.

1. *The machine does not provide coffee if there has not been pressed the button in the previous action;*
2. *The machine does not provide coffee if it has not been powered up;*

3. *The machine does not provide coffee if it has been switched off and has not been powered on again;*
4. *The machine cannot be switched off before being powered up;*
5. *The machine cannot be switched off after having been switched off without having been powered up again;*
6. *The machine cannot be powered up if it has been already powered up and it has not been switched off;*
7. *The machine cannot be switched off if there has been pressed the button in the previous action;*
8. *the machine cannot be powered up without being switched off;*

Let us now translate those requirements in a more formal way, using η *to refer to the language recognized by* RE_{more}*:*

1. $\eta \cap \Sigma^*(\Sigma \setminus \{B\})C\Sigma^* = \emptyset$
2. $\eta \cap (\Sigma \setminus \{P\})^*C\Sigma^* = \emptyset$
3. $\eta \cap \Sigma^*S(\Sigma \setminus \{P\})^*C\Sigma^* = \emptyset$
4. $\eta \cap (\Sigma \setminus \{P\})^*S\Sigma^* = \emptyset$
5. $\eta \cap \Sigma^*S(\Sigma \setminus \{P\})^*S\Sigma^* = \emptyset$
6. $\eta \cap \Sigma^*P(\Sigma \setminus \{S\})^*P\Sigma^* = \emptyset$
7. $\eta \cap \Sigma^*BS\Sigma^* = \emptyset$
8. $\eta \cap \Sigma^*P(\Sigma \setminus \{S\})^* = \emptyset$

The other sequences are accepted.

So the language recognized by RE_{more} *is:*

$$\eta = \neg(\Sigma^*(\Sigma \setminus \{B\})C\Sigma^*) \cap \neg((\Sigma \setminus \{P\})^*C\Sigma^*) \cap \\ \neg(\Sigma^*S(\Sigma \setminus \{P\})^*C\Sigma^*) \cap \neg((\Sigma \setminus \{P\})^*S\Sigma^*) \cap \neg(\Sigma^*S(\Sigma \setminus \{P\})^*S\Sigma^*) \cap \\ \neg(\Sigma^*P(\Sigma \setminus \{S\})^*P\Sigma^*) \cap \neg(\Sigma^*BS\Sigma^*) \cap \neg(\Sigma^*P(\Sigma \setminus \{S\})^*)$$

As proved in previous examples, the language Σ^* *is LTO, as any language containing only a string of a single character or of two characters. It is trivial to see that, for any character* $a \in \Sigma$*, the language* $(\Sigma \setminus \{a\})^*$ *is LT, thus also LTO. With this in mind, looking at the definition of* η *language, it is easy to see that it is a (complex) composition of LTO* *languages via boolean operators, so also η is LTO, which turns out to demonstrate that the language η is Non-Counting.*

The most important result for the LTO class of languages, is:

Theorem 2.2. *The class of LTO languages coincides with the class of NC languages.*

This theorem adds to theorem 2.2 the fact that any NC language is also LTO. Although the importance of this result, the proof is too long to be reported here, and it can be found in [20] among lot of oter results and information about the LTO family of languages.

Given theorem 2.2, theorem 1.5, theorem 1.7 and theorem 1.6, there can be summarized:

Corollary 2.3. *Those classes of languages are the same:*

- *the class of (Regular) Non-Counting languages;*
- *the LTO family of languages;*
- *the Star Free languages;*
- *the class of languages defined via First Order Logic;*
- *the class of languages recognizable by acyclic finite state automata.*

Example 2.10. *Here it is a quick example to give a hint about the corollary's application: The language of example 2.9 described by $RE_{more} = ((B)^*(P((B)^+C)^*S)^*)^*$ can be translated in the automata of fig. 2.1 that recognizes the language. The construction of this automaton from the Regular Expression is easier than what has been done in example 2.9 to prove that the language is LTO. Those two examples are a perfect example of the utility of having many equivalent representations: it simplifies the study of languages and their properties.*

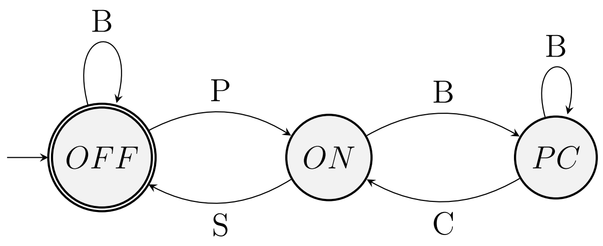


Figure 2.1: FSM recognizing $\text{RE}_{\text{more}} = ((B)^*(P((B)^+C)^*S)^*)^*$

3 | Context Free Languages

As it has been stated earlier, the goal of this work is to find a class of languages that is a sub-class of Operator Precedence Languages and such that the Non-Counting property is valid for its languages. However, in definition 1.33 there has been given the definition of Non-Countingness, and it has been proven that a language that is Non-Counting is also a Regular Language, so how is it possible to relate the family of Operator Precedence Languages (OPL) with the Non-Countingness concept?
The goal of this section is to provide an answer to that question. In order to define the Non-Counting property (and the relative family of languages) for the Operator Precedence languages, we will discuss how the Non-Countingness could be extended to the Context Free Languages to which the Operator Precedence belongs.

Inside this section, there could be found the definition and main results for the class of Context-Free Languages (CFL), corresponding to the *type-2* of Chomsky's classification, then there will be presented the sub-class of Structured Context-Free Languages with its properties and peculiarities, and finally, there will be discussed how to relate the Non-Countingness with this family of languages.

Another topic presented in this section that plays a central role in CFL and OPL, is the structure of the string. The main definitions and results presented in this section can be found in books regarding formal languages, like [21], while the ones regarding Structured Context-Free Languages can be found in [14].

3.1. Context Free Languages

Context-Free Languages, or CFLs, together with Regular Languages, constitute the widest chapter in the formal language literature. In particular, CFL have been introduced by Noam Chomsky in the 1950s in order to find a formalism capable of representing natural languages, and they have been effectively used, for example, to describe the structure of sentences.

Example 3.1 (CFL in NLP). *In Natural Language Processing (NLP), CFL is used also*

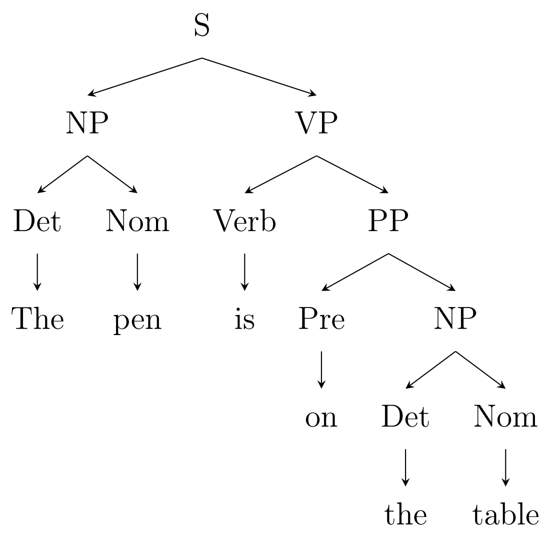


Figure 3.1: Tree structure of the phrase "The pen is on the table"

to describe the relations holding between different words used to build sentences and give them sense. In fig. 3.1 can be found an example of the structure of a sentence that can be described via CFL. For more information about this field, referr to [12].

The CFL family expands the expressiveness of RL, introducing the possibility of effectively expressing the structure of strings inside languages. This particular feature, which is proper of CFL, enables an algorithm to extract a property of natural languages that may not be immediately visible and that, somehow, contains the meaning of sentences[1]. For this reason, the CFL family has been used to structure and describe high-level programming languages. In honor of the chief scientist of the team that developed FORTRAN, one of the first programming languages, one meta-syntax that can be used to express Contxt-Free grammars, usually used to describe the programming languages' syntax rules, is called *Backus-Naur Form* or BNF.

Example 3.2 (BNF of C). *Here is an example of how BNF can be used to describe a (simplified) version of C's syntax:*

⟨program⟩ ::= ⟨declaration⟩ | ⟨function-definition⟩

⟨declaration⟩ ::= ⟨type-specifier⟩ ⟨declarator-list⟩ ;

⟨type-specifier⟩ ::= ***int*** *|* ***float*** *|* ***double*** *|* ***char***

⟨declarator-list⟩ ::= ⟨declarator⟩ , ⟨declarator-list⟩ | ⟨declarator⟩

⟨declarator⟩ ::= ⟨identifier⟩ | ⟨identifier⟩ [⟨expression⟩]

[1]Or at least part of the meaning of the sentence. Unluckily there has not been found a formal way to analyze natural languages that resolves ambiguity and can be practically used

$\langle$*expression*$\rangle$ *::=* $\langle$*identifier*$\rangle$ */* $\langle$*number*$\rangle$ */* $\langle$*expression*$\rangle$ *+* $\langle$*expression*$\rangle$ */* $\langle$*expression*$\rangle$ *-* $\langle$*expression*$\rangle$

$\langle$*logical-expression*$\rangle$ *::=* $\langle$*expression*$\rangle$ *==* $\langle$*expression* */* *<*$\langle$*expression*$\rangle$ *>=* $\langle$*expression*$\rangle$ */* *!*$\langle$*logical-expression*$\rangle$ */* $\langle$*identifier*$\rangle$

$\langle$*function-definition*$\rangle$ *::=* $\langle$*type-specifier*$\rangle$ $\langle$*identifier*$\rangle$ *(* $\langle$*parameter-list*$\rangle$ *)* $\langle$*compound-statement*$\rangle$ */* $\langle$*type-specifier*$\rangle$ $\langle$*identifier*$\rangle$ *( )* $\langle$*compound-statement*$\rangle$

$\langle$*parameter-list*$\rangle$ *::=* $\langle$*parameter*$\rangle$ *,* $\langle$*parameter-list*$\rangle$ */* $\langle$*parameter*$\rangle$

$\langle$*parameter*$\rangle$ *::=* $\langle$*type-specifier*$\rangle$ $\langle$*identifier*$\rangle$

$\langle$*compound-statement*$\rangle$ *::=* *{* $\langle$*declarator-list*$\rangle$ $\langle$*statement-list*$\rangle$ *}*

$\langle$*statement-list*$\rangle$ *::=* $\langle$*statement*$\rangle$ */* $\langle$*statement*$\rangle$ $\langle$*statement-list*$\rangle$

$\langle$*statement*$\rangle$ *::=* $\langle$*expression*$\rangle$ *;* */* $\langle$*if-statement*$\rangle$ */* $\langle$*while-statement*$\rangle$ */* $\langle$*return-statement*$\rangle$

$\langle$*if-statement*$\rangle$ *::=* *if (* $\langle$*logical-expression*$\rangle$ *)* $\langle$*statement*$\rangle$ */* *if (* $\langle$*expression*$\rangle$ *)* $\langle$*compound-statement*$\rangle$ *else* $\langle$*compound-statement*$\rangle$

$\langle$*while-statement*$\rangle$ *::=* *while (* $\langle$*logical-expression*$\rangle$ *)* $\langle$*compound-statement*$\rangle$

$\langle$*return-statement*$\rangle$ *::=* *return ;* */* *return* $\langle$*expression*$\rangle$ *;*

As stated in definition 1.23, the CFL family contains the RL family, which implies that it is more expressive than RL. This implies that any Regular Language is also a Context Free Language, but also that there exist some Context-Free Languages that recognize a set of strings that is not recognizable by any Regular Language. In other words, by augmenting the expressiveness, there is the possibility to be more precise on which sting to accept and which not to accept, so there are more tools to restrict the language, which is somehow counter-intuitive: the more expressive a family of language is, the fewer strings each language of the family can be able to recognize.
To prove the fact that the CFL family strictly contains the RL one, there is provided an example of a language that is CFL but not RL, which, combined with the fact that any language that is RL that it is also CFL, suffices to prove that the CFL family strictly contains the RL one, extending its expressiveness.

Example 3.3. *Consider the alphabet* $\Sigma = \{[,]\}$ *and the language* $\eta = [^n]^n$, $n \geq 1$. *It is trivial to see that it can be generated by the grammar:*
$S = []|[S]$
that is a Context Free Grammar, and, using the pumping lemma, it is trivial to see that the language is not a Regular Laguage.

This example, although simple, uses a subset of a known language, the Dyck Language, that is presented in section 3.2 to discuss some of its features.

For the sake of completeness over the expressiveness of the CFL family, here is an example of a language that is not Context-Free. There is not provided the proof of the following example.

Example 3.4. *Consider the alphabet $\Sigma = \{a, b, c\}$ and the language $\eta = a^n b^n c^n$, $n \geq 1$. It is trivial to see that it can be generated by the grammar:*
$S = aT|aST$
$aT = abC$
$CT = TC$
$bC = bc$
$cC = cc$
This language is not CFL.

The previous example expressed a subset of the Bach languages that are discussed in [19], hwere there can be found the proof for it.

The big feature that the CFL adds with respect to RL, is the possibility to nest a number of (possibly infinite) strings in a hierarchical way. This nesting, provided by the Context-Free grammar's derivation, also enables to describe a "structure" of the strings recognized by Context-Free Languages. Here are some examples about this concept:

Example 3.5 (arithmetic)**.** *Consider the CF grammar $G_{aritm1} = \langle V_T, V_N, P_1, S\rangle$ with $V_T = \{e, +, \times\}$, $V_N = \{S, A, B\}$ having as the production set P:*
$S \rightarrow B$
$A \rightarrow A \times A$
$A \rightarrow e$
$B \rightarrow A + B$
$B \rightarrow A$
This grammar generates the correct strings of sums and multiplications of the element e. For example, the strings $e + +e$ or $e + e + e \times e+$ are not generated by G_{aritm1}.

Consider the string $e + e + e \times e + e \times e + e$ that is generated by G_{aritm1}. There can be defined a "structure" over this string relying on the grammar G_{aritm1}; this structure, which is the sequence of rules and derivations used to generate or, equivalently, recognize the string, is reported in fig. 3.2.

In this way, there can be embedded in the grammar not only the generation (and acceptance) of the strings, but also the construction of a structure over them. In the previous

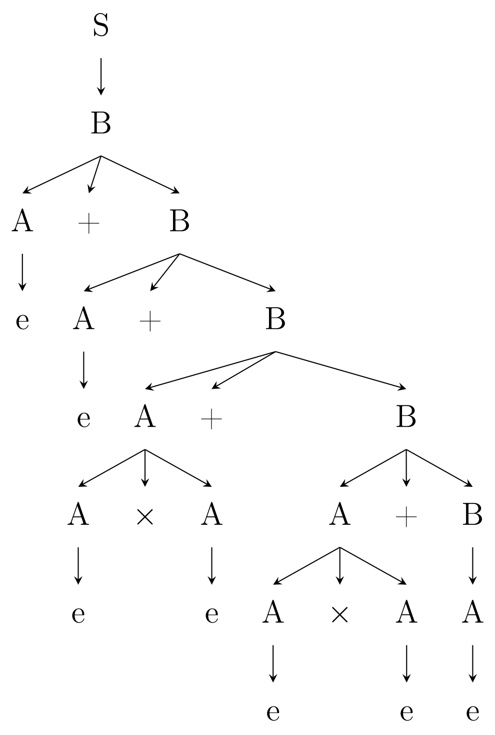


Figure 3.2: Structure of $e + e + e \times e + e \times e + e$ with respect to G_{aritm1}

example the structure is given only because of the grammar, not by the usual conventions. Thus there can be defined a grammar that generates the same language, but that builds structures that are the opposite of the ones built by the grammar of example 3.5.

Example 3.6 (opposite arithmetic). *Consider* $G_{aritm2} = \langle V_T, V_N, P_2, S\rangle$ *with all the elements as the ones of* G_{aritm1} *of example 3.5 except for* P_2 *that will contains the rules:*
$S \to A$
$B \to B + B$
$B \to e$
$A \to B \times A$
$A \to B$
which are the "opposite" of G_{aritm1}*. Also in this case this grammar generates the strings of correct sums and multiplications of the element e. For example, the strings* $e + +e$ *or* $e + e + e \times e+$ *are not accepted neither by* G_{aritm2}*.*

Consider the string of example example 3.5: $e + e + e \times e + e \times e + e$*, it is accepted also by* G_{aritm2}*, but section 3.1 is one possible structure that* G_{aritm2} *defines over the string, and comparing it with the one provided by* G_{aritm1} *and present in fig. 3.2, it is trivial to see that* G_{aritm1} *defines a structure compatible with the usual precedence of operations, so having* $(e + (e + ((e \times e) + ((e \times e) + e))))$ *while* G_{aritm2} *builds a structure that is the*

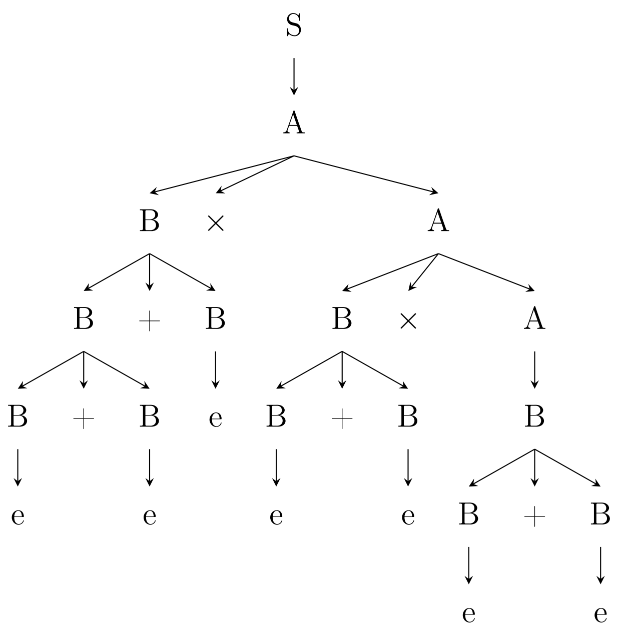


Figure 3.3: Structure of $e + e + e \times e + e \times e + e$ with respect to G_{aritm2}

opposite of what is usually used, having $(((e + e) + e) \times ((e + e) \times (e + e)))$.

Moreover, there is also the possibility to have CF Grammars that can associate more than one structure to the same string: those grammars are said to be *ambiguous*. The problem of ambiguity is one of the bigger ones for the CFL family and the ones being more expressive. There have been many attempts to solve it, but sometimes it is not possible because of its intrinsic nature of the described phoenomena. However, although this topic is a hot topic in computing science, it will not be furthermore discussed in this work, mainly because the families that are presented here, are not affected by the ambiguity problem. Operator Precedence Languages, as Structured Context Free Languages, have been studied in order to overcame this problem. Interestingly enough, the solution of ambiguity brought some great results in those families, which means that ambiguity is not only a pervasive problem, but it also causes some other properties, like closure under Boolean operators, not to hold.

Example 3.7. *Here is presented a simple attempt to build an ambiguous grammar* $_{ambigAritm}$ *starting from* G_{aritm1} *and* G_{aritm2} *is:*
$G_{ambigAritm} = \langle V_T, V_N, P_3, S\rangle$ *with* P_3 *containing:*
$S \rightarrow A \mid B$
$A \rightarrow B \times A \mid B \mid A \times A \mid e$
$B \rightarrow A + B \mid A\ B + B \mid e$
It is trivial to see that using the grammar $G_{ambigAritm}$ *the two structures in fig. 3.2 and*

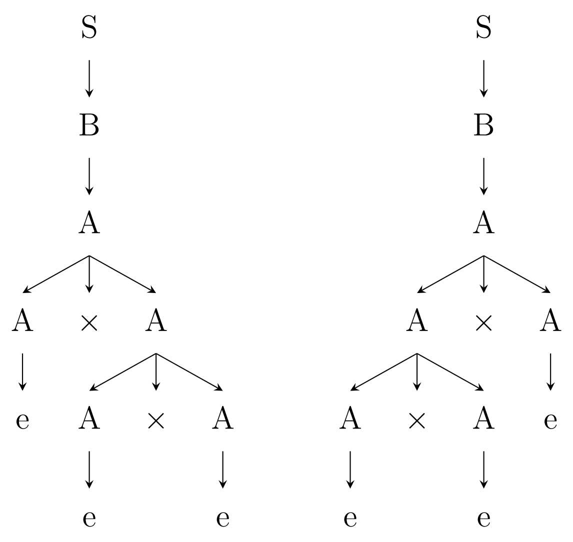


Figure 3.4: Structure of $e \times e \times e$ with respect to G_{aritm1}

section 3.1 relative to the string $e+e+e\times e+e\times e+e$ *can be built, having thus that the same string has more than one structure.*

However, ambiguity is also present in the two grammars G_{aritm1} *and* G_{aritm2} *and it can be proved by building the structure of* $e\times e\times e$ *with* G_{aritm1}*, which has been done in fig. 3.4, and* $e+e+e$ *with* G_{aritm2}*, which has been done in fig. 3.5.*

The activity of checking for a string appartenance to a CF language and the building of the structure related to it, also referenced as its *syntax tree*, is called *parsing*.

One important result, taken from the literature, is the following:

Theorem 3.1 (CFL closure)**.** *CFL are closed under union, concatenation, Kleene***, homomorphism, and inverse homomorphism, but not under complement and intersection.*

Thus, CFL enriches RL but loses some of the most important features as the closure over all boolean operations. In the following section, itis presented how to restrict the CFL family and gaining again those properties for the sub-family.

There must be stated a last known result: the extension of the Pumping Lemma also in the word of Context-Free Languages, also known as the *Bar-Hillel lemma*, in honor of the mathematician who proposed it in [2].

Theorem 3.2 (Bar-Hillel Lemma)**.** *Given a Context Free Language* $\eta \subseteq \Sigma^*$*, there exists an integer* $p > 0$ *depending solely on* η *such that for each string* $z \in \eta$ *with* $|z| \geq p$ *it can be written as:*

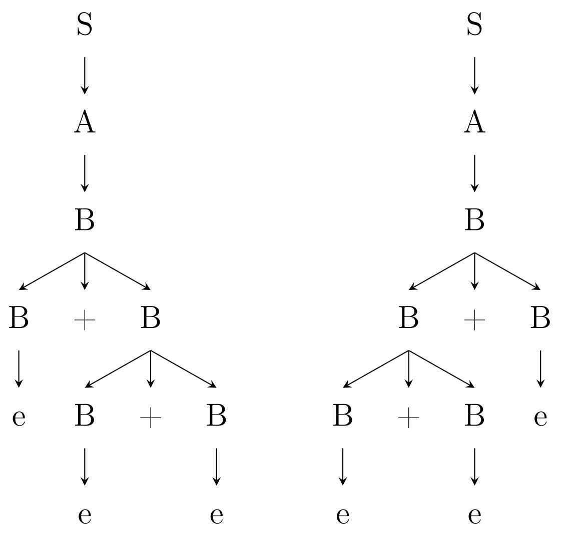


Figure 3.5: Structure of $e + e + e$ with respect to G_{aritm2}

$z = uvwxy$
In such a way that:

- $|vwx| \leq p$;
- $|vx| \geq 1$;
- $uv^iwx^iy \in \eta$ *for any* $i \geq 0$.

3.2. Structured Context-Free Languages

In this section, is presented the concept of Structured Context-Free Languages, introduced in 1967 by R. McNaughton in his work [16], where he proposed the idea of embedding the structure inside the string itself. The aim of this idea is to make somehow explicit the frontiers of the syntax tree, to remove the ambiguity problems and gain some of the properties that the RL family has but the CFL one does not.
This property of languages is either called *Structured CFL* or *visible structure CFL*. There are presented some of its major results with the reason behind each one of them. The goal of the section, is to understand if and how those concepts can be used also in other classes of languages, as in the Operator Precedence one. Here is a simple example of a language that can be classified as structured CFL, which is the Dyck language:

Example 3.8 (Dyck language). *Consider the alphabet* $\Sigma = \{(,)\}$*, the Dyck language is defined via the grammar:*

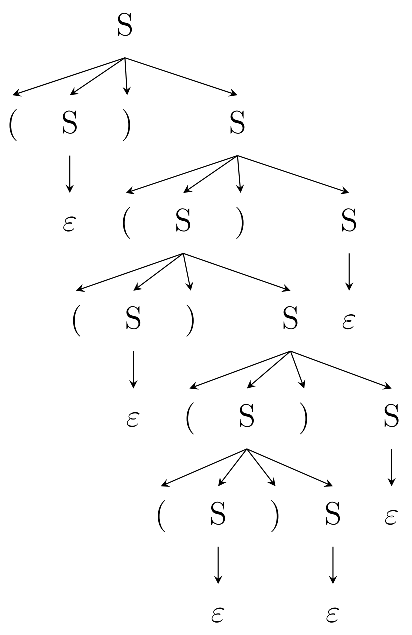


Figure 3.6: Structure of ()(()(())) with G_{Dyck}

$G_{Dyck} = \langle \Sigma, \{S\}, \{S \to \varepsilon\,,\, S \to (S)S\}, S\rangle$.

It is trivial to see that the language generated by such grammar is the language of the well-parenthesized strings over Σ^**, and it is also easy to see that this grammar is not ambiguous.*

In fig. 3.6 is reported the structure of the string ()(()(()))

As the first definition of a structured CFL, R. McNaughton proposed the *parenthesis languages* one, where each sub-tree of the syntax tree is delimited by two paired parentheses.

Definition 3.1 (parenthesis grammar). *A grammar*
$\tilde{G} = \langle V_T \cup \{\lfloor, \rfloor\}, V_N, \tilde{P}, S\rangle$ *is called parenthesis grammar (Par Gram) if the rhs of every rule in* $\tilde{P}$ *is parenthesized, so if each rhs is in the form* $\lfloor\beta\rfloor$, $\beta \in (V_T \cup V_N)^*$.

$\tilde{G}$ *is called the parenthesized version of* G*, if* $\tilde{P}$ *consists of all rules* $B \to \lfloor\beta\rfloor$ *such that* $B \to \beta$ *is in* P.

The language generated by a parenthesis grammar $\tilde{(G)}$, $L(\tilde{G})$*, is called parenthesis language and is defined over* Σ_{par}.

Here is an example changing the grammar of example 3.5 to be parenthesized, to see what happens to its ambiguity:

Example 3.9. *To transform*

P_1		P_{1Par}
$S \to B$	$\Rightarrow$	$S \to \lfloor B \rfloor$
$A \to A \times A$	$\Rightarrow$	$A \to \lfloor A \times A \rfloor$
$A \to e$	$\Rightarrow$	$A \to \lfloor e \rfloor$
$B \to A + B$	$\Rightarrow$	$B \to \lfloor A + B \rfloor$
$B \to A$	$\Rightarrow$	$B \to \lfloor A \rfloor$

Table 3.1: Parenthesizing the rules of P_1 into P_{1Par}

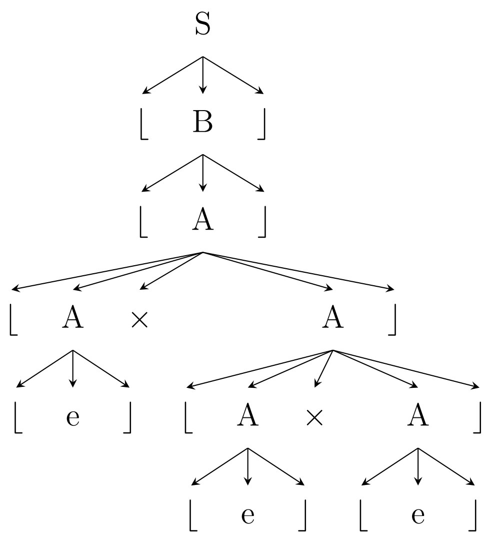


Figure 3.7: Structure of $\lfloor\lfloor\lfloor e\rfloor \times \lfloor\lfloor e\rfloor \times \lfloor e\rfloor\rfloor\rfloor\rfloor$ with respect to $G_{aritm1Par}$

$G_{aritm1} = \langle V_T, V_N, P_1, S\rangle$ into
$G_{aritm1Par} = \langle V_{T,Par}, V_N, P_{1Par}, S\rangle$
it is needed to extend V_T to $V_{T,Par} = V_T \cup \{\lfloor, \rfloor\}$ and to change P_1 to P_{1Par}, and that is done by changing each rule of P_1 as done in table 3.1.

Let us now use $G_{aritm1Par}$ to generate the string using the same derivations used by G_{aritm1} in the first case of fig. 3.4. The generated string will be:
$\lfloor\lfloor\lfloor e\rfloor \times \lfloor\lfloor e\rfloor \times \lfloor e\rfloor\rfloor\rfloor\rfloor$.
It is trivial to see that the structure of the new string, with respect to $G_{aritm1Par}$, is unique. The structure is the one of fig. 3.7.

All the languages for which there is a visible structure inside the string are Structured CFL, thus all the languages for which, given a string belonging to the language itself, it is derivable the unique syntax tree corresponding to it.
There are some significant languages that are structured CFL, from the parenthesis ones to the *tree languages* passing through the *Operator Precedence Languages*. Note that there is no ambiguity for those languages because their strings' structure is explicit.

The concept of *well structured* is a key concept for Structured Context-Free Languages, and is applicable both on strings and on their substrings.

Definition 3.2 (well structured). *Given a structured CFL η and a string s, s is well-structured with respect to η if the syntax tree associated with s via η is complete.*
Given a structured CFL η, a string s and a substring t of s, t is well structured on s with respect to η if the part of the syntax tree of s that spans over t is a complete subtree.

Corollary 3.1. *Given a structured CFL η and a string s, s is well-structured with respect to η iff $s \in \eta$.*

The just given definition of well-structured is generic for any language that is Structured CFL, and it strongly relies on the concept of structure. As previously said, there exist different families of languages that are structured CFL, thus there exists for eachone of them a specific concept of structure, and then the definition of well-structured can be tailored to suit the specific family. For example, there is reported how the well-structured definition can be modified for parenthesis languages.

Definition 3.3. *Given a parenthesis language η and a string s, s is well-structured with respect to η if s is well-parenthesized.*
Given a parenthesis language η, a string s and a substring t of s, t is well structured on s with respect to η if t is well-(sub)parenthesized in s.

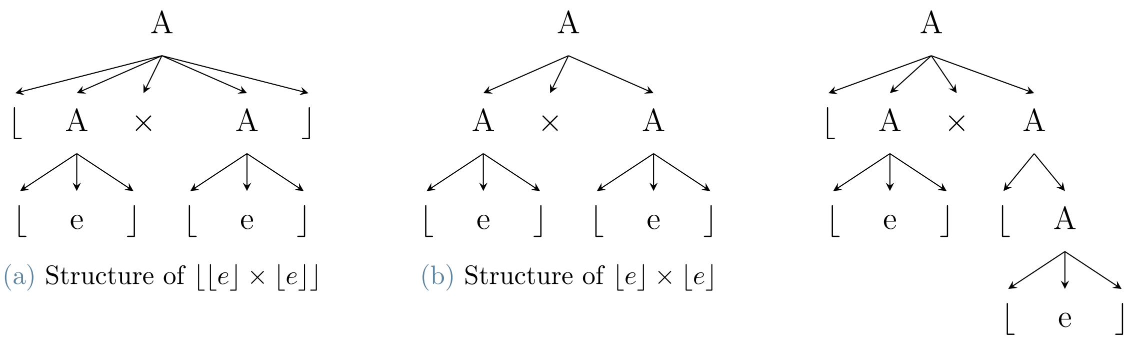
Later on, how to decline the well-structured property for Operator Precedence Languages will be discussed.

Here there are some examples over parenthesis languages:

Example 3.10. *Consider the parenthesized language $L_{aritm1Par}$ defined by the grammar $G_{aritm1Par}$ of example 3.9. As said in the previous example, the string $\lfloor\lfloor\lfloor e\rfloor \times \lfloor\lfloor e\rfloor \times \lfloor e\rfloor\rfloor\rfloor\rfloor$ is recognized by the grammar, which means that it is also well parenthesized.*
Also the string $\lfloor\lfloor\lfloor e\rfloor\rfloor\rfloor$ is well parenthesized and belongs to the language.
The string $\lfloor\lfloor\lfloor e\rfloor + \lfloor e\rfloor$ is not well parenthesized on $L_{aritm1Par}$.

Inside the string $\lfloor\lfloor\lfloor e\rfloor \times \lfloor\lfloor e\rfloor \times \lfloor e\rfloor\rfloor\rfloor\rfloor$, the substring $\lfloor\lfloor e\rfloor \times \lfloor e\rfloor\rfloor$ is well parenthesized, while the whole string is not well parenthesized, and neither the substring $\lfloor e\rfloor \times \lfloor e\rfloor$ and neither $\lfloor\lfloor e\rfloor \times \lfloor\lfloor e\rfloor$ is. In fig. 3.8 are reported the syntax trees regarding those substrings. Please note that the syntax tree of $\lfloor\lfloor e\rfloor \times \lfloor e\rfloor\rfloor$ is a complete subtree of fig. 3.7, while the others are not, which is the reason why they are not well parenthesized.

A feature of Structured CFLs, is that they are closed under the intersection, which is not



(a) Structure of $\lfloor\lfloor e\rfloor \times \lfloor e\rfloor\rfloor$ (b) Structure of $\lfloor e\rfloor \times \lfloor e\rfloor$ (c) Structure of $\lfloor\lfloor e\rfloor \times \lfloor\lfloor e\rfloor$

Figure 3.8: Structure of substrings using $L_{aritm1Par}$

the case for generic CFLs.

In order to prove the closure with respect to negation, McNaughton relied on the following milestones:

1. The possibility to apply all the operations within the structure universe instead of the whole universe of strings. This can be possible thanks to the stencil concept:

   Definition 3.4. *A stencil of a terminal alphabet Σ is a string in $(\Sigma \cup \{N\})^*)$ -where $N \notin \Sigma$-. For any given CFG G a stencil grammar G_s is naturally derived therefrom by projecting any nonterminal of G into the unique nonterminal N, possibly deleting duplicated productions.*

   Given this definition, there can be now defined the *structure universe*

   Definition 3.5. *For any ParGram G the structure universe of G is the -parenthesis-language $L(G_s)$. For any set of PG, S, its structure universe is the union of the structure universes of its members.*

2. The possibility of building a normal form of any parenthesis grammar, the Backward Deterministic Normal Form (BDNF) with no repeated rhs, as done in definition 1.22 and theorem 1.2.

Here is an example of how the structure universe can be computed:

Example 3.11. *Consider the grammar $G_{aritm1Par}$ of example 3.9. To define its structure universe we must before calculate its stencil version:*
$G_{aritm1ParS} = \langle V_{T,Par}, \{N\}, P_{1ParS}, N\rangle$,

P_{1Par}	$\Rightarrow$	P_{1ParS}
$S \to \lfloor B \rfloor$	$\Rightarrow$	$N \to \lfloor N \rfloor$
$A \to \lfloor A \times A \rfloor$	$\Rightarrow$	$N \to \lfloor N \times N \rfloor$
$A \to \lfloor e \rfloor$	$\Rightarrow$	$N \to \lfloor e \rfloor$
$B \to \lfloor A + B \rfloor$	$\Rightarrow$	$N \to \lfloor N + N \rfloor$
$B \to \lfloor A \rfloor$	$\Rightarrow$	$N \to \lfloor N \rfloor$

Table 3.2: Mapping P_{1Par} to P_{1ParS}

obtained by mapping P_{1Par} to P_{1ParS}, as done in table 3.2. [2]

The structure universe of $G_{aritm1Par}$ is the language generated by $G_{aritm1ParS}$.

With those results, assuming, without loss of generality, that a ParGram G is in BDNF, it is possible to define the complement of $L(G)$ with respect to its structure universe by:

- Let $A_{\text{err}} \notin V_n$ be a new nonterminal and let h be the homomorphism that maps each element of $V_n \cup \{A_{\text{err}}\}$ into N and every element of Σ into itself; then let's add to G's production set P the productions $A_{\text{err}} \to \alpha$ for all $\alpha \in (\Sigma \cup V_n \cup \{A_{\text{err}}\})^*$ such that $h(\alpha)$ is in the production set of G_s but α is not in P;
- The new (renaming) productions with S as the lhs have as rhs the complement w.r.t. V_n of the original ones plus $S \to A_{\text{err}}$.

Here is an example of this procedure:

Example 3.12. *Let us use the result o example 3.11, in particular, the production sets P_{1Par} and P_{1ParS}. We want to build a new production set P_A that contains some rules in the form $A_{err} \to \alpha$, where α is such that its stencil is in the rhs of P_{1ParS} but α is not in the rhs of P_{1Par}.*
To do so let us analyze the rhs of P_{1ParS} and build from them the new rules:

- $N \to \lfloor N \rfloor$ *enables to add* $A_{err} \to \lfloor A_{error} \rfloor$*;*
- $N \to \lfloor N \times N \rfloor$ *enables to add* $A_{err} \to \lfloor A \times A_{err} \rfloor | \lfloor A \times B \rfloor |$
$\lfloor A_{err} \times A \rfloor | \lfloor A_{err} \times A_{err} \rfloor | \lfloor A_{err} \times B \rfloor | \lfloor B \times A \rfloor | \lfloor B \times A_{err} \rfloor | \lfloor B \times B \rfloor$*;*
- $N \to \lfloor N + N \rfloor$ *enables to add* $A_{err} \to \lfloor A + A \rfloor | \lfloor A + A_{err} \rfloor |$
$\lfloor A_{err} + A \rfloor | \lfloor A_{err} + A_{err} \rfloor | \lfloor A_{err} + B \rfloor | \lfloor B + A \rfloor | \lfloor B + A_{err} \rfloor | \lfloor B + B \rfloor$.

The language defined by the grammar with as production set P_A containing the new gen-

[2] note that during mapping the rules $S \to \lfloor B \rfloor$ and $B \to \lfloor A \rfloor$ are transformed in the same rule, and thus one of them can be eliminated

erated productions, $S \to \lfloor A_{err} \rfloor$, and the rules of P_{1Par} except for those with S as lhs ($s \to \lfloor B \rfloor$) substituted by their complement w.r.t. V_N ($S \to \lfloor A \rfloor$), is the grammar defining the complement language of $G_{aritm1Par}$ with respect to its structure universe.

At this point, there can be defined the intersection of two parenthesized languages using DeMorgan's rule:
$A \cap B = \neg((\neg A) \cup (\neg B))$.
Note that this definition of intersection requires that the 2 initial languages share the same structure universe. So, by making evident the universe structure on which the negation operation works, the previous formula can be transformed into:
$A \cap_U B = \neg_U((\neg_U A) \cup (\neg_U B))$.

3.3. Non-Counting for Structured Context-Free Languages

In this section, the Non-Countingness concept, described in definition 1.33, is expanded from being contained inside the RL family, to define a subclass of Context-Free Languages. To pursue this goal, the well-structured concept will be leveraged. For this reason, the definition will rely on Structured CFL.

Definition 3.6 (NC_S). *A language $\eta \subseteq \wp(\Sigma_{any}^*)$ that is structured CFL, is NC_S (non counting) iff $\exists n > 1$ such that, for all strings*
$x, u, w, v, y \in \Sigma_{any}^*,\ m \in \mathbb{N}$
where w and uwv are well structured on xu^nwv^ny with respect to η then
$xu^nwv^ny \in \eta \Leftrightarrow xu^{n+m}wv^{n+m}y \in \eta$.

As done also for Regular Languages, also in this case a language that is not Non-Counting is said to be Counting.

This definition, relying on the concept of structure, can be tailored to better fit the specific class of languages using it, as it has been done for the well-structured one. For example, the class NC_{Par} is the class of Non-Counting Parenthesized languages, that has a particular definition specifying in the particular case what well-structured means. Here is the definition:

Definition 3.7 (NC_{Par}). *A language $\eta \subseteq \wp(\Sigma_{any}^*)$ that is a prenthesized language, is NC_{par} (non counting) iff $\exists n > 1$ such that, for all strings*
$x, u, w, v, y \in \Sigma_{any}^*,\ m \in \mathbb{N}$
where w and uwv are well parenthesised on xu^nwv^ny w.r.t η then

$xu^nwv^ny \in \eta \Leftrightarrow xu^{n+m}wv^{n+m}y \in \eta$.

Example 3.13. *Consider the parenthesized grammar*
$G_{par} = \langle \{a, b\}, \{S, T\}, P, S\rangle$ *with* P *containing:*
$S \to \lfloor \varepsilon \rfloor$
$S \to \lfloor aTb \rfloor$
$T \to \lfloor aSb \rfloor$
It is trivial to see that this grammar generates the language $\lfloor (a\lfloor)^{2n}(\rfloor b)^{2n}\rfloor$*, which, intuitively, is counting, however, let us prove it:*
For any number n *there can be choosen a string* $s = u^n \cdot w \cdot v^n$ *such that:*

- *If* $n \equiv 0 \mod 2$*, then* $u = \lfloor a,\ v = b\rfloor,\ w = \lfloor\rfloor$*;*
- *If* $n \equiv 1 \mod 2$*, then* $u = \lfloor a,\ v = b\rfloor,\ w = \lfloor a\lfloor\rfloor b\rfloor$.

It is easy to see that $s \in L(G_{par})$*, but* $u^{+1}n \cdot w \cdot v^{n+1} \notin L(G_{par})$*, which proves that* $L(G_{par})$ *is Counting.*

Example 3.14. *Let us reshape the grammar example 3.13 to be Non-Counting. To achieve such a goal it is needed to modify only the production set of* G_{par}*. So it will be*
$G_{parNC} = \langle \{a, b\}, \{S\}, P_{NC}, S\rangle$ *with* P_{NC} *containing:*
$S \to \lfloor \varepsilon \rfloor$
$S \to \lfloor aSb \rfloor$
It is trivial to see that this language is NC_{Par}.

For the sake of simplicity, there is proven the equivalence of definition 3.8 with another easier formula that makes it easier to prove a language to be NC_S. This result has been inspired by Corollary 1 of [7]:

Theorem 3.3. *The following statements are equivalent to a backward deterministic reduced grammar* G*. There exists* $n \geq 0$ *such that for any* $x, v, w, u, y \in \Sigma^*$ *where* w, vwu *are w.p., and for any integer* $m \geq 0$*,*

1. $xu^nwv^ny \in L(G) \Leftrightarrow xu^{n+m}wv^{n+m}y \in L(G)$ *(i.e.* $L(G)$ *is non-counting),*
2. $xu^nwv^ny \in L(G) \Rightarrow xu^{n+m}wv^{n+m}y \in L(G)$*,*
3. $xu^{n+m}wv^{n+m}y \in L(G) \Rightarrow xu^nwv^ny \in L(G)$.

This result can be further expanded with the following result, inspired by lemma 1.1:

Lemma 3.1. *Given a language $\eta \in \wp(\Sigma^*)$ and an integer n, the following statements are equivalents:*

A $\forall_{m\in\mathbb{N},\, x,u,w,v,y\in\Sigma^*} xu^n wv^n y \in \eta \Rightarrow xu^{n+m}wv^{n+m}y \in \eta$

B $\forall_{x,u,w,v,y\in\Sigma^*} xu^n wv^n y \in \eta \Rightarrow xu^{n+1}wv^{n+1}y \in \eta$

Proof. It is trivial to see that point A implies B.
The fact that point B implies A is a bit sneaky: assuming that B is valid, there is that for each $xu^nwv^ny \in \eta$ implies $xu^{n+1}wv^{n+1}y \in \eta$, but this is the same of saying that $(xu)u^nwv^n(vy) \in \eta$, which, having B that holds, implies that also $(xu)u^{n+1}wv^{n+1}(vy) \in \eta$, so $xu^{n+2}wv^{n+2}y \in \eta$, and here can go on and on until reaching any number m, which implies $xu^{n+m}wv^{n+m}y \in \eta$ for any value of m, which corresponds to the initial thesis. □

This result, together with the fact that for any grammar, there exists an equivalent one in BDNF, enables us to formulate a newer definition of NC_S that is equivalent to definition 3.6.

Definition 3.8 (NC_S). *A language $\eta \subseteq \wp(\Sigma^*_{any})$ that is structured CFL, is also NC_S (non counting) iff $\exists n > 1$ such that, for all strings*
$x, u, w, v, y \in \Sigma^*_{any}$, $m \in \mathbb{N}$ where w and uwv are well structured on xu^nwv^ny with respect to η then
$xu^nwv^ny \in \eta \Rightarrow xu^{n+1}wv^{n+1}y \in \eta$ (i.e. $L(G)$).

This definition, although it is simpler than definition 3.6, is also more versatile: it can be used to demonstrate that a language is NC_S requiring only one case to be studied.

Other results about Structured CFL that are worth to be reported are:

Theorem 3.4. *It's decidable whether a structured CFL is NC or not.*

Theorem 3.5. *A structured CFL is NC iff is BDR grammar has no counting derivation.*

The main source of information for this section has been [7], where there is also reported that NC_S is closed under complement, from [15] there can be found that NC_S is closed under union, thus also under intersection. Moreover in [15] there is that NC_{OP} [3] is closed under concatenation.

In order to summarize, thus, it can be stated:

Theorem 3.6. *Any NC_S is closed under boolean operations.*

[3]This class will be introduced later

The NC_{OP} family is closed under boolean operations and concatenation.

4 | Operator Precedence Languages

The Operator Precedence Languages, or OPL, are presented in this chapter. OPLs are structured CFL. OPLs have been introduced by R.Floyd in his work [10] in the '60s, before the presentation of R. Mc Naughton's parenthesis languages. Floyd took inspiration from arithmetic expressions and their way to uniquely bind structures to the corresponding expressions based on the precedence of the operators. He understood that the concept of operator precedence could be taken outside of the algebraic world, and could be brought to the field of formal languages. In particular, in OPL, each character can be seen as a mathematical operator, havinng precedence relations defining the order in which it has to be computed. This is the main idea behind the OPL family.

Given a string belonging to an OPL, there is no explicit structure embedded in the string itself, however, it can be parsed in a linear time, resulting in a unique structure associated to the string. An other result concerning the OPL's structures, is the *Local Parsability Property*. This property ensures that the structure of any string belonging to an OPL, is made from an appropriate combination of the structures of the substrings belonging to the initial string. This feature is the one that can be leveraged to parallelize the computation for OPLs.

In this section, the definition of Operator Precedence Languages is presented alongside some of its major results. There is reported also the way Operator Precedence builds structures for strings and the Non-Counting class of languages, together with the characterization that have been proposed for this family in the literature.

4.1. Operator Precedence Languages Definition

In order to introduce the Operator Precedence Languages, there must be defined the concept of Operator Precedence relations. The easiest, and most used, way to introduce those relations is by using the grammars and their derivation rules. Thus, there are described the features requested to grammars in order to introduce the Operator Precedence Relations.

Definition 4.1 (Operator Grammar). *A grammar rule is said to be in operator form if its rhs has no adjacent non-terminals.*
A grammar with all the rules of its production set in operator form is an operator grammar (OG).

A known result presented in [11] is that any CF grammar, which has been defined in definition 1.23, can be recursively transformed into an equivalent OG. Here is an example about Operator Grammar.

Example 4.1 (OG Grammar). *Consider the grammars presented in example 3.5 and example 3.7, the rules inside their production sets (P and P_3 respectively) are in the Operator form:*
$P:$
$S \rightarrow B$
$A \rightarrow A \times A \mid e$
$B \rightarrow A + B \mid A$
$P_3:$
$S \rightarrow A \mid B$
$A \rightarrow B \times A \mid B \mid A \times A \mid e$
$B \rightarrow A + B \mid A\ B + B \mid e$
Thus, both the grammars are Operator Grammars.

Another essential concept, together with Operator Grammar, for the definition of OP relations, is the Left/Right non-terminal set.

Definition 4.2 (Left/Right non-terminal sets). *For any OG $G = \langle \Sigma, V_N, P, S \rangle$, given a non-terminal $A \in V_N$, the left and right terminal sets of A with respect to G are:*

- $\mathcal{L}_G(A) = \{a \in \Sigma \mid A \Rightarrow^*_G Ba\alpha\}$
- $\mathcal{R}_G(A) = \{a \in \Sigma \mid A \Rightarrow^*_G \alpha aB\}$

where $B \in V_N \cup \{\varepsilon\}$, $\alpha \in (V_N \cup \Sigma)^*$.
In the notation, the footer G will be omitted if there is no risk of confusion.

Corollary 4.1. $\mathcal{L}_G(A), \mathcal{R}_G(A) \subseteq \Sigma^*$

Thus, collecting the previous definitions, there can be defined the Operator Precedence Relations.

Definition 4.3 (Operator Precedence Relations)**.** *For any OG*
$G = \langle \Sigma, V_N, P, S \rangle$*,*
let $\alpha, \beta \in (V_N \cup \Sigma)^*$*, and* $a, b \in \Sigma$*,*
there will be defined the three binary relations, called Operator Precedence Relations: equals in precedence ($\doteq$)*, takes precedence (gtrdot), and yields precedence*($\lessdot$)*.*
Those relations will be defined as follows:

$$\textit{equals in p.: } a \doteq b \iff \exists A \to \alpha a B b \beta,\ B \in V_N \cup \{\varepsilon\}$$

$$\textit{takes p.: } a \gtrdot b \iff \exists A \to \alpha D b \beta,\ D \in V_N \textit{ and } a \in \mathcal{R}_G(D)$$

$$\textit{yields p.: } a \lessdot b \iff \exists A \to \alpha a D \beta,\ D \in V_N \textit{ and } b \in \mathcal{L}_G(D).$$

Here there are some examples of how to extract the precedence relations from Operator Grammars:

Example 4.2. *Consider the OG*
$G = \langle V_T, V_N, P_{OG}, S \rangle$ *with* $V_T = \{e, +, \times\}$*,* $V_N = \{S, A, B, E\}$ *and with the production set* P_{OG} *containing:* $S \to B$
$A \to E \times A \mid E$
$B \to A + B \mid A$
$E \to e$

Note that this grammar is very similar to the grammar defined on example 3.5 with as a minor modification the fact that the rule $E \to e$ *is the only rule producing* e*. Let us calculate the* $\mathcal{L}_G$ *and* $\mathcal{R}_G$ *sets:*
$\mathcal{L}_G(S) = \{e, +, \times\}$
$\mathcal{L}_G(A) = \{e, \times\}$
$\mathcal{L}_G(B) = \{e, +, \times\}$
$\mathcal{L}_G(E) = \{e\}$
$\mathcal{R}_G(S) = \{e, +, \times\}$
$\mathcal{R}_G(A) = \{e, \times\}$
$\mathcal{R}_G(B) = \{e, +, \times\}$
$\mathcal{R}_G(E) = \{e\}$

Now there can be determined the Operator Precedence Relations of this grammar:

- *Consider the relations between* $+$ *and* $+$*. There is* $B \to A + B$ *as the only rule with* $+$*. Being that it has only one* $+$*, there cannot be* $+ \doteq +$*; being* $+ \notin \mathcal{R}_G(A)$*, then* *there does not hold the* $+ \gtrdot +$*; being* $+ \in \mathcal{L}_G(B)$*, then* $+ \lessdot +$.

- *Consider the relations between* $+$ *and* $\times$*. We have* $B \to A + B$ *as the only rule with* $+$*. Being that it has only* $+$ *and not* $\times$*, there cannot be* $+ \doteq \times$*; being* $\times \in \mathcal{L}_G(B)$*, then there it holds* $+ \lessdot \times$*; being* $A \to E \times A$ *the only production with* $\times$*, and being* $+ \notin \mathcal{R}_G(E)$*, then it does not hold* $+ \gtrdot \times$.
- *Consider the relations between* $+$ *and* e*. There is* $B \to A + B$ *as the only rule with* $+$*. Being that it has only* $+$ *and not* e*, there cannot be* $+ \doteq e$*; being* $e \in \mathcal{L}_G(B)$*, then there it holds* $+ \lessdot e$*; being* $E \to e$ *the only production with* e*, then it is trivial to see that it does not hold* $+ \gtrdot e$.
- *Consider the relations between* $\times$ *and* $+$*. There is* $A \to E \times A$ *as the only rule with* $\times$*. Being that it has only* $\times$ *and not* $+$*, there cannot be* $\times \doteq +$*; being* $+ \notin \mathcal{L}_G(A)$*, then there it does not hold* $\times \lessdot +$*; being* $B \to A + B$ *and being* $\times \in \mathcal{R}_G(A)$*, then it holds* $\times \gtrdot +$.
- *Consider the relations between* $\times$ *and* $\times$*. There is* $A \to E \times A$ *as the only rule with* $\times$*. Being that it has only one occurrence of* $\times$*, there cannot be* $\times \doteq \times$*; being* $\times \in \mathcal{L}_G(A)$*, then it holds* $\times \lessdot \times$*; being* $\times \notin \mathcal{R}_G(E)$*, then it does not hold* $\times \gtrdot \times$.
- *Consider the relations between* $\times$ *and* e*. There is* $A \to E \times A$ *as the only rule with* $\times$*. Being that it has only* $\times$ *and not* e*, there cannot be* $\times \doteq e$*; being* $e \in \mathcal{L}_G(A)$*, then there it holds* $\times \lessdot e$*; being* $E \to e$ *the only production with* e*, then it is trivial to see that it does not hold* $\times \gtrdot e$.
- *Consider the relations between* e *and* $+$*. There is* $B \to A + B$ *as the only rule with* $+$*. Being* $e \in \mathcal{R}_G(B)$*, it holds* $e \gtrdot +$.
- *Consider the relations between* e *and* $\times$*. There is* $A \to E \times A$ *as the only rule with* $\times$*. Being* $e \in \mathcal{R}_G(E)$*, it holds* $e \gtrdot \times$.
- *Being that the* e *appears only in* $A \to e$*, then it has no other relations.*

To summarize, there has been found the following Operator Precedence Relations:
$\lessdot \; = \{++, +\times, +e, \times\times, \times e\}$
$\gtrdot \; = \{\times +, e+, e\times\}$
$\doteq \; = \emptyset$

Example 4.3. *Consider the Operator Grammar defined on example 3.5 which has as production set:* P :
$S \to B$

$A \to A \times A \mid e$
$B \to A + B \mid A$

Calculate the $\mathcal{L}_G$ and $\mathcal{R}_G$ sets:
$\mathcal{L}_G(S) = \{e, +, \times\}$
$\mathcal{L}_G(A) = \{e, \times\}$
$\mathcal{L}_G(B) = \{e, +, \times\}$
$\mathcal{R}_G(S) = \{e, +, \times\}$
$\mathcal{R}_G(A) = \{e, \times\}$
$\mathcal{R}_G(B) = \{e, +, \times\}$

In order to determine the Operator Precedence relations for this grammar, then:

- *Consider the relations between $+$ and $+$. There is $B \to A + B$ as the only rule with $+$. Being that it has only one $+$, there cannot be $+ \doteq +$; being $+ \notin \mathcal{R}_G(A)$, then there does not hold the $+ \gtrdot +$; being $+ \in \mathcal{L}_G(B)$, then $+ \lessdot +$.*
- *Consider the relations between $+$ and $\times$. We have $B \to A + B$ as the only rule with $+$. Being that it has only $+$ and not $\times$, there cannot be $+ \doteq \times$; being $\times \in \mathcal{L}_G(B)$, then there it holds $+ \lessdot \times$; being $A \to A \times A$ the only production with $\times$, and being $+ \notin \mathcal{R}_G(A)$, then it does not hold $+ \gtrdot \times$.*
- *Consider the relations between $+$ and e. There is $B \to A + B$ as the only rule with $+$. Being that it has only $+$ and not e, there cannot be $+ \doteq e$; being $e \in \mathcal{L}_G(B)$, then there it holds $+ \lessdot e$; being $A \to e$ the only production with e, then it is trivial to see that it does not hold $+ \gtrdot e$.*
- *Consider the relations between $\times$ and $+$. There is $A \to A \times A$ as the only rule with $\times$. Being that it has only $\times$ and not $+$, there cannot be $\times \doteq +$; being $+ \notin \mathcal{L}_G(A)$, then there it does not hold $\times \lessdot +$; being $B \to A + B$ and being $\times \in \mathcal{R}_G(A)$, then it holds $\times \gtrdot +$.*
- *Consider the relations between $\times$ and $\times$. There is $A \to A \times A$ as the only rule with $\times$. Being that it has only one occurrence of $\times$, there cannot be $\times \doteq \times$; being $\times \in \mathcal{L}_G(A)$, then it holds $\times \lessdot \times$; being $\times \in \mathcal{R}_G(A)$, then it holds $\times \gtrdot \times$.*
- *Consider the relations between $\times$ and e. There is $A \to A \times A$ as the only rule with $\times$. Being that it has only $\times$ and not e, there cannot be $\times \doteq e$; being $e \in \mathcal{L}_G(A)$, then there it holds $\times \lessdot e$; being $A \to e$ the only production with e, then it is trivial to see that it doesn't hold $\times \gtrdot e$.*
- *Consider the relations between e and $+$. We have $B \to A + B$ as the only rule with*

$+$. Being $e \in \mathcal{R}_G(B)$, it holds $e \gtrdot +$.

- *Consider the relations between e and $\times$. There is $A \to A \times A$ as the only rule with $\times$. Being $e \in \mathcal{R}_G(A)$, it holds $e \gtrdot \times$.*
- *Being that the e appears only in $A \to e$, then it has no other relations.*

To summarize, there have been found the following Operator Precedence Relations:
$\lessdot\ = \{++, +\times, +e, \times\times, \times e\}$
$\gtrdot\ = \{\times+, \times\times, e+, e\times\}$
$\doteq = \emptyset$

In order to represent the OP relations in a more human-readable way, there has been introduced the concept of Operator Precedence Matrix, or OPM:

Definition 4.4 (Operator Precedence Matrix). *For an OG $G = \langle \Sigma, V_N, P, S \rangle$, its corresponding operator precedence matrix (OPM) $M = OPM(G)$ is a $|\Sigma| \times |\Sigma|$ array such that, for each ordered pair $(a, b) \in (\Sigma \times \Sigma)$, the element M_{ab} is the set of OP relations holding between a and b.*

Example 4.4 (calculating OPM). *Consider the previous examples example 4.2 and example 4.3, here will be represented the OPMs relative to each grammar.*
The OPM of the grammat of example 4.2 is:

$$\begin{array}{cccc} & + & \times & e \\ + & \lessdot & \lessdot & \lessdot \\ \times & \lessdot & \gtrdot & \gtrdot \\ e & \gtrdot & \gtrdot & \end{array}$$

The OPM of the grammat of example 4.3 is:

$$\begin{array}{cccc} & + & \times & e \\ + & \lessdot & \lessdot & \lessdot \\ \times & \lessdot & \gtrdot\lessdot & \gtrdot \\ e & \gtrdot & \gtrdot & \end{array}$$

There have been presented all the concept needed to introduce the definition of Operator Precedence Grammar and Language.

Definition 4.5 (Operator Precedence Grammar and Language). *An OG $G = \langle \Sigma, V_N, P, S \rangle$ is an Operator Precedence Grammar, or Floyd grammar, iff its Operator Precedence Matrix, $OPM(G)$, is a conflict-free matrix, i.e., $\forall_{a,b \in \Sigma} |M_a b| \leq 1$.*

An Operator Precedence Language (OPL) is a language generated by an OPG.

Please note that in the previous examples, the grammar of example 4.2 satisfies the OPG definition, while example 4.3 does not. This is evident in the OPMs extracted in example 4.4, where it is trivial to see that for example 4.3 the OPM is not conflict free: $M_{\times,\times} = \{\lessdot, \gtrdot\}$.

4.2. Operator Precedence Languages Structure

As previously said, OPLs are structured Context-Free Languages, which implies that there must be a procedure that associates to each string belonging to an OPL its structure. Moreover, as it has been done for Structured CFL, also for OPL the concept of *universe structure* is defined.

The aim of this section is to define the structure and structure universe for OPL. However, in order to achieve such a goal, there is needed to introduce also other concepts, like OP-alphabet.

Definition 4.6 (OP-alphabet). *An operator Precedence Alphabet is a pair* (Σ, M) *where* Σ *is an alphabet and* M *is a conflict-free operator precedence matrix, i.e., a* $|\Sigma_\#| \times |\Sigma_\#|$ *array that associates at most one of the operator precedence relations:* $\doteq$, $\lessdot$ *or* $\gtrdot$ *to each ordered pair* $(a, b) \in \Sigma_\# \times \Sigma_\#$.

The # character, introduced in definition 2.1, is used as a delimiter of the string, so it yields precedence to other terminals and takes precedence over them (with a special case for $\# \doteq \#$).

As previously stated, there is a way to bind each string to its unique structure, and thus to the sequence of derivations performed by an OPG to generate the string. This procedure, for OPLs, relies on the concept of *chain*, which definition is the following:

Definition 4.7 (chain). *Let* (Σ, M) *be an OP-alphabet.*

- *A simple chain is a word* $a_0 a_1 a_2 \cdots a_n a_{n+1}$*, written as* $^{a_0}\lfloor a_1 a_2 \cdots a_n \rfloor^{a_{n+1}}$*, such that:* $a_0,\ a_{n+1} \in \Sigma_\#$*,* $a_i \in \Sigma$ *for every* $i : 1 \leq i \leq n$*,* $M_{a_0 a_{n+1}} \neq 0$*, and* $a_0 \lessdot a_1 \doteq a_2 \doteq \cdots \doteq a_n \gtrdot a_{n+1}$.
- *A composed chain is a word* $a_0 x_0 a_1 x_1 a_2 \cdots a_n x_n a_{n+1}$*, with* $x_i \in \Sigma^*$*, where* $^{a_0}\lfloor a_1 a_2 \cdots a_n \rfloor^{a_{n+1}}$ *is a simple chain, and either* $x_i = \varepsilon$ *or* $^{a_i}\lfloor x_i \rfloor^{a_{i+1}}$ *is a chain (either simple or composed), for every* $i : 0 \leq i \leq n$*. Such a composed chain will be written as* $^{a_0}\lfloor x_0 a_1 x_1 a_2 \cdots a_n x_n \rfloor^{a_{n+1}}$.
- *The body of a chain* $^{a}\lfloor x \rfloor^{b}$*, simple or composed, is the word* x.

Definition 4.8. *Given an OP-alphabet (Σ, M) and a string $s \in \Sigma^*$, with $M(s)$ is indicated the chain (if exists) built on $\# \cdot s \cdot \#$ using the OP-alphabet (Σ, M'), where M' is M with the addition of $M'_{\#a} = \{\lessdot\}$ and $M'_{b\#} = \{\gtrdot\}$ for each $a, b \in \Sigma \mid M_{\#a} = \emptyset$ and $M_{b\#} = \emptyset$.*

The definitions given so far are sufficient to prove the following results:

Corollary 4.2. *There cannot exists a chain similar to $\lfloor \cdots \rfloor \lfloor \cdots \rfloor$ but there must be at least one terminal between $\rfloor$ and $\lfloor$.*

Proof. The proof of this corollary relies on definition 4.7 and definition 4.1:

Noticing that $\lfloor \cdots \rfloor \lfloor \cdots \rfloor$ implies that there is a derivation with 2 consecutive non-terminals in the rhs of the rule used, it is in contradiction with the requirement of the grammar to be OG.

Noticing that $a \rfloor \lfloor b$ implies that $\cdots a \rfloor^b$ is a chain and ${}^a\lfloor b \cdots$ is a chain, it is trivial that in the OPM M, the set M_{ab} contains both $\lessdot$ and $\gtrdot$, which implies it is not conflict free, so (Σ, M) isn't an OP-alphabet. □

For more information about Operator Precedence Language and chains, refer to [3].

Corollary 4.3. *Given a OP-alphabet (Σ, M), 2 strings $u, v \in \Sigma^*$ and 2 terminals $c_0, c_l \in \Sigma$ such that ${}^{c_0}\lfloor u \rfloor^{c_l}$ and ${}^{c_0}\lfloor v \rfloor^{c_l}$ are chains, and a chain s containing as substring $c_0 u c_l$, then also ${}^{c_0}\lfloor u \rfloor^{c_l}$ is a chain inside s, and that the string z built replacing in s $c_0 u c_l$ with $c_0 v c_l$, is still a chain containing ${}^{c_0}\lfloor v \rfloor^{c_l}$ in place of ${}^{c_0}\lfloor u \rfloor^{c_l}$.*

Example 4.5. *Consider the language defined in example 4.2 and let us extend its OPM with the # character:*

	$+$	$\times$	e	$\#$
$+$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$
$\times$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
e	$\gtrdot$	$\gtrdot$		$\gtrdot$
$\#$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$

Consider the string $\#e + e \times e + e\#$ on which there can be trivially built
$\# \lessdot e \gtrdot + \lessdot e \gtrdot \times \lessdot e \gtrdot + \lessdot e \gtrdot \#$.
The first simple chain that can be seen is the $\lfloor e \rfloor$;
Then it can be extended with the complex chain $\lfloor \lfloor e \rfloor \times \lfloor e \rfloor \rfloor$; Then there can be added the first + obtaining $\lfloor \lfloor e \rfloor + \lfloor \lfloor e \rfloor \times \lfloor e \rfloor \rfloor \rfloor$;
Finally there can be built $\#\lfloor \lfloor \lfloor e \rfloor + \lfloor \lfloor e \rfloor \times \lfloor e \rfloor \rfloor \rfloor + \lfloor e \rfloor \rfloor \#$

Till now there has been provided only the concept of structure not yet the one of structure universe. In the following the concept of MaxLanguage is presented, that defines the structure universe for an Operator Precedence Language. Note the similarities in the construction of Max Language with the construction of the universe structure presented for structured CFL.

Definition 4.9. *Given an OP-alphabet (Σ, M), a word $w \in \Sigma^*$ is said to be compatible with M iff the two following conditions hold:*

- *For each pair of letters c, d, consecutive in w, $M_{cd} \neq 0$.*
- *For each substring x of $\#w\#$ such that $x = a_0 x_0 a_1 x_1 a_2 \cdots a_n x_n a_{n+1}$, if $a_0 \lessdot a_1 \doteq a_2 \cdots a_{n-1} \doteq a_n \gtrdot a_{n+1}$ and, for every$0 \leq i \leq n$, either $x_i = \varepsilon$ or ${}^{a_i}\lfloor x_i \rfloor^{a_{n+1}}$ is a chain (simple or composed), then $M_{a_0 a_{n+1}} \neq 0$.*

Definition 4.10 (MaxLanguage)**.** *Given an OP-alphabet (Σ, M), the set of all its compatible strings is called Max Language of M, or, $MaxLang_M$.*

Example 4.6. *Given the alphabet $\Sigma = \{a, b\}$ and the OPM M:*

$$\begin{array}{cccc} & a & b & \# \\ a & \lessdot & \doteq & \\ b & & \gtrdot & \gtrdot \\ \# & \lessdot & & \doteq \end{array}$$

The MaxLangage defined via the OP-Alphabet (Σ, M) is the set $\{a^n b^n \mid n \geq 0\}$. For example:
$M(ab) = \#\lfloor a \doteq b \rfloor\# \Rightarrow ab \in ML$
$M(aaabbb) = \#\lfloor a \lfloor a \lfloor a \doteq b \rfloor b \rfloor b \rfloor \# \Rightarrow aaabbb \in ML$
$M(aabbb) =$ *is not defined* $\Rightarrow aabbb \notin ML$.

As it has been done with languages, also with OPM there can be defined the set's operations to work on them.

Definition 4.11. *Given two conflict-free OPMs M_1 and M_2, there are defined the set inclusion and union:*
$M_1 \subseteq M_2$ *if* $\forall_{a,b \in \Sigma_\#} (M_1)_{ab} \subseteq (M_2)_{ab}$
$M = M_1 \cup M_2$ *iff* $\forall_{a,b \in \Sigma_\#} M_{ab} = (M_1)_{ab} \cup (M_2)_{ab}$.
Two matrices are compatible if their union is conflict-free. [1]
A matrix is total (or complete) if it contains no empty cell.

[1]If the matrices are not conflict-free, their union does not define an OPM.

Definition 4.12. *Given an OPL η and a OP-alphabet (Σ, M), then:*

- *If the OPM of η is contained in M, η is said to be contained or defined in the OP-alphabet (Σ, M);*
- *If the OPM of η is compatible with M, η is said to be compatible with the OP-alphabet (Σ, M).*

Corollary 4.4. *Given an OPL η and a OP-alphabet (Σ, M), if η is contained in the OP-alphabet (Σ, M), then $\eta \subseteq MaxLang_M$.*

Note that the chain definition does not depend on a particular language/grammar. This is due to the fact that chaining relies only on the OPM, thus on the OP-alphabet. So, consider any two OP languages $\eta_1, \eta_2 \subseteq \Sigma^*$ compatible with each other and having as OPMs M_1 and M_2 respectively, and the OPM $M = M_1 \cup M_2$ that defines an OP-alphabet (Σ, M). For any string $s \in MaxLang_M$, then:

- if $s \in \eta_1$, than the structure of s with respect to M_1 is the same built with respect to M;
- if $s \in \eta_2$, than the structure of s with respect to M_2 is the same built with respect to M;
- if $s \in (\eta_1 \cap \eta_2)$, than the structure of s with respect to M_1 is the same built with respect to M_2 which is the same built with respect to M.

Corollary 4.5. *Given an OP-alphabet (Σ, M) and a string $s \in \Sigma^*$, $M(s)$, if exists, starts with $\lfloor$ and ends with $\rfloor$.*

To end the excursus over the MaxLanguage, there must be reported the following results from [8], [7], [5], and [14]

Proposition 4.1.

- *If an OPM M is total, then $MaxLang_M = \Sigma^*$.*
- *Let (Σ, M) be an OP-alphabet where M is $\doteq$-acyclic. The class $\mathcal{G}_M$ contains an OPG, called the maxgrammar of M, denoted by $G_{max,M}$, which generates the maxlanguage $MaxLang_M$. For all grammars $G \in \mathcal{G}_M$, the inclusions $L(G) \subseteq MaxLang_M$ and $L(G_p) \subseteq L(G_{p,max,M} = L_p(M)$ hold, whereG_p and $G_{p,max,M}$ are the parenthesized versions of G and $G_{max,M}$, and $L_p(M)$ is the parenthesized version of $L(M)$.*
- *the closure properties of the family $\mathcal{L}_M$ of (Σ, M)-compatible OPLs defined by a total OPM are the following:*

- *$\mathcal{L}_M$ is closed under union, intersection, and set-difference, therefore also under complement.*
- *$\mathcal{L}_M$ is closed under concatenation.*
- *if matrix M is $\doteq$-acyclic, $\mathcal{L}_M$ is closed under Kleene star.*

Corollary 4.6. *The inclusion problem between OPL with compatible OPM is decidable.*

4.3. Non-Countingness for Operator Precedence Languages

In this section, the concept of Non-Countingness is defined for the class of OPL languages. Once the NC_{OP} family has been presented, there are reported some results regarding it. Note that the goal of the thesis is to find a particular subclass of this family of languages, so the concepts of this section will play a relevant role later.

In order to tailor the NC_S to become NC_{OP}, there must be defined the concept of *well structured* for OPL. There is presented the concept of *well-chained* that plays the role of well structured, and there is then reported the definition of NC_{OP} using the newly introduced concept.

Definition 4.13 (Well chained). *Given an OP- alphabet (Σ, M), a string $w \in MaxLang_M$, given a sub-string x of w and $a, b \in \Sigma_\#$ such that the occurrence of x in $\#.w.\#$ is preceded by a and followed by b, we say that x is well-chained on w iff ${}^a\lfloor x \rfloor^b$ is a chain.*

The string w is well-chained if $w \in MaxLang_M$.

From now on, talking of OPL, there can be used interchangeably the concepts of well-structured or well-chained features.

It is now trivial to mutate the definition 3.8 into the definition of NC_{OP} by using well chained instead of well-structured concept.

Definition 4.14 (NC_{OP}). *Given a OP-alphabet(Σ, M) and an OP language $\eta \subseteq \Sigma^*_{any}$, η is NC_{OP} (non counting) iff $\exists n > 1$ such that, for all strings $x, u, w, v, y \in \Sigma^*_{any}$, $m \in \mathbb{N}$ where w and uwv are well chained on xu^nwv^ny w.r.t M then $xu^nwv^ny \in \eta \Leftrightarrow xu^{n+m}wv^{n+m}y \in \eta$.*

One remarkable result taken from [15] is:

Theorem 4.1. *It is decidable whether an OPL is NC_{OP} or not.*

One interesting and useful result that can be proven here regarding the well-chained property is the following:

Theorem 4.2. *Given an OP-alphabet* (Σ, M) *and a string* $s \in MaxLang_M$ *such that* $s = x.u^3.w.v^3.y$ *and in which* w *and* $u.w.v$ *are well-chained in* s*, then* $u^2.w.v^2$ *is well-chained on* s*.*

Before giving the proof of the theorem, there is reported an example:

Example 4.7. *Consider the OP-alphabet* (Σ, M)*, where* $\Sigma = \{a, b\}$ *and* M *:*

$$\begin{array}{cccc} & a & b & \# \\ a & \lessdot & \doteq & \gtrdot \\ b & \doteq & \gtrdot & \gtrdot \\ \# & \lessdot & \lessdot & \doteq \end{array}$$

Consider the string $s = \#baaaabbbba\#$ *belonging* $MaxLang_M$*, and consider* $x = b$*,* $u = a$*,* $w = ab$*,* $v = b$*,* $y = a$*.*
$M(a.w.b) = M(aabb) =^{a}\lfloor ab\rfloor^{b}$ *and*
$M(a.u.w.v.b) = M(aaabbb) = {}^{a\lfloor a}\lfloor ab\rfloor^{b\rfloor b}$
thus w *and* $u.w.v$ *are well chained on* s*, so, the theorem says that also* $u^2.w.v^2$ *is well chained on* s*. Let us see if it is true:*
$M(a.u^2.w.v^2.b) = M(aaaabbbb) = {}^{a\lfloor a\lfloor a}\lfloor ab\rfloor^{b\rfloor b\rfloor b}$*, which is well chained.*
To conclude this example, the structure of s *is:*
$M(s) = \#\lfloor ba\lfloor a\lfloor a\lfloor ab\rfloor b\rfloor b\rfloor ba\rfloor\#$

Proof. Calling u_l, v_0 the last and the initial character of u and v respectively, by hypothesis it is that ${}^{u_l}\lfloor w\rfloor^{v_0}$ is a chain and that ${}^{u_l}\lfloor uwv\rfloor^{v_0}$ is a chain. By the local parsability of OPLs, it is that uwv contains the chain ${}^{u_l}\lfloor w\rfloor^{v_0}$, which, for corollary 4.3, implies that substituting the chain ${}^{u_l}\lfloor w\rfloor^{v_0}$ with ${}^{u_l}\lfloor uwv\rfloor^{v_0}$ there can be obtained ${}^{u_l}\lfloor uuwvv\rfloor^{v_0}$ which is a chain, and in particular, being ${}^{u_l}\lfloor u^2.w.v^2\rfloor^{v_0}$ a chain, $u^2.w.v^2$ is well chained on s. □

Please note that, in the previous theorem and proof, the assumption of having M total has not be made.

Example 4.8. *Modify the OPM* M *of example example 4.7 to become* M_{inc} *:*

$$\begin{array}{cccc} & a & b & \# \\ a & \lessdot & \doteq & \\ b & & \gtrdot & \gtrdot \\ \# & \lessdot & & \doteq \end{array}$$

In this case M_{inc} *is not complete, which implies that its MaxLanguage does not con-*

tain all the strings built over Σ. One example is $\#baaaabbbba\#$ that does not belong to $MaxLang_{M_{inc}}$.

Consider $s = \#aaaaaabbbbbb\# \in MaxLang_{M_{inc}}$ and $x = aa$, $u = a$, $w = ab$, $v = b$, $y = bb$, $s = x.u^3.w.v^3.y$.
$M_{inc}(a.w.b) = M_{inc}(aabb) =^a \lfloor ab \rfloor^b$
$M_{inc}(a.u.w.v.b) = M_{inc}(aaabbb) =^{a\lfloor a} \lfloor ab \rfloor^{b\rfloor^b}$ thus w and $u.w.v$ are well chained on s, thus, the theorem says that also $u^2.w.v^2$ is well chained on s. Let us check if it is true:
$M_{inc}(a.u^2.w.v^2.b) = M_{inc}(aaaabbbb) =^{a\lfloor a \lfloor a} \lfloor ab \rfloor^{b\rfloor^{b\rfloor^b}}$ so it is well chained.

To conclude this example, the structure of s is:
$M_{inc}(s) = \#\lfloor a\lfloor a\lfloor a\lfloor a\lfloor a\lfloor ab\rfloor b\rfloor b\rfloor b\rfloor b\rfloor b\rfloor\#$

Given theorem 4.2, one trivial result is:

Corollary 4.7 (Structure of $x.u^n.w.v^n.y$). *Given an OP-alphabet (Σ, M) and a string $s = x.u^n.w.v^n.y$ with w and $u.w.v$ that are well-structured, there is that $\forall i \in \mathbb{N}, i < n$ $u^i.w.v^i$ is well chained on s.*

Example 4.9. *Consider example 4.8:*
by choosing $x = y = \varepsilon$ and keeping $u = a$, $w = ab$, $v = b$, the string
$s = \#aaaaaabbbbbb\#$ could be seen as
$s = x.u^5.w.v^5.y$, on which w and $u.w.v$ are well chained. From the structure of s, which is
$M_{inc}(s) = \#\lfloor a\lfloor a\lfloor a\lfloor a\lfloor a\lfloor ab\rfloor b\rfloor b\rfloor b\rfloor b\rfloor b\rfloor\#$, it is trivial to see that not only $u^2.w.v^2$ is well chained on s, but also $u^3.w.v^3$ and $u^4.w.v^5$ are so. Also $u^5.w.v^5$ is well chained on s, but it is so by relying not on the last character of u and the first of v, thus it is quite lucky and this result cannot be generalized.

Lemma 4.1 ($MaxLang_M$ is NC_{OP}). *Given an OP-alphabet (Σ, M), the maxlanguage of M is NC_{OP}.*

Proof. The proof of this theorem is done by leveraging the NC_{OP} definition.
Given $s = xu^nwv^ny \in MaxLang_M$ with
w, uwv well-structured on s and $n \geq 3$, there can be considered
$s = x'uwvy'$ with $x' = xu^{n-1}y' = v^{n-1}y$ and
$z = x'u^2wv^2y'$.
For theorem 4.2 there is that, given ${}^{u_l}\lfloor uwv\rfloor^{v_0}$, the chain on s of uwv, such that uwv is well-structured, then also ${}^{u_l}\lfloor uuwvv\rfloor^{v_0}$ is a chain, and so, for corollary 4.3, there can be substituted in s the chain ${}^{u_l}\lfloor w\rfloor^{v_0}$ with ${}^{u_l}\lfloor uwv\rfloor^{v_0}$ still having a chain. The new string is equal

to z, and, being $z = xu^{n+1}wv^{n+1}y$, it has been proven that $xu^nwv^ny \in MaLang(M) \Rightarrow xu^{n+1}wv^{n+1}y \in MaxLang_M$. □

Said that, the following can be stated:

Corollary 4.8 ($LTO \cap MaxLang_M \subseteq NC_{OP}$)**.** *Given an LTO $\eta \in \Sigma^*$ and a OP-alphabet (Σ, M), then the language $\xi = \eta \cap MaxLang_M$ is NC_{OP}.*

Proof. Given a string
$s = xu^nwv^ny \in \xi$ such that w, uwv are well-parenthesized over M, then also
$s' = xu^{n+1}wv^{n+1}y \in \xi$.

This is due to the fact that if $s \in MaxLang_M$, then also
$s' \in MaxLang_M$, because $MaxLng_M$ is NC_{OP}.
Being $s \in \eta$, and being η NC, then also
$xu^{n+1}wv^ny \in \eta$, and thus $s' = xu^{n+1}wv^{n+1}y \in \eta$.
Thus, being s' recognized by η and $MaxLang_M$, then s' is recognized by the conjunction of those 2 languages. □

Example 4.10. *Consider the alphabet $\Sigma = \{a, b, c, [,], .\}$, ad a grammar G on it that has as production set:*
$S \to [A(.E)^+]$
$E \to c|[B(.E)^+]$
$A \to a$
$B \to b$
This grammar is OG, and we can build its OPM M:

	a	b	c	$[$	$]$	$.$	$\#$
a						$\gtrdot$	
b						$\gtrdot$	
c					$\gtrdot$	$\gtrdot$	
$[$	$\lessdot$	$\lessdot$				$\doteq$	
$]$					$\gtrdot$	$\gtrdot$	$\gtrdot$
$.$			$\lessdot$	$\lessdot$	$\doteq$	$\doteq$	
$\#$					$\lessdot$		

Please note that $L(G) \in MaxLang_M$, but $L(G) \neq MaxLang_M$, because, for instance, $[b.c] \in MaxLang_M \cap (\neg G)$, and, moreover, $[.] \in MaxLang_M$.
There is the possibility to write an LTO language to enforce the constraints that are missing over $MaxLang_M$ and that are present on G:
$LTO_1 : LT_3(\{\#[a, [a., a.c, a.[, .[b, [b., b.c, b.[, .c., c.c, c.[,$

$.c], c]], c].,].c,].[,]]], c]\#,]]\#\})$
LTO_1 will recognize all the strings of G, and will discard those that are malformed, for instance $[b.c] \notin LTO_1$, $[.] \notin LTO_1$; However LTO_1 will recognize some strings not recognized by G, for example $[a.[b.c.[b.c] \in LTO_1$.
By mixing the features of LTO_1 with the ones of $MaxLang_M$, there can be built the language $MaxLang_M \cap LTO_1$, which is equivalent to the language defined by G. Although this work may seem unnecessarily complex, please note that in this way the OPL part will calculate the structure of the string and LT the semantic correctness, and both those families of languages have good properties for parsing, making the parsing of long strings not only fast but also possible to split the workload on different computers.

Let us now check if $MaxLang_M \cap LTO_1$ in NC_{OP} or not. There will be done an example giving an explanation on how can be intuitively seen that the language is NC_OP.
Consider $s = [a.[b.[b.[b.c]]]]$ and analyze if $s \in MaxLang_M \cap LTO_1$;
Then check if $s' = [a.[b.[b.[b.[b.c]]]]]$ is contained in $MaxLang_M \cap LTO_1$.

Please note that $s \in LTO_1$, which implies, being LTO_1 NC, that also $s' \in LTO_1$.
Let us now analyze $M(s) = \#\lfloor[\lfloor a\rfloor.\lfloor[\lfloor b\rfloor.\lfloor[\lfloor b\rfloor.\lfloor[\lfloor b\rfloor.\lfloor c\rfloor]\rfloor]\rfloor]\rfloor]\rfloor\#$,
so $s \in MaxLang_M$.
Calling $x = [a.,\ y =],\ u = [b.,\ v =],\ w = [b.c]$, there is
$s = x.u^2.w.v^\cdot y$ with w and $u.w.v$ well chained on s.
$M(s') = \#\lfloor[\lfloor a\rfloor.\lfloor[\lfloor b\rfloor.\lfloor[\lfloor b\rfloor.\lfloor[\lfloor b\rfloor.\lfloor[\lfloor b\rfloor.\lfloor c\rfloor]\rfloor]\rfloor]\rfloor]\rfloor]\rfloor\#$,
so $s' \in MaxLang_M$, which is not surprising because $MaxLang_M$ is NC_{OP}.

Thus $s \in MaxLang_M \cap LTO_1$ and $s' \in MaxLang_M \cap LTO_1$.

This excursus over NC_{OP} family can now terminate by reporting some other results inspired by [15].

Theorem 4.3. *Counting and non-counting parenthesis languages are closed with respect to complement. Thus counting and non-counting OPLs are closed with respect to complement with respect to the MaxLanguage defined by OPM.*

Theorem 4.4. *Non-counting parenthesis languages and non-counting OPLs are closed with respect to union and therefore with respect to intersection.*

Theorem 4.5. *NC_{OP} is closed with respect to concatenation.*

Please note that a NC_{OP} language, strictly relies on the OPM:

Example 4.11. *Given the OP-alphabet (Σ, M), where $\Sigma = \{a\}$ and having M:*

$$\begin{array}{ccc} & \# & a \\ \# & \doteq & \lessdot \\ a & \gtrdot & \doteq \end{array}$$

The MaxLanguage defined over the OP-alphabet (Σ, M) is the language defined by a^*. Although this language is NC with respect to Regular Languages, it is not NC_{OP} because, for each string $s \in ML$, s has only one substring that is well chained on it, that is the string s itself.
Each string's chain will be $\#\lfloor aa \cdots a \rfloor \#$.

Example 4.12. *Given the OP-alphabet (Σ, M), where $\Sigma = \{a\}$ and having M:*

$$\begin{array}{ccc} & \# & a \\ \# & \doteq & \lessdot \\ a & \gtrdot & \lessdot \end{array}$$

The MaxLanguage defined over the OP-alphabet (Σ, M) is the language defined by a^* as in the example 4.11; However, despite in the other example, this time the language is NC_{OP}, because, inside strings belonging to the language, there are substrings well-chained and thus the definition definition 4.14 can be put in place.

4.4. Other Operator Precedence Languages Representations

As it has been done for the families of languages belonging to Chomsky's classification, also for OPLs there have been studies providing different ways to characterize the languages and to investigate their properties. In particular, there have been works to provide the characterization of automata recognizing OPLs, like [3], to generalize the concept of Regular Expressions to describe also OPLs, like [14], to find a logical characterization of OPLs, like [15], and to expand the results of OPLs to a more general class of languages, like [6].

For all those new characterizations for OPLs, there have been studies to prove their expressiveness and relate it with each other and with the Floyd's defined OPL family. The presence of all those representations provides a variety of equivalent notations useful to prove some properties in a simpler way.

In this section, there is presented the definition of Operator Precedence Expressions and Monadic Second Order Logic, how they relate with each other, and with the NC_{OP} family. To find more information about the characterizations here presented, refer to [15].

4.4.1. Operator Precedence Expressions

As it has been done for Regular Languages with Regular Expressions, also for OPLs there has been defined a way to represent languages with a series of expressions: the Operator Precedence Expressions, or OPE. The idea is to describe a language in such a way that hints at the form of strings belonging to that language. This is done using not only the alphabet on which the language is defined but extending it with also a meta alphabet used to express how to mix the different expressions together, in order to get a string belonging to the desired language. This meta alphabet extends the one used by Regular Expressions with the meta-characters [and], which are used to define the *fence* operation, one concept that will inspire definition 6.1 for LTOP.

Definition 4.15 (OPE). *Given an OP alphabet (Σ, M), where M is complete, an OPE E and its language $L_M(E) \subseteq \Sigma^*$ are defined as follows.*
The meta-alphabet of OPE uses the same symbols as REs, together with the two symbols '[' and ']'. Let E_1 and E_2 be two OPEs:

1. *$a \in \Sigma$ is an OPE with $L_M(a) = a$.*
2. *$\neg E_1$ is an OPE with $L_M(\neg E) = MaxLang(M) \setminus L_M(E)$.*
3. *$a[E_1]b$, called the fence operation, i.e., we say E_1 in the fence a, b, is an OPE with:*
   if $a, b \in \Sigma$: $L_M(a[E_1]b) = a \cdot \{x \in L_M(E_1) \mid M(a \cdot x \cdot b) = \lfloor a \cdot M(x) \cdot b \rfloor\} \cdot b$;
   if $a = \#, b \in \Sigma$: : $L_M(\#[E_1]b) = \{x \in L_M(E_1) \mid M(x \cdot b) = \lfloor M(x) \cdot b\} \cdot b$;
   if $a \in \Sigma, b = \#$: $L_M(a[E_1]\#) = a \cdot \{x \in L_M(E_1) \mid M(a \cdot x) = \lfloor a \cdot M(x) \rfloor\}$.
   Where E_1 must not contain $\#$.
4. *$E_1 \cup E_2$ is an OPE with $L_M(E_1 \cup E_2) = L_M(E_1) \cup L_M(E_2)$.*
5. *$E_1 \cdot E_2$ is an OPE with $L_M(E_1 \cdot E_2) = L_M(E_1) \cdot L_M(E_2)$, where E_1 doesn't contain $a[E_3]\#$ and E_2 does not contain $\#[E_3]a$, for some OPE E_3 and $a \in \Sigma$.*
6. *E_1^* is an OPE defined by $E_1^* = \bigcup_{n=0}^{\inf} E_1^n$, where $E_1^0 = \{\varepsilon\}$, $E_1^1 = E_1$, $E_1^n = E_1^{n-1} \cdot E_1$; $E_1^+ = E_1 \cdot E_1^*$.*

Among the operations defining OPEs, concatenation has the maximum precedence; set-theoretic operations have the usual precedence, and the fence operation is dealt with as a normal parenthesis pair.
A Star Free OPE is one OPE that does not use * or $^+$.

A major result for the class of languages defining via OPE, is the following:

Theorem 4.6. *The class of languages definable via OPEs coincides with the class of*

OPLs.
The class of languages definable via Star Free OPEs coincides with the class of NC_{OP}.

There can be now introduced another result that will be helpful for a later proofs:

Lemma 4.2 (derived operations)**.** *There can be used the following derived operations inside the OPEs:*

- $a \triangle b := a[\Sigma^+]b$
- $a \nabla b := \neg(a \triangle b) \cap a \cdot \Sigma^+ \cdot b$

It is trivial to see $a[E]b = a\triangle b \cap a \cdot E \cdot b$. Keeping in mind the fact that Σ^+ is a Star Free language, it can be proved the following:

Lemma 4.3 (Flat Normal Form)**.** *Any Star-Free OPE can be written in the following form, called flat normal form:*
$\bigcup_i \bigcap_j t_{i,j}$
Where the elements $t_{i,j}$ have either the form $L_{i,j}a_{i,j}\triangle b_{i,j}R_{i,j}$, or $L_{i,j}a_{i,j}\nabla b_{i,j}R_{i,j}$, or $H_{i,j}$, for $a_{i,j}, b_{i,j} \in \Sigma$, and $L_{i,j}, R_{i,j}, H_{i,j}$ star-free regular expressions.

4.4.2. Monadic Second Order Logic

The Monadic Second Order Logic is now presented with its definition and results. As the name suggests, this characterization stays to symbolic logic characterization as the Operator Precedence Expression stays to Regular Expressions. To find more information about this family, refer to [13].

Definition 4.16 (Monadic Second Order Logic for OPLs)**.** *Let (Σ, M) be an OP-alphabet, $\mathcal{V}_1$ a set of firs-order variables, and $\mathcal{V}_2$ a set of second-order (or set) variables. The $MSO_{(\Sigma,M)}$ (monadic second order logic over (Σ, M)) is defined by the the following syntax (the OP-alphabet will be omitted unless necessary to prevent confusion):*

$\theta := c(x) | x \in X | x < y | x \curvearrowright y | \neg\theta | \theta_1 \vee \theta_2 | \exists x\, \theta | \exists X\, \theta$
Where $c \in \Sigma_{\#}$, $x, y \in \mathcal{V}_1$, and $X \in \mathcal{V}_2$.

A MSO formula is interpreted over a (Σ, M) string w compatible with M, with respect to assignments $\mathfrak{v}_1 : \mathcal{V}_1 \to \{0, 1, \ldots, |w|+1\}$ and $\mathfrak{v}_2 : \mathcal{V}_2 \to \wp(\{0, 1, \ldots, |w|+1\})$, in this way:

- $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}_2 \models c(x)$ *iff* $\#w\# = w_1 c w_2$ *and* $|w_1| = \mathfrak{v}_1(x)$.
- $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}_2 \models x \in X$ *iff* $\mathfrak{v}_1(x) \in \mathfrak{v}_2(X)$.

- $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}_2 \models x < y$ *iff* $\mathfrak{v}_1(x) < \mathfrak{v}_1(y)$.
- $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}_2 \models x \curvearrowright y$ *iff* $\#w\# = w_1 a w_2 b w_3$, $|w_1| = \mathfrak{v}_1(x)$, $|w_1 a w_2| = \mathfrak{v}_1(y)$, *and* w_2 *is the frontier of a subtree of the syntax tree of* w.
- $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}_2 \models \neg\theta$ *iff* $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}_2 \not\models \theta$.
- $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}_2 \models \theta_1 \vee \theta_2$ *iff* $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}_2 \models \theta_1$ *or* $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}_2 \models \theta_2$.
- $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}_2 \models \exists x\, \theta$ *iff* $\#w\#, M, \mathfrak{v}'_1, \mathfrak{v}_2 \models \theta$ *for some* $\mathfrak{v}'_1$ *with* $\mathfrak{v}'_1(y) = \mathfrak{v}_1(y)$ *for all* $y \in \mathcal{V}_1 - \{x\}$.
- $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}_2 \models \exists X\, \theta$ *iff* $\#w\#, M, \mathfrak{v}_1, \mathfrak{v}'_2 \models \theta$ *for some* $\mathfrak{v}'_2$ *with* $\mathfrak{v}'_2(Y) = \mathfrak{v}_2(Y)$ *for all* $Y \in \mathcal{V}_2 - \{X\}$.

To improve readability there will be dropped $M, \mathfrak{v}_1, \mathfrak{v}_2$ *and* $\#$ *from notation whenever there is no risk of ambiguity; Furthermore there will be used standard abbreviations in formulae, like* $\wedge, \forall, \oplus(xor), x+1, x-1, x=y, x \leq y$.
The language of a formula ψ *without free variables is* $L(\psi) = \{w \in L(M) \mid w \models \psi\}$.

As done for OPEs, also in this case there are some interesting results, in particular:

Theorem 4.7 (MSO = OPE). *The class of languages defined via MSO corresponds to the OPL class of languages.*

The proof, given in [15], relies on lemma 4.3, and the idea of using the Flat Normal Form to characterize OPEs will be used (also) later for the purpose to relate OPE with the new family LTOP.

Another important result is the fact that by restricting MSO to formulae of First Order (FO), then the class of languages defined corresponds to the class of NC_{OP} languages. To find more information about MSO, referr to [15].

Theorem 4.8 (*FO* $\subseteq$ *SFOPE*). *For every FO formula* ψ *on an OP-alphabet* (Σ, M) *there is a star-free OPE* E *on* (Σ, M) *such that* $L(E) = L(\psi)$.

Theorem 4.9 ($NC_{OP} \subseteq FO$). *Non-Counting (Aperiodic) Operator Precedence languages are FO-definable.*

Theorem 4.10. *The following classes of languages are equivalent:*

- NC_{OP};
- *the family of languages definable via Star Free OPE;*

- *the family of languages definable via FO.*

5 Locally Testable Extended Languages

The goal of this thesis is to find a family of languages that is both NC_{OP} and definable in an LTO-like manner. This is done in order to study if the properties of Non-Counting Regular Languages can be transferred into the Operator Precedence word.

In this section, the LTEO family of languages is introduced. This class is useful to forge the new operator, *fence-subs* that is presented in the next chapter, in fact here is discussed how there can be applied substitution inside languages. This chapter and the LTEO family will play a central role in the discussion about decidability.

5.1. Substitution

Here is presented the substitution. The scope of application of this concept is to substitute inside strings generated by a certain language some particular characters with other specified strings. The characters that will be substituted are the elements of Φ. Here there is an example:

Example 5.1. *The goal of the substitution is to give the possibility to reuse languages to describe other languages, so, describing where and how to substitute other languages in the target one.*

Given a "meta language" $\xi = a\eta_1(a\eta_1 + a\eta_2)^*a$ *where*
$\eta_1 = bc^*b$, $\eta_2 = cb^*c$, *the substitution will be the tool to map* ξ *in* Σ^*, *without having the fancy characcters* η_1 *or* η_2 *inside the strings of* ξ. *So:*
$\xi = \{a\eta_1 a, a\eta_1 a\eta_1 a, a\eta_1 a\eta_2 a, a\eta_1 a\eta_1 a\eta_1 a, \ldots\}$
$\eta_1 = \{bb, bcb, bccb, bcccb, \ldots\}$
$\eta_2 = \{cc, cbc, cbbc, cbbbc, \ldots\}$
$subs(\xi) =$
$\{abba, abcba, abccba, abcccba, \ldots,$
$abbabba, abbabcba, abbabccba, \ldots, abcbabba, abccbabba, \ldots, abcbabcba, abcbabccba, \ldots,$

$abbacca, abbacbca, abbacbbca, abbacbbbca, \dots, abcbacca, abccbacca, \dots, \}$

To define substitution, there are some steps to be done:

- Defined how to identify blanks (defined in definition 2.1 as elements of Φ) inside a string or inside a language;
- Defined the concept of substitution applicable over only one string;
- Defined how to apply the substitution over whole languages.

5.1.1. Finding used Blanks

In order to identify blanks inside strings, there is defined a function ϕ such that, given a language $\eta \subseteq \Sigma_\Phi^*$, or a string $s \in \Sigma_\Phi^*$, it extracts the set of all and only the elements of Φ that are present in the string itself or on any string of the language:

Definition 5.1 (ϕ). *Given a string $s \in \Sigma_\Phi^*$, there will be defined*
$\phi(s) = \{c \in \Phi \mid s \notin (\Sigma_\Phi \setminus \{c\})^*\}$.

Given a language $\eta \subseteq \Sigma_\Phi^*$, there will be defined
$\phi(\eta) = \{c \in \Phi \mid \eta \not\subseteq (\Sigma_\Phi \setminus \{c\})^*\}$.

Please note that if η is such that $\eta = LT_k(S) \subseteq \Sigma_\Phi^*$ and S is clean$_{\mathrm{LT}}$, then $\phi(\eta)$ is the smallest subset of Φ such that $S \subseteq (\Sigma \cup \phi(\eta))^*$.

Example 5.2. *Analyze the languages $\eta \subseteq \Sigma_\Phi^*$ defined as*
$\eta = a -_1 (a -_1 + a -_2)^* a$ *and*
$S_1 = \{\#a-_1, a -_1 a, -_1a-_2, -_1a-_1, a -_2 a, -_2a-_1, -_2a-_2, -_1a\#\}$ *with the language* $\xi_1 = LT_3(S_1)$ *and*
$S_2 = \{\#a-_1, a -_1 a, -_1a-_2, -_1a-_1, a -_2 a, -_2a-_1, -_2a-_2, -_1a\#, -_1a-_3\}$ *with the language* $\xi_2 = LT_3(S_2)$.

Analyze the ϕ function on those languages:
$\phi(\eta) = \{-_1, -_2\}$
$\phi(\xi_1) = \phi(LT_3(S_1)) = \{-_1, -_2\}$
$\phi(\xi_2) = \phi(LT_3(S_2)) = \{-_1, -_2\}$

Note that $\xi_1 = \xi_2$ and $\phi(\xi_1) = \phi(\xi_2)$, however
$S_1 \neq S_2$ *and* $S_2 \not\subseteq (\Sigma \cup \phi(LT_3(S_2)))^*$, *that is because S_2 is not clean$_{LT}$.*

5.1.2. Single String Substitution

Here is formulated the concept of substituting the elements of Φ inside a string. It is done incrementally, starting from the simple case of substituting each occurrence of one same *blank* with the same string, then substituting the occurrences of the same *blank* with a string taken from a language, and, finally, extending it to substitute the occurrences of different *blanks* each one with a string from a different language.

The following definition of the function $stringSub(t, -, s)$, takes as input two strings $t, s \in \Sigma_\Phi^*$ and a blank $- \in \Phi, - \notin \phi(s)$, and returns the string t' that is equal to t where of all the occurrences of $-$ that are replaced with s. In the following definitions, for the sake of simplicity, are used t that stands for "target", s stands for "substitution", and $-$ is the default representation of any blank belonging to Φ.

Definition 5.2 (Substitution of a blank with a string). *$stringSub : \Sigma_\Phi^* \times \Phi \times \Sigma_\Phi^* \to \Sigma_\Phi^*$ where there is $z = stringSub(t, -, s) \Leftrightarrow$*

1. *If t does not contain $-$, then $z = t$;*
2. *Else there exist 2 strings $w_1 \in (\Sigma_\Phi \setminus \{-\})^*$, $w_2 \in \Sigma_\Phi^*$ such that $t = w_1 \cdot - \cdot w_2$, then $z = w_1 \cdot s \cdot stringSub(w_2, -, s)$.*

Example 5.3. *Consider $\Sigma = \{a, b, c\}$, $\Phi = \{-_1, -_2\}$, and the strings*
$t = a -_1 a -_2 a -_2 -_1 a$, $s_1 = bc$ and $s_2 = cb$.
Let us use the stringSub to see how it works:
$y = stringSub(t, -_1, s_1) = a \cdot bc \cdot stringSub(a -_2 a -_2 -_1 a, -_1, s_1) =$
$abc \cdot a -_2 -_2 \cdot bc \cdot stringSub(a, -_1, s_1) = abca -_2 -_2 bca$
$x = stringSub(y, -_2, s_2) = abca \cdot cb \cdot stringSub(-_2 bca, -_2, s_2) =$
$abcacb \cdot cb \cdot stringSub(bca, -_2, s_2) = abcacbcbbca$

the previous concept is now extended in order to substitute inside the original string t not always the same string s but an entire language. In this case, the *stringSub* function returns not a single string but the language composed by all the strings produced by all possible substitutions:

Definition 5.3 (Substitution of a blank with a language). *$stringSub : \Sigma_\Phi^* \times \Phi \times \wp(\Sigma_\Phi^*) \to \wp(\Sigma_\Phi^*)$ where there is $z \in stringSub(t, -, \eta) \Leftrightarrow$*

1. *If t does not contains $-$, then $z = t$;*
2. *Else there exists 4 strings*
   $w_1 \in (\Sigma_\Phi \setminus \{-\})^*$, $w_2 \in \Sigma_\Phi^*$, $s \in \eta$, $z' \in stringSub(w_2, -, \eta)$ such that

$t = w_1 \cdot - \cdot w_2$, *then* $z = w_1 \cdot s \cdot z'$.

Example 5.4. *Consider* $\Sigma = \{a, b, c\}$, $\Phi = \{-\}$, *te string* $t = a - a - a$ *and the language* $\eta = b^*$. *Let us use the stringSub to see how it works:*
$stringSub(t, -, \eta) = \{a\varepsilon \cdot stringSub(a - a, -, \eta), ab \cdot stringSub(a - a, -, \eta),$
$abb \cdot stringSub(a - a, -, \eta), abbb \cdot stringSub(a - a, -, \eta), \dots\};$
$stringSub(a - a, -, \eta) = \{a\varepsilon \cdot stringSub(a, -, \eta), ab \cdot stringSub(a, -, \eta),$
$abb \cdot stringSub(a, -, \eta), abbb \cdot stringSub(a, -, \eta), \dots\}$
$stringSub(a, -, \eta) = \{a\}$
$stringSub(a - a, -, \eta) = \{aa, aba, abba, abbba, \dots\}$
$stringSub(t, -, \eta) = \{aaa, aaba, aabba, aabbba, \dots, abaa, ababa, ababba,$
$ababbba, \dots, abbaa, abbaba, abbabba, abbabbba, \dots\};$

One last step to be done is to define how to substitute multiple blanks inside a string, assigning to each one of them a specific language from which extract the strings to substitute in the target. Here is an example.

Example 5.5. *Consider* $\Sigma = \{a, b, c\}$, $\Phi = \{-_1, -_2\}$, *the string* $t = a -_1 b -_2 -_1$ *and the languages* $\eta_1 = b^*$ *and* $\eta_2 = c^*$. *The new definition of stringSub will be the one that permits to substitute* $-_1$ *with* η_1 *and* $-_2$ *with* η_2, *so to define the language* $a\eta_1 b\eta_2\eta_1 = ab^*bc^*b^*$.

The function *stringSub* needs to have a way to map each blank of $\phi(s)$ to a target language, from which take the string with which perform the substitution. In order to do so, the *stringSub* takes as input a function $map : \Phi \to \wp(\Sigma_\Phi^*)$ instead of the blank and the language to substitute.

Definition 5.4 (Substitution of blanks with a languages)**.** $stringSub : \Sigma_\Phi^* \times (\Phi \to \wp(\Sigma_\Phi^*)) \to \wp(\Sigma_\Phi^*)$ *where there is*
$z \in stringSub(t, map) \Leftrightarrow$

1. *If* $|\phi(t)| = 0$, *then* $z = t$;
2. *Else there exist 4 strings,* $w_1 \in \Sigma^*$, $w_2 \in \Sigma_\Phi^*$, $s \in map(-_i)$, $z' \in stringSub(w_2, -_i)$, *and the blank* $-_i \in \Phi$, *such that* $t = w_1 \cdot -_i \cdot w_2$, *then* $z = w_1 \cdot s \cdot z'$.

Please note that $stringSub(t, -, \eta) = stringSub(t, \{- \to \eta\})$, so there can be used only definition 5.4 to cover all the other definitions given for *stringSub*.

Here is an example:

Example 5.6. *Consider* $\Sigma = \{a, b, c\}$, $\Phi = \{-_1, -_2\}$, *the string* $t = a -_1 b -_2 -_1$, *and the languages* $\eta_1 = b^*$ *and* $\eta_2 = c^*$.
Let us see how to calculate $stringSub(t, \{-_1 \to \eta_1;\ -_2 \to \eta_2\})$*:*
$stringSub(a -_1 b -_2 -_1, \{-_1 \to b^*;\ -_2 \to c^*\}) = \{$
$a \cdot \varepsilon \cdot stringSub(b -_2 -_1, \{-_1 \to b^*;\ -_2 \to c^*\}),$
$a \cdot b \cdot stringSub(b -_2 -_1, \{-_1 \to b^*;\ -_2 \to c^*\}),$
$a \cdot bb \cdot stringSub(b -_2 -_1, \{-_1 \to b^*;\ -_2 \to c^*\}), \ldots\}$
$stringSub(b -_2 -_1, \{-_1 \to b^*;\ -_2 \to c^*\}) = \{$
$b \cdot \varepsilon \cdot stringSub(-_1, \{-_1 \to b^*;\ -_2 \to c^*\}),$
$b \cdot c \cdot stringSub(-_1, \{-_1 \to b^*;\ -_2 \to c^*\}),$
$b \cdot ccc \cdot stringSub(-_1, \{-_1 \to b^*;\ -_2 \to c^*\}),$
$b \cdot cccc \cdot stringSub(-_1, \{-_1 \to b^*;\ -_2 \to c^*\}), \ldots\}$
$stringSub(-_1, \{-_1 \to b^*;\ -_2 \to c^*\}) = \{\varepsilon, b, bb, bbb, \ldots\}$
$stringSub(b -_2 -_1, \{-_1 \to b^*;\ -_2 \to c^*\}) = \{$
$b, bb, bbb, \ldots, bc, bcb, bcbb, \ldots, bcc, bccb, bccbb, \ldots\}$
$stringSub(a -_1 b -_2 -_1, \{-_1 \to b^*;\ -_2 \to c^*\}) = \{$
$ab, abb, abbb, abc, abcb, abcbb, abcc, abccb, abbc, abbbc, abbbcc, abbbbccccbb, \ldots\}$

5.1.3. Substitution for Languages

Here is presented the definition of *stringSub* allowing it to work also over languages.

The case of $\Phi = \{-\}$ is the easiest one, thus there can be examined first:

Definition 5.5 (substitution over language with one only blank)**.** *Given a language* $\eta \subseteq \Sigma^*_\Phi$ *and a language* $\xi \in \Sigma^*$, *the language* $stringSub(\eta, \xi)$ *is defined as:*
$stringSub(\eta, \xi) = \{s \in \Sigma^*_\Phi \mid \exists t \in \eta \land s \in stringSub(t, -, \xi)\}$

Example 5.7. *Consider the alphabet* $\Sigma = \{a, b\}$, $\Phi = \{-\}$, $\eta = a(-a)^*$, $\xi = b^*$, *then the language* $stringSub(\eta, \xi) = \{stringSub(a, -, \xi), stringSub(a - a, -, \xi), stringSub(a - a - a, -, \xi), \ldots\} = \{a, aa, aba, abba, abbba, aaa, aaba, abaa, ababa, \ldots\}$.

In the more general case, with $|\Phi| > 1$, another approach should be used. As it has been done for definition 5.4, there must be defined a function that maps each (used) element of Φ into a language. So the definition of the new substitution would be:

Definition 5.6 (substitution over language with more blanks)**.** *Given a language* $\eta \subseteq \Sigma^*_\Phi$ *and a function*

map : $\Phi \to \wp(\Sigma_\Phi^*)$ *with* $\forall_{-\in\phi(\eta)}\phi(map(-)) \cap \phi(\eta) = \emptyset$, *the language* $stringSub(\eta, map)$ *is defined as:*
$stringSub(\eta, map) = \{s \in \Sigma_\Phi^* \mid \exists t \in \eta \wedge s \in stringSub(t, map)\}$.

Please note that when $\Phi = \{-\}$, the two are equivalent:
$stringSub(\eta, \xi) = stringSub(\eta, \{- \to \xi\})$.

Example 5.8. *Consider the alphabet* $\Sigma = \{a, b, c\}$,
$\Phi = \{-_1, -_2\}$, $\eta = a(-_1a + -_2a)^*$, $\xi_1 = b^*$, $\xi_2 = c^*$, *then the language*
$stringSub(\eta, \{-_1 \to \xi_1;\ -_2 \to \xi_2) =$
$\{stringSub(a, \{-_1 \to \xi_1;\ -_2 \to \xi_2)),$
$stringSub(a -_1 a, \{-_1 \to \xi_1;\ -_2 \to \xi_2)),$
$stringSub(a -_2 a, \{-_1 \to \xi_1;\ -_2 \to \xi_2)),$
$stringSub(a -_1 a -_2 a, \{-_1 \to \xi_1;\ -_2 \to \xi_2)) \ldots\} =$
$\{a, aa, aba, abba, aca, acca, aaa, aba, aaca, acaba, \ldots\}$.

5.2. Language Definition

The Locally Testable Extended (LTE) family of languages is here defined by leveraging the *stringSub* concept presented before. The idea behind this family is to build a LT language over Σ_Φ and than apply substitution over the language in order to obtain a new language over Σ^*. The goal behind this is to embed somehow the fence operator inside the substitution, in order to bind LTOs with OPs, however, how it will be discussed later: by simply applying the *stringSub* and then intersecting the obtained language with a Max Language there is not obtained what we are looking for.

Definition 5.7 (LTE)**.** *The family of LTE languages is the smallest set containing a language* $\eta \in \Sigma^*$ *such that:*

1. $\eta = \emptyset$; *or*

2. η *is a LT language; or*

3. *There exists a LT language* $\xi \subseteq \Sigma_\Phi^*$ *and a function* $map : \Phi \to LTE$ *such that* $\eta = stringSub(\xi, map)$.

Here there are a few quick examples:

Example 5.9. *Consider the alphabet* $\Sigma = \{a, b, c\}$, $\Phi = \{-_1, -_2\}$,
$S = \{\#a-_1, a -_1 a, -_1a-_2, -_1a-_1, a -_2 a, -_2a-_1, -_2a-_2, -_1a\#\}$ *and*
$\eta_1 = b^*$, $\eta_2 = c^*$.

Noticing that both η_1 *and* η_2 *are LT, the language*
$stringSub(LT_3(S), \{-_1 \to \eta_1;\ -_2 \to \eta_2\})$ *is LTE, and is defined as:*
$LT_3(S) = a -_1 (a -_1 + a -_2)^* a$
$stringSub(LT_3(S), \{-_1 \to \eta_1;\ -_2 \to \eta_2\}) = ab^*(ab^* + ac^*)^* a$

Example 5.10. *Consider* $\Sigma = \{a\},\ \Phi = \{-\}$*, the language*
$\eta = \{aaa\}$ *and* $\xi = LT(S),\ S = \{\#a, a-, -a, a\#\}$.
It is trivial to see that $\eta \in LTE$.
Consider $map = \{- \to \eta\}$*, then the language defined as*
$stringSub(\xi, map)$ *is LTE. In particular we have:*
$\xi = \{a, a-a, a-a-a, a-a-a-a, ...\} = a(-a)^*$
$stringSub(\xi, \eta) = \{a, aaaaa, aaaaaaaaa, ...\} = a(aaaa)^*$

As it has been done for the LT languages with the definition of LTO, let us extend the expressiveness of LTE with LTEO:

Definition 5.8 (LTEO)**.** *LTEO is the smallest set of languages such that:*

1. *LTEO contains LTE;*
2. *LTEO contains* $stringSub(\xi, map)$ *with* ξ *that is LT on* Σ_Φ^* *and* $map : \Phi \to LTEO$*;*
3. *LTEO is closed under Boolean operators and concatenation.*

5.3. Main Results

As it has been said at the beginning of this chapter, there are no particular results for this family of languages, however, it represents a step that has been taken in the direction of LTOP, and that will be useful for later proof of LTOP. Lots of the following results point out features that are not present in this family and that are desirable in the LTOP one.

Please note that the substitution in the LTEO family works something like the concatenation inside Regular Expressions, with the difference that for LTEO the expressions are generated by an LT language.

Example 5.11 (LTE and RegExp)**.** *Consider the Regular Expressions:*
$RE_1 = (ab)^+$
$RE_2 = (b^+.RE_1.a^+)^+$
it is trivial to translate those regular expressions into LTE:
$LTE_1 = LT_2(\{\#a, ab, ba, b\#\}$
$LTE_i = LT_2(\{\#b, bb, b-, -a, aa, ab, a\#\})$

$LTE_2 = stringSub(LTE_i, LTE_1)$
With this construction, it is easier to translate Regular Expressions into LT-like format, thus thanks to the fact that this new representation is more expressive than LTO.

Lemma 5.1 ($LTO \subseteq LTEO$)**.** *LTO family is contained in LETO.*

Which implies

Corollary 5.1 ($NC \subseteq LTEO$)**.** *NC is contained in LTEO.*

At this point, it may arise the question: "is it possible that LTEO defines NC?". To answer this question, please note that if $LTEO \subseteq NC$, having $LTO \subseteq LTEO$ and being $NC = LTO$, then it would be $LTEO = LTO$, which seems unfeasible. However, here is a result that will answer the question:

Theorem 5.1 ($LTE \not\subseteq NC$)**.** *Not all LTE languages are Non-Counting*

Proof. The proof of this theorem consists of the constatation that $a(aaa)^*$ is not a NC language but it is LTE, as proved in example 5.10. □

So, $LTEO \neq LTO$. This also enables to state the following:

Corollary 5.2 ($NC \subset LTEO$)**.** *NC is strictly contained in LTEO, but there are LTEO languages that are not NC.*

Let us now add some examples relating LTO with OPLs:

Example 5.12. *Given the OP-alphabet* (Σ, M)*, where* $\Sigma = \{a\}$ *and having M:*

$$\begin{array}{ccc} & \# & a \\ \# & \doteq & \lessdot \\ a & \gtrdot & \doteq \end{array}$$

Given $\eta = LT_2(\{\#a, a-, -a, -\#\})$*,* $\xi = \{a\}$*, then the LTE language generated by* $stringSub(\eta, \xi) = a^{2n}$*, which, absurdely, is* NC_{OP} *because no substring in* $s \in stringSub(\eta, \xi)$ *is well chained.*

Example 5.13. *Given the OP-alphabet* (Σ, M)*, where* $\Sigma = \{a\}$ *and having M:*

$$\begin{array}{ccc} & \# & a \\ \# & \doteq & \lessdot \\ a & \gtrdot & \lessdot \end{array}$$

Given $\eta = LT_2(\{\#a, a-, -a, -\#\})$, $\xi = \{a\}$, then the LTE language generated by $stringSub(\eta, \xi) = a^{2n}$, which trivially isn't NC_{OP} because, even if its substrings are well-chained, if $a^m \in stringSub(\eta, \xi)$, then $a^{m+1} \notin stringSub(\eta, \xi)$.

Example 5.14. *Given the OP-alphabet (Σ, M), where $\Sigma = \{a, b, c\}$ and having M:*

	$\#$	a	b	c
$\#$	$\doteq$	$\lessdot$		
a			$\lessdot$	
b				$\lessdot$
c	$\gtrdot$	$\lessdot$		

*Given $\eta = LT_4(\{\#abc, abca, bca-, ca-c, a-ca, -cab, cabc, abc\#\})$,
$\xi = \{b\}$, then the LTE language generated by $stringSub(\eta, \xi) = (abc)^{2n}$, which trivially is not NC_{OP} because, even if its substrings are well-chained, if
$(abc)^m \in stringSub(\eta, \xi)$, then $(abc)^{m+1} \notin stringSub(\eta, \xi)$.*

6 | Locally Testable Languages over Operator Precedence

Here is presented the LTOP family of languages. In order to define it, there are used some of the LTO's features, and some of the OPE's ones, to get a family that is NC_{OP}. After having defined it, there is presented the way it relates to the other characterizations of OP, with particular attention to the ones already presented and their Non-Counting versions, and, leveraging those comparisons, there will be proved that the LTOP family is not only contained into NC_{OP}, but it coincides with the NC_{OP} family.

As previously hinted, the LTOP family is an evolution of the LTEO family, using the substitution, but, because of the great expressiveness of *stringSub*, shown in corollary 5.2, there is set a restriction on the way with which the substitutions are performed: the substitutions are done in a fence-like manner. So, after having defined the new substitution function, the *fence-subs*, there are presented the definition of LTOP, its results, and its relationship with the families of MSO and OPE definable languages.

6.1. Fence Substitution

In order to define the LTOP family, it is needed to introduce a new operation that collimates the peculiarities of *stringSub* and the fence operation of OPEs. The idea is to build a *stringSub* that substitutes only substrings that constitute a complete subtree of the tree structure of the final string, or, equivalently, if the part to be substituted is well-chained over the final string.

Example 6.1. *Given the OP-alphabet defined via $\Sigma = \{a, b, c\}$ and the OPM M:*

$$\begin{array}{ccccc}
 & a & b & c & \# \\
a & \lessdot & \lessdot & \doteq & \\
b & & \doteq & \gtrdot & \\
c & \lessdot & & & \gtrdot \\
\# & \lessdot & & & \doteq
\end{array}$$

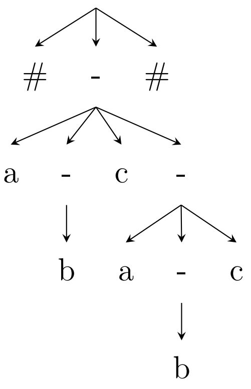


Figure 6.1: Structure of *abcabc*

The string $s = abcabc$, for which there is $M(s) = \#\lfloor a\lfloor b\rfloor c\lfloor a\lfloor b\rfloor c\rfloor\rfloor\#$, which means that the structure of the string is the one reported in fig. 6.1.

In the figure there have been indicated with "-" the nodes for which there are the only nodes for which there is the possibility to apply the fence-subs operation, because, even if the substitution has no limitations, the fence requires to substitute substrings that are well-chained on the final string, so substrings that in the structure tree, are part of a complete subtree of it.

As it has been done for *stringSub*, the definition of *fence-subs* is given before operating over a single string, and then it is expanded to operate over a whole language.

Definition 6.1 (*fence-subs*). *Given an OP-alphabet (Σ, M), a string $s \in \Sigma_\Phi$ and function map : $\Phi \to \Sigma^*$ that maps each element of $\phi(s)$ to one string $x_i \in \Sigma^*$, the string fence-subs(s, map), if exists, will be the string s in which each ocurrence of each element of $\phi(s)$, $-_j$, is substituted by $map(-_j) = x_j$, only if, given that $-_j$ in $\#\cdot s\cdot\#$ is surrounded by the characters $a, b \in \Sigma_{\Phi,\#}$, then:*

- *if $a, b \in \Sigma$: $M(a \cdot map(-_j) \cdot b) = \lfloor a \cdot M(map(-_j)) \cdot b\rfloor$;*
- *if $a = \#, b \in \Sigma$: : $M(map(-_j) \cdot b) = \lfloor map(-_j) \cdot b\rfloor$;*
- *if $a \in \Sigma, b = \#$: $M(a \cdot map(-_j)) = \lfloor a \cdot map(-_j)\rfloor$.*

Example 6.2. *Given the OP-alphabet defined via $\Sigma = \{a, b, c\}$ and the OPM M:*

$$\begin{array}{ccccc} & a & b & c & \# \\ a & \lessdot & \lessdot & \doteq & \\ b & & \doteq & \gtrdot & \\ c & \lessdot & & & \gtrdot \\ \# & \lessdot & & & \doteq \end{array}$$

Given the string $s = a - c$, consider fence-subs$(s, \{- \rightarrow b\})$:
$M(abc) = \lfloor a \lfloor b \rfloor c \rfloor$, $M(b) = \lfloor b \rfloor$, thus $aca =$ fence-subs$(s, \{- \rightarrow c\})$.

Consider fence-subs$(s, \{- \rightarrow bcab\})$:
$M(abcabc) = \lfloor a \lfloor b \rfloor c \lfloor a \lfloor b \rfloor c \rfloor \rfloor$, $M(bcab) = \lfloor \lfloor b \rfloor c \lfloor a \lfloor b \rfloor \rfloor \rfloor$, and it is trivial to see that bcab is not well-chained on abcabc, then fence-subs$(s, \{- \rightarrow bcab\})$ is not defined.

definition 6.1 defines how *fence-subs* works t substitute a single string, however, it is needed to extend this definition to work also with languages:

Definition 6.2. *Given an OP-alphabet (Σ, M), a string $s \in \Sigma_\Phi$ and a function $map : \Phi \rightarrow \wp(\Sigma^*)$ that maps each element of $\phi(s)$ to one language $\eta_i \subseteq \Sigma^*$, the language fence-subs(s, map) will be the set of all the possible strings derived from s in which each occurrence of each element of $\phi(s)$, $-_j$, is substituted by $x_j \in \eta_j = map(-_j)$, only if, given that the occurrence of $-_j$ in $\# \cdot s \cdot \#$ is surrounded by the characters $a, b \in \Sigma_{\Phi,\#}$, then:*

- *if $a, b \in \Sigma$: $M(a \cdot x_j \cdot b) = \lfloor a \cdot M(x_j) \cdot b \rfloor$;*
- *if $a = \#, b \in \Sigma$: $M(x_j \cdot b) = \lfloor M(x_j) \cdot b \rfloor$;*
- *if $a \in \Sigma, b = \#$: $M(a \cdot x_j) = \lfloor a \cdot M(x_j) \rfloor$.*

Example 6.3. *Given the OP-alphabet defined via $\Sigma = \{a, b, c\}$ and the OPM M:*

$$\begin{array}{ccccc} & a & b & c & \# \\ a & \lessdot & \lessdot & \doteq & \\ b & & \doteq & \gtrdot & \\ c & \lessdot & & & \gtrdot \\ \# & \lessdot & & & \doteq \end{array}$$

Given the string $s = a -_1 c -_2 a$ and the languages $\eta_1 = \{b, c\}$,
$\eta_2 = (ab)^* + c^*$, then let us analyze the language fence-subs$(s, \{-_1 \rightarrow \eta_1;\ -_2 \rightarrow \eta_2\})$. Please note that c cannot be fence-substituted in place of $-_1$, because it will not be well-chained, while b can, so:
fence-subs$(s, \{-_1 \rightarrow \eta_1;\ -_2 \rightarrow \eta_2\}) = \{$
$a \cdot \{b\} \cdot$ fence-subs$(c -_2 a, \{-_1 \rightarrow \eta_1;\ -_2 \rightarrow \eta_2\})\}$
By noticing that any repetition of c (exception given for ε) will not be well-chained in place of $-_2$ inside $c -_2 a$, then c^* will not be used to generate the language; however, any

any string belonging to $(ab)^*$ *will be well-chained, so:*
fence-subs$(c -_2 a, \{-_1 \rightarrow \eta_1;\ -_2 \rightarrow \eta_2\}) = \{ca, caba, cababa, \dots\}$
Which implies that:
fence-subs$(s, \{-_1 \rightarrow \eta_1;\ -_2 \rightarrow \eta_2\}) = \{abca, abcaba, abcabcaba, \dots\}$

Lastly, the *fence-subs* can be extended to work not only on a single string s on which perform the substitution but it can also be extended to accept an entire language ξ:

Definition 6.3 (*fence-subs*). *Given an OP-alphabet* (Σ, M)*, a language* $\xi \in \wp(\Sigma_\Phi)$ *and a function* $map : \Phi \rightarrow \wp(\Sigma^*)$ *that maps each element of* $\phi(\xi)$ *to one language* $\eta_i \subseteq \Sigma^*$*, the language fence-subs*(ξ, map) *is defined as:*
$\bigcup_{s \in \xi}$ *fence-subs*(s, map)

Given those definitions and corollary 4.2, there can be seen that the operation *fence-subs* does not apply to strings s such that $-_i -_j$ is a substring of s.

Corollary 6.1. *Given any OP-alphabet* (Σ, M)*, a function* $map : \Phi \rightarrow \wp(\Sigma^*)$ *and a string* $s \in \Sigma_\Phi$*, then fence-subs*(s, map) *may contain at least a string only if there does not exist any two (possibly equal) blanks* $-_i, -_j \in \Phi$ *such that* $-_i -_j$ *is a substring of* s*.*

Example 6.4. *Consider the OP-alphabet used in example 6.3 and change it by adding the relation* $c \gtrdot c$*, so having as alphabet* $\Sigma = \{a, b, c\}$ *and the OPM M:*

	a	b	c	$\#$
a	$\lessdot$	$\lessdot$	$\doteq$	
b		$\doteq$	$\gtrdot$	
c	$\lessdot$		$\boldsymbol{\gtrdot}$	$\gtrdot$
$\#$	$\lessdot$			$\doteq$

Consider the languages $\eta_1 = b^*$, $\eta_2 = a^+c^+$ *and* $\xi = ((a -_1 c)^*(a -_2 c)^*)^*$ *and see how to build fence-subs*$(\xi, \{-_1 \rightarrow \eta_1, -_2 \rightarrow \eta_2\})$.
Please note that inside the string $a -_2 c \in \xi$*, the* $-_2$ *character can be substituted only by strings in the form of* $a^n b^n$.
$\eta_1 = \{\varepsilon, b, bb, \dots\}$
$\eta_2 = \{ac, aac, aaac, acc, accc, aacc, \dots\}$
$\xi = \{\varepsilon, a -_1 c, a -_1 ca -_1 c, a -_2 c, a -_2 ca -_2 c, a -_1 ca -_2 c, \dots\}$
To build fence-subs$(\xi, \{-_1 \rightarrow \eta_1, -_2 \rightarrow \eta_2\})$ *we need to calculate the union of the fence-substitution of all the strings recognized by* ξ*, so:*
fence-subs$(\varepsilon, \{-_1 \rightarrow \eta_1, -_2 \rightarrow \eta_2\}) = \{\varepsilon\}$
fence-subs$(a -_1 c, \{-_1 \rightarrow \eta_1, -_2 \rightarrow \eta_2\}) = \{ac, abc, abbc, abbbc, \dots\}$
fence-subs$(a -_1 ca -_1 c, \{-_1 \rightarrow \eta_1, -_2 \rightarrow \eta_2\}) = \{acac, abcac, acabc, abbcabbbbc, \dots\}$

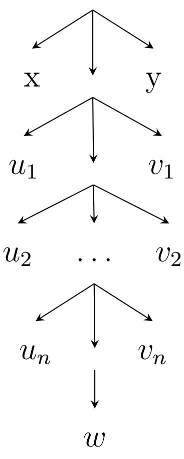


Figure 6.2: Structure of $xu_1u_2\cdots u_nwv_nv_{n-1}\cdots v_1y$

fence-subs$(a -_2 c, \{-_1 \to \eta_1, -_2 \to \eta_2\}) = \{aacc, aaaccc, aaaacccc, \ldots\}$
fence-subs$(a -_1 ca -_2 c, \{-_1 \to \eta_1, -_2 \to \eta_2\}) = \{acaacc, abbcaacc, abbbcaaaacccc, \ldots\}$
By calculating the union of all those languages, there could be found the language *fence-subs*$(\xi, \{-_1 \to \eta_1, -_2 \to \eta_2\})$.

Let us now analyze some results on the structure of *fence-subs* strings:

Corollary 6.2. *Given an OP-alphabet* (Σ, M), *and a string* $s = xu^nwv^ny = xu_1u_2\cdots u_nwv_nv_{n-1}\cdots v_1y$ *such that* w *and* uwv *are well-chained, then there do not exist any string* $o \in \Sigma_\Phi^*$ *and function* $map : \Phi \to \wp(\Sigma^*)$ *such that* $s \in$ *fence-subs*(o, map) *and such that there is a fence-subs between a* u_i (v_i) *and* u_l *(resp.* v_l*) with* $i \neq l$ *in which either the last (beginning) part of* u_i (v_i) *and the beginning (last) part of* u_l (v_l) *are parts of the same substitution or there is an entire* u_{i+1} (v_{i-1}) *that is contained in the substitution.*

Proof. Thanks to corollary 4.7 it is that the structure of $xu_1u_2\cdots u_nwv_nv_{n-1}\cdots v_1y$ is similar to the one of fig. 6.2. Given this fact, it can be easily seen that there is no possibility for a fence substitution to span over more than one repetition of u (v) and to span only over u^n (v^n).

□

Corollary 6.3. *Given an OP-alphabet* (Σ, M), *and a string* $s = xu^nwv^ny$ *such that* w *and* xwv *are well-chained and having* $u, v \in \Sigma^+$ *(so they cannot be* ε*), then if there exist any strings* $o \in \Sigma_\Phi^*, z, j, k \in \Sigma^*$ *such that* u^n *isn't a substring of* j *and* v^n *is not a substring* *of k and $s = \textit{fence-subs}(o, \{- \to z\})$ and $o = xj - ky$ then $u^l w v^h$ is a substring of z, and $l = h \geq 0$.*

Proof. Thanks to corollary 4.7 there is that the structure of
$xu_1u_2 \cdots u_n w v_n v_{n-1} \cdots v_1 y$ is similar to the one of fig. 6.2. Given this, it is trivial to see that the fence substitution, in order to span over an entire sub-tree, must start from somewhere inside u_i and end over v_i. If this is not the case, then it will span over only part of a subtree which is not acceptable. $\square$

Given an OP-alphabet (Σ, M), we can extend it to (Σ_Φ, M_Φ) by adding the relation $-_i \gtrdot a_j$ and $a_j \lessdot -_i$ for each element $-_i \in \Phi$ and $a_j \in \Sigma_\#$. In this case, it is useful to note that given a string $s \in \Sigma_\Phi$ it has the same structure of the string obtained via *fence-subs* on s.

Example 6.5. *Consider the language used in example 6.4 and the string*
$s = abbcaaccaaacccabc$ built starting from $s' = a -_1 ca -_2 ca -_2 ca -_1 c$; Extend the OPM M in M_Φ :

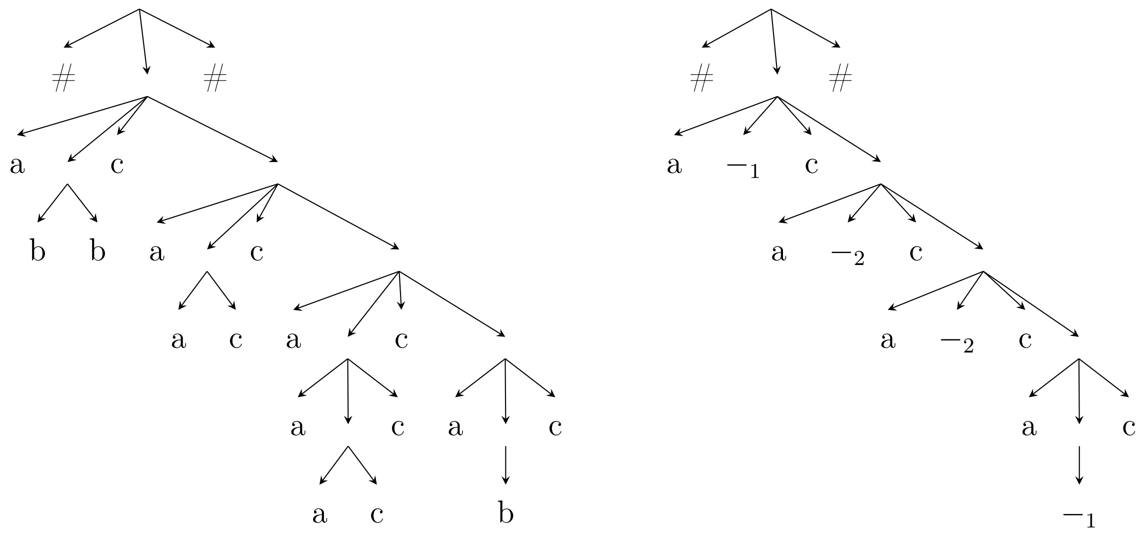
$$\begin{array}{c|cccccc}
 & a & b & c & -_1 & -_2 & \# \\
\hline
a & \lessdot & \lessdot & \doteq & \lessdot & \lessdot & \\
b & & \doteq & \gtrdot & \lessdot & \lessdot & \\
c & \lessdot & & \gtrdot & \lessdot & \lessdot & \gtrdot \\
-_1 & \gtrdot & \gtrdot & \gtrdot & & & \gtrdot \\
-_2 & \gtrdot & \gtrdot & \gtrdot & & & \gtrdot \\
\# & \lessdot & & & \lessdot & \lessdot & \doteq
\end{array}$$

Analyze the structure of s and s' over the corresponding matrices (note that $M(s) = M_\Phi(s)$):
$M_\Phi(s) = \#\lfloor a\lfloor bb\rfloor c\lfloor a\lfloor ac\rfloor c\lfloor a\lfloor a\lfloor ac\rfloor c\rfloor c\lfloor a\lfloor b\rfloor c\rfloor\rfloor\rfloor\rfloor\rfloor\#$
$M_\Phi(s') = \#\lfloor a\lfloor -_1\rfloor c\lfloor a\lfloor -_2\rfloor c\lfloor a\lfloor -_2\rfloor c\lfloor a\lfloor -_1\rfloor c\rfloor\rfloor\rfloor\rfloor\#$

As can be seen also on fig. 6.3, the structure of s and s' are equal without considering the replaced parts.

Please note that in the substitution there have been used 2 different names to refer to the function in the case it was applied to a string or to a language, while it has not been done for the *fence-subs* function. This choice has been taken in order to not make the notation heavier than it could be, and because there are no substantial differences between one definition of *fence-subs* and the other. In particular, it can be seen that:

- $\textit{fence-subs}(s, map_s = \{-_i \to x_i \mid - \in \Phi, x_i \in \Sigma^*\}) =$
  $\textit{fence-subs}(s, map_l = \{-_i \to \eta_i \mid -_i \in \Phi, \eta_i = \{x_i\}\}$;



(a) Structure of $M_\Phi(s)$ (b) Structure of $M_\Phi(s')$

Figure 6.3: Structure of s and s'

- $\textit{fence-subs}(s, map_l \subseteq (\Phi \times \wp(\Sigma^*))) = \textit{fence-subs}(\{s\}, map_l)$

Note also that this way of defining the *stringSub* and *fence-subs* is not much different from the concept of concatenation or fence inside the expressions, with exception given for the fact that expressions are defined somehow in a "static" way, while for *stringSub* and *fence-subs*, it is like the expressions are built by a language, and then they are evaluated.

6.2. Fence Substitution Restriction

Now that the *fence-subs* function has been defined, there is one last topic to be introduced: Is it sufficient to add the fence like "restriction" over the substitution in order to make the language NC_{OP}?
In order to answer this question, let us recall some of the non NC_{OP} examples done with LTEO and see if they are recognizable using the *fence-subs* operation instead of the *stringSub* one.

Example 6.6. *Consider example 5.13, substituting the stringSub function with fence-subs. Now fence-subs will recognize the language $\{aa\}$, because on any other string $s \in \eta$ different from $a-$, the fence-subs function could not be applicable.*

Example 6.7. *Consider example 5.14, substituting the stringSub function with fence-subs. Now fence-subs would not be applicable because there is no way to apply it on $(abca - c)^+$*

in order not to violate the fence restriction.

Given those examples, it may seem that it has been a success to introduce the fence restriction, however, there will be reported another example that proves it is yet not sufficient.

Example 6.8. *Given the OP-alphabet (Σ, M), where $\Sigma = \{a, b, c\}$ and having M:*

$$\begin{array}{c|cccc} & \# & a & b & c \\ \hline \# & \doteq & \lessdot & & \\ a & & & \lessdot & \\ b & & & & \gtrdot \\ c & \gtrdot & \lessdot & & \end{array}$$

Given $\eta = LT_4(\{\#abc, abca, bca-, ca-c, a-ca, -cab, cabc, abc\#\})$, $\xi = \{b\}$, then the language generated by fence-subs(η, ξ) is $(abc)^{2n}$, which trivially is not NC_{OP}.

From the previous example, it can be seen that fence-like restriction applied to the string substitution is not yet strict enough to guarantee that a language, produced with the application of *fence-subs* over NC_{OP} languages, is NC_{OP} too. For this reason, with the goal to restrict the expressive power of the new class of languages, there can be introduced another restriction: the *Fence Substitution Restriction*, or FSR.
This restriction has been built with particular attention to avoid the same cases of the example 6.8, and to easily prove how LTOP languages relate to OPEs. Before presenting FSR, however, it is useful to introduce the concept of SubString Language:

Definition 6.4 (SubString Language)**.** *Given a language $\eta \in \wp(\Sigma^*_{any})$, the SubString Language of η is the language defined as:*
$SSL(\eta) = \{x \in \Sigma^*_{any} \mid \exists w \in \eta$ such that x is a substring of $w\}$.

So, given any language η, $SSL(\eta)$ is the set of all the substrings of all the strings inside the language η.

Example 6.9. *For example,*
$SSL(\{abc, def\}) = \{a, b, c, ab, bc, abc, d, e, f, de, ef, def\}$;
$SSL(a^*) = a^*$;
$SSL(ab^*cd^*) = ab^* \cup cd^* \cup b^*cd^* \cup ab^*cd^*$.

Given the definition of SSL, the Fence Substitution Restriction can be presented:

Definition 6.5 (Fence Substitution Restriction)**.** *Given an OP-alphabet (Σ, M) and $\tau =$ fence-subs(ξ, map), it is said that τ is Fence Sub Restricted (FSR) iff, for each element*

$s \in SSL(\tau) \cap (\Sigma_{\#}^3 \cdot \Sigma^*)$ there exists at most one string $o \in SSL(\xi) \cap \Sigma_{\#,\Phi}^3 \cdot \Sigma_{\#,\Phi}^*$ such that $s = \textit{fence-subs}(o, map)$.

Example 6.10. *Consider the OP-alphabet (Σ, M), where $\Sigma = \{a, b, c\}$ and M :*

	a	b	c	$\#$
a	$\lessdot$	$\lessdot$	$\gtrdot$	
b		$\doteq$	$\gtrdot$	$\gtrdot$
c	$\lessdot$		$\doteq$	$\gtrdot$
$\#$	$\lessdot$			$\doteq$

And the language $\xi = (a - c)^*|(aac)^*|(abb)^*$, then:

$\eta_1 = a^*$, $\textit{fence-subs}(\xi, \{- \to \eta_1\})$ does not satisfy FSR, because $a - c, aac \in SSL(\xi)$ and $aac \in \textit{fence-subs}(a - c, \{- \to \eta_1\})$ and $aac \in \textit{fence-subs}(aac, \{- \to \eta_1\})$

$\eta_2 = a^*b^+$, $\textit{fence-subs}(\xi, \{- \to \eta_2\})$ does not satisfy FSR, because $ca-, caab \in SSL(\xi)$ and $caab \in \textit{fence-subs}(ca-, \{- \to \eta_2\})$ and $caab \in \textit{fence-subs}(caab, \{- \to \eta_2\})$;

*$\eta_3 = a^+b^+$, $\textit{fence-subs}(\xi, \{- \to \eta_3\})$ does satisfy FSR, because the substitution is inside $a - c$ and it will provide, at least $aabc$, but $\{abc, bbc, aaa, aab\} \cap SSL(\xi) = \emptyset$.
Please note that, although $ca- \in SSL(\xi)$ and $caab \in \textit{fence-subs}(ca-, \{- \to \eta_3\})$, thus $caa \in SSL(\textit{fence-subs}(ca-, \{- \to \eta_3\}))$, and $caa \in SSL(\xi)$, however $\textit{fence-subs}(\xi, \{- \to \eta_3\})$ satisfies FSR because $caa \notin \textit{fence-subs}(ca-, \{- \to \eta_3\})$.*

The main concept behind FSR is to add to the language via *fence-subs* only new strings that cannot be expressed in other ways. Its concept can somehow remind the restriction over BDNF rules for grammar.

The FSR restriction enables to state the following two theorems that will be the basis for the main results of LTOP results:

Corollary 6.4. *Given an OP-alphabet (Σ, M), a string $s = xu^nwv^ny \in \textit{fence-subs}(o, map)$, if uwv and w are well-chained on s, then if in o there is a substring u'_i being fence-substituted into u_i inside u^n in s, then if $u = \varepsilon$ then $u'_i = \varepsilon$, else $u'_i \in \Sigma_\Phi^* \Sigma$.*

The same is valid for v, with $v'_i \in \Sigma\Sigma_\Phi^*$.

Proof. For the case of $u_i = \varepsilon$, it's trivial to see that there is not possible to have $u'_i = -$ or $u'_i \in \Sigma^+$, thus u'_i must beε.

In the other cases, if u'_i would terminate with an element of Φ, there would be a chain ending while the u ends, so $u \cdot u$ would be $\ldots \lfloor \ldots \rfloor \lfloor \ldots$ which is not possible, so u'_i must be either empty or end with a character of Σ. $\square$

Theorem 6.1. *Given an OP-alphabet (Σ, M), a string $s = xu^n wv^n y \in$ fence-subs(o, map) for which is valid FSR, if uwv and w are well-chained on s, then if in o there is a substring $u_1' u_2' \cdots u_m'$ being fence-substituted into $u_1 u_2 \cdots u_m$ inside $u^n = u_1 u_2 \cdots u_m u_{m+1} \cdots u_n$ in s, and being $m \geq 5$, then $u_1' = u_2' = u_3' = \cdots = u_m'$.*

Proof. Consider $u_1' u_2' u_3' u_4' u_5'$, thanks to corollary 6.4,
$u_1' = u_1'' a$, $u_2' = u_2'' a$, $u_3' = u_3'' a$, $u_4' = u_4'' a$ and $u_5' = u_5'' a$, with $a \in \Sigma$ and
$u_1'', u_2'', u_3'', u_4'', u_5'' \in \Sigma_\Phi^*$. Consider $o_1 = u_1' u_2' u_3' u_4'$ and $o_2 = u_2' u_3' u_4' u_5'$.
It is trivial to see $u^4 \in$ *fence-subs*(o_1, map) and $u^4 \in$ *fence-subs*(o_2, map), thus, being FSR valid, then $o_1 = o_2$, which implies that:

$\begin{array}{cccccccc} u_1'' & \text{a} & u_2'' & \text{a} & u_3'' & \text{a} & u_4'' & \text{a} \\ u_2'' & \text{a} & u_3'' & \text{a} & u_4'' & \text{a} & u_5'' & \text{a} \end{array}$ $u_1'' = u_2'' = u_3'' = u_4'' = u_5''$, which implies that $u_1' = u_2' = u_3' = u_4' = u_5'$. □

It is now presented one of the major results regarding LTOP:

Theorem 6.2 (*fence-subs*, FSR and NC_{OP}). *Given an OP-alphabet (Σ, M), and $\tau =$ fence-subs(ξ, map), $\xi \in LT_{k,\Sigma_\Phi}$, if $\forall -_i \in \phi(\xi)$ $map(-_i) \in NC_{OP}$ and τ is FSR, then $\tau \in NC_{OP}$.*

Proof. To prove the theorem, there will be hypothesized that exists
$s = xu^n wv^n y \in$ *fence-subs*$_M(\xi, map)$ and we will verify that
$z = xu^{n+1} wv^{n+1} y \in$ *fence-subs*$_M(\xi, map)$.

Given the definition of *fence-subs*, it can be seen that $s \in$ *fence-subs*$_M(\xi, map)$ implies that exists $o \in \Sigma_\Phi^*$ such that $o \in \xi$ and $s =$ *fence-subs*$_M(o, map)$.

In this proof, there will be enumerated all the possibilities of the composition of o and they will be analyzed one by one. The first classification that can be done is:

1. *fence-subs* is not used to produce $s \longrightarrow o \in \Sigma^*, o = s$;
2. *fence-subs* is used to produce $s \longrightarrow o \notin \Sigma^*, o \neq s$;

In the case of item 1 where *fence-subs* is not used, using the fact that $\xi \in LT_{k,\Sigma_\Phi}$, and having $s \in \xi \in NC$, it is trivial to see that $z \in \xi$, thus $z \in$ *fence-subs*$_M(\xi, map)$.

Let us now analyze the case of item 2, to study which, there will be examined different cases. Let be $x_1, x_2, y_1, y_2 \in \Sigma^*$, $x_1^{'}, y_2^{'}, w_e^{'} \in \Sigma_\Phi^*$ such that $x_1 \cdot x_2 = x$, $y_1 \cdot y_2 = y$, $x_1 =$

$\textit{fence-subs}_M(x_1^{'}, \textit{map}),\ y_2 = \textit{fence-subs}_M(y_2^{'}, \textit{map}),\ x_1^{'} \cdot w_e^{'} \cdot y_2^{'} = o$, in this case we there can be:

1. $w_e^{'} \in \Sigma^* \longrightarrow x_2u^nwv^ny_1 = w_e^{'}$;
2. $w_e^{'} = -_i \longrightarrow x_2u^nwv^ny_1 \in \eta_i = \textit{map}(-_i)$;
3. $w_e^{'} \notin (\Sigma^* \cup \Phi)$.

Start from item 1. In this case there is $x_2u^nwv^ny_1 = w_e^{'}$, asking that $n > k$, having $\xi \in LT_{k,\Sigma_\Phi}$, thanks to theorem 2.2 and corollary 4.8, then $x_1^{'}x_2u^nwv^ny_1y_2^{'} \in \xi \Rightarrow x_1^{'}x_2u^{n+1}wv^{n+1}y_1y_2^{'} \in \xi$, which implies that:
$z = x_1x_2u^{n+1}wv^{n+1}y \in \textit{fence-subs}_M(x_1^{'}x_2u^{n+1}wv^{n+1}y_1y_2^{'}, \eta_1, \cdots, \eta_m)$.

Discuss the item 2: in that case there is that $\exists_{-_i\in\phi(o)}x_2u^nwv^ny_1 \in \eta_i = \textit{map}(-_i)$, asking that $n \geq n_i$ [1] , there is that $x_2u^{n+1}wv^{n+1}y_1 \in \eta_i$, which means that $z = x_1x_2u^{n+1}wv^{n+1}y \in \textit{fence-subs}_M(x_1^{'}w_e^{'}y_2^{'}, \textit{map})$.

Finally analyze the item 3. Please note that being u^nwv^n well-chained, there ca not be a *fence-subs* on part of x_2 (or y_1) that includes in its substitution also part of u^n (or, respectively, v^n) without including the whole u^nwv^n, thus falling in case item 2. For this reason, for the case in the study, there will be considered $x_2 = y_1 = \varepsilon$. So $u^nwv^n \in \textit{fence-subs}(w_e^{'}, \textit{map})$. For corollary 6.3 we have that if there exists a fence-substitution on $w_e^{'}$ such that its substitution contains u^lwv^k, then $k = l$. This means that $w_e^{'} = u_e^{'}w^{'}v_e^{'}$. In this new writing, $w^{'}$ is such that in its substitution it is contained $u^{l+1}wv^{l+1}$ where l is the number of u (or v) in the substitution.

Thanks to corollary 6.4 we have that there cannot be present fence-substitutions between one u (v) and another. This means that $u_e^{'}$ ($v_e^{'}$) is the concatenation of $n-l-1$ substrings that can be *fence-subs* to u (v). For FSR, we have that each one of these substrings in $u_e^{'}$ ($v_e^{'}$) is equal to the other ones, so $u_e^{'} = u^{'\,n-l-1}$ ($v_e^{'} = v^{'\,n-l-1}$).

This means that $w_e^{'} = u^{'\,n-l-1}w^{'}v^{'\,n-l-1}$. If $l > n_{\eta_i}$, then $u^{l+1}wv^{l+1}$ is in the substitution of $w^{'}$, then $u^{l+2}wv^{l+2} \in SSL(\textit{fence-subs}(w^{'}, \textit{map}))$, and thus $z \in \textit{fence-subs}(o, \textit{map})$. Else, if $n-l-1 > n_\xi$, then the string that adds one $u^{'}$ and one $v^{'}$ to o is contained in ξ, because $\xi \in NC$, thus $z \in \textit{fence-subs}(\xi, \textit{map})$.

[1]Denoting with n_i the the number n of NC_{OP} definition with respect to the language $\eta_i = \textit{map}(-_i)$, that, being $-_i \in \phi(o) \subseteq \phi(\xi)$, then η_i must be NC_{OP}.

In the end, due to the fact that there are no other possibilities to build o, it is sufficient to choose $\forall -_i \in \phi(\xi)\ n > n_\xi + n_{map(-_i)} + 1$ to have that the language $\textit{fence-subs}(\xi, map) \in NC_{OP}$. □

Let us now analyze example 6.8. It is trivial that in that specific case, the FSR was violated, so we reached our goal of restricting the expressiveness of these languages, however, it is sufficient to guarantee that any LTOP is NC_{OP}? Is this restriction too strict?Is the FSR property decidable?
The following of this work will answer those questions.

6.3. Language Definition

How can LTOP family be defined?

definition 6.5 and theorem 6.2 gave a hint on it, however, how it has been done for LTOs, there will be defined the LTOP family as the boolean operators and concatenation closure on the family defined via those previous results:

Definition 6.6 (LTOP). *Given an OP-alphabet (Σ, M) we define as LTOP family of languages the smallest set of languages that contains:*

1. *the empty language: $\emptyset$*
2. *The language $\tau = \textit{fence-subs}(\xi, map)$ where ξ is LT_{k,Σ_Φ} and $map : \Phi \to LTOP$ and where there subsists the FSR on τ*

And such that it is closed under Boolean operations and concatenation.

Corollary 6.5. *The LTOP family contains the LTO family.*

Proof. To prove this theorem, note that it is trivial to see $LT \subseteq LTOP$, thus, being $LTOP$ closed with respect to boolean operations and concatenation, then $LTO \subseteq LTOP$. □

The main result that can now be stated is:

Theorem 6.3 ($LTOP \subseteq NC_{OP}$). *The family of languages defined by LTOP is a subset of the NC_{OP} family.*

Proof. To prove this theorem we will take inspiration from section 5 of [15].

The goal is to prove that any language $\eta \in LTOP$ implies that $\eta \in NC_{OP}$. As the first thing, it will be pointed out that if $\eta \in LT$, then $\eta \in NC$, and, thus, $\eta \in NC_{OP}$.

Then, given $\eta, \xi \in LT$, then $\neg\eta$, $\eta \cup \xi$, $\eta \cap \xi$, $\eta \cdot \xi \in LTO$, which implies that they are also NC_{OP}.

Lastly note that any language that is LTOP is built, via *fence-subs*, over other languages that are either LTO or LTOP, but also those LTOP languages must be composed over LTO or LTOP languages; however during this sequence of LTOP languages, there must be present one last language only relying on a set of LTO languages on which $map : \Phi \to LTO$, so, for theorem 6.2 the language defined via *fence-subs* using the function map mapping Φ in NC_{OP} is NC_{OP}.

Being NC_{OP} family closed with respect to boolean operators and concatenation, any language that is derived as a boolean operation or the concatenation of more NC_{OP} languages, it is itself a NC_{OP} language, which will end the proof. □

So, there is $LTOP \subseteq NC_{OP}$, which is a great result, however, in the following section there is proved more amazing results from the comparison of LTOP with the other characterization of OPLs given in the literature.

6.4. Relations with the Literature

Here is presented how the newly defined LTOP class of languages relates with the other representations of NC_{OP}, in particular with SF OPE and FO as presented in [15]. This section, although its shortness, is very important because describes in which way the newly defined class relates to the representations presented in the literature, and thus how it can propose new tools for future research.

As a first thing, let us discuss how LTOP relates to SF OPE. For the following theorem, it is used the characterization of SF OPE given in lemma 4.3.

Theorem 6.4 ($SFOPE \subseteq LTOP$). *The family of languages defined by Star Free OPEs is contained into LTOP family.*

Proof. To prove this theorem, we can rely on lemma 4.3.
It is trivial to see that languages expressible via Regular Expressions are contained into LTOP, thus the $L_{i,j}, R_{i,j}, Hi, j$ are also LTOP languages.

The language defined by $a[\Sigma^+]b = a\triangle b$ is LTOP (and satisfies the FSR clause).
Being LTOP closed with respect to boolean operations and concatenation then also $\neg(a[\Sigma^+]b) \cap a \cdot E \cdot b = a\triangledown b$ is LTOP.
At this point it is trivial to see that any language defined via $\bigcup_i \bigcap_j t_{i,j}$ with $t_{i,j} = L_{i,j} \cdot a_{i,j}\triangle b_{i,j} \cdot R_{i,j} | L_{i,j} \cdot a_{i,j}\triangledown b_{i,j} \cdot R_{i,j} | H_{i,j}$ is also LTOP. □

Thanks to the previous result, also the following can be stated:

Theorem 6.5. *The class of LTOP languages coincides with the class of Non-Counting languages over OP.*

Proof. To prove the theorem, it is used theorem 4.9 and theorem 4.8 to have $NC_{OP} \subseteq FO \subseteq SFOPE$, from theorem 6.4 we have $SFOPE \subseteq LTOP$, and for theorem 6.3 it is $LTOP \subseteq NC_{OP}$ which implies $NC_{OP} \subseteq FO \subseteq SFOPE \subseteq LTOP \subseteq NC_{OP}$, which implies $NC_{OP} = FO = SFOPE = LTOP$. □

So, there has been proven that the LTOP family is not only contained in the NC_{OP} one, but it coincides with it. Moreover, thanks to theorem 6.4, there has been proven how this new characterization relates to the other ones present in the literature.

7 | Decidability

In this section, it is discussed if the problem of deciding if an LTOP language is FSR is decidable or not. As an anticipation of the final result, the problem is decidable, but the path there will be followed in order to reach this result is neither short nor simple. Before proceeding on, please note that the goal of this section is only to prove that the problem is decidable, not to provide a suitable way to solve the problem. There surely must be other ways to prove this same result.

The steps that will be performed are:

- Find an equivalent notation for FSR such that the check for a language if respects FSR consists in a finite number of set comparisons;
- Find a way to build languages equivalent to LTE languages, such that they are regular languages;
- Find a way to build languages equivalent to LTEO languages, such that they are regular languages;
- Find a way to build languages equivalent to LTOP languages (without the check for FSR), such that they are regular languages;
- Prove that the problem of deciding whether or not a language satisfies the FSR is a decidable problem because it is equivalent to the problem of deciding whether or not a finite set of Regular Languages intersected with a Max Language are all empty or not.

As already stated, please note that this proof provides to practically check if an LTOP satisfies the FSR or not, however, the process described in the following pages is not intended to be used, and thus there has been put no attention to the complexity of computation or memory consumption.

7.1. FSR Semplification

Here is discussed how to reformulate the FSR to be easier handled and make its condition correspond to a check of the emptiness of a finite number of languages. In order to do so, we can start with the following result describing the SSL language of an LT language:

Theorem 7.1 (SSL LT)**.** *Given an LT language η, its SSL is also an LT language.*

Proof. Suppose that, as described in definition 2.4, $\eta = LT_k(\alpha, \beta, \gamma, \delta)$ for some sets of strings $\alpha, \beta, \gamma \subseteq \Sigma^k$, and $\delta \subseteq \bigcup_{i=0}^{k-1} \Sigma^i$.
The language $SSL(\eta)$ can be characterized as a LT language defined over $\alpha', \beta', \gamma', \delta'$, having:

- δ' containing all the possible substrings shorter than k of all the elements of the sets $\alpha, \beta, \gamma, \delta$;
- α' containing all the elements of α, β, γ;
- β' containing all the elements of β;
- γ' containing all the elements of α, β, γ.

□

Thanks to this result, we are guaranteed not only that the $SSL(\xi)$ language of definition 6.5 is a regular language, but also that it is an LT language, and thus, NC. However, this result does not guarantee that the number of set comparisons to be done in order to check the FSR property is finite. To overcome this issue, the following can be stated:

Theorem 7.2. *Given an OP-alphabet (Σ, M) and*
$\tau = \textit{fence-subs}(\xi, map)$, with ξ that is LT, being
$\alpha = SSL(\xi) \cap \Sigma^3_{\#,\Phi} \cdot \Sigma^*_{\#,\Phi}$,
$\beta = SSL(\tau) \cap \Sigma^3_{\#} \cdot \Sigma^*_{\#}$,
$\delta = SSL(\xi) \cap \Sigma^3_{\#,\Phi}$,
the following are equivalent:

1. *for each element $s \in \beta$ there exists at most one string $o \in \alpha$ such that $s = \textit{fence-subs}(o, map)$;*
2. *for each element $o \in \alpha$ the set $\textit{fence-subs}(o, map) \cap \textit{fence-subs}(\alpha \setminus \{o\}, map)$ is empty;*

3. *for each element $o \in \delta$ the set*
   $\textit{fence-subs}(o, map) \cap \textit{fence-subs}(\alpha \setminus \{o\}, map)$ is empty.

Proof. In order to prove this theorem, let us go step by step.
The equivalence of the first and second points is trivial.
Let us now analyze the second and the third ones, and in particular, let us prove that:

- the second point implies the third;
- the third point implies the second.

Being $\delta \subseteq \alpha$, it is trivial to see that the second point implies the third.

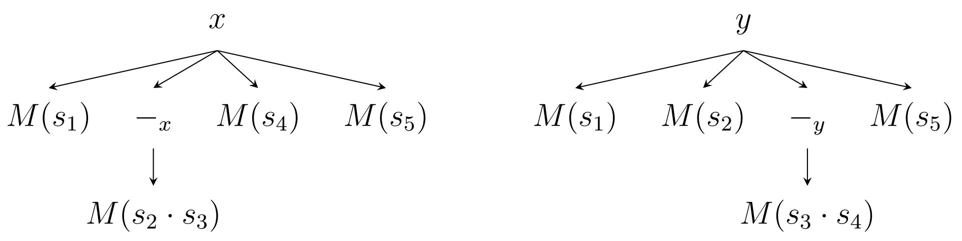
To prove that the third point implies the second, we will prove that if the second point is not valid, then neither the third is. So, we want to prove that if there exist two strings $x, y \in \alpha$ such that $x \neq y$ and
$\textit{fence-subs}(x, map) \cap \textit{fence-subs}(y, map) \neq \emptyset$, then there also exists $w \in \delta, z \in \alpha$ such that $w \neq z$ and $\textit{fence-subs}(w, map \cap \textit{fence-subs}(z, map) \neq \emptyset$.

Obviously, if $|x| = 3$, $|y| = 3$ or $x = y$, the proof will become trivial.
Let us then suppose that $|x| \geq |y| > 3$, and, that $x \neq y$.

If $x, y \in \Sigma^*$, then it is trivial to see that their substrings of length 3 will provide the proof we are searching.

Consider $x \notin \Sigma^*$. In this case, if also $y \notin \Sigma^*$, then note that there cannot be *interposed* substitutions in x and y, i.e., it cannot be that there is the following situation:
$x = x_1 -_x x_4 x_5$
$y = y_1 y_2 -_y y_5$
$\exists s = s_1 s_2 s_3 s_4 s_5 \mid s \in \textit{fence-subs}(x, map) \wedge s \in \textit{fence-subs}(y, map)$
where $s_1 \in \textit{fence-subs}(x_1, map) \wedge s_1 \in \textit{fence-subs}(y_1, map) \wedge$
$s_2 \in \textit{fence-subs}(y_2, map) \wedge$
$s_2 s_3 \in \textit{fence-subs}(-_x, map) \wedge$
$s_3 s_4 \in \textit{fence-subs}(-_y, map) \wedge$
$s_4 \in \textit{fence-subs}(x_4, map) \wedge$
$s_5 \in \textit{fence-subs}(x_5, map) \wedge s_5 \in \textit{fence-subs}(y_5, map) \wedge$
$s_2 \neq \varepsilon \wedge s_4 \neq \varepsilon$.

This can be proved by relying on the definition of *fence-subs* and the fact that the structure of a string over a fixed OP alphabet is unique, while in the case of interposed *fence-subs* then they will be as in fig. 7.1. Please note that the previous holds also by setting $s_3 = \varepsilon$, leveraging corollary 4.2. So we proved that between 2 *fence-subs* over x



(a) Structure of $x_1 -_x x_4 x_5$

(b) Structure of $y_1 y_2 -_y y_5$

Figure 7.1: Structure of x and y

and y not included one inside the other, there must be at least one character of offset from where the first one ends and where the second one starts.

By also noting that:

- the first and last characters of x cannot belong to Φ because the *fence-subs* will not be defined;
- there must be one character a belonging to x and such that $a \in \Phi$ and for which its *fence-subs* can be mapped on a substring y' of y such that $y' \neq a$

Then also for the substring $x' = a_{-1}aa_{+1}$, where a_{-1}, a_{+1} are the characters preceeding and following a in x, there exists a substring y'' of y such that $y'' \neq x'$, and $\exists s' \mid s' \in$ *fence-subs*$(x', map) \wedge s' \in$ *fence-subs*(y'', map).

So, it has been proved that if there is $x \in \alpha$ such that there exists $y \in \alpha \setminus \{x\}$ such that *fence-subs*$(x, map) \cap$ *fence-subs*(y, map) contains a string s, then there must also be $x' \in \delta$ such that there exists $y'' \in \alpha \setminus \{x'\}$ such that *fence-subs*$(x', map) \cap$ *fence-subs*(y'', map) contains a string; which proves that the third point of the theorem implies the second one, which ends the proof. $\square$

From the previous result, it is trivial to state the following:

Corollary 7.1. *There exists a finite number of strings of length 3 that are substrings of any language $\eta \subseteq \Sigma^*$ if $|\Sigma|$ is finite.*

Corollary 7.2. *There exists a finite number of set comparisons to be done to verify if the FSR property holds for a certain LTOP.*

Said that, our focus will now become to prove that the problem of deciding whether or not *fence-subs*$(o, map) \cap$ *fence-subs*$(\alpha \setminus \{o\}, map) = \emptyset$ with $o \in \delta$ and α, δ defined as in theorem 7.2, is decidable.

7.2. Regularizing Locally Testable Extended Languages

The goal of this section is to provide a way to represent LTE languages by leveraging some Regular Language. In order to achieve such a goal, there must be stated some results over LT languages:

Definition 7.1 (once reliant)**.** *Given an LT $\eta = LT_k(S) \subseteq \Sigma_{any}^*$ and a character $a \in \Sigma_{any}^*$, then, being x any element of $S \cap \Sigma_{any}^* \cdot \{a\} \cdot \Sigma_{any}^*$, if $LT_k(S \setminus \{x\}) \subseteq (\Sigma_{any} \setminus \{a\})^*$, the language η is said to be once reliant on a.*

Definition 7.2 (eclittic LT)**.** *Given a language $\eta \subseteq \Sigma_{any}^*$ defined on a generic alphabet Σ, and a character $a \in \Sigma$, we define $\eta_{\tilde{a}} = \eta \cap (\Sigma \setminus \{a\})^*$.*

Corollary 7.3. *Given an LT language η and a character a, then $\eta_{\tilde{a}}$ is also an LT language.*

The proof of the previous corollary is trivial. Now there can be presented the following result, upon which the whole section will rely:

Theorem 7.3. *Given an alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$ with $\Sigma_1 \cap \Sigma_2 = \emptyset$, then being η_1, η_2 two LT languages such that $\eta_1 \subseteq (\Sigma_1 \cup \{-\})^*$ and $\eta_2 \subseteq \Sigma_2^*$, and being η_1 once reliant on $-$, then $\tau = stringSub(\eta_1, \{- \to \eta_2\})$ is also LT.*

Proof. To prove this theorem, let us suppose, without loss of generality, that $\eta_1 = LT_{k_1}(\alpha_1, \beta_1, \gamma_1, \delta_1)$, $\eta_2 = LT_{k_2}(\alpha_2, \beta_2, \gamma_2, \delta_2)$. It is trivial that there could be defined $k = k_1 + k_2$ and that there can be built the sets $\alpha_1', \alpha_2', \beta_1', \beta_2', \gamma_1', \gamma_2', \delta_1', \delta_2'$ such that $\eta_1 = LT_k(\alpha_1', \beta_1', \gamma_1', \delta_1')$, $\eta_2 = LT_k(\alpha_2', \beta_2', \gamma_2', \delta_2')$. For corollary 7.3 there can be built also $\eta_{1,\tilde{-}} = LT_k(\alpha_1'', \beta_1'', \gamma_1'', \delta_1'')$

Let us now build the sets:

- α such that contains all the possible beginnings of length k of the strings belonging to τ;
- γ such that contains all the possible endings of length k of the strings belonging to τ;
- δ such that contains all the possible strings of length k belonging to τ;
- β such that contains all the elements of β_1'', β_2', and, for each element $el \in \beta_1' \setminus \beta_1''$:

- β contains each substring of lenght k of $stringSub(el, \delta'_2)$;
- β contains each substring i of lenght k of $stringSub(el, \alpha'_2)$ such that $i \in \Sigma_1^* \cdot \sigma_2^*$;
- β contains each substring e of lenght k of $stringSub(el, \gamma'_2)$ such that $i \in \Sigma_2^* \cdot \sigma_1^*$.

It is now trivial to see that $\tau = LT_k(\alpha, \beta, \gamma, \delta)$.

Please note that the *once reliant* assumption guarantees that the substitution of strings of η_2 are always placed inside the "same context[1]", which enables τ to be LT. □

In order to use the previous result, there are lots of limitations:

- $|\Phi| = 1$;
- The language η_1 must be once reliant on $-$;
- The languages η_1 and η_2 must be defined over different languages.

All those restrictions limit the applicability of the result. In order to use the theorem and leverage it, there is presented how to overcome each one of those imitations. In particular, there is analyzed how to overcome them in the opposite order with respect to the one with which they have just been exposed.

7.2.1. Define Languages on Different Alphabets

In order to leverage the result of theorem 7.3 there must be found a way to translate a language $\eta_2 \subseteq \Sigma^*$ into another one defined over a different alphabet, and a way to do the backward operation, so to translate a language from the newer alphabet to the older one.

In this section, there it is presented how to do such operations. In order to start, let us define a way to generate new alphabets:

Definition 7.3 (melter). *Given an alphabet Σ and a set E, E is said to be meltable with respect to Σ if $(\Sigma \times E) \cap \Sigma = \emptyset$, where $\times$ indicates the cartesian product.*

Given an alphabet Σ and a set E that is meltable with respect to Σ, the function melt is defined as $M(\Sigma, E) = \Sigma \times E$.
There can be used the notation $M^0(\Sigma, E) = M(\Sigma, E) \cup \Sigma$.

It is be used the function $M(\Sigma, E)$ to define the new alphabets to be used. The concept of *meltable* has been introduced to avoid having, for instance, $M(M^0(\Sigma, E), E)$. Please note also that $M(\Sigma, \emptyset) = \emptyset$, $M^0(\Sigma, \emptyset) = \Sigma$.

[1]defined in definition 1.18

Corollary 7.4. *Given an alphabet Σ and two sets $E_1 \subseteq E_2$that are both meltable with respect to Σ, then $M(E_1, \Sigma) \subseteq M(E_2, \Sigma)$.*

Here is defined a function that enables the translation of characters, strings, and languages from being defined over $M^0(\Sigma, E)$ to be defined over Σ.

Definition 7.4 (flattener)**.** *Given an alphabet Σ,two sets E_s and E_t such that $E_s \subseteq E_t$, the function flattener f_{Σ,E_t} is defined such that:*

- *Being $c \in \Sigma$, then $f_{\Sigma,E_t}(c, E_s) = c$;*
- *Being $c \in M(\Sigma, E_s)$, then*
  $f_{\Sigma,E_t}(c, E_s) = c' \mid \exists e \in E_s \; c = \langle c, e\rangle$;
- *Being $c \in M(\Sigma, E_t \setminus E_s)$, then*
  $f_{\Sigma,E_t}(c, E_s) = c$;
- *Being $s = c_0c_1 \ldots c_n, c_i \in M^0(\Sigma, E_t)$, then*
  $f_{\Sigma,E_t}(s, E_s) = c'_0c'_1 \ldots c'_n \mid c'_i \in f_{\Sigma,E_t}(c_i, E_s)$;
- *Being $\eta = \{s_1, s_2, \ldots\} \subseteq M^0(\Sigma, E_t)^*$, then*
  $f_{\Sigma,E_t}(\eta, E_s) = \{s'_1, s'_2, \ldots\} \mid s'_i \in f_{\Sigma,E_t}(s_i, E_s)$

Please note that definition 7.4, gives the possibility to translate a language from being defined over $M^0(\Sigma, E_t)$ to be defined over $M^0(\Sigma, (E_t \setminus E_p))$.

Another concept that can be introduced here is:

Definition 7.5 (f-equivalence)**.** *Given two languages $\eta_1 \subseteq \Sigma^*$, $\eta_2 \subseteq M^0(\Sigma, E)$ for some Σ, E, then they are said to be f-equivalent if $\eta_1 = f_{\Sigma,E}(\eta_2, E)$.*

Now there is presented a function that enables, with the other things, to map a language from being defined over Σ to be defined over $M(\Sigma, E_p)$.

Definition 7.6 (producer)**.** *Given an alphabet Σ, three sets E_s, E_p and E_t such that $E_s \subseteq E_t$ and a set S such that $S = M(\Sigma, E_s)$ or $S = M^0(\Sigma, E_s)$, then we can define the function producer P as:*

- *Being $c \notin S$, then $P_{\Sigma,E_t}(c, S, E_p) = \{c\}$;*
- *Being $c \in (\Sigma \cap S)$, then*
  $P_{\Sigma,E_t}(c, S, E_p) = \{c' \mid \exists e \in E_p \; c' = \langle c, e\rangle \in M(\Sigma, E_p)\}$;
- *Being $c \in (S \setminus \Sigma)$, then*
  $P_{\Sigma,E_t}(c, S, E_p) = \{c' \mid \exists e \in E_p \; c' = \langle f_{\Sigma,E_t}(c, E_s), e\rangle \in M(\Sigma, E_p)\}$;

- *Being* $s = c_0c_1 \dots c_n, c_i \in M^0(\Sigma, E_t)$, *then* $P_{\Sigma,E_t}(s, S, E_p) = \{c'_0c'_1 \dots c'_n \mid c'_i \in P_{\Sigma,E_t}(c_i, S, E_p)\}$
- *Being* $\eta = \{s_1, s_2, \dots\} s_i \in M^0(\Sigma, E_t)^*$, *then* $P_{\Sigma,E_t}(\eta, S, E_p) = \{s'_1, s'_2, \dots \mid s'_i \in P_{\Sigma,E_t}(s_i, S, E_p)\}$

So, in order to translate the language $\eta \subseteq \Sigma^*$ to a new f-equivalent language η' defined over $E = \{e\}$, it is sufficient to set $\eta' = P_{\Sigma,E}(\eta, \Sigma, E)$.

Definition 7.7 (Language Producer). *Given an alphabet Σ, three sets E_t, E_p, E_s such that $E_p, E_s \subseteq E_t$, and a set $S = M(\Sigma, E_s)$ or $S = M^0(\Sigma, E_s)$, then $L_{\Sigma,E_t}(\eta, S, E_p)$ is defined as the function returning the set of all the possible languages η' such that $\eta = f_{\Sigma,E_t}(\eta', E_p)$.*

With those definitions, there can be expanded the theorem 7.3 to not rely on the languages to be defined over separate alphabets:

Corollary 7.5. *Given two LT languages $\eta_1 \subseteq \Sigma^*_\Phi$ and $\eta_2 \subseteq \Sigma^*$, with $\Phi = \{-\}$ and η_1 that is once reliant on $-$, then there exists a LT language that is f-equivalent to $stringSub(\eta_1, \{- \to \eta_2\})$.*

7.2.2. Transform Languages to Once Reliant

Here is presented how an LT language $\eta \subseteq \Sigma^*_\Phi$ that is not Once Reliant on $-_i$ and a function $map : \Phi \to \wp(\Sigma^*)$, can be transformed into a language η' that is Once Reliant on $-_i$ and a function map', such that $stringSub(\eta, map) = stringSub(\eta', map')$.

Consider a language $\eta = LT_k(S) \subseteq \Sigma^*_\Phi$, $-_i \in \phi(\eta)$ such that η is not once reliant on $-_i$, and a function $map : \phi(\eta) \to \wp(\Sigma^*)$, and let S', R, T, U, V be sets, map' be a function, then:

1. Set $T = S \cap \Sigma^* \cdot -_i \cdot \Sigma^*$, $S' = S \setminus T$, $map' = map$, $U = R = V = \emptyset$;
2. Choose $t = t_1t_2 \dots t_{l-1} -_i t_{l+1}t_{l+2} \dots t_k \in T \setminus V$ such that
   - $t_1 = \#$ or
   - $t_k = \#$ or
   - There is no m such that the number of elements $e' \in T \setminus V$ having as m-th character $-_i$ is greater than the number of characters having $-_i$ as l-th character.
3. Choose $-_j \in (\Phi \setminus (\phi(\eta) \cup U))$;

4. Select each element $e = e_1 e_2 \dots e_{m-1} -_i e_{m+1} e_{m+2} \dots e_k \in T$ such that:

   $l \leq m \ \forall_{x=1}^{l-1} t_x = e_{x+m-l}$ and $\forall_{x=l+1}^{k-m+l} t_x = e_{x+m-l}$

   $l > m \ \forall_{x=1}^{m-1} t_{x+l-m} = e_x$ and $\forall_{x=l+1}^{k-l+m} t_{x+l-m} = e_x$

   and add it to V and add the new element
   $e_1 e_2 \dots e_{m-1} -_j e_{m+1} e_{m+2} \dots e_k$ to R;

5. Add t to V, and $-_j$ to U;

6. Repeat from point 2 until $T \setminus V$ is not empty;

7. Add T to S';

8. For each element $-_j \in U$, add to map' the relation $-_j \rightarrow map(-_i)$

The language $\eta' = LT_k(S')$ is then LT, for each element $-_j$ of U, η' is once reliant on $-_j$, and $stringSub(\eta, map) = stringSub(\eta', map')$. So we can state:

Theorem 7.4. *Given an LT language $\eta \subseteq \Sigma_\Phi^*$ and a function $map : \phi(\eta) \rightarrow \wp(\Sigma^*)$, there exist an LT language $\eta' \subseteq \Sigma_\Phi^*$ and a function $map' : \phi(\eta') \rightarrow \wp(\Sigma^*)$ such that η' is single reliant on every element of $\phi(\eta')$, and $stringSub(\eta, map) = stringSub(\eta', map')$. Moreover there can be defined the function $ip : \Phi \rightarrow \Phi$ that associates to each element of $\phi(\eta')$ the corresponding element of $\phi(\eta)$.*

This result solves the problem of once reliability, however, it needs to enlarge the dimension of the set Φ.

7.2.3. Accept More Blanks

Here is discussed that the restriction of having $|\Phi| = 1$ in theorem 7.3 can be avoided.

Theorem 7.5. *Given an LT language $\eta \subseteq \Sigma_\Phi^*$ and a function $map : \Phi \rightarrow \wp(\Sigma^*)$ such that map maps each element of $\phi(\eta)$ into an LT language, then there exists an LT language that is f-equivalent to $stringSub(\eta, map)$.*

Proof. Consider an LT language $\eta \subseteq \Sigma_\Phi^*$ and a function $map : \Phi \rightarrow \wp(\Sigma^*)$ such that map maps each element of $\phi(\eta)$ into an LT language, and take an element $-_i \in \phi(\eta)$ and such that for each element of $- \in \phi(\eta)$, η is once reliant on $-$. Even if there is not such a condition, for theorem 7.4, there can be found the language and function that satisfies such conditions and such that the language $stringSub(\eta, map)$ is the same.

At this point, there can be demonstrated by induction that the theorem is valid.

Let us suppose that $|\Phi| = 1$, then, it is the case of theorem 7.3, so the theorem is true.

Let us suppose that $|\Phi| = n$, and that the theorem is valid for $n-1$.
Take $- \in \Phi$, and consider $\Sigma' = \Sigma \cup \{-\}$, $\Phi' = \Phi \setminus \{-\}$, and let us define $map' : \Phi' \to \wp(\Sigma'^*) = map \setminus \{- \to map(-)\}$. It is trivial to see that $\eta \subseteq (\Sigma' \cup \Phi')^*$ and that, for each element $-' \in \Phi'$, $map(-') \subseteq \Sigma'^*$, thus, there exists, by the induction hypothesis, an LT language $\eta' \subseteq M^0(\Sigma', E)^*$ such that it is f-equivalent to $stringSub(\eta, map')$. Now, by applying theorem 7.3 over $stringSub(\eta', \{- \to map(-)\})$, it is trivial to see that we obtain an f-equivalent language with respect to the initial $stringSub(\eta, map)$.

Let us call $\Sigma' = \Sigma_\Phi \setminus \{-_i\}$, then $\eta \in (\Sigma' \cup \{-_i\})^*$, but also for each $- \in \phi(\eta)$, $map(-) \subseteq \Sigma'^*$. Thus, we can apply to have tat there exists a $\square$

All those steps, together with the decidability results previously reported for Regular Languages, allow to state the following:

Corollary 7.6. *Given two LT languages η_1, η_2, it is decidable if $stringSub(\eta_1, \{- \to \eta_2\})$ is empty or not.*

Given an LTE language η, the problem of deciding if $\eta = \emptyset$ is decidable.

Given a string s and an LTE language η, the problem of deciding whether $s \in \eta$ or not.

Given an LTE language, there exists a number n such that for all the strings s having $|s| > n$, $s \in \eta$ iff there exists w, y, z such that $1 \leq |y| \leq n$, $s = wyz$, $wz \in \eta$ and $\forall i \in \mathbb{N} wy^iz \in \eta$.

7.3. Regularizing LTEO

In the previous section, it has been presented how LTE languages can be translated into other f-equivalent LT languages, and thus which problems for them are decidable. Let us proceed with this journey by discussing f-equivalent representations of LTEO languages. In particular, being the LTEO family of languages the closure of the LTE one with respect to boolean operations and concatenation, the following steps are now made:

1. Discussion of the concatenation;
2. Discussion of the negation operation;
3. Discussion of the intersection operation;
4. Definition of the LTR family of languages;

5. Discussion over LTEO and LTR.

7.3.1. Concatenation

Let us start with the introduction of the concatenation operation. This operation is the first one to be presented for mainly two reasons: it is the simplest and it does not change with respect to the traditional definition. Thus, there can be omitted the definition of concatenation.

Theorem 7.6 (concatenation). *Given two LT languages η_1, η_2 and two sets E_1, E_2 such that $E_1 \cap E_2 = \emptyset$ and $\eta_1 \subseteq M(\Sigma, E_1)^*$, $\eta_2 \subseteq M_{\Sigma,E_2}(\Sigma, E_2)^*$, then also the language $\eta_1 \cdot \eta_2$ is LT.*

The proof of the previous theorem is trivial considering that the two languages η_1 and η_2 are defined over different alphabets.

Theorem 7.7. *Given two languages $\eta_1 \subseteq M(\Sigma, E_1)^*$, $\eta_2 \subseteq M(\Sigma, E_2)^*$ with $E_1 \cap E_2 = \emptyset$, the following languages are equivalent:*

- $f_{\Sigma,(E_1 \cup E_2)}(\eta_1 \cdot \eta_2, (E_1 \cup E_2))$;
- $f_{\Sigma,E_1}(\eta_1, E_1) \cdot f_{\Sigma,E_2}(\eta_2, E_2)$.

Proof. To prove this theorem, note that being $E_1 \cap E_2 = \emptyset$, then for each element $s_1 \in \eta_1$ and $s_2 \in \eta_2$, then $f_{\Sigma,(E_1 \cup E_2)}(s_1 \cdot s_2, (E_1 \cup E_2)) = f_{\Sigma,E_1}(s_1, E_1) \cdot f_{\Sigma,E_2}(s_2, E_2)$ and there are no other strings belonging to $f_{\Sigma,(E_1 \cup E_2)}(\eta_1 \cdot \eta_2, (E_1 \cup E_2))$ or $f_{\Sigma,E_1}(\eta_1, E_1) \cdot f_{\Sigma,E_2}(\eta_2, E_2)$. □

Corollary 7.7. *Given two languages $\eta_1, \eta_2 \subseteq \Sigma^*$, there exist a f-equivalent representation of $\eta_1 \cdot \eta_2$.*

7.3.2. Negation

Regarding the negation operation, note that usually, having as alphabet Σ, the $\neg\eta$ is equivalent to $\Sigma^* \setminus \eta$, however, this concept cannot be applied for our purposes, because the alphabet may change, and there can be the needing to express the negation concept over a subset of the alphabet.

Definition 7.8 (Negation). *Let η be a language defined over $M(\Sigma, E_t)$, and let E_p, E_s, E_t be three sets such that$E_s, E_p \subseteq E_t$, then:*

$\neg_{E_t,E_s,E_p}\eta = \{s \mid s \in M(\Sigma, E_t) \wedge \forall s' \in P_{\Sigma,E_t}(s_i, S, E_p) s' \notin \eta\}$

Theorem 7.8. *Given a language $\eta \subseteq \Sigma^*$, a set E, and being $\eta' \in L_{\Sigma,E}(\eta, \Sigma, E)$, the following languages are equivalent:*

- $\neg\eta$
- $f_{\Sigma,E}(\neg_{E,E,E}\eta', E)$

Proof. To prove the theorem, we will proceed by proving the following steps:

- If $s \notin \neg\eta$, then also $s \notin f_{\Sigma,E}(\neg_{E,E,E}\eta', E)$;
- If $s \notin f_{\Sigma,E}(\neg_{E,E,E}\eta', E)$, then also $s \notin \neg\eta$

If $s \notin \neg\eta$, then $s \in \eta$, thus for each $s' \in P_{\Sigma,E}(s, \Sigma, E)$ it is valid $s' \in \eta'$ which implies $s' \notin \neg_{E,E,E}\eta'$, which implies that $s \notin f_{\Sigma,E}(\neg_{E,E,E}\eta', E)$.

If $s \notin f_{\Sigma,E}(\neg_{E,E,E}\eta', E)$ then for each $s' \in P_{\Sigma,E}(s, \Sigma, E)$, $s' \notin \neg_{E,E,E}\eta'$ which implies that there exists at least one string $s'' \in P_{\Sigma,E_t}(s', M(\Sigma, E), E)$ then $s'' \in \eta'$, which implies that $f_{\Sigma,E}(s'', E) \in \eta$, which implies $f_{\Sigma,E}(s'', E) \notin \neg\eta$ and, by noticing that $f_{\Sigma,E}(s'', E) = s$, finally proves $s \notin \neg\eta$. □

Corollary 7.8. *Given a language $\eta \subseteq M(\Sigma, E)^*$, the following languages are equivalent:*

- $\neg\eta$
- $\neg_{E,E,E}\eta$

7.3.3. Intersection

Another operation that can be introduced is the intersection. Please note that, because we are working on a wider alphabet, as it has been done for the negation, the standard definition must be adapted.

Definition 7.9 (intersection). *Given two languages $\eta_1, \eta_2 \subseteq M(\Sigma, E_t)$, and five sets $E_p, E_1, E_2 \subseteq E_t$, $S = M^0(\Sigma, E_p)$, then:*
$\eta_1 \bigcap_{E_t,E_p,E_1,E_2} \eta_2 = \{s \mid \exists s' \in P_{\Sigma,E_t}(s, S, E_1)\ s' \in \eta_1 \wedge \exists s'' \in P_{\Sigma,E_t}(s, S, E_2)\ s'' \in \eta_2\}$

As it has been done for both negation and concatenation, let us prove that the just stated definition can be linked to the normal intersection:

Theorem 7.9. *Given two languages $\eta_1, \eta_2 \subseteq \Sigma^*$, and two sets $E_1, E_2 \in E_t$, being $\eta_1' \in L_{\Sigma,E_t}, (\eta_1, \Sigma, E_1)$, $\eta_2' \in L_{\Sigma,E_t}, (\eta_2, \Sigma, E_2)$, then the following two languages are equivalent:*

- $\eta_1 \cap \eta_2$
- $\eta_1' \bigcap_{E_t,\emptyset,E_1,E_2} \eta_2'$

Proof. In order to prove the theorem, we will proceed by proving:

- If $s \in \eta_1 \cap \eta_2$, then also $s \in \eta_1' \bigcap_{E_t,\emptyset,E_1,E_2,} \eta_2'$;
- If $s \in \eta_1' \bigcap_{E_t,\emptyset,E_1,E_2,} \eta_2'$, then also $s \in \eta_1 \cap \eta_2$.

If $s \in \eta_1 \cap \eta_2$, then $s \in \eta_1$ and $s \in \eta_2$, which means there exists $s' \in P_{\Sigma,E_t}(s,\Sigma,E_1)$, $s'' \in P_{\Sigma,E_t}(s,\Sigma,E_2)$ such that $s' \in \eta_1'$ and $s'' \in \eta_2'$, which, by construction, implies that $s \in \eta_1' \bigcap_{E_t,\emptyset,E_1,E_2,} \eta_2'$.

If $s \in \eta_1' \bigcap_{E_t,\emptyset,E_1,E_2,} \eta_2'$ it means that there exist $s' \in P_{\Sigma,E_t}(s,\Sigma,E_1)$, $s'' \in P_{\Sigma,E_t}(s,\Sigma,E_2)$ such that $s' \in \eta_1'$ and $s'' \in \eta_2'$, which means that $f_{\Sigma,E_t}(s',E_1) \in \eta_1$, and $f_{\Sigma,E_t}(s'',E_2) \in \eta_2$, but noticing that $f_{\Sigma,E_t}(s',E_1) = f_{\Sigma,E_t}(s'',E_2) = s$, then $s \in \eta_1 \cap \eta_2$. □

Corollary 7.9. *Given two languages $\eta_1, \eta_2 \subseteq M(\Sigma,E)^*$, the following two languages are equivalent:*

- $\eta_1 \cap \eta_2$
- $\eta_1 \bigcap_{E,\emptyset,\emptyset,\emptyset} \eta_2'$

Please note that also with this formulation of negation and intersection, the De Morgan rule is valid:

Corollary 7.10. *Given any two languages $\eta_1, \eta_2 \subseteq M^0(\Sigma,E)^*$, then:*
$\eta_1 \cup \eta_2 = \neg_{E,E,E}((\neg_{E,E,E}\eta_1) \bigcap_{E,\emptyset,\emptyset,\emptyset} (\neg_{E,E,E}\eta_2))$.

7.3.4. LTR

Here is defined the LTR class of languages. In this definition, there are given only the main features of LTR, and then the LTR expressiveness can be expanded thanks to later theorems.

Definition 7.10 (LTR). *The class of LTR languages contains all the languages $\eta \subseteq M(\Sigma,E)^*$ such that:*

- *η is LT;*
- *$\eta = \eta_1 \bigcap_{E,E_p,E_1,E_2} \eta_2$ with η_1 being $\eta_1 \in M(\Sigma,E)^*$, $\eta_2 \in M(\Sigma,E)^*$ two LTR languages, with and $E_1, E_2, E_p \subseteq E$;*

- *$\eta = \neg_{E_t,E_s,E_t}\eta_1$ with $\eta_1 \subseteq M(\Sigma, E)$ an LTR language and with $E_s, E_p \subseteq E_t$.*

Corollary 7.11. *Given a LT language $\eta \subseteq \Sigma^*$, then also any language $\eta' \in L_{\Sigma,E}(\eta, \Sigma, E)$ is LT.*

Corollary 7.12. *Given an alphabet Σ and a set E, then the language $M(\Sigma, E)^*$ is LT, and thus also LTR.*

Proof. It is trivial to see that the LT language defined as $LT_2(\alpha, \beta, \gamma, \delta)$ with:
$\alpha = \beta = \gamma = M(\Sigma, E)^2$;
$\delta = M(\Sigma, E) \cup \{\varepsilon\}$
defines the language $M(\Sigma, E)^*$. □

In the definition of LTR, there has not been reported the closure of the family with respect to concatenation, however, with the following result, it will be clear that LTR is closed also under it.

Theorem 7.10. *Given two LTR $\eta_1 \subseteq M(\Sigma, E_1)^*$, $\eta_2 \subseteq M(\Sigma, E_2)^*$ with $E_1 \cap E_2 = \emptyset$, then also $\eta_1 \cdot \eta_2$ is LTR.*

Proof. To prove this theorem, we will proceed by discussing case by case.

1. If η_1, η_2 are both LT, then, thanks to theorem 7.6, also $\eta_1 \cdot \eta_2$ is LT and thus LTR;
2. If $\eta_1 = \neg_{E_1,E_{p1},E_{s1}}\eta'_1$ and η_2, η'_1 are LT such that $\eta'_1 \cdot \eta_2$ is LTR, then:
   - $\neg_{(E_1 \cup E_2),E_{p1},E_{s1}}\eta'_1 \cdot \eta_2$ is trivial to see it is LTR;
   - $\neg_{(E_1 \cup E_2),E_{p1},E_{s1}}\eta'_1 \cdot \eta_2$ defines the same language of $\eta_1 \cdot \eta_2$.
3. If $\eta_2 = \neg_{E_2,E_{p2},E_{s2}}\eta'_2$ and η_1, η'_2 are LT such that $\eta_1 \cdot \eta'_2$ is LTR, it can be proven in a similar way of how it has been done for the previous point;
4. If $\eta_1 = \neg_{E_1,E_{p1},E_{s1}}\eta'_1$, $\eta_2 = \neg_{E_2,E_{p2},E_{s2}}\eta'_2$ and η'_1, η'_2 are LT for which the concatenation with an other LTR generates a LTR language, then:
   - $\neg_{(E_1 \cup E_2),E_{p1},E_{s1}}(\eta'_1 \cdot M(\Sigma, E_2)^*) \bigcap_{(E_1 \cup E_2),\emptyset,\emptyset,\emptyset}$ $\neg_{(E_1 \cup E_2),E_{p2},E_{s2}}(M(\Sigma, E_1)^* \cdot \eta'_2)$ is LTR, because $M(\Sigma, E)^*$ is an LTR, for hypothesis, both $\eta'_1 \cdot M(\Sigma, E_2)^*$ and $M(\Sigma, E_1)^* \cdot \eta'_2$ are LTR, thus also $\eta_1 \cdot \eta_2$ is LTR.
   - $\neg_{(E_1 \cup E_2),E_{p1},E_{s1}}(\eta'_1 \cdot M(\Sigma, E_2)^*) \bigcap_{(E_1 \cup E_2),\emptyset,\emptyset,\emptyset}$ $\neg_{(E_1 \cup E_2),E_{p2},E_{s2}}(M(\Sigma, E_1)^* \cdot \eta'_2)$ defines the same language of $\eta_1 \cdot \eta_2$. This point

is quite trivial after noticing that the $\bigcap_{E_t,\emptyset,\emptyset,\emptyset}$ coincides to use the standard boolean operator $\cap$ over two languages defined over $M(\Sigma, E_t)$.

5. If $\eta_1 = \eta_1' \bigcap_{E_1,E_{1,s},E_{1,1},E_{1,2}} \eta_1''$ where $E_{1,s}, E_{1,1}, E_{1,2} \subseteq E_1$, and $\eta_1', \eta_1'', \eta_2$ are LT such that $\eta_1' \cdot \eta_2$ and $\eta_1'' \cdot \eta_2$ are both LTR, then:

   - $(\eta_1' \cdot \eta_2) \bigcap_{(E_1 \cup E_2),E_{1,s},E_{1,1},E_{1,2}} (\eta_1'' \cdot \eta_2)$, being $(\eta_1' \cdot \eta_2)$ and $(\eta_1'' \cdot \eta_2)$ LTR, is LTR too;

   - $(\eta_1' \cdot \eta_2) \bigcap_{(E_1 \cup E_2),E_{1,s},E_{1,1},E_{1,2}} (\eta_1'' \cdot \eta_2)$ defines the same language of $\eta_1 \cdot \eta_2$. For the sake of simplicity, let us call $\xi = (\eta_1' \cdot \eta_2) \bigcap_{(E_1 \cup E_2),E_{1,s},E_{1,1},E_{1,2}} (\eta_1'' \cdot \eta_2)$. To prove $\xi = \eta_1 \cdot \eta_2$, we will firstly prove that $\xi \subseteq \eta_1 \cdot \eta_2$ and then that $\eta_1 \cdot \eta_2 \subseteq \xi$.
     To prove the first step, let us suppose that there exists $s \in \xi$, thus
     $\exists s' \in P_{\Sigma,(E_1 \cup E_2)}(s, M(\Sigma, E_{1,s}), E_{1,1}) \wedge s' \in (\eta_1' \cdot \eta_2)$, and
     $\exists s'' \in P_{\Sigma,(E_1 \cup E_2)}(s, M(\Sigma, E_{1,s}), E_{1,2}) \wedge s'' \in (\eta_1'' \cdot \eta_2)$. Moreover, being $E_1 \cap E_2 = \emptyset$, there must be $s = s_1 \cdot s_2$ with $s_1 \in M(\Sigma, E_1)^*, s_2 \in M(\Sigma, E_2)^*$, and being $E_{1,s} \subseteq E_1$, then $P_{\Sigma,(E_1 \cup E_2)}(s_2, M(\Sigma, E_{1,s}), E_{1,1}) = P_{\Sigma,(E_1 \cup E_2)}(s_2, M(\Sigma, E_{1,s}), E_{1,2}) = s_2$, thus also $s' = s_1' \cdot s_2$ and $s'' = s_1'' \cdot s_2$, with $s_1' \in P_{\Sigma,(E_1 \cup E_2)}(s_1, M(\Sigma, E_{1,s}), E_{1,1}) \subseteq M(\Sigma, E_1)$, thus $s_1' \in \eta_1'$, and $s_1'' \in P_{\Sigma,(E_1 \cup E_2)}(s_1, M(\Sigma, E_{1,s}), E_{1,2}) \subseteq M(\Sigma, E_1)$, thus $s_1'' \in \eta_1''$, which implies that $s_1 \in \eta_1$, which proves that if $s \in \xi$ then $s \in \eta_1 \cdot \eta_2$.
     To prove the second step, instead, we want to prove that if $s \in \eta_1 \cdot \eta_2$, then $s \in \xi$. If $s \in \eta_1 \cdot \eta_2$, then $s = s_1 \cdot s_2 \mid s_1 \in M(\Sigma, E_1)^* \wedge s_1 \in \eta_1 \wedge s_2 \in M(\Sigma, E_2)^* \wedge s_2 \in \eta_2$, which implies that there exist $s_1' \in P_{\Sigma,E_1}(s_1, M(\Sigma, E_{1,s}), E_{1,1})$ and $s_1'' \in P_{\Sigma,E_1}(s_1, M(\Sigma, E_{1,s}), E_{1,2})$ such that $s_1' \in \eta_1'$ and $s_1'' \in \eta_1''$. Now, being $E_1 \cap E_2 = \emptyset$, then it is trivial to see that $s_1' \cdot s_2 \in P_{\Sigma,(E_1 \cup E_2)}(s, M(\Sigma, E_{1,s}), E_{1,1})$ and $s_1'' \cdot s_2 \in P_{\Sigma,(E_1 \cup E_2)}(s, M(\Sigma, E_{1,s}), E_{1,2})$, which implies that $s \in \xi$, which was what we were searching for.

6. If $\eta_2 = \eta_2' \bigcap_{E_2,E_{2,s},E_{2,1},E_{2,2}} \eta_2''$ where $E_{2,s}, E_{2,1}, E_{2,2} \subseteq E_2$, and $\eta_2', \eta_2'', \eta_1$ are LT such that $\eta_1 \cdot \eta_2'$ and $\eta_1 \cdot \eta_2''$ are both LTR, then the proof that $\eta_1 \cdot \eta_2$ is LTR is the same of the previous point by changing the order of the concatenation;

7. If $\eta_1 = \eta_1' \bigcap_{E_1,E_{1,s},E_{1,1},E_{1,2}} \eta_1''$ where $E_{1,s}, E_{1,1}, E_{1,2} \subseteq E_1$ and $\eta_2 = \eta_2' \bigcap_{E_2,E_{2,s},E_{2,1},E_{2,2}} \eta_2''$ where $E_{2,s}, E_{2,1}, E_{2,2} \subseteq E_2$ where $\eta_1', \eta_1'', \eta_2', \eta_2''$ are LT, then:

   - $(\eta_1 \cdot M(\Sigma, E_2)^*) \bigcap_{(E_1 \cup E_2),\emptyset,\emptyset,\emptyset} (M(\Sigma, E_1)^* \cdot \eta_2)$ is LTR, because $M(\Sigma, E_2)^*$ and $M(\Sigma, E_1)^*$ are LT, thus also $(\eta_1 \cdot M(\Sigma, E_2)^*)$ and $(M(\Sigma, E_1)^* \cdot \eta_2)$ are LTR, and the intersection of two LTR languages is LTR;

   - $(\eta_1 \cdot M(\Sigma, E_2)^*) \bigcap_{(E_1 \cup E_2),\emptyset,\emptyset,\emptyset} (M(\Sigma, E_1)^* \cdot \eta_2)$ is trivially equivalent to $\eta_1 \cdot \eta_2$

because each string it recognizes can be split in two, so $s = s_1 \cdot s_2$ where $s_1 \in M(\Sigma, E_1)^*$ and $s_2 \in M(\Sigma, E_2)^*$, with $s_1 \in \eta_1$ and $s_2 \in \eta_2$, thus the string also belongs to $\eta_1 \cdot \eta_2$, and the same can be stated for the other side: each string of $s \in \eta_1 \cdot \eta_2$ is composed of two parts $s = s_1 \cdot s_2$ such that $s_1 \in \eta_1$ and $s_2 \in \eta_2$, and thus $s_1 \in M(\Sigma, E_1)^*$ and $s_2 \in M(\Sigma, E_2)^*$ which implies it is recognizable by $(\eta_1 \cdot M(\Sigma, E_2)^*) \bigcap_{(E_1 \cup E_2), \emptyset, \emptyset, \emptyset} (M(\Sigma, E_1)^* \cdot \eta_2)$.

8. If $\eta_1 = \eta_1' \bigcap_{E_1, E_{1,s}, E_{1,1}, E_{1,2}} \eta_1''$ where $E_{1,s}, E_{1,1}, E_{1,2} \subseteq E_1$ and $\eta_2 = \neg_{E_2, E_{p2}, E_{s2}} \eta_2'$, and $\eta_2', \eta_1', \eta_1''$ are LT, then:

   - $(\eta_1 \cdot M(\Sigma, E_1)^*) \bigcap_{(E_1 \cup E_2), \emptyset, \emptyset, \emptyset} (M(\Sigma, E_2)^* \cdot \eta_2)$ is LTR;
   - $(\eta_1 \cdot M(\Sigma, E_1)^*) \bigcap_{(E_1 \cup E_2), \emptyset, \emptyset, \emptyset} (M(\Sigma, E_2)^* \cdot \eta_2)$ is equivalent to $\eta_1 \cdot \eta_2$. To prove it, there can be noticed that in the proof of item 7, where the same result is reported, there is no restriction on the fact that η_1 or η_2 are intersections of LTR and not other LTR languages.

9. If $\eta_1 = \neg_{E_1, E_{p1}, E_{s1}} \eta_1'$ and $\eta_2 = \eta_2' \bigcap_{E_2, E_{2,s}, E_{2,1}, E_{2,2}} \eta_2''$ where $E_{2,s}, E_{2,1}, E_{2,2} \subseteq E_2$, and $\eta_1', \eta_2', \eta_2''$ are LT, then the same reasoning of the previous point can be applied by switching the order of languages.

$\square$

With the previous results, there can be stated:

Theorem 7.11. *Given an LTO language η, there can be built an LTR language η' that is f-equivalent to η.*

Proof. The proof consists of noticing that:

- Given an LT language, there can be built an f-equivalent language;
- Given two languages eta_1, η_2 and two f-equivalents $\eta_1' \subseteq M(\Sigma, E_1)^*, \eta_2' \subseteq M(\Sigma, E_2)^*$, with $E_1 \cap E_2 = \emptyset$, $E_1 \cup E_2 \subseteq E$, then are also f-equivalent $\neg \eta_1$ and $\neg_{E_1, E_1, E_1} \eta_1$, $\eta_1 \cap \eta_2$ and $\eta_1' \bigcap_{E, E, E_1, E_2} \eta_2'$, $\eta_1 \cdot \eta_2$ and $\eta_1' \cdot \eta_2'$;

$\square$

7.3.5. LTEO

Here is discussed how to relate LTEO and LTR

Theorem 7.12. *Given an LTOE language η, there can be built an LTR language η' that is f-equivalent to η.*

Proof. To prove the theorem we can proceed by proving that there exists a language f-equivalent to $stringSub(\tau, map)$ with τ LT over Σ_Φ, with the function map that maps each element of $\phi(\tau)$ into a language over Σ^* for which there exists a f-equivalent LTR. If this holds, being LTR closed under concatenation, intersection, and negation, then there can be described in LTR form all the possible LTEO languages.

There will be considered $|\Phi| = 1$ and τ once reliant over $-$, as done in theorem 7.3, without loss of generality.

By definition, an LT τ can be rewritten in LTR form as $\tau' \in L_{\Sigma,E}(\eta, \Sigma, E)$, so if $\phi(\tau) = \emptyset$, the theorem is valid.

Let us consider the language $\theta = stringSub(\tau, \{- \to \eta\})$:

- If η is LT, then, thanks to theorem 7.3, there can be built an f-equivalent LTR to θ that is also LT;

- If $\eta = \neg\xi$ such that
  $stringSub(\tau, \{- \to \xi\}) = f_{\Sigma,E}(stringSub(\tau', \{- \to \xi'\}), E)$ where
  $stringSub(\tau', \{- \to \xi'\})$ is LTR and
  $\xi' \in M(\Sigma, E')^*,\ E' \in E, \tau' \in M(\Sigma_\Phi, (E \setminus E'))$, then
  $f_{\Sigma,E}((stringSub(\tau', \{- \to M(\Sigma, E')^*\}) \bigcap_{E,E',E'} stringSub(\tau', \{- \to \xi'\})) \cup \tau_{\dot{-}}, E)$
  is LTR and it is f-equivalent to θ;

- If $\eta = \xi_1 \cap \xi_2$ such that
  $stringSub(\tau, \{- \to \xi_1\}) = f_{\Sigma,E}(stringSub(\tau', \{- \to \xi_1'\}), E)$ and
  $stringSub(\tau, \{- \to \xi_2\}) = f_{\Sigma,E}(stringSub(\tau', \{- \to \xi_2'\}), E)$ are LTR with
  $\xi_1' \in M(\Sigma, E_1)^*, \xi_2' \in M(\Sigma, E_2)^*, E_p = E_1 \cup E_2$, then also
  $stringSub(\tau', \{- \to \xi_1'\}) \bigcap_{E_t,E_p,E_1,E_2} stringSub(\tau', \{- \to \xi_2'\})$ is LTR, and it is f-equivalent to θ;

- If $\eta = \eta_1 \cdot \eta_2$ with η_1, η_2 for which there exist two corresponding f-equivalent LTR languages, then also for $stringSub(\tau, \{- \to \eta\})$ there is an f-equivalent LTR.

□

7.3.6. Regularizing LTOP

Here is discussed how the concept of *fence-subs* can be merged with the one of LTR.

As a first thing, let us take a look at the just stated theorem 7.12. From it, there can be seen that for the substring that is substituted inside the f-equivalent LTR language, this substring has a unique alphabet, used only for the other substrings taken from the same language mapped by the function *map*, that can be used in other substitutions. This means that, in the LTR language, it is clear which parts of a string have been substituted, from which language they came, and so on. And all that is possible because those substrings are defined over an alphabet that is different from time to time.

At this point, consider an OP-alphabet (Σ, M). Obviously, it defines structure over strings belonging to Σ^*, but we need to expand this domain, to work with strings belonging to the generic $M^0(\Sigma, E)^*$ in order to identify stings for which each substring that does not belongs to Σ is well chained in the string. To achieve such a goal, let us call M' the OPM that defines the OP-alphabet $(M^0(\Sigma, E), M')$.
It is trivial that M must be contained in M', and that for each element $e \in E$, the elements of M' relative to $M(\Sigma, \{e\})^2$, must coincide with the ones of the f-equivalent elements of M.
At this point, by leveraging the fact that we can consider the LT language on which *fence-subs* is applied, as once reliant on each element of Φ, then there is trivial to see that each element $- \in \Phi$, and thus each substitution, will be preceded and followed always by the same characters $a, b \in \Sigma$. Thus, leveraging this fact, we can define the elements of M' connecting the alphabet $M(\Sigma, E_1)$ to an other alphabet, by leveraging the fact that each substring containing characters of $M(\Sigma, E_1)$, will be included in the two characters a, b.

At this point is trivial to see that the way to define an f-equivalent language of an LTOP one, is to find the LTR language f-equivalent to the LTEO version of the LTOP language, and intersect the LTR language with the Max Language pf the newly built M', which will in turn guarantee that the substrings produced by substitutions are well chained over the string.

So, to summarize, the following can be stated:

Theorem 7.13. *Given a language η that is LTOP (without the constraint on FSR), there exist a max language $ML_{M'}$ a language LTR η' such that $\eta' \cap MaxLang_{M'}$ is f-equivalent to η.*

7.4. Decidability

In this section, it is discussed the decidability problem. In order to do so, there are presented two sections: one discussing the decidable problems of LTR, and one describing the decidability problem of FSR.

7.4.1. Decidability for LTR

The LTR definition is done in such a way to guarantee the following results. Please note that the proof of theorem 7.14 will become trivial after the theorem 7.15.

Theorem 7.14 (decidability of string belonging to LTR)**.** *The problem of deciding whether or not a certain string s belongs to an LTR language η, is decidable.*

Proof. To prove the theorem, we will proceed with the following steps:

- We will discuss the problem for the base case of LTR;
- We will prove that, given a string s, the set $P_{\Sigma,E_t}(s, S, E_p)$ is finite;
- Assuming that the problem is decidable for two generic LTR languages $\xi_1\xi_2$, we will prove it is decidable also for the negation of ξ_1 or the intersection of ξ_1, ξ_2.

As the first thing, we want to prove that for the simplest LTR, the problem of deciding whether or not a string belongs to the language, is decidable. Given that the simplest case of an LTR language is a language that is LT, and being LT regular languages, we can rely on theorem 1.3 that says that this problem is decidable for Regular languages.

As a second thing, we need to prove the following:

Corollary 7.13. *Given three sets $E_p \subseteq E_t$ and S, and a string $s \in M^0(\Sigma, E_t)^*$, the set $P_{\Sigma,E_t}(s, S, E_p)$ is finite.*

Proof. There may be many ways to prove this corollary, however, one consists of noticing that the elements of $P_{\Sigma,E_t}(s, S, E_p)$ have the same, finite, length of s. This, together with the fact that they are defined over an alphabet, $M^0(\Sigma, E_t)$, with a finite number of elements, will enable us to calculate the maximum number of elements of the set, so having
$|P_{\Sigma,E_t}(s, S, E_p)| \leq M^0(\Sigma, E_t)| \times |s|$. □

Now, the final steps left to end the proof, are to prove that, given two LTR languages for which it is decidable if a string s belongs to them or not, it is decidable if a string s belongs to the negation of one of the languages or to the intersection of them. Let us say that the two LTR languages are ξ_1, ξ_2, then:

- The solution of deciding if a string s belongs to the $\neg_{E_t,E_s,E_p}\xi_1$ can be computed by checking, for each element of $P_{\Sigma,E_t}(s,S,E_p)$, if it belongs to ξ_1 or not. By hypothesis, this problem is decidable, and thus deciding if $s \in \neg_{E_t,E_s,E_p}\xi_1$ relies on a finite number of decidable problems, so the problem is decidable;
- The solution of deciding if a string s belongs to the $\xi_1 \bigcap_{E_t,E_p,E_1,E_2} \xi_2$ can be computed by checking for each element of $P_{\Sigma,E_t}(s,S,E_1)$, if it belongs to ξ_1, and for each element of $P_{\Sigma,E_t}(s,S,E_2)$, if it belongs to ξ_2. Being those two decidable problems per hypothesis, then the original problem can be solved by checking a finite number of times a decidable problem; thus it is decidable.

In this way, we proved that the problem of belonging is decidable for the LTR family of languages. □

Although the previous result may appear a good result, we need something stronger: we want the problem of deciding whether or not an LTR is empty, to be decidable.

Let us now introduce one really important result:

Theorem 7.15 *($LTR \subseteq RL$). LTR is a subfamily of Regular Languages.*

Proof. The proof of this theorem consists of 3 parts:

1. Prove the base case of LTR is RL;
2. Prove that if a LTR η is RL, then also $\neg_{E_t,E_s,E_p}\eta$ is RL;
3. Prove that if two LTR η_1, η_2 are RL, then also $\eta_1 \bigcap_{E_t,E_p,E_1,E_2} \eta_2$ is RL;

Let us start with the first point. Being the base case of LTR an LT language, then it is known that it is also RL.

For the second step, let us take for granted that η is RL; thus there exists a Finite State Automata recognizing η.
It is trivial to see that there can be built a non-deterministic Finite Step Automata that, given as input a string s, writes on the output a string inside $P_{\Sigma,E_t}(s, M(\Sigma,E_s), E_p)$. Thus the language recognized by $\neg_{E_t,E_s,E_p}\eta$ is the same not recognized by the composition of the two automata.
It is known that the composition of two Finite State Automata has the same expressive power of one single Finite State Automata, and that for any non-deterministic Finite State Automata, there exists an equivalent one that is deterministic.
Thus in this case we proved that there exists a (deterministic) Finite State Automata

recognizing the language $\neg_{E_t,E_s,E_p}\eta$, however, being the class of languages recognized by Finite State Automata corresponding to the class of Regular Languages, then $\neg_{E_t,E_s,E_p}\eta$ is a Regular Language.

To prove the intersection, the process is quite equivalent to the one we followed for the negation. □

The previous result makes it trivial the following one:

Theorem 7.16 (*f-equivalent* of LTR is Regular)**.** *If a given language η is f-equivalent to any LTR language ξ, then η is a Regular Language*

Proof. Notice that, given any alphabet Σ such that $\eta \subseteq \Sigma^*$, and any set E such that $\xi \subseteq M^0(\Sigma, E)^*$, there can be built a (non-deterministic) Finite State Automaton that is able to "translate" any string $s \in \Sigma^*$ into any *f-eqivalent* string $s' \in M^0(\Sigma, E)^*$. This, combined with the fact that there exists a Finite State Automaton that recognizes the LTR ξ, implies that there exists a non-deterministic Finite State Automaton that ends the computation in a final state if receives as input a string $s \in \Sigma$ that is *f-equivalent* to a string $s' \in M^0(\Sigma, E)^*$ and such that $s' \in \xi$.

It is notorious that for any non-deterministic Finite State Automaton, there is an equivalent deterministic one. Thus there exists a deterministic Finite State Automaton that defines the same language of η. Given that the class of languages recognizable with deterministic Finite State Automaton coincides with the class of Regular Languages, it is trivial to see that also η is a Regular Language. □

Corollary 7.14 ($LTEO \subseteq RL$)**.** *Given any LTEO language η, it is also a Regular Language.*

With the previous results, we are guaranteed that not only the problem of deciding whether or not an LTR is empty is decidable, but we also gain the fact that all the results of Regular Languages are also valid for LTR, in particular the pumping lemma.

7.4.2. FSR decidability

Here is discussed the decidability of the problem of recognizing whether or not a LTOP language satisfies the FSR.

Theorem 7.17. *The problem of deciding whether or not a language satisfies the FSR property corresponds to decide whether or not there exists a not empty language in a finite* *set of LTR languages intersected with a MaxLanguage.*

Proof. The proof of this theorem relies on the fact that for corollary 7.2 the FSR property relies on a finite number of comparisons of languages, and for theorem 7.13 those languages are empty only if are empty also the intersections of two languages that are LTOE using *fence-subs* instead of *stringSubs*, and a MaxLanguage. □

Theorem 7.18. *Given a LTR η and a $MaxLanguage_{M'}$, it is decidable whether $\eta \cap ML_{M'}$ is empty or not.*

Proof. To prove this theorem, let us consider η. From theorem 7.15, η is a Regular language, thus it is also a Context Free Language, which means that it can be defined by a suitable Context Free Grammar G, such that it is an Operator Grammar.

Let us now calculate the Operator Precedence Relations holding over G, and let us remove all those rules that will provide relations not contained in M'. The resulting grammar G' is a Context Free Grammar that generates the language resulting from the intersection of η and $MaxLanguage_{M'}$.

Now, by applying theorem 3.2, the problem of deciding whether or not $L(G')$ is empty, is decidable, because it relies on the checking if there exists at least one string z of length shorter than p, such that $z \in L(G')$. Thus there is a finite number of strings to be checked, and, being the problem of deciding whether or not a particular string s belongs to a CFL decidable, then the initial problem relies on a finite number of decidable problems. □

And, thus, finally:

Theorem 7.19. *The problem of deciding whether or not a given LTOP language η satisfies FSR is decidable.*

Proof. The proof relies on theorem 7.17 and theorem 7.18, making this problem decidable because composed by a finite set of finite problems. □

So, there has been proved that the problem of recognizing whether or not a given LTOP language satisfies the FSR, is decidable.

Conclusions

The goal of this work was to find a class of Operator Precedence Non-Counting languages starting from LTO over RL, by trying to mimic its definition, and to relate it with the other representation present in the literature; However, the final result, which was somehow unexpected, showed that not only the found LTOP is a subclass of NC_{OP}, but it coincides with NC_{OP} itself. This unexpected result could be explained by the lots of ideas that were abandoned because not enough expressive (someone turned out to be less expressive than $LTO \cap ML$), or too expressive. The idea of LTOP took place from a previous one that was more expressive than NC_{OP}, recognizing the language of example 6.8. Then, with the introduction of FSR, things became more complex, but in the end, it was worth it because it limited the expressiveness enough to guarantee LTOP to be contained in NC_{OP}, but not too much, ensuring NC_{OP} is contained in LTOP. It has been also studied the decidability problem of FSR, proving that it is decidable whether or not a language is LTOP.

The work has been done by reading the materials, studying the more important parts, and researching the things that were left unclear or on which there could be useful deeper knowledge. During this work, there has been a flight over lots of concepts and ideas in the field of computer science, and each one of them has been just hinted at because there are really a lot of works and researches on each one of them, however this gave the possibility to go deeper in concepts that during courses have been presented.

Bibliography

[1] *REPRESENTATION OF EVENTS IN NERVE NETS AND FINITE AUTOMATA*, pages 3–42. Princeton University Press, 1956. ISBN 9780691079165. URL `http://www.jstor.org/stable/j.ctt1bgzb3s.4`.

[2] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Z. Phonetik Sprachwiss. Kommunikationsforsch.*, 14:143–172, 1961. ISSN 0044-331X.

[3] M. Chiari, D. Mandrioli, and M. Pradella. Model-checking structured context-free languages. In A. Silva and K. R. M. Leino, editors, *CAV '21*, volume 12760 of *LNCS*, pages 387—410. Springer, 2021. doi: 10.1007/978-3-030-81688-9_18.

[4] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956. doi: 10.1109/TIT.1956.1056813.

[5] S. Crespi Reghizzi and D. Mandrioli. Operator Precedence and the Visibly Pushdown Property. *J. Comput. Syst. Sci.*, 78(6):1837–1867, 2012.

[6] S. Crespi Reghizzi and M. Pradella. Beyond operator-precedence grammars and languages. *Journal of Computer and System Sciences*, 113:18–41, 2020. doi: 10.1016/j.jcss.2020.04.006.

[7] S. Crespi Reghizzi, G. Guida, and D. Mandrioli. Noncounting Context-Free Languages. *J. ACM*, 25:571–580, 1978.

[8] S. Crespi Reghizzi, D. Mandrioli, and D. F. Martin. Algebraic Properties of Operator Precedence Languages. *Information and Control*, 37(2):115–133, May 1978.

[9] V. Diekert and P. Gastin. First-order definable languages. In *Logic and Automata: History and Perspectives, Texts in Logic and Games*, pages 261–306. Amsterdam University Press, 2008.

[10] R. W. Floyd. Syntactic Analysis and Operator Precedence. *J. ACM*, 10(3):316–333, 1963.

[11] M. A. Harrison. *Introduction to Formal Language Theory.* Addison Wesley, 1978.

[12] D. Jurafsky and J. H. Martin. *Speech and Language Processing.* 3rd draft edition, 2023.

[13] V. Lonati, D. Mandrioli, F. Panella, and M. Pradella. Operator precedence languages: Their automata-theoretic and logic characterization. *SIAM J. Comput.*, 44(4):1026–1088, 2015.

[14] D. Mandrioli and M. Pradella. Generalizing input-driven languages: Theoretical and practical benefits. *Computer Science Review*, 27:61–87, 2018. doi: 10.1016/j.cosrev.2017.12.001.

[15] D. Mandrioli, M. Pradella, and S. Crespi Reghizzi. Star-freeness, first-order definability and aperiodicity of structured context-free languages. In *Proceedings of ICTAC 2020*, Lecture Notes in Computer Science. Springer, 2020.

[16] R. McNaughton. Parenthesis Grammars. *J. ACM*, 14(3):490–500, 1967.

[17] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, 1960. doi: 10.1109/TEC.1960.5221603.

[18] E. Mendelson. *Introduction to Mathematical Logic.* D. Van Nostrand Company, Princeton, New Jersey, 1964.

[19] G. K. Pullum. Context-freeness and the computer processing of human languages. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 1–6, Cambridge, Massachusetts, USA, June 1983. Association for Computational Linguistics. doi: 10.3115/981311.981313. URL `https://aclanthology.org/P83-1001`.

[20] S. P. Robert McNaughton. *Counter-Free Automata.* The MIT Press, 1971. ISBN 9780198520115.

[21] A. K. Salomaa. *Formal Languages.* Academic Press, New York, NY, 1973.

List of Figures

List of Tables

Acknowledgements

I would like to express my deepest appreciation to prof. Matteo Pradella, who gave me the possibility to end my university path with a theoretical thesis, thus giving me the possibility to see a field that maybe there will be not many opportunities to dive into.

I am also grateful to Pay Reply, which gave me the possibility to conclude my university career while starting the working one. In particular, I'm very grateful to all my colleagues.

Thanks also should go to my friends for supporting me on this path; to my classmates that inspired me to do better; to the CEO of GRM IT; to all the people with whom I had the honor to work with because each one of them has something to teach me. Particularly Niccolo F., Fabrizio M., Gabriele B., Davide S., and many others.

Lastly, I would be remiss in not mentioning my family for everything they did. In particular, I would thank my mother, who gave me the possibility to achieve this goal in life.