National and Kapodistrian University of Athens

Graduate Program in Logic, Algorithms and Computation

M.Sc. Thesis:

Conjunctive and Boolean Grammars

Alexandros Palioudakis

Supervised by:
Panos Rondogiannis

Athens, July 2010

Acknowledgements

First and foremost, I would like to thank my sister Irini Palioudaki for her help in whatever I needed. I should thank her for the figures in this thesis, for her help in my English, for her support and for being always there. Also, I would like to thank my father for his financial support, that without it I couldn't realize my studies. Moreover, I would like to thank Dimitris Paparas and Aristotelis Misios for helping me when they could. Furthermore, I would like to thank my professors in MPLA for transmiting us their knowledge. Like professor Elias Koutsoupias whose lectures were inspiring. Last but not least, I feel the need to thank my thesis advisor Panos Rondogiannis, who believed in me, supported me and carefully corrected my thesis, which before had plenty mistakes. I am sorry if I am forgetting someone who helped me durring this thesis and I don't have in mind at the moment.

This work is dedicated to my professors, who influenced me the most,
Cornelia Kalfa and George Chantampis.

Contents

Abstract

In this thesis we examine four interesting classes of formal languages. The two of them, namely the regular and context-free ones, are well-known classes that have been widely studied in the literature [21, 22, 23, 24, 25]. The other two, namely conjunctive and Boolean languages, are more recent ones and appear to possess many interesting properties [1, 3, 15]. Conjunctive and Boolean languages can be produced by conjunctive and Boolean grammars respectively which are natural extensions of context-free grammars introduced by A. Okhotin in [1] and [15]. The basic idea behind of these new formalisms is to allow intersection, in conjunctive grammars, and intersection and negation, in Boolean grammars, in the right-hand side of rules. It is immediately obvious that the classes of conjunctive and Boolean languages are proper supersets of the class of context-free languages.

In this thesis we also examine three models of abstract machines closely related to the above classes of formal languages. These machines are: automata, pushdown automata and synchronized alternating pushdown automata. Each one of them corresponds in terms of expressive power to one of the previous classes of languages. More specifically, automata correspond to regular languages, pushdown-automata to context-free languages and synchronized alternating pushdown automata to conjunctive languages. At present, there is no known machine model that corresponds to Boolean languages.

Nowadays, there exist certain well-understood techniques for demonstrating that a language is not regular or context-free. Unfortunately this does not hold for conjunctive and Boolean languages. After ten years of study, no technique is presently known for showing that a language is *not* conjunctive or Boolean. Possibly this is the most important open problem in the area.

Most of the ideas presented in this thesis are based on existing articles, books and lecture notes. We were mostly influenced by Okhotin's lectures [27]. Our contributions can be outlined as follows:

- Lemmas 2.3.1 and 3.2.1 in which we present a language that can be produced by a one non terminal conjunctive grammar but not by a one non terminal context-free grammar.
- Propositions 2.3.1 and 3.2.1 which give us a more general criterion that can be used to show that certain languages can not be produced by one non terminal context-free and one non terminal conjunctive grammars.

- Subsection 3.3 which describes an approach which we believe that is promising for establishing that a specific language is not conjunctive.
- Subsection 3.4 in which we propose a slightly different model for synchronized alternating pushdown automata (SAPDA) than the one introduced in [11]. Moreover, we obtain and present in detail a new proof of the equivalence of the SAPDA model to conjunctive grammars.

We believe that a further study of conjunctive and Boolean grammars will prove to be very rewarding, since the area appears to have both a theoretical and a practical significance.

Chapter 1

Preliminaries

1.1 Basics Notions

1.1.1 Baby Sets and Functions

We shall begin with the definition of an important concept, the set. Almost, every mathematical book starts with sets, we will also do the same. For the needs of this thesis, it is sufficient to use the definition which says that a set is a collection of things. We denote a set by a letter or with a collection of elements inside the symbols '{' and '}'. We denote membership by the symbol '$\in$'. We write '$t \in A$' to denote that t is a member (or an element) of A, we write '$t \notin A$' to denote that t is not a member of A. For example, there is a set whose members are exactly the numbers $2, 3, 5$ and 7, and we write this set as:

$$\{2, 3, 5, 7\}$$

Then $t \in \{2, 3, 5, 7\}$ means that $t = 2$ or $t = 3$ or $t = 5$ or $t = 7$. We say that two sets are equal if they have exactly the same elements. Some times we want to specify the set whose elements have a specific property, let us call it P, so we write $\{x \mid x$ has the property P$\}$ (we denote by $\{x \in A \mid x$ has the property P$\}$ the set $\{x \mid x \in A$ and x has the property P$\}$). The set with no elements is called *empty set* and denoted by $\emptyset$.
Now, we will give four operations on sets. The *union* of the sets A and B is the set $A \cup B$ of all elements that are members of A or B (or both). Similarly, the *intersection* of A and B is the set $A \cap B$ of all elements that are members of both A and B. The *difference* A of B is the set $A - B$ (or A/B) of all elements that are members of A but not of B. The *Cartesian product* (or *product set*) of the sets A and B is the set $A \times B$ of all pairs (x, y), where $x \in A$ and $y \in B$. The Cartesian product can be generalized to the n-ary Cartesian product over n sets $A_1, \ldots, A_n$ as $A_1 \times \ldots \times A_n = \{\,(x_1, \ldots, x_n) \mid x_i \in A_i\}$. The number of elements of A denoted by $|A|$.
Another important relation between sets is the *subset*, denoted by the symbol '$\subseteq$'. We write $A \subseteq B$ and we read 'A is subset of B', if every element of A is

also a element of B. The set of all subsets of a set A, is called power set $\mathscr{P}(A)$ of A.[1] We denote by $\mathbb{N}$ the set of natural numbers (i.e. $\{ 0,\ 1,\ 2,\ \ldots \}$).

We end this paragraph with some important concepts in mathematics, *relations* and *functions*.

Definition 1.1.1. *A relation is any subset of a Cartesian product.*

For instance, a subset of $A \times B$, called a "binary relation from A to B", is a collection of ordered pairs (a, b) with first components from A and second components from B. For a binary relation R, we often write aRb to mean that (a, b) is in R.

Definition 1.1.2. *A function is a relation that uniquely associates members of one set with members of another set. More formally, a function from A to B is an object f such that every a in A is uniquely associated with an object $f(a)$ in B.*

A function is therefore a many-to-one (or sometimes one-to-one) relation. The set A of values at which a function is defined is called its domain, while the set B of values that the function can produce is called its range. We write $f : A \to B$ to denote that function f has domain the set A and its range is the set B. We normally denote functions with small latin letters f,g,h etc.
Moreover, if for every two different elements $x_1 \neq x_2$ of A, $f(x_1) \neq f(x_2)$ then we say that f is *injective* or *one-to-one* function. We say that a function f is *surjective* or *onto*, if for every element $y \in B$ there exists an element $x \in A$ such that $f(x) = y$. Finally, we say that a function is *bijective* if it is injective and surjective.

Definition 1.1.3. *Let $f : A \to B$ be a function. The inverse image of a subset D of B is the subset of the domain A defined by $f^{-1}(D) = \{ x \in A \mid f(x) \in D \}$.*

Notice that if f is a bijection there exists a unique inverse function $f^{-1} : B \to A$ of f, where for each x, y such that $f(x) = y$ we have $f^{-1}(y) = x$.

1.1.2 Alphabets, Words and Languages

Now, we will give some basic definitions regarding formal languages.

Definition 1.1.4. *A finite, non empty set $\Sigma = \{a_1, \ldots, a_m\}$ is called alphabet. Normally, we denote an alphabet by a capital greek letter Σ, Γ, etc.*

Definition 1.1.5. *A word (or a string) over an alphabet $\Sigma = \{a_1, \ldots, a_m\}$ is a finite sequence $a_{i_1} \ldots a_{i_l}$ of symbols from Σ, where $l \geq 0$ and $i_1, \ldots, i_l \in \{1, \ldots, m\}$. The number l is called the length of the string and it is denoted by $|a_{i_1} \ldots a_{i_l}|$. The empty string, when $l = 0$, is denoted by ϵ.*

[1] In the literature the power set of A is often denoted by 2^A. Normally, B^A is the set of functions from A to B. The power set denoted some times by 2^A because for every subset of A we have a unique function from A to $\{0, 1\}$ and from such a function we have a unique subset of A.

The set of all strings over an alphabet Σ, is denoted by Σ^*, where $w \in \Sigma^*$ iff there is $m \in \mathbb{N}$ such that $w \in \Sigma^m$.[2] The letters w, u, v, x, y, z and sometimes some others, are used to denote strings. The string x is a *substring* of a string y, if there exist strings u, v such that $y = uxv$, notice that u, or v or both could also be the empty string ϵ.

Definition 1.1.6. *For an alphabet Σ, every subset of Σ^* is called a language over Σ. The set of all languages is thus the set of all subsets of Σ^*, denoted by $\mathscr{P}(\Sigma^*)$.*

The letters L,K,M,N and sometimes some others are used to denote languages. We define an additional operation over languages, the *complement.* The *complement* of a language L denoted by $\overline{L}$ is the language $\Sigma^* - L$. The *concatenation* of two words, u and v, is the word uv, and the *concatenation* of two languages L_1, L_2 is the language $L_1L_2 = \{\, uv \mid u \in L_1 \text{ and } v \in L_2\}$. Given a language L (or an alphabet Σ), we write L_ϵ (or Σ_ϵ) to denote the language $L \cup \{\epsilon\}$ (or the set $\Sigma \cup \{\epsilon\}$). We also define the powers of a word w to be for every $n \geq 0$, $w^n = \underbrace{w \ldots w}_{n}$, where $w^0 = \epsilon$, for every $w \in \Sigma^*$. Moreover, we define the (Kleene) star ‘ $*$ ’ of a language L to be:

$$L^* = \bigcup_{n \in \mathbb{N}} L^n$$

Note that according to this definition we have that $\emptyset^* = \epsilon$. Additionally we define $L^+ = LL^*$.

1.2 Finite Automata

In this section we study simple machines without memory. These theoretical machines are called *finite automata.* We study which languages can be produced by them and the limitations of those machines. Formally we have the definition:

Definition 1.2.1. *A deterministic finite automaton (DFA) is a quintuple $A = (Q, \Sigma, \delta, q_0, T)$, where*

- *Q is a finite non empty set of states.*
- *Σ is an input alphabet.*
- *$\delta : Q \times \Sigma \to Q$ is the transition function.*
- *$q_0 \in Q$ is the initial state.*
- *$T \subseteq Q$ is the set of final states , also called acceptance states.*

The transition function is extended to $\delta^* : Q \times \Sigma^* \to Q$ as follows:

[2]For every alphabet Σ, we denote $\Sigma^0 = \{\epsilon\}$.

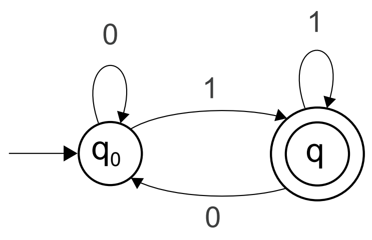

Figure 1.1: State diagram of an automaton.

$$\begin{aligned}\delta^*(q,\epsilon) &= q\\ \delta^*(q,aw) &= \delta^*(\delta(q,a),w), \text{ where } a\in\Sigma \text{ and } w\in\Sigma^*\end{aligned}$$

The language recognized by a DFA A is defined as follows:

$$L(A) = \{\, w \in \Sigma^* \mid \delta^*(q_0, w) \in T\}$$

In order to understand better how an automaton works, we can present it with a state diagram. The states of an automaton are represented with nodes, and the transition function is denoted by named directed edges. Finally, the initial state is denoted by an arrow and when a state is a final one, we denote it by a cycle around it. We give an example in Figure 1.1. We start in the state q_0 and we stay in that state as long as we read 0's, we change state to q when we read a ' 1 '. This automaton produces every word over $\{0,1\}$, which ends with ' 1 '.

Definition 1.2.2. *A non deterministic finite automaton (NFA) is a quintuple $B = (Q, \Sigma, \Delta, q_0, T)$, where Q, Σ, q_0 and T are as above, while $\Delta \subseteq Q \times \Sigma \times Q$ is a relation instead of a function*[3].

The language recognized by a NFA is defined as follows:

$$L(B) = \{\, w \in \Sigma^* \mid w = a_1 \dots a_n, \text{ where each } a_i \in \Sigma, \text{ there are } r_0, \dots, r_n \in Q \text{ where } r_0 = q_0,\ (r_{i-1}, a_i, r_i) \in \Delta \text{ and } r_n \in T\}.$$

It's obvious that a DFA can be regarded as a special case of an NFA. But in the following lemma we will see that they are equivalent in computational power.

[3]We can consider Δ as a function, $\Delta : Q \times \Sigma \to \mathscr{P}(Q)$.

Lemma 1.2.1. *Let $B = (Q, \Sigma, \Delta, q_0, T)$ be a NFA. Then the DFA $A = (\mathscr{P}(Q), \Sigma, \delta, \{q_0\}, T')$ with $\delta : \mathscr{P}(Q) \times \Sigma \to \mathscr{P}(Q)$ and $\delta(C, a) = D$ iff $(C, a, D) \subseteq \Delta$ and finally $T' = \{ C \in \mathscr{P}(Q) \mid C \cap T \neq \emptyset\}$, generates the same language as B.*

Proof. Upon reading a string w, A computes the set of states reached in all possible computations of B on w. □

Definition 1.2.3. *A non deterministic finite automata with ϵ-transitions (ϵ-NFA) is a quintuple $C = (Q, \Sigma, \Delta, q_0, T)$, where Q, Σ, q_0 and T are as above, while $\Delta \subseteq (Q, \Sigma \cup \{\epsilon\}, Q)$ is the transition relation.*

An ϵ-NFA C recognizes the following language:

$$L(C) = \{ w \in \Sigma^* \mid w = u_1 \dots u_n \text{ of each } u_i \in \Sigma \cup \{\epsilon\}, \text{ there are } r_0, \dots, r_n \in Q \text{ where } r_0 = q_0,\ (r_{i-1}, u_i, r_i) \in \Delta \text{ and } r_n \in T\}$$

Lemma 1.2.2. *Let $C = (Q, \Sigma, \Delta, q_0, T)$ be an ϵ-NFA. For every state $q \in Q$, let $Closure(q) \subseteq Q$ be the set of states reachable from q by zero or more ϵ-transitions. Then the NFA $B = (Q, \Delta', q_0, T')$ with $\Delta = \{ (p, a, q) \mid p, q \in Q, a \in \Sigma$, there are $p_1, p_2 \in Q$, $p_1 \in Closure(p)$, $q \in Closure(p_2)$ and $(p_1, a, p_2) \in \Delta\}$ with $T' = \{ q \in Q \mid Closure(q) \cap T \neq \emptyset\}$ generates the same language as C.*

Proof. The transition relation Δ simulates a sequence of ϵ-transitions followed by a transition by an input symbol. The set of acceptance states T represents acceptance in all states, from which one can reach an acceptance state by a sequence of ϵ-transitions. □

From the previous lemmas we have shown that the computational models of DFA, NFA and $\epsilon-$NFA are equivalent in computational power. So, it makes it easy for us to use any of these models, depending of what it's convenient to us in each case.

1.3 Regular Languages

Regular languages are expressions over constant languages $\emptyset$, $\{\epsilon\}$ and $\{a\}$, for all $a \in \Sigma$, connected using union, concatenation and star. We will see in this paragraph that the class of regular languages is equivalent to the class of languages recognized by automata. For further study of automata theory and regular languages see [21, 22, 24, 25, 26]. Formally, regular expressions can be defined as follows:

Definition 1.3.1. *We define regular expressions by induction, where*

- *$\emptyset$ and $\{\epsilon\}$ are regular expressions*
- *$\{a\}$ is a regular expression for every $a \in \Sigma$*
- *if α and β are regular expressions, then $\alpha \cup \beta$, $\alpha\beta$ and α^* are regular expressions too.*

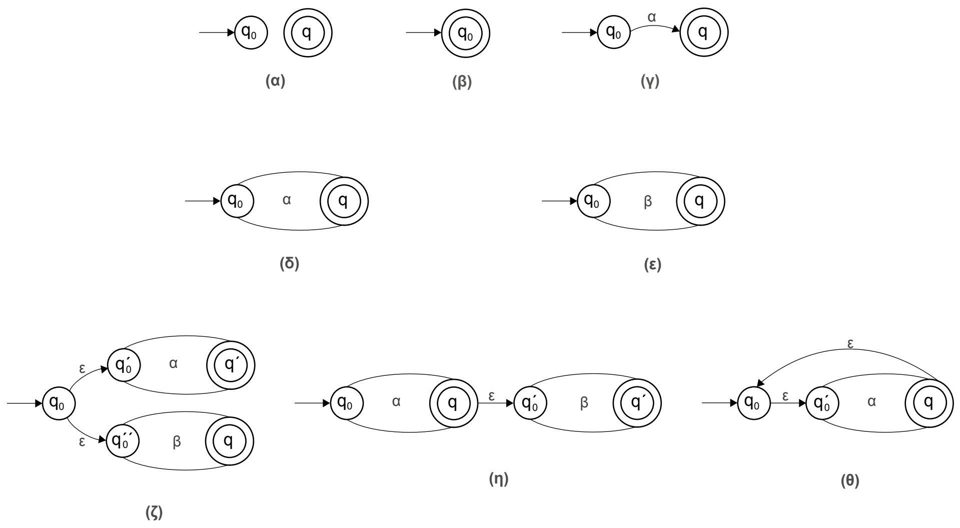

Figure 1.2: Automata for the simplest regular expressions

In order to simplify things, we write a instead of $\{a\}$. Also, the star has the highest priority, followed by concatenation and the union has lowest priority. For example we write $(a \cup bc^*)d$ instead of $(\{a\} \cup (\{b\}(\{c\}^*)))\{d\}$.

Lemma 1.3.1. *For every regular expression there exists an ϵ-NFA recognizing the same language.*

Proof. We prove the lemma with induction on the size of the expression. For every regular expression, an equivalent ϵ-NFA with a unique acceptance state is constructed. Figure 1.2 gives ϵ-NFAs for the three base cases of regular expressions, as well as the three cases of the induction step, where α and β are smaller regular expressions for which ϵ-NFAs exist by the induction hypothesis. □

Before we continue we have to define what is language equations. A language equation is like a numerical equation but the value of the variable is a language. Also, instead of numbers we have known languages and the accepted operations are union, complement, intersection and difference instead of addition, multiplication, abstraction, ect.

Lemma 1.3.2. *For every ϵ-NFA there exists an equivalent regular expression.*

Proof. The basic idea here is that we can see each state of the ϵ-NFA as a variable of a language equation. For every state q we create the variable Ξ_q.

Now, we have the system of language equations, for all $p \in Q$

$$\Xi_p = \bigcup_{\substack{q \in Q \\ \delta(p,a)=q}} a\Xi_q$$

and for terminal stages p we have the equations

$$\Xi_p = (\bigcup_{\substack{q \in Q \\ \delta(p,a)=q}} a\Xi_q) \cup \{\epsilon\}$$

We notice that the least solution[4] to the language equation

$$X = LX \cup M$$

is the language L^*M, in [25] is has a more analytical explanation. From the last equation we can solve the above system of equations and we can also notice that any solution is a regular expression. □

From the lemmas above we have the following theorem.

Theorem 1.3.1. *A language is recognized by a DFA if and only if it is generated by a regular expression.*

The set of languages representable by these equivalent formalisms is known as the family of regular languages, and any such language is called a *regular language.*

1.4 Closure Properties and Pumping Lemma for Regular Languages

Let $\mathscr{L}$ be a family of languages, let $f : \mathscr{P}(\Sigma^*) \times \ldots \times \mathscr{P}(\Sigma^*) \to \mathscr{P}(\Sigma^*)$ be an n-argument function on languages. Then $\mathscr{L}$ is said to be closed under f if for all $L_1, \ldots, L_n \in \mathscr{L}$, we have $f(L_1, \ldots, L_n) \in \mathscr{L}$.

Theorem 1.4.1. *The family of regular languages are closed under the operation of union, complement, intersection, concatenation and star.*

Proof. Closer under union, concatenation and star is immediate by the definition of regular expressions. In order to prove that the family of regular languages is closed under complement we use Theorem 1.3.1. Assume we have a DFA $A = (Q, \Sigma, \Delta, q_0, T)$ which recognizes the regular language L. Then the DFA $A' = (Q, \Sigma, \Delta, q_0, Q - T)$ recognizes $\overline{L}$. For closure under intersection, we use de Morgan's law $K \cap L = \overline{\overline{K} \cup \overline{L}}$ and the closure of regular languages under complementation and union. □

[4]When $\epsilon \notin L$, is the only solution.

Now the question that comes naturally is which languages are not regular. This question is answered with the help of the next theorem.

Theorem 1.4.2 (Pumping Lemma)**.** *For every regular language $L \subseteq \Sigma^*$ there exists a constant $p \geq 1$, such that for every string $w \in L$ with $|w| \geq p$ there exists a factorization $w = xyz$, where y is non empty and $|xy| \leq p$, such that $xy^iz \in L$ for all $i \in \mathbb{N}$.*

Proof. Let $A = (Q, \Sigma, \delta, q_0, T)$ recognizing the language L and $p = |Q|$. Let $w = w_1w_2 \dots w_n$ be a word in L of length n, where $n \geq p$. Let $q_0q_1 \dots q_n$ be the sequence of states that A enters while processing w, so $q_{i+1} = \delta(q_i, w_{i+1})$ for $0 \leq i \leq n-1$. This sequence has length $n+1$, which is at least $p+1$. By the pigeonhole principle, at least one state of A repeats itself in the sequence $q_0q_1 \dots q_n$. We call the first of these q_j and the second q_l. Because q_l occurs among the first $p+1$ places in the sequence, we have $l \leq p+1$. Now let $x = w_1 \dots w_j$, $y = w_{j+1} \dots w_l$ and $z = w_{l+1} \dots w_n$.
As x takes A from q_0 to q_j, y takes A from q_j to q_j and z takes A from q_j to q_n, which is a final state, A must accept xy^iz for $i \geq 0$. We know that $j \neq l$, so $|y| > 0$ and $l \leq p+1$, so $|xy| \leq p$. $\square$

Now, we can see some examples of non-regular languages.

Example 1.4.1. *The language $L = \{a^nb^n | n \in \mathbb{N}\}$ is not regular. From the previous theorem there exists a number $p \geq 1$, such that for every string $w \in L$ with $|w| \geq p$ there exists a factorization $w = xyz$, where y is non empty and $|xy| \leq p$, such that $xy^iz \in L$ for all $i \geq 0$. But $a^pb^p \in L$, so exists m, $0 < m \leq p$, $y = a^m$, and $a^{p-m}b^p \in L$. This is a contradiction, because $p - m \neq p$.*

Example 1.4.2. *The language $L = \{a^{4^n} | n \in \mathbb{N}\}$ is not regular. Again, from the previous theorem for every $w \in L$ there exist $x, y, z \in a^*$, with $y \neq \epsilon$, which $w = xyz$, such that $x = a^{m_1}$, $y = a^{m_2}$ and $z = a^{m_3}$, then for all $k \in \mathbb{N}$, we have that $a^{m_1+km_2+m_3} \in L$, for every $k \in \mathbb{N}$, there is a sequence $x_k \geq 0$, such that $4^{x_k} = m_1 + km_2 + m_3$.*
Let $p = m_1 + m_3$ and $q = m_2$, so we have for every $k \in \mathbb{N}$, that $p + kq = 4^{x_k}$. This implies that for every $k \in \mathbb{N}$, we have $x_{k+1} > x_k$, so $x_{k+1} \geq x_k + 1$, then $4^{x_{k+1}} \geq 4^{x_k+1} \Rightarrow 3 \times 4^{x_k} \leq 4^{x_{k+1}} - 4^{x_k} = p + (k+1)q - p - kq$ so we have that for every $k \in \mathbb{N}$, it would be $3 \times 4^{x_k} \leq q$ which is a contradiction, because we have for every $k \in \mathbb{N}$, that $x_{k+1} > x_k$.

Chapter 2

Context-Free Grammars

2.1 Basic Notions

In this paragraph we will give the well studied concept of context-free grammars. For further study of context-free grammars see [21, 22, 23, 24, 25, 26].

Definition 2.1.1. *A* context-free grammar *is a quadruple* $G = (\Sigma, N, P, S)$ *, in which:*

- Σ *and* N *are disjoint finite non-empty sets of* terminal *and* non-terminal *symbols, respectively*
- P *is a finite set of grammar rules, each of the form* $A \to u$*, with* $A \in N$ *and* $u \in (\Sigma \cup N)^*$
- $S \in N$ *is a non-terminal designated as the start symbol*

Multiple rules for a single non-terminal are often written using the notation

$$A \to \alpha_1 \,|\, \ldots \,|\, \alpha_m$$

in which the vertical line is, in essence, disjunction.

One approach to defining the semantics of context-free grammars is rewriting of strings over $(\Sigma \cup N)^*$.

Definition 2.1.2. *Let* $G = (\Sigma, N, P, S)$ *be a context-free grammar. Define a relation '*$\Rightarrow$*' of one-step derivability on* $(\Sigma \cup N)^*$ *as follows:*

$$s_1 A s_2 \Rightarrow s_1 \alpha s_2, \textit{ for all } A \to \alpha \in P \textit{ and } s_1, s_2 \in (\Sigma \cup N)^*$$

Let $\Rightarrow^*$ *be the reflexive and transitive closure of* $\Rightarrow$*. For every string* $\alpha \in (\Sigma \cup N)^*$*, we define the set of strings over* Σ *derivable from* α*:*

$$L_G(\alpha) = \{\, w \in \Sigma^* \mid \alpha \Rightarrow^* w\}$$

The language generated by the grammar is the set of strings derivable from S:

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

A language L is context-free if there exists a context-free grammar G, where $L = L(G)$.

Another equivalent definition of the semantics of context-free grammars is given by language equations:

Definition 2.1.3. *Let $G = (\Sigma, N, P, S)$ be a context-free grammar, let $N = \{A_1, \dots, A_n\}$ and consider the associated system of n equations, in which the unknowns $\{A_1, \dots, A_n\}$ assume values of languages over Σ:*

$$A_i = \bigcup_{A_i \to s_1 \dots s_l \in P} (s_1 \dots s_l), \quad \textit{where} \quad 1 \leq i \leq n$$

and where each symbol $s_t = A_j \in N$ in the right-hand side of the equation represents a variable, while each symbol $s_t = a \in \Sigma$ represents a constant language $\{a\}$. Let $(L_1, \dots, L_n)$ be the least solution of the previous system, that is, a component wise intersection of all the solutions of the system. Then the language generated by each non-terminal A_i is defined as $L_G(A_i) = L_i$.

The correctness of this definition requires a proof: it should be shown that any system of language equations associated to a context-free grammar has a solution, and that the component wise intersection of all solutions is a solution as well. These properties of language equations we are going to establish right now.
Let Σ be an alphabet, let $n \geq 1$. Define a partial order $\sqsubseteq$ on the set $(\mathscr{P}(\Sigma^*))^n$ of vectors of n languages as $(K_1, \dots, K_n) \sqsubseteq (L_1, \dots, L_n)$ if and only if $K_i \subseteq L_i$. The least element is $\perp = (\emptyset, \dots, \emptyset)$.
Let $X_i = \phi_i(X_1, \dots, X_n)$ $(1 \leq i \leq n)$ be a system of language equations, where $\phi_i : (\mathscr{P}(\Sigma^*))^n \to \mathscr{P}(\Sigma^*)$ are any functions on languages. Let $\phi = (\phi_1, \dots, \phi_n)$ be a vector function representing the right-hand side of the system. Assume that ϕ has the following two properties:

- ϕ is *monotone*, in the sense that for any two vectors K and L, the inequality $K \sqsubseteq L$ implies $\phi(K) \sqsubseteq \phi(L)$.
- ϕ is $\bigcup$-continuous, in the sense that for every sequence of vectors of languages $\{L^{(i)}\}_{i=1}^{\infty}$ it holds that

$$\bigcup_{i=1}^{\infty} \phi(L^{(i)}) = \phi\Big(\bigcup_{i=1}^{\infty} L^{(i)}\Big).$$

We have the following lemmas.

Lemma 2.1.1. *If ϕ is monotone and $\bigcup$-continuous, then the least solution of a system $X = \phi(X)$ is the vector $\bigcup_{k=0}^{\infty} \phi^k(\perp)$*

Lemma 2.1.2. *If each ϕ_i is a composition of variables and any constant languages using union and concatenation, then ϕ is monotone and $\bigcup$-continuous.*

The above lemma, also holds if ϕ_i contains intersections and Kleene stars. The following theorem shows that the previous definitions of context-free grammars are equivalent.

Theorem 2.1.1. *Let $G = (\Sigma, N, P, S)$ be a context-free grammar and let $X = \phi(X)$ be the associated system of language equations. Then, for every $A_i \in N$ and $w \in \Sigma^*$, $A_i \Rightarrow^* w$ if and only if $w \in [\bigcup_{k\geq 0} \phi^k(\perp)]_i$, where $[\Gamma]_i$ denotes the i-th component of the vector Γ.*

Proof. (Outline) To show the "only if" part, we perform induction on the length of the derivation of w. For every w derivable in l steps from A_i it is shown that $w \in [\phi^k(\perp)]_i$ for some $k(l)$.
To show the "if" part, we perform induction on the number of steps in the iteration. For every $w \in [\phi^k(\perp)]_i$ it is shown that w is derivable in l steps from A_i for some $l(k)$. □

Let us see some examples of context-free languages.

Example 2.1.1. *The language $\{\, a^n b^n \mid n \geq 0\}$ is generated by the grammar $S \to aSb \mid \epsilon$.*

Example 2.1.2. *The language of balanced parentheses is generated by the grammar $S \to SS \mid (S) \mid \epsilon$.*

Example 2.1.3. *The language of even-length palindromes $\{\, ww^R \mid w \in \{a,b\}^*\}$ is generated by the grammar $S \to aSa|bSb|\epsilon$.*

Example 2.1.4. *Let $\Sigma = \{a, b\}$. The language $\overline{\{\, ww \mid w \in \{a,b\}^*\}}$ is generated by the grammar*

$$\begin{array}{rcl} S & \to & AB \mid BA \mid A \mid B \\ A & \to & XAX \mid a \\ B & \to & XBX \mid b \\ X & \to & a \mid b \end{array}$$

Theorem 2.1.2. *The intersection of a regular language R with a context-free language L is context-free language.*

Proof. Since R is regular, there exists an automaton $A = (Q, \Sigma, \delta, q_0, T)$ which recognizes it. Also, since L is context-free, there exists a context-free grammar $G = (\Sigma, N, P, S)$, such that $L = L(G)$. Our goal is to creat a context-free grammar G' which generates $L \cap R$. The set of non-terminals of this grammar will be $(Q \times N \times Q) \cup \{\overline{S}\}$, where $\overline{S}$ is a new non-terminal. The rules of the grammar G' will be of the form[1]:

[1]We can assume that the rules of a grammar are of the form $A \to BC$, $A \to a$ or $A \to \epsilon$, where A, B, C are non-terminals and a a terminal. We can assume that because we can replace every rule $A \to C$ with the rules $A \to EC$ and $E \to \epsilon$, where E is a new non-terminal. Every rule $A \to ubu'$, where $u, u' \in (\Sigma \cup N)^*$ and $b \in \Sigma$, with the rules $A \to uBu'$ and $B \to b$, where B is a new non-terminal. Every rule $A \to u$, where $u \in N^*$ and $|u| > 2$, with the rules $A \to BC$ and $C \to u'$, when $u = Bu'$.

$$\begin{array}{rcll} (q,A,q') & \rightarrow & (q,B,q_1)(q_1,C,q') & \text{where } A \rightarrow BC \in P \text{ and } q,q_1,q' \in Q \\ (q,A,q') & \rightarrow & a & \text{where } A \rightarrow a \in P,\ q,q' \in Q \text{ and } \delta(q,a) = q' \\ (q,A,q) & \rightarrow & \epsilon & \text{where } A \rightarrow \epsilon \in P \text{ and } q \in Q \end{array}$$

Now, with induction we can show that $(q,A,q') \Rightarrow^*_{G'} w$ iff $A \Rightarrow^*_G w$ and $\delta^*(q,w) = q'$. We end the proof by adding the rules of the initial symbol $\overline{S}$ of the grammar G'. These rules will be of the form:

$$\overline{S} \longrightarrow (q_0, S, q_t), \text{ where } q_t \in T$$

□

Now, we will see a special case of context-free grammars, which are called *right regular grammar.*

Definition 2.1.4. *A context-free grammar $G = (\Sigma, N, P, S)$, is called it right regular if all the rules in P are of the form:*

$$\begin{array}{rcl} A & \rightarrow & aB \\ A & \rightarrow & a \\ A & \rightarrow & \epsilon \end{array}$$

where $a \in \Sigma$ and $A, B \in N$.

Theorem 2.1.3. *A language is regular if and only if it is generated by a right regular grammar.*

Proof. If a language L is regular, then there is an automaton $A = (Q, \Sigma, \delta, q_0, T)$ which recognizes L. For all the transitions $\delta(q,a) = q'$, we add the rule $A_q \rightarrow aA_{q'}$. We also add the rule $A_q \rightarrow \epsilon$, if $q \in T$. It is easy to show that $\delta^*(q_0, w) \in T \Leftrightarrow A_{q_0} \Rightarrow^* w$.
On the other direction, if the language L is generated by a right regular grammar $G = (\Sigma, N, \delta, S)$, then we create the NFA where $Q = N \cup \{F\}$, for a symbol $F \notin N$. For every rule of the form $A \rightarrow aB$, we take the transition $\delta(A,a) = B$. For every rule of the form $A \rightarrow a$, where $a \in \Sigma_\epsilon$, we take the transition $\delta(A,a) = F$. Finally, $T = \{F\}$ and S is the initial state. Then it is easy to show that the grammar and the NFA are equivalence. □

Corollary 2.1.1. *If L is a regular language, then L is context-free.*

Often we represent a rule

$$A \rightarrow t_1 \ldots t_n, \text{ where } A \in N \text{ and } t_i \in \Sigma \cup N \cup \{\epsilon\}$$

by a tree called *parse tree*, as it's showed in Figure 2.1(α). In this tree, every node which has children is named with a non-terminal. Every leaf can be a node whose name is a terminal or a non-terminal. A parse tree conveys the meaning of a string according to the grammar. It can also be seen as a snapshot of a derivation of the grammar. For a grammar G and $w \in L(G)$, we say that a parse tree yields w if the root of the tree is the grammar's initial symbol and

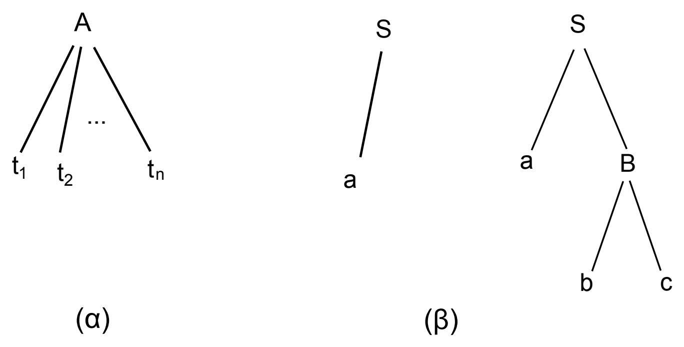

Figure 2.1: Parse trees

the concatenation leaves yields the word w. We give en example of two parse trees for the grammar $S \to aB$, $B \to cb \,|\, \epsilon$ in Figure 2.1(β). A context-free grammar G is called *unambiguous* if every string $w \in L(G)$ has a unique parse tree. It's easy to see the following proposition.

Proposition 2.1.1. *A word $w \in L(G)$ if and only if there is a parse tree with yield w.*

Example 2.1.5. *The following context-free grammar generates the language $\{\, a^k b^l c^m \mid k = l \text{ or } l = m\}$:*

$$\begin{array}{lcl} S & \to & AB \,|\, DC \\ A & \to & aA \,|\, \epsilon \\ B & \to & bBc \,|\, \epsilon \\ C & \to & cC \,|\, \epsilon \\ D & \to & aDb \,|\, \epsilon \end{array}$$

This grammar is ambiguous because every string of the form $a^n b^n c^n$ can be obtained both using the rule $S \to AB$ and using the rule $S \to DC$.

2.2 Pumping Lemma for Context-Free Grammars

In this section we answer the natural question of which are the limitations of context-free languages. It is the corresponding lemma for the regular languages, as we have seen in Theorem 1.4.2. Also, we give some examples of non context-free languages. We can answer these questions with the following theorem.

Theorem 2.2.1 (Pumping lemma for context-free languages)**.** *For every context-free language $L \subseteq \Sigma^*$ there exists a constant $p \geq 1$, such that for every string $w \in L$ with $|w| \geq p$ there exists a factorization $w = xuyvz$, where $|uv| > 0$ and $|uyv| \leq p$, such that $xu^iyv^iz \in L$ for every $i \in \mathbb{N}$.*

Proof. Let $G = (\Sigma, N, P, S)$ be a context-free grammar generating L. Let $m = max_{(A \to \alpha \in P)}|\alpha|$ and define $p = m^{|N|} + 1$. Consider any string $w \in L$ of length at least p and consider its parse tree. Let an internal node s in the tree be called *non-trivial* if the partition of its subtree induced by its sons non trivially divides the terminal leaves (that is, it is not the case that one of the sons has all terminal leaves and the others have none).
Then the parse tree should contain a path with at least $|N| + 1$ non trivial nodes, some non-terminal A must repeat twice in this path and the section of the tree between these two instances of A can be repeated 0 or more times, thus obtainning parse trees of xu^iyv^iz. This section of the tree represents a derivation of uAv from A. □

Example 2.2.1. *The language $L = \{a^nb^nc^n | n \geq 0\}$ is not context-free.*

Proof. Suppose that it is context-free and let $p \geq 1$ be the constant given by the pumping lemma. Consider $w = a^pb^pc^p$. Then there exists a factorization $w = xuyvz$. There are several cases:

- Either u or v is not in $a^* \cup b^* \cup c^*$, that is, the string spans over the boundary between as, bs and cs. Then $xu^2yv^2z \notin a^*b^*c^*$ and cannot be in L.
- If $u, v \in a^*$, then $xyz = a^{p-|uv|}b^pc^p \notin L$. The cases of $u, v \in b^*$ and $u, v \in c^*$ are similar.
- If $u \in a^*$ and $v \in b^*$, then $xu^0yv^0z = a^{p-|u|}b^{p-|v|}c^p \notin L$. The cases of $u \in a^*$, $v \in c^*$ and $u \in b^*$, $v \in c^*$ are similar.

In each case a contradiction is obtainned. □

In the rest of the paragraph we will study the closure properties of context-free languages.

Proposition 2.2.1. *The context-free languages are closed under union, concatenation and star.*

Proof. If $G_i = (\Sigma, N_i, P_i, S_i)$ with $i \in \{1, 2\}$ and $N_1 \cap N_2 = \emptyset$ are context-free grammars. Then the grammars $(\Sigma, N_1 \cup N_2 \cup \{S\}, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S)$ and $(\Sigma, N_1 \cup N_2 \cup \{S\}, P_1 \cup P_2 \cup \{S \to S_1S_2\}, S)$ generate $L(G_1) \cup L(G_2)$ and $L(G_1)L(G_2)$, respectively, while the grammar $(\Sigma, N_1 \cup \{S\}, P_1 \cup \{S \to S_1S, S \to \epsilon\}, S)$ generates $L(G_1)^*$. The correctness of all constructions is easily proved using language equations. □

Proposition 2.2.2. *The context-free languages are not closed under intersection and complementation.*

Proof. Consider the following two grammars, which generate the languages $\{\, a^i b^l c^l \mid i, l \geq 0\}$ and $\{\, a^m b^m c^j \mid j, m \geq 0\}$, respectively:

$$\begin{array}{rcl} S_1 & \to & aS_1 \,|\, A \\ A & \to & bAc \,|\, \epsilon \\ \\ S_2 & \to & S_2c \,|\, B \\ B & \to & aBb \,|\, \epsilon \end{array}$$

Suppose context-free languages are closed under intersection. Then the language $\{\, a^i b^l c^l \mid i, l \geq 0\} \cap \{\, a^m b^m c^j \mid j, m \geq 0\} = \{\, a^n b^n c^n \mid n \geq 0\}$ should be context-free as well, which contradicts Example 2.2.1.
Consider the following grammar generating the language $\{\, a^k b^l c^m \mid k \neq l$ or $l \neq m\}$

$$\begin{array}{rcl} S & \to & AE \,|\, DC \\ A & \to & aA \,|\, \epsilon \\ B & \to & bB \,|\, \epsilon \\ C & \to & cC \,|\, \epsilon \\ D & \to & aDb \,|\, aA \,|\, bB \\ E & \to & bEc \,|\, bB \,|\, cC \end{array}$$

The language

$$L = \{a^k b^l c^m | k \neq l \text{ or } l \neq m\} \cup \overline{a^* b^* c^*} = \overline{\{a^n b^n c^n | n \geq 0\}}$$

is context-free as well, as a union of two context-free languages. Then, if context-free languages were closed under complementation, $\overline{L}$ should be context-free as well, which is known to be untrue. □

Though the intersection of two context-free languages is not necessarily context-free, it is context-free in case one of these languages is regular as we have seen in Theorem 2.1.2.
One more very simple closure result. Let Σ and Γ be alphabets. A mapping $h : \Sigma^* \to \Gamma^*$ is called a *homomorphism* if $h(uv) = h(u)h(v)$ for all $u, v \in \Sigma^*$. This definition, in particular, implies that $h(\epsilon) = \epsilon$, and that a homomorphism is completely defined by the images of all symbols from Σ.

Proposition 2.2.3. *Let Σ and Γ be alphabets, let $h : \Sigma^* \to \Gamma^*$ be a homomorphism and let $G = (\Sigma, N, P, S)$ be any context-free grammar. Define $h(A) = A$ for all $A \in N$. Then the grammar $G' = (\Gamma, N, P', S)$ with $P' = \{A \to h(\alpha) | A \to \alpha \in P\}$ generates the language $h(L(G))$.*

2.3 Special Cases

In this section we study the power of context-free languages when we have bounded resources. The resources of a context-free language are the number of symbols or terminals. Let us see some examples of this.

Lemma 2.3.1. *The context-free language $a^* \cup b^*$ can not be produced by a context-free grammars with only one symbol.*

Proof. We assume that there exists a context-free grammar $G = (\{a, b\}, \{S\}, P, S)$ which produces $a^* \cup b^*$. Let m be the maximum number of symbols and terminals that we have in the right part of all rules in P. Then we have that $a^{m+1} \in L$, so $S \Rightarrow^* a^{m+1}$. We also have the number of computation steps are greater than one, because of how we choose m. Then it is true that we have $u, v \in \{a, b, S\}^*$, with $|uv| > 0$, such that $S \Rightarrow^* uSv \Rightarrow^* a^{m+1}$. But we reach to contradiction, because since we also have that $b \in L(S)$, then we should also have that $a^k b a^l \in L(S)$, for some k, l where $k + l < m + 1$. □

In the previous example we saw a language which can not be produced by any context-free grammar with only one non-terminal. It comes naturally the question, do we have a proposition which implies such negatives results. The answer is the next proposition, which we will also use in the next chapter.

Proposition 2.3.1. *Let $G = (\Sigma, N, P, S)$ be a context-free grammar. We define the set $U = \{u \in \Sigma^* \mid A \to u \in P,$ with $A \in N\}$. Then for all $w \in L(G)$, there exists an element of U which is a substring of w.*

The validation of the previous proposition is very easy, we let it to the reader (with induction on the number of steps of the computation). As an easy result of the previous proposition we have that the language $a^+b^+c^+$ can not be produced by a context-free grammar with only one variable.
Another well known result for context-free languages, over unary alphabets, is the following proposition.

Proposition 2.3.2. *Every context-free language over a unary alphabet is regular.*

Proof. Consider an infinite context-free language L over a unary alphabet, $\{a\}$ (the proof is immediate if L is finite). Let n be the pumping lemma constant for L. From the pumping lemma, we have that for every $w \in L$, with $|w| \geq n$, there exists a factorization of w, $w = xuyvz$, with $|uv| > 0$ and $|uyv| \leq n$ such that for every $i \geq 0$, $xu^i yv^i z \in L$. Since L is over an unary alphabet, then there exist numbers $p, p_1, p_2, q_1, q_2 \in \mathbb{N}$ such that $x = a^{p_1}, y = a^p, z = a^{p_2}, u = a^{q_1}$ and $v = a^{q_2}$. So, from the pumping lemma we have $a^{(p+p_1+p_2)+i(q_1+q_2)} \in L$ for every $i \in \mathbb{N}$, which is also a regular expression.
We have seen so far, that for every word greater than n we have a set of the form $\{a^{p+iq} \mid i \geq 0\}$ for some specific p, q. This set is a subset of L and we also have that q is smaller than n, but we don't have a bound for p. We have to notice that if $p = kq + u$, for some positive integers k, u, it holds

$$\{a^{p+iq} \mid i \geq 0\} = \{a^{u+iq} \mid i \geq k\}$$

From the first step of this proof we have that L consists of perhaps some words of length less that n plus a union of sets of the form $\{a^{p+iq} \mid i \geq 0\}$. From the last remark, it is easy to see that the union of the previous sets is finite, since we have finite pairs of (p, q) with $p \leq q \leq n$. If we want to be more clear, we also have to notice that $\{a^{p+iq} \mid i \geq l\} \subseteq \{a^{p+iq} \mid i \geq k\}$, for $k \leq l$. This finite union is regular expression, so L is regular. $\square$

2.4 Pushdown Automata

In this section we introduce another computational model, called *pushdown automata* (PDA). This is a generalization of automata which we already know. These new automata are like finite automata but they have an extra component, called *stack*. The stack provides additional memory and it gives more "computational power" than an ordinary automaton. We will also see, that the class of languages accepted by pushdown automata is equivalent to the class of context-free languages. Here, we give the definition of pushdown automata as in [21, 24]. In [22] a different but equivalent definition for PDAs is given.

2.4.1 Definition

Definition 2.4.1. *A pushdown automaton (PDA) is a six-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ and F are all finite sets, and*

- *Q is a set of states*
- *Σ is the input alphabet*
- *Γ is the stack alphabet*
- *$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathscr{P}(Q \times \Gamma_\epsilon)$ is the transition function*[2]
- *$q_0 \in Q$ is the initial state*
- *$F \subseteq Q$ is the set of accept states, or final states.*

As we can see, the definition of a pushdown automaton is similar to that of a finite automaton, except for the stack, which we can find in the transition function. The alphabet of the pushdown automaton is Σ and the stack's alphabet is Γ.
In order to understand how the pushdown automaton works, we have to understand its transition function. The domain of the function is $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$. Thus the current state, next input symbol read, and top symbol of the stack determine the next move of a pushdown automaton. Either symbol may be ϵ, causing the machine to move without reading a symbol from the input or without reading a symbol from the stack. We also see from the definition of transition function that we allow non determinism in this model.
A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. It accepts an input w when: $w = w_1 w_2 \ldots w_m$, where each $w_i \in \Sigma_\epsilon$ and sequences of

[2]Recall that $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$

states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ exist such that to satisfy the following three conditions. The string s_i represent the sequence of stack contents that M has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = \epsilon$. This condition signifies that M starts out properly in the start state and with an empty stack.
2. For $i = 0, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack and next input symbol.
3. $r_m \in F$. This condition states that an acceptance state occurs at the input end.

More formally we can define the *configuration* of a PDA. A configuration is a snapshot of a computation of the PDA. A configuration is a triple (q, w, u), where $q \in Q$ is a state, $w \in \Sigma^*$ is the remaining input to be read and $u \in \Gamma^*$ is the stack content. We denote that a configuration (q, w, u) goes in one step to the configuration (q', w', u') by $(q, w, u) \rhd (q', w', u')$. So, we could say that an accepting computation on input w in terms of configuration, is the one which starts with (q_0, w, ϵ), every computation step is of the form $(p, aw', bu') \rhd (q, w', cu')$ if $(q, c) \in \delta(p, a, b)$, where $a \in \Sigma_\epsilon$, $b, c \in \Gamma_\epsilon$, $w' \in \Sigma^*$ and $u' \in \Gamma^*$ and ends with a configuration (p, ϵ, u'') where p is an acceptance state. Now, lets see some examples to help us understand PDAs better.

Example 2.4.1. *In this example, we will give the formal description of the PDA that recognizes the language* $\{\, 0^n 1^n \mid n \geq 0\}$. *Let* M_1 *be* $(Q, \Sigma, \Gamma, \delta, q_0, F)$, *where*
$Q = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{0, 1\}$
$\Gamma = \{0, \#\}$
$F = \{q_0, q_3\}$, *and*
δ *is given by the following table:*

Input:	*0*			*1*			ϵ		
Stack:	*0*	#	ϵ	*0*	#	ϵ	*0*	#	ϵ
q_0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{(q_1, \#)\}$
q_1	$\emptyset$	$\emptyset$	$\{(q_1, 0)\}$	$\{(q_2, \epsilon)\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{(q_2, \epsilon)\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{(q_3, \epsilon)\}$	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

We can also use a state diagram to describe a PDA. The state diagram of a PDA is similar with a finite state automaton diagram, the only difference being in the representation of the transition function. Instead of naming the directed edge only with the input symbol, we name it with the form of "$a, b \to c$". We write it, to signify that when the machine is reading an a from the input it may replace the symbol b on the top of the stack with a c. Any of them can be the symbol ϵ. If a is ϵ, the machine may make this transition without reading any

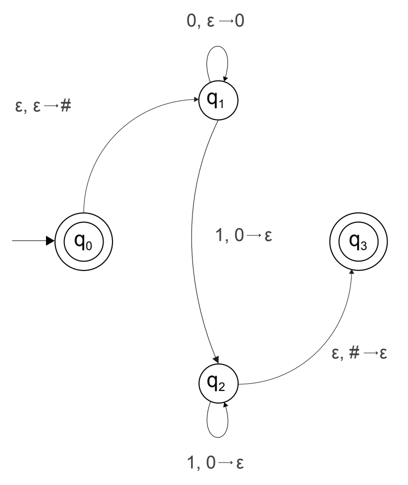

Figure 2.2: State diagram for PDA M_1 that recognizes $\{\, 0^n1^n \mid n \geq 0\}$

symbol from the input. If b is ϵ, the machine may make this transition without reading or replacing any symbol from the stack. If c is ϵ, the machine does not write any symbol on the stack when going along this transition. We can see an example of the above in the state diagram of the Example 2.4.1, in Figure 2.2. The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack. In the previous example we saw a 'trick', which allows us to do that test. Initially, we place a special symbol '#' on the stack, then if we ever see that # again, we know that the stack is empty. Subsequently, when we refer to testing for an empty stack in an informal description of a PDA, we implement the procedure which we just described and saw in the previous example. Additionally, we have another mechanism to test if we have reached the end of the input word. This mechanism is able to achieve this effect by not letting any transition from the acceptance states. Then, it accepts only if we have reached at the end of the input word at the same time that we have reach an acceptance state.
Now, we are going to see one more example, in which we need to use nondeterminism.

Example 2.4.2. *In this example, we will see the PDA that recognizes the language* $\{\, ww^R \mid w \in \{0,1\}^*\}$[3]*. Let* $M_2 = (Q, \Sigma, \Gamma, \delta, q_0, F)$*, where* $Q = \{q_0, q_1, q_2, q_3\}$*,* $\Sigma = \{0,1\}$*,* $\Gamma = \{0,1,\#\}$ *and* $F = \{q_0, q_3\}$*.*

The informal description of the PDA is that it begins by putting the symbols which are read onto the stack. At each point non deterministically, it guesses whether the middle of the string has been reached. If its guess is a 'yes', it

[3]Recall that w^R means w written in reverse order.

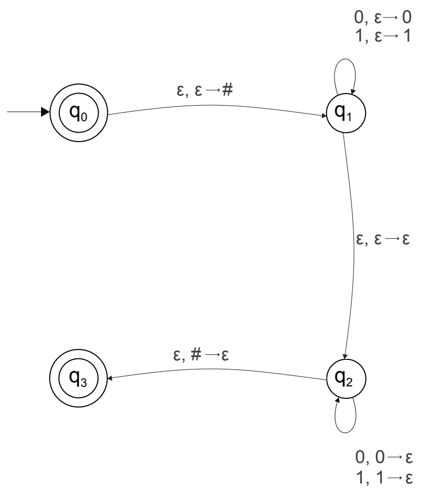

Figure 2.3: State diagram for PDA M_2 that recognizes $\{\, ww^R \mid w \in \{0,1\}^* \}$

changes state into popping off the stack for each symbol read, checking to see that they are the same. If always the same symbol is found and the stack empties at the same time as the input is finished, it accepts; otherwise it rejects. We can see the state diagram of this PDA in Figure 2.3*.*

2.4.2 Equivalence with context-free grammars

In this paragraph we show that context-free grammars and pushdown automata are equivalent in power. Both of them are capable of describing the class of context-free languages. We will show how to convert any context-free grammar into a pushdown automaton that recognizes the same language and vice versa. Recall that we defined a context-free language to be any language that can be described with a context-free grammar. Our goal in this paragraph is to show the following theorem.

Theorem 2.4.1. *A language is context-free iff some pushdown automaton recognizes it.*

In order to simplify things, we split the previous theorem into the two following propositions.

Proposition 2.4.1. *If a language is context-free, then some pushdown automaton recognizes it.*

Proof. Let L be a context-free language, then we know that L has a context-free grammar G which generates it. We will convert the context-free grammar G

into a pushdown automaton P.
The PDA P will accept an input w, if and only if the grammar G generates it. It will follow the grammar's rules. In order to do that, we will use the PDA's stack to store the substitutions made as the grammar G generates a string. These choices will be made from PDA the P non deterministically.
Lets give some more details. The following is an informal description of P.

1. We start with putting into the stack the symbol # and then G's start symbol S. So, we will have into the stack '$S\#$'.
2. Repeat the following steps forever.
   (a) If the top of the stack is a variable symbol A, non deterministically select one of the rules for A and replace A by the string on the right-hand side of the rule.
   (b) If the top of the stack is a terminal symbol α, read the next symbol from the input and compare it to α. If the symbols on stack and in input are the same, go to the next input symbol and delite the one of the stack. If they are not the same, reject on this branch of the nondeterminism.
   (c) If the top of the stack is the symbol #, enter the accept state. Doing so accepts the input if it has all been read.

We see that in the beginning of the simulation of the grammar G from PDA P, we put into the stack the mark of the stack to know when the stack will be empty. Then we put into the stack the start symbol of the grammar to simulate the initial state of the grammar. The directions given in 2 are how the simulation works. When the top symbol of the stack is a variable, the things are simple, we just replace that variable by the right-hand side of a rule. That rule, which has that variable as its head, will be chosen non-deterministically. In other words, the PDA P will guess the rule which the grammar uses to do its one step derivation. As we can know only the top symbol of the stack, it will be difficult to do substitutions with variables which are not in the top. We solve this problem by comparing at once the terminal symbol which appears on the top of the stack, with the following symbol of the input. As we can see, the PDA P will accept its input when we will have into the stack the input word followed by #, and only then. By the description of P, we see that the only words which can be created into the stack are the ones that can be produced by the grammar. Then, the language that P recognizes is the same as the language that G generates.
Someone may argue about how can we put a long string into a stack in one step of the machine, instead of just a symbol. This is simple, since we can simulate this action by introducing additional states to write the string one symbol at a time. For example, let q and r be states of the PDA and let $\alpha \in \Sigma_\epsilon$ and $s \in \Gamma_\epsilon$. Say that we want the PDA to go from q to r when it reads α and pops s. Furthermore we want it to push the entire string $u = u_1 \ldots u_k$ into

the stack at the same time. We can implement this action by introducing new states $q_1 \dots q_{k-1}$ and setting the transition function as follows:

$$\begin{array}{ccc} (q_1, u_k) & \in & \delta(q, \alpha, s) \\ \delta(q_1, \epsilon, \epsilon) & = & \{(q_2, u_{k-1})\} \\ \delta(q_2, \epsilon, \epsilon) & = & \{(q_3, u_{k-2})\} \\ & \vdots & \\ \delta(q_{k-1}, \epsilon, \epsilon) & = & \{(r, u_1)\} \end{array}$$

The PDA P has three basic states, q_{start}, q_{loop} and q_{accept}, and other peripheral ones. We use these states to put into the stack the strings as we described earlier. The initial state of the machine is q_{start}, from which the machine goes to q_{loop} as it places $S\#$ into the stack. It stays to that state until the top symbol of the machine will be # and goes to q_{accept}. Then the machine accepts only when the input word agrees with the stack word. $\square$

Now we prove the other direction of Theorem 2.4.1. We just showed how to convert a context-free grammar into a PDA. We will show now, how to convert a PDA into a context-free grammar. This task is a bit more difficult.

Proposition 2.4.2. *If a pushdown automaton recognizes a language, then it is context-free.*

Proof. In this proof we need to make a context-free grammar G, which will generate the same language with a given pushdown automaton P. This means, that we want to show that for a word w which the pushdown automaton P accepts, to those it can be produced by the grammar G.
In order to do that, we will prove something more general. We make a grammar G, which for every two states p, q has a variable A_{pq}. This variable will generate all the strings that can take P from state p with an empty stack to state q with an empty stack. We have to notice that A_{pq} can also generate strings that take P from p to q, regardless of the stack contents at p, leaving the stack at q in the same condition as it was at p.We need to simplify our task by modifying P. We need P to satisfy the three following conditions:

- It has a single accept state, 'q_{accept}'.
- It empties its stack before accepting.
- Every transition either pushes a symbol into the stack or pops one off the stack, but it doesn't do both at the same time.

Since we have a non deterministic machine, we can easily satisfy the first two conditions. In the first one, we can put a new state q_{accept} which will be the only terminal state. We change the transition function and from the old terminal states we go to the new one without reading something from the input or from the stack. In the second one, we do the 'trick' to test if the stack is empty as we have already seen. We modify the transition function just before to accept and it goes to a state that empties the stack. Then, it goes to the accept state.

We satisfy the third condition by replacing each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state. We also replace each transition that neither pops nor pushes with a transition sequence that pushes then pops an arbitrary stack symbol. So, we can assume that the pushdown automaton P is of the form $(Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$. For all strings that take P from p to q, starting and ending with an empty stack, P's first move is to push a symbol into the stack. P's last move is to pop a symbol off the stack, because in every move of P either it pushes or it pops and it ends up with an empty stack. We have two possibilities during that computation. Either the stack isn't empty for the whole computation, or it empties at a point. In the first case, the symbol that was pushed into the stack at the beginning is the symbol that was popped at the end. We simulate that possibility with the rule $A_{pq} \to aA_{rs}b$, where a is the input read at the first move, b is the input read at the last move, r is the next state that P goes after p when it reads the input with an empty stack and s is the previous state before q. In the other case, the stack empties at a point. Let r be the state which P is when it empties its stack. We simulate this possibility with the rule $A_{pq} \to A_{pr}A_{rq}$.
The set of G's variables is $\{\, A_{pq} \mid (p, q)$ is a pair of P's states $\}$. The start symbol of G is $A_{q_0,q_{accept}}$. We make G's as follows:

- For every $p, q, r, s \in Q$, $t \in \Gamma$ and $a, b \in \Sigma_\epsilon$, if $(r, t) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, t)$, we take the rule $A_{pq} \to aA_{rs}b$.
- For every $p, q, r \in Q$, we take the rule $A_{pq} \to A_{pr}A_{rq}$.
- For every $p \in Q$, we take the rule $A_{pp} \to \epsilon$.

The only thing that we still have to do, is to explain why our construction works. We do that by proving the following two sentences, 'If A_{pq} generates x, then x can bring P from p with an empty stack to q with an empty stack' and 'If x can bring P from p with an empty stack to q with an empty stack, then A_{pq} generates x'. We prove both sentences with induction on the number of steps in the derivation of x from A_{pq} and on the number of steps in the computation of P that goes from p to q with an empty stack on input x. For the first one we have that, if the derivation has one step, then we have to use a rule which on the right-hand side contains no variables. The only rules in G where no variables occur on the right-hand side are $A_{pp} \to \epsilon$. Input ϵ takes P from p with an empty stack to p with an empty stack. If the derivation is of length $k + 1$, for $k \geq 1$, we assume that the sentence holds for derivations of length at most k. Suppose that $A_{pq} \overset{k+1}{\Longrightarrow} x$. The first step of this derivation is either $A_{pq} \Rightarrow aA_{rs}b$ or $A_{pq} \Rightarrow A_{pr}A_{rq}$. In the first case, we consider the factorization of x, $x = ayb$. So,we have that $A_{rs} \overset{k}{\Longrightarrow} y$, but from the induction hypothesis we have that P can go from r with empty stack to s with empty stack. Because $A_{pq} \to aA_{pq}b$ is a rule of G, $(r, t) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, t)$, for some stack symbol t. Hence, if P starts at p with an empty stack, after reading a it can go to state r and push t into the stack. Then reading string y can bring it to s and leave t on the stack. Then after reading b it can go to state q and pop t off the stack. In

the second case, we consider the factorization of x, $x = yz$, where A_{pr} generates y and A_{rq} generates z. Because both $A_{pr} \overset{*}{\Rightarrow} y$ and $A_{rq} \overset{*}{\Rightarrow} z$ are most of k steps, we have from induction hypothesis that y can bring P from p to r and z can bring from r to q with an empty stack. Which is what we want to show.
For the second sentence, we have that if the computation has 0 steps, it starts and ends at the same state, say p. In 0 steps, P only has time to read the empty string, so $x = \epsilon$. By construction, G has the rule $A_{pp} \to \epsilon$. We assume that the sentence holds for computations of length at most k, where $k \geq 0$.We will prove it for computations of length $k+1$. We suppose that P has a computation wherein x brings p to q with an empty stack in $k+1$ steps. Either the stack is empty only at the beginning and end of this computation, or it becomes empty somewhere too.
In the first case, the symbol that is pushed at the first move must be the same as the symbol that is popped at the last move, let t be that symbol. We let a be the input read in the first move, b be the input input read in the last move, r be the state after the first move, and s be the state before the last move. Then, $(r, t) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, t)$, so the rule $A_{pq} \to aA_{rs}b$ is in G. We have the factorization of x, $x = ayb$. Input y can bring P from r to s without touching the symbol t that is on the stack and P can go from r with empty stack to s with empty stack on input y. We have removed the first and last steps of the $k+1$ steps of the original computation on x, so the computation on y has $(k+1)-2 = k-1$ steps. From the induction hypothesis we have $A_{rs} \overset{*}{\Rightarrow} y$, so we have $A_{pq} \overset{*}{\Rightarrow} x$.
In the second case, we let r be the state where the stack becomes empty other than at the beginning or the end of the computation on x. Then the portions of the computation from p to r and from r to q each contains at most k steps. We assume that y is the input read during the first portion and z is the input read during the second portion. We have from induction hypothesis that $A_{pr} \overset{*}{\Rightarrow} y$ and $A_{rq} \overset{*}{\Rightarrow} z$. Because the rule $A_{pq} \to A_{pr}A_{rq}$ is in G, we have $A_{pq} \overset{*}{\Rightarrow} x$, which completes our proof. □

We have just proved that pushdown automata recognize the class of context-free languages. This proof gives us a different explanation of Corollary 2.1.1. Because every regular language is recognized by a finite automaton and every finite automaton is a pushdown automaton that simply ignores its stack.

Chapter 3

Conjunctive Grammars

Conjunctive grammars are a natural extension of context-free grammars. The main difference between the two formalisms is that conjunctive grammars allow conjunctions on the right-hand side of the rules. This class of grammars, defined by A. Okhotin in [1], is more recent and appears to possess many interesting properties. For further study of conjunctive grammars see [1, 2, 3, 4, 5, 6, 7, 10, 11, 12].

3.1 Definitions

Definition 3.1.1. *A conjunctive grammar is a quadruple $G = (\Sigma, N, P, S)$, in which:*

- *Σ and N are disjoint finite non empty sets of terminal and non-terminal symbols respectively*
- *P is a finite set of grammar rules, each of the form*

  $$A \to \alpha_1 \& \ldots \& \alpha_n$$

  where $A \in N$, $n \geq 1$ and $\alpha_1, \ldots, \alpha_n \in (\Sigma \cup N)^*$
- *$S \in N$ is a non-terminal designated as the start symbol.*

The semantics of conjunctive grammars is defined using derivations, more or less in the same way as in the context-free case. The only difference is in the objects being transformed: while context-free derivations operate on strings over $\Sigma \cup N$, which are terms over concatenation, derivations in conjunctive grammars use terms over concatenation and conjunction. Let us denote such terms as strings over an extended alphabet $\Sigma \cup N \cup \{(, \&,)\}$, assuming that none of the three special symbols is in $\Sigma \cup N$. The set of valid string representations is defined inductively as follows:

1. ϵ is a term

2. Every symbol from $\Sigma \cup N$ is a term

3. If u and v are terms, then uv is a term

4. If $u_1, \dots, u_n$ $(n \geq 1)$ are terms, then $(u_1 \& \dots \& u_n)$ is a term.

Definition 3.1.2. *Given a grammar G, define the relation $\overset{G}{\Longrightarrow}$ of immediate derivability on the set of terms:*

1. *A non-terminal can be rewritten by the body of some rule enclosed in parentheses, that is, if $s_1 A s_2$ with $A \in N$ is a term, then, for every rule $A \to \alpha_1 \& \dots \& \alpha_n \in P$,*

$$s_1 A s_2 \overset{G}{\Longrightarrow} s_1(\alpha_1 \& \dots \& \alpha_n) s_2$$

2. *A conjunction of several identical terms enclosed in parentheses can be replaced by one such term without the parentheses, that is, if u is a term, then*

$$s_1(\underbrace{u \& \dots \& u}_{n}) s_2 \overset{G}{\Longrightarrow} s_1 u s_2$$

With $\overset{G}{\Longrightarrow}^n$ we denote the result of n steps of $\overset{G}{\Longrightarrow}$. Let $\overset{G}{\Longrightarrow}^*$ be the reflexive and transitive closure of $\overset{G}{\Longrightarrow}$. From now on, we will write $\Longrightarrow$ instead of $\overset{G}{\Longrightarrow}$ and $\Longrightarrow^*$ instead of $\overset{G}{\Longrightarrow}^*$, when G is obvious from the context. The language generated by a term A is the set of all strings over Σ derivable from its start symbol in a finite number of steps:

$$L_G(A) = \{\, w \mid w \in \Sigma^*,\ A \Longrightarrow^* w\}$$

The language generated by a grammar is the language generated by the star symbol S of the grammar:

$$L(G) = L_G(S) = \{\, w \mid w \in \Sigma^*,\ S \Longrightarrow^* w\}$$

Let us now consider a representation of conjunctive grammars by language equations:

Definition 3.1.3. *For every conjunctive grammar $G = (\Sigma, N, P, S)$, the associated system of language equations is a system of equations in variables N, in which each variable assumes a value of a language over Σ, and which contains the following equation for every variable A:*

$$A = \bigcup_{A \to \alpha_1 \& \dots \& \alpha_m \in P} \ \bigcap_{i=1}^{m} \alpha_i\,, \qquad A \in N$$

Each instance of a symbol $\alpha \in \Sigma$ in such a system defines a constant language $\{a\}$, while each empty string denotes a constant language $\{\epsilon\}$. A solution of such a system is a vector of languages $(\ldots, L_C, \ldots)_{C \in N}$, such that the substitution of L_C for C, for all $C \in N$, turns each equation of Definition 3.1.3 into an equality.
It is known that every such system has a non-empty set of solutions, and among them the least solution consists of exactly the languages generated by the non-terminals of the original conjunctive grammar: $(\ldots, L_G(C), \ldots)_{C \in N}$.
Now, we will give some examples of conjunctive grammars. First, we will construct a conjunctive grammar for the most common example of a non context-free language.

Example 3.1.1. *The following conjunctive grammar generates the language* $\{\, a^n b^n c^n \mid n \geq 0\}$*:*

$$\begin{array}{lcl} S & \longrightarrow & AB\&DC \\ A & \longrightarrow & aA \,|\, \epsilon \\ B & \longrightarrow & bBc \,|\, \epsilon \\ C & \longrightarrow & cC \,|\, \epsilon \\ D & \longrightarrow & aDb \,|\, \epsilon \end{array}$$

The grammar is based upon the representation of this language as an intersection of two context-free languages: $\{\, a^n b^n c^n \mid n \geq 0\} = \{\, a^i b^j c^k \mid j = k\} \cap \{\, a^i b^j c^k \mid i = j\}$ According to this grammar, the string *abc* can be derived in the following way:

$$S \Longrightarrow (AB\&DC) \Longrightarrow ((aA)B\&DC) \Longrightarrow ((aA)(bBc)\&DC) \Longrightarrow$$
$$((aA)(bBc)\&(aDb)C) \Longrightarrow ((aA)(bBc)\&(aDb)(cC)) \Longrightarrow^4$$
$$((a())(b()c)\&(a()b)(c())) \Longrightarrow^4 ((a)(bc)\&(ab)(c)) \Longrightarrow^2 (abc\&abc) \Longrightarrow abc$$

In essence, here two context-free derivations are done in parallel, and the same string has to be derived from AB and from DC in order to do the last step of the derivation. Another common example of a non context-free language, $\{\, wcw \mid w \in \{a, b\}^*\}$, forms a more interesting case, because, as demonstrated in [8, 9], it is not expressible as a finite intersection of context-free languages. Let us give a conjunctive grammar for this language and explain how it works.

Example 3.1.2. *The following conjunctive grammar generates the language* $\{\, wcw \mid w \in \{a, b\}^*\}$*:*

$$\begin{array}{lcl} S & \longrightarrow & C\&D \\ C & \longrightarrow & aCa \,|\, aCb \,|\, bCa \,|\, bCb \,|\, c \\ D & \longrightarrow & aA\&aD \,|\, bB\&bD \,|\, cE \\ A & \longrightarrow & aAa \,|\, aAb \,|\, bAa \,|\, bAb \,|\, cEa \\ B & \longrightarrow & aBa \,|\, aBb \,|\, bBa \,|\, bBb \,|\, cEb \\ E & \longrightarrow & aE \,|\, bE \,|\, \epsilon \end{array}$$

The non-terminal C generates $\{\, xcy \mid x, y \in \{a, b\}^*$ and $|x| = |y|\}$ and thus ensures that the string consists of two equal-length parts separated by a center marker. D takes one symbol from the left and uses A or B to compare it to the corresponding symbol at the right. At the same time, D recursively refers to itself in order to apply the same rule to the rest of the string. Formally, A generates $\{\, xcvay \mid x, v, y \in \{a, b\}^*,\ |x| = |y| \}$, B generates $\{\, xcvby \mid x, v, y \in \{a, b\}^*,\ |x| = |y| \}$ and therefore D produces $\{\, uczu \mid u, z \in \{a, b\}^* \}$ (the last result may be obtainned by a straightforward induction on the length of the string). Finally, $\{\, xcy \mid x, y \in \{a, b\}^* \text{ and } |x| = |y| \} \cap \{\, uczu \mid u, z \in \{a, b\}^* \} = \{\, wcw \mid w \in \{a, b\}^* \}$

Let us construct a derivation of the string *abcab* and thus formally demonstrate that it is generated by the given grammar:

$$\begin{gathered}
S \Longrightarrow (C\&D) \Longrightarrow ((aCb)\&D) \Longrightarrow ((a(bCa)b)\&D) \Longrightarrow ((a(b(c)a)b)\&D) \Longrightarrow \\
((a(bca)b)\&D) \Longrightarrow ((abcab)\&D) \Longrightarrow (abcab\&D) \Longrightarrow (abcab\&(aA\&aD)) \Longrightarrow \\
(abcab\&(a(bAb)\&aD)) \Longrightarrow (abcab\&(a(b(cEa)b)\&aD)) \Longrightarrow \\
(abcab\&(a(b(c()a)b)\&aD)) \Longrightarrow (abcab\&(a(b(ca)b)\&aD)) \Longrightarrow \\
(abcab\&(a(bcab)\&aD)) \Longrightarrow (abcab\&(abcab\&aD)) \Longrightarrow \\
(abcab\&(abcab\&a(bB\&bD))) \Longrightarrow (abcab\&(abcab\&a(b(cEb)\&bD))) \Longrightarrow \\
(abcab\&(abcab\&a(b(c(aE)b)\&bD))) \Longrightarrow \\
(abcab\&(abcab\&a(b(c(a())b)\&bD))) \Longrightarrow \\
(abcab\&(abcab\&a(b(c(a)b)\&bD))) \Longrightarrow (abcab\&(abcab\&a(b(cab)\&bD))) \Longrightarrow \\
(abcab\&(abcab\&a(bcab\&bD))) \Longrightarrow (abcab\&(abcab\&a(bcab\&bD))) \Longrightarrow \\
(abcab\&(abcab\&a(bcab\&b(cE)))) \Longrightarrow (abcab\&(abcab\&a(bcab\&b(c(aE))))) \Longrightarrow \\
(abcab\&(abcab\&a(bcab\&b(c(a(bE)))))) \Longrightarrow \\
(abcab\&(abcab\&a(bcab\&b(c(a(b()))))))) \Longrightarrow \\
(abcab\&(abcab\&a(bcab\&b(c(a(b)))))) \Longrightarrow \\
(abcab\&(abcab\&a(bcab\&b(c(ab))))) \Longrightarrow (abcab\&(abcab\&a(bcab\&b(cab)))) \Longrightarrow \\
(abcab\&(abcab\&a(bcab\&bcab))) \Longrightarrow (abcab\&(abcab\&abcab)) \Longrightarrow \\
(abcab\&abcab) \Longrightarrow abcab
\end{gathered}$$

It is important to note that the construction essentially uses the center marker, and therefore this method cannot be applied to construct a conjunctive grammar for the language $\{\, ww \mid w \in \{a, b\}^* \}$. The question of whether $\{\, ww \mid w \in \{a, b\}^* \}$ can be specified by a conjunctive grammar remains an open problem. In subsection 3.3 we will see an attempt to prove that the language $\{\, ww \mid w \in \{a, b\}^* \}$ can not be produced by any conjunctive grammar.

3.2 Special Cases

In this section we will study the power of conjunctive languages when we have bounded resources. The resources of a conjunctive language are the number of symbols or terminals. Similar with the context-free case, we will study both cases, when we bound the variable symbols and when we bound the terminal symbols. First, we present an example which demonstrates that if we compare context-free grammars with one variable, with conjunctive grammars with one variable, still conjunctive ones are more powerful. We then introduce Proposition 3.2.1 which can be used to infer negative expressibility results conserning univariate conjunctive grammars. To our knowledge, this result (despite its simplicity) hasn't been mentioned in the literature. We end this paragraph with Example 3.2.1 introduced by A. Jeż in [4]. It was the first example of a non-regular unary language and until then it was believed that conjunctive grammars over one terminal produce only regular languages.

Lemma 3.2.1. *The following conjunctive grammar with only one non-terminal symbol generates the language $a^* \cup b^*$:*

$$\begin{array}{lcl} S & \longrightarrow & aaS \,\&\, aSa \\ S & \longrightarrow & bbS \,\&\, bSb \\ S & \longrightarrow & a \\ S & \longrightarrow & b \\ S & \longrightarrow & \epsilon \end{array}$$

In Lemma 2.3.1 we saw, that the language $a^* \cup b^*$ can not be produced by a context-free grammar with only one variable. The above example demonstrates that conjunctive grammars are more powerful than context-free ones even if we restict attetion to one non-terminal symbol.
Now, lets give a proposition equivalent with 2.3.1 one. We will use it to show negative results for conjunctive grammars with only one non-terminal symbol.

Proposition 3.2.1. *Let $G = (\Sigma, N, P, S)$ be a conjunctive grammar. We define the set $U = \{u \in \Sigma^* \mid A \to u \in P, \text{ with } A \in N\}$. Then for all $w \in L(G)$, there exists an element of U which is a substring of w.*

We leave the proof of this proposition to the reader. As a result of the previous proposition we have that the language $\{\, a^n b^n c^n \mid n \geq 1\}$ can not be produced by any conjunctive grammar with only one non-terminal symbol. Additionally, we have that neither the language $\{\, a^n b^m a^n b^m \mid m, n \geq 1\}$ can be produced by any conjunctive grammar with only one variable. An idea of how we have these results is that if k is the biggest number of strings that appear on the right-hand side of all rules, then in the first language the word $a^k b^k c^k$ has to have a substring of the set U of Proposition 3.2.1. But, this is not possible since all strings of U are of the form $a^l b^l c^l$, where $3l \leq k$. So far, the negative results that we have seen from Proposition 3.2.1 are about non context-free languages. Then, someone may ask about the relation between conjunctive grammars with only one non-terminal symbol and general context-free grammars. But, as a result of the previous proposition one can easily see that the simple regular language ab^*a can not be produced by any conjunctive grammar with only one non-terminal. The follow lemma, given by A. Alhazov, shows that we can produce every unary regular language with a conjunctive grammar with only one non-terminal.

Lemma 3.2.2. *Every unary regular language is generated by a one-nonterminal conjunctive grammar.*

Proof. Let $K \cup (a^p)^+ L$ be the given language, where $p \geq 1$ and $K, L \subseteq \{\epsilon, a, \ldots, a^{p-1}\}$. Then the required grammar is

$$\begin{array}{lcl} S & \longrightarrow & a^i, \text{ where } a^i \in K \cup a^pL \cup a^{2p}L \\ S & \longrightarrow & a^pS\&a^{2p}S \end{array}$$

In other words, the language equation

$$X = K \cup a^pL \cup a^{2p}L \cup (a^pX \cap a^{2p}X)$$

has the unique solution $K \cup (a^p)^+L$. □

Until now, we have studied the case of bounding the non-terminal symbols. What happens in the case of bounding the terminal symbols? We have seen in context-free languages that if we bound them to only one terminal they are equivalent to regular languages. It was believed that the same holds in the case of conjunctive grammars. But, in the next example we will see that conjunctive grammars over one terminal can produce non regular languages.

Example 3.2.1. *The following conjunctive grammar with the start symbol A_1 generates the language $\{ a^{4^n} \mid n \geq 0\}$:*

$$\begin{array}{lcl} A_1 & \longrightarrow & A_1A_3 \,\&\, A_2A_2 \,|\, a \\ A_2 & \longrightarrow & A_1A_1 \,\&\, A_2A_6 \,|\, aa \\ A_3 & \longrightarrow & A_1A_2 \,\&\, A_6A_6 \,|\, aaa \\ A_6 & \longrightarrow & A_1A_2 \,\&\, A_3A_3 \end{array}$$

Each non-terminal A_i generates the language $\{a^{i4^n} | n \geq 0\}$.

This construction is best understood in terms of 4−base positional notation of the lengths of the strings. Then A_1, A_2, A_3 and A_6 generate strings of length $(10\ldots0)_4$, $(20\ldots0)_4$, $(30\ldots0)_4$ and $(120\ldots0)_4$, respectively.
The goal of the rule $A_1 \to A_1A_3\&A_2A_2$ is to represent every next power of four by summing up some multiples of the previous power of four. The concatenation A_1A_3 contains strings of length $4^j + 34^l$; if $j = l$, this will be exactly 4^{j+1}, which is the desired number. However, the concatenation A_1A_3 also produces some junk strings of length $4^j + 3 \times 4^l$ with $j \neq l$, and the base−4 notation of these strings is of the form $(10\ldots030\ldots0)_4$ or $(30\ldots010\ldots0)_4$. The second concatenation A_2A_2 consists of all strings of length $2 \times 4^j + 2 \times 4^l$, and in the case of $j = l$ this gives the required number 4^{j+1}. The junk strings obtainned in this concatenation have length with base−4 notation $(20\ldots020\ldots0)_4$, and it is easy to see that the sets of junk strings in these two concatenations are disjoint. Thus the conjunction $A_1A_3\&A_2A_2$ filters out all strings of length other than the next power of four.
To simplify the reasoning, it is useful to define the positional notation formally. Let $\Sigma_k = \{0, 1, \ldots, k-1\}$ be digits. For every string of digits $w = a_{l-1}\ldots a_1a_0 \in \Sigma_k^*$, let $(w)_k = \Sigma_{i=0}^{l-1} a_ik^i$ be the number defined by this string. For every language L of positional notations of numbers, the set of numbers it defines is denoted by $(L)_k = \{(w)_k | w \in L\}$. Finally, for any two sets of numbers $S, T \subseteq \mathbb{N}$, define their sum as $S + T = \{ m + n \mid m \in S, n \in T\}$.
Now the above conjunctive grammar can be represented as the following system of equations over sets of natural numbers

$$\begin{array}{rcl} X_1 & = & (X_1+X_3) \cap (X_2+X_2) \cup \{1\} \\ X_2 & = & (X_1+X_1) \cap (X_2+X_6) \cup \{2\} \\ X_3 & = & (X_1+X_2) \cap (X_6+X_6) \cup \{3\} \\ X_6 & = & (X_1+X_2) \cap (X_3+X_3) \end{array}$$

and the claim is that the least solution of this system is $((10^*)_4, (20^*)_4, (30^*)_4, (120^*)_4)$. The first equation has already been verified before, and in the new notation this calculation is as follows:

$$((10^*)_4 + (30^*)_4) \cap ((20^*)_4 + (20^*)_4) =$$
$$((10^*30^*)_4 \cup (100^*)_4 \cup (30^*10^*)_4) \cap ((20^*20^*)_4 \cup (100^*)_4) = (100^*)_4$$

Then, the right-hand side evaluates to $(10^*)_4$. The rest of equations hold by similar calculations:

$$((10^*)_4 + (10^*)_4) \cap ((20^*)_4 + (120^*)_4) =$$
$$((10^*10^*)_4 \cup (20^*)_4) \cap ((20^*120^*)_4 \cup (200^*)_4 \cup (320^*)_4 \cup (120^*20^*)_4) = (200^*)_4$$
$$((10^*)_4 + (20^*)_4) \cap ((120^*)_4 + (120^*)_4) =$$
$$((10^*20^*)_4 \cup (30^*)_4 \cup (20^*10^*)_4) \cap ((120^*120^*)_4 \cup (1320^*)_4 \cup (300^*)_4) = (300^*)_4$$
$$((10^*)_4 + (20^*)_4) \cap ((30^*)_4 \cup (30^*)_4) =$$
$$((10^*20^*)_4 \cup (30^*)_4 \cup (20^*10^*)_4) \cap ((30^*30^*)_4 \cup (120^*)_4) = (120^*)_4$$

3.3 An Attempt to Find a Non-Conjunctive Language

In all computational models the main question is what is their limitations. Which are the languages that can be produced by these models and which can't. Still, it remains an open problem to find a language that is not conjunctive. The only answer we can give in this question is that we have a cubic time algorithm which recognizes conjunctive grammars. From the time hierarchy theorem we know that there are languages which aren't in $Time(O(n^3))$. Then, we know that these languages can not be produced by any conjunctive grammar, but we don't have a method of checking if a language, which belongs to $Time(O(n^3))$, is conjunctive or not. Additionally, because the language $\{\, wcw \mid w \in \{a,b\}^* \}$ is conjunctive we can't expect to have a theorem like pumping lemma, as we have in automata or in context-free grammars.
In this paragraph I will make an attempt to find a language which is not conjunctive. The hypothesis is that the language which is not conjunctive is $WW = \{\, ww \mid w \in \{a,b\}^* \}$. The idea of showing that this language is not conjunctive, is to find a classification on conjunctive grammars. This will be a function from the set of conjunctive grammars to natural numbers. If we denote the set of conjunctive grammars by CG, then a classification could be a function $F : CG \to \mathbb{N}$. Moreover, we want to give successive approximation of WW with a family of languages L_i, where $WW = \bigcup_{i\in\mathbb{N}} L_i$, and for $i < j$ we want $F(L_i) < F(L_j)$. Someone may notice that we put a language as an

element of the domain of the function F instead of a grammar. But, we denote by $F(L)$, for a conjunctive language L, the least $F(G)$ where G is a conjunctive grammar and $L = L(G)$, $F(L) = min\{\, F(G) \mid L = L(G)\}$. From now on, we will use that notation for any classification that we define. Before we define this family of languages we have to define something else. Let w be a word over Σ^*. We define $[w]$ to be the number of distinct letters alternating into the word w, for example $[a^5b^3a^7c^2] = 4$. Now, we can define the family of languages L_i:

$$L_i = \{\, ww \mid w \in \{a,b\}^* \quad and \quad [w] = i\}$$

The first attempt of finding a classification is the function $NT : CG \to \mathbb{N}$, where NT will take a grammar as an input and it will give the number of its non-terminal symbols. For example, the grammar $G = (\{a,b\}, \{S\}, \{S \to aaS\&aSa, S \to bbS\&bSb, S \to \epsilon\}, S)$ has $NT(G) = 1$. We notice that G produces L_1 and from Proposition 3.2.1 we know that $NT(L_2) > 1$. My first assumption was that it holds $NT(L_{i+1}) > i$, for every $i \in \mathbb{N}$.
Lets assume $NT(L_{i+1}) > i$ is true for every $i \geq 3$, then we will have that WW is not conjunctive. Lets assume for the sake of contradiction that that WW is conjunctive. Then for a natural number m it will be $NT(WW) = m$. Then we will make the grammar G' with the rules:

$$\begin{array}{rcl} S_1 & \longrightarrow & \underbrace{ABAB\ldots AB}_{4m+8} \mid \underbrace{BABA\ldots BA}_{4m+8} \\ A & \longrightarrow & aA \mid a \\ B & \longrightarrow & bB \mid b \end{array}$$

Then G' produces the language $\{\, w \in \{a,b\}^* \mid [w] = 4m+8\}$. Now, if we combine the grammar G' and a grammar which produces WW, say that is G'' with start symbol S_2, we can take the grammar G which has all the rules of G' and G'' and one more rule $S \to S_1\&S_2$.[1] We will have then that $L_{2m+4} = L(G)$, but $NT(G) = m+4$ which contradicts to the conjecture that $NT(L_{2m+4}) > 2m+3$ because we have for every number of non-terminals m that $m+4 < 2m+4$. This gives us the conclusion that if the conjecture is true, WW is not conjunctive.
Until now, we haven't shown that for each $i \in \mathbb{N}$, L_i is conjunctive language. From the following grammars we will see that the statement ' for every $i \in \mathbb{N}$ we have that $NT(L_{i+1}) > i$', is not true. We will give a grammar G_i which will produce the language L_i, and each grammar G_i will have five non-terminals. Let S_i be the initial symbol of every G_i respectively. In each G_i we will use the following non-terminals and their rules:

$$\begin{array}{rcl} A & \longrightarrow & aA \mid a \\ B & \longrightarrow & bB \mid b \\ A_n & \longrightarrow & aA_na \mid \underbrace{BAB\ldots}_{n} \\ B_n & \longrightarrow & aB_na \mid \underbrace{ABA\ldots}_{n} \end{array}$$

[1]We can assume that grammars G' and G'' don't have common non-terminals, if they have, we just change names of the non-terminals of one of the grammars.

An example for specific n of the above is A_3 will have the rule $A_3 \to BAB$, and B_4 will have the rule $B_4 \to ABAB$. We have already seen that L_1 can be produced by the grammar $S_1 \to aaS\&aSa \mid bbS\&bSb \mid \epsilon$. It is easy to see that the grammar which produces L_2 is $S_2 \to A_1B\&AB_1 \mid B_1A\&BA_1$. For general $i \in \mathbb{N}$ we will have the following grammars which produce L_i:
For i even

$$\begin{aligned}
S_i &\longrightarrow A_{i-1}\underbrace{BAB\ldots}_{i-1}\&AB_{i-1}\underbrace{ABA\ldots}_{i-2}\&\ldots\&\underbrace{ABA\ldots}_{i-1}B_{i-1}\\
S_i &\longrightarrow B_{i-1}\underbrace{ABA\ldots}_{i-1}\&BA_{i-1}\underbrace{BAB\ldots}_{i-2}\&\ldots\&\underbrace{BAB\ldots}_{i-1}B_{i-1}
\end{aligned}$$

For i odd

$$\begin{aligned}
S_i &\longrightarrow A_{i-2}A_{i-2}\&AB_{i-2}\underbrace{ABA\ldots}_{i-2}\&\ldots\&\underbrace{ABA\ldots}_{i-2}B_{i-2}A\\
S_i &\longrightarrow B_{i-2}B_{i-2}\&BA_{i-2}\underbrace{BAB\ldots}_{i-2}\&\ldots\&\underbrace{BAB\ldots}_{i-2}A_{i-2}B
\end{aligned}$$

We notice that each G_i needs only five non-terminals, S_i, A, B and A_m, B_m for $m = i-1$ or $m = i-2$, so we need another way of classification. This could be to count the number of & that a grammar has. But, probably it is not a good way to do so, since languages with similar 'complexity' will have different measures. Like in case of the obvious grammars which produce L_2 and $\{\, a^n b^m a^n b^m \mid n, m \geq 0\}$, where they are similar languages but in the case of L_2 we need the double number of & because we have a disjunction of the first letter a or b. So, in the next definition we will give the following, and probably better, classification.

Definition 3.3.1. *We define the function DAP over a conjunctive grammar $G = (\Sigma, N, P, S)$, $DAP : N \times \mathscr{P}(N) \to \mathbb{N}$, by the formula*

$$DAP(A, Y) = \max_{u \in P_A} \{DAP(u, Y \cup \{A\}) + k\}$$

where k is the number of occurrences of & in u and by convention $DAP(u, M) = \sum_{B \in (N-M)} DAP(B, M)$, where $u \in (\Sigma \cup N \cup \{\&\})^*$ and B appears in u and finally $P_A = \{\, u \in (\Sigma \cup N \cup \{\&\})^* \mid A \to u \in P\}$.
We define the degree of G by $DAP(G)$, where $DAP(G) = DAP(S, \{S\})$.

The previous definition is a little complicated, so we will give some examples of this degree for specific grammars. In Example 3.1.1, we have that its degree is 1. Because $DAP(S, \{S\}) = DAP(AB\&DC, \{S\}) + 1 = DAP(A, \{s\}) + DAP(B, \{s\}) + DAP(C, \{s\}) + DAP(D, \{s\}) + 1$ and for every $X \in \{A, B, C, D\}$, we have $DAP(X, \{S\}) = 0$. Similarly, we have the degree of the grammar in Example 3.1.2 that is 2. Finally, lets calculate the DAP degree of a little more complicated grammar, the one of Example 3.2.1.

$$\begin{gathered}
DAP(A_1, \{A_1\}) = DAP(A_2, \{A_1, A_2\}) + DAP(A_3, \{A_1, A_3\}) + 1 = \\
DAP(A_6, \{A_1, A_2, A_6\}) + DAP(A_2, \{A_1, A_2, A_3\}) + DAP(A_6, \{A_1, A_3, A_6\}) \\
+ 3 = DAP(A_3, N) + DAP(A_6, N) + DAP(A_2, N) + 6 = 9
\end{gathered}$$

Now, we will give some propositions which are easy to prove and we leave that job at the reader.

Proposition 3.3.1. *Let L_1 and L_2 two conjunctive languages with $DAP(L_1) = m$ and $DAP(L_2) = n$. Then it holds that $DAP(L_1 \cap L_2) \leq m + n + 1$ and that $DAP(L_1 \cup L_2) \leq max\{m, n\}$.*

Proposition 3.3.2. *Let $G = (\Sigma, N, P, S)$ be a conjunctive grammar. For every $A, B \subseteq N$, and $C \in N$, where $A \subseteq B$, we have*
$DAP(C, A) \geq DAP(C, B)$.

Proposition 3.3.3. *Let $G = (\Sigma, N, P, S)$ be a conjunctive grammar. For every $A \in N$ and $B \subseteq N$, we can calculate $DAP(A, B)$ in a finite number of steps.*

Proposition 3.3.4. *For a conjunctive language L we have that, $DAP(L) = 0$ iff L is a context-free language.*

The claim here is that the degree of the language L_{i+2} is at least i, i.e. $DAP(L_{i+2}) \geq i$. If this claim is true, we have managed to prove that the language WW is not conjunctive. For the sake of contradiction, we assume that the language WW is a conjunctive language, then it has a degree. Let m be that degree. Then the language $L_{2(m+2)} = WW \cap ((aa^*bb^*)^{2(m+2)} \cup (bb^*aa^*)^{2(m+2)})$ has degree at most $m + 1$ which contradicts the claim.
Lets think about $DAP(L_i)$, we start our thinking in small i. For $i = 1$, we have the language $L_1 = (a^2)^* \cup (b^2)^*$, where the grammar with the rules $S \to A \,|\, B$, $A \to aaA \,|\, \epsilon$ and $B \to bbB \,|\, \epsilon$, produces it. From this grammar we have that $DAP(L_1) = 0$, where the claim holds. For the language L_2 we know that it's not context-free so we know that $DAP(L_2) \geq 1$, and from the grammar

$$\begin{array}{lcl} S & \longrightarrow & AD\&EB \,|\, DA\&BE \\ A & \longrightarrow & aAa \,|\, D \\ B & \longrightarrow & bBb \,|\, E \\ D & \longrightarrow & bB \,|\, \epsilon \\ E & \longrightarrow & aE \,|\, \epsilon \end{array}$$

we have that $DAP(L_2) = 1$, where again the claim holds. Unfortunately, I haven't found a way to continue with the rest L_i, for $i \geq 3$. A thought was to try induction, but we can't use L_i for speaking about L_{i+1} because of their definition. Then, I tried to prove the claim for some more general family of language than L_i, this form of languages could be of the form $L'_i = \{\, w_1ba^{n_1}bw_2ba^{n_2}b \ldots ba^{n_i}bw_{n+1}ba^{n_1}bw_{n+2}ba^{n_2}b \ldots ba^{n_i}bw_{2n+1} \;\mid\; w_k \in \{a, b\}^*$ for $k = 1, \ldots 2n + 1\}$. But again no results. Continuing in this way, we introduce a class of conjunctive languages, we will call them *generalized context-free (GCF)* languages. This class of languages seems to contain L_i, and if this is true then again we have proven my claim.

Definition 3.3.2. *Let a given conjunctive grammar $G = (\Sigma, N, P, S)$. Then, $D \in N$ is called a GCF symbol if*

- *$L(D)$ is a context-free language (or similarly $DAP(L(D)) = 0$)*
- *All the non-terminal symbols which appear on the right-hand side of all rules with head D, are GCF symbols.*

The conjunctive grammar G is GCF grammar, if S is a GCF symbol. A language L is GCF language, if there exists a GCF grammar G, where $L = L(G)$ and $DAP(L) = DAP(G)$.

For example, the grammar of Example 3.1.1 is a GCF grammar but the grammars from Examples 3.1.2 and 3.2.1 are not. Lets see some propositions for GCF grammars, again they are easy to verify and we leave it to the reader.

Proposition 3.3.5. *Let L be a conjunctive language. Then L is a GCF language iff L can be written as a finite intersection of context-free languages.*[2]

Proposition 3.3.6. *Let L be a GCF language and R a regular language. Then if $DAP(L) = m$ we have that $DAP(L \cap R) \leq m$.*

Proposition 3.3.7. *Let L be a unary language. Then L is a GCF language iff L is regular.*

Two words about the proof of the second proposition is that we prove it with induction on DAP. Where, at basis step we know it by Theorem 2.1.2 and it is easy to continue on the next steps. Now, we will see another proposition a little more complicated and more interesting.

Proposition 3.3.8. *Let L be a GCF language. Then $DAP(L) = m$ iff $L \in \big(\sqcup(\sqcap_{m+1}CF) - \sqcup(\sqcap_m CF)\big)$.*[3]

Proof. We begin with the "only if" part. Then, we have that $DAP(L) = m$ for a GCF language L and we want to show that $L \in \sqcup(\sqcap_{m+1}CF)$ and $L \notin \sqcup(\sqcap_m CF)$. We have that if $DAP(L) = m$ then $L \in \sqcup(\sqcap_{m+1}CF)$ from induction on m and with the help of the identity $(A\cup B)\cap(C\cup D) = (A\cap C)\cup(B\cap C)\cup(A\cap D)\cup(B\cap D)$ for all sets A, B, C, D. In our case, A, B are sets of $\sqcup(\sqcap_{s_1}CF)$ and C, D are sets of $\sqcup(\sqcap_{s_2}CF)$, where $s_1 + s_2 + 1 \leq m$. Then we can see that each $A \cap C$, $B \cap C$, $A \cap D$ and $B \cap D$ have the most m intersections.
We have that $L \notin \sqcup(\sqcap_m CF)$ by contradiction. For contradiction sake we assume that there are L_{ij} context-free languages which L is of the form

$$L = (L_{11} \cap \ldots \cap L_{1m}) \cup \ldots \cup (L_{n1} \cap \ldots \cap L_{nm})$$

where $1 \leq i \leq n$ and $1 \leq j \leq m$ for a natural number n. Then, for every language L_{ij} we have a context-free grammar $G_{ij} = (\Sigma, N_{ij}, P_{ij}, S_{ij})$ which produces it. We can assume that for every two $(i,j) \neq (k,l)$ it's true that $N_{ij} \cap N_{kl} = \emptyset$. Then we can create the conjunctive grammar:

$$G = (\Sigma, (\bigcup_{ij} N_{ij}) \cup \{S\}, (\bigcup_{ij} P_{ij}) \cup \{S \to S_{i1} \& \ldots \& S_{im} \mid 1 \leq i \leq n\}, S)$$

[2]From now on we will use the notation $\sqcup$ for finite union and $\sqcap$ for finite intersection. Moreover, we will use the notation $\sqcup_k$ for at most k unions and $\sqcap_k$ for at most k intersections.
[3]We remind to the reader that by CF we denote the set of context-free grammars.

But we have that $L = L(G)$ and $DAP(G) = m - 1$, which contradicts with $DAP(L) = m$. The "if" part is obvious from the previous part. $\square$

We give all these definitions because in [8] we have a similar family of languages with L_i. In this family of languages we have that as we take more complicated languages we need more intersections of context-free languages in order to produce them. Then, if we prove that L_i are GCF languages we will have proven that WW is not a conjunctive language. We end this paragraph with a notice. Every word over $\{a, b\}$ can take us at a unique vector $\sigma \in \mathbb{N}^*$ and vice versa.[4] For example the word $a^3b^2a^4$ gives us the vector $(3, 2, 4)$ and the vector $(0, 5, 2, 1)$ gives as the word $b^5a^2b^1$. We can see that every language over $\{a, b\}$ has a unique correspondence to an element of $\mathscr{P}(\mathbb{N}^*)$ and vice versa. The question here is can we find some properties on elements of $\mathscr{P}(\mathbb{N}^*)$, where its correspondence to be GCF languages or conjunctive languages.

3.4 Synchronized Alternating Pushdown Automata

In this section we introduce another computational model, called *synchronized alternating pushdown automata* (SAPDA). This is similar to the pushdown automata of 2.4. As, we have shown that the class of context-free grammars is equal to the class of languages accepted by PDAs, we will show the equivalence of the class of conjunctive languages with the class of languages accepted by SAPDA. The original paper introducing this concept is [11]. For making our model similar to automata and to pushdown automata as introduced in 2.4, we give a different but equivalent definition of SAPDA as it's given in [11]. So, the proof of equivalence of the class of conjunctive grammars with the class of SAPDA is also different than the one given in the orininal paper. Moreover, I believe that the proof which I give here is simpler and easier for someone to understand.

3.4.1 Definition

Definition 3.4.1. *A synchronized alternating pushdown automaton*[5] *is a six-tuple $A = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ and F are all finite sets, and*

- Q *is a set of states*
- Σ *is the input alphabet*
- Γ *is the stack alphabet*

[4]Here, we assume that every word starts with an a. In different case, we will take n-tuple of natural numbers with 0 at the first position.

[5]Note that the given definition is different from the original paper [11]. We give a different definition of SAPDA for symmetry with automata and PDA. The main difference is that in the paper is defined the SAPDA which accept an input if every leaf node empties its stack. Here, SAPDA is defined to accept an input if every leaf node is in an acceptance state when it reads its input.

- *$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathscr{P}(\{(q_1, u_1) \wedge \ldots \wedge (q_k, u_k) | k = 1, 2, \ldots, i = 1, \ldots, k,$ $q_i \in Q$ and $u_i \in \Gamma\})$ is the transition function*
- *$q_0 \in Q$ is the initial state*
- *$F \subseteq Q$ is the set of accept states.*

We describe the current stage of the automaton computation as a directed labelled tree. In this tree we encode the stack contents, the current states of the stack-branches, and the remainning input to be read for each stack-branch. Each leaf node has a processing head which reads the input and writes to its branch independently. When a multiple-state conjunctive transition is applied, the stack branch splits into multiple branches, one for each conjunct. The branches process the input independently. When all children leaves are exactly the same, they collapse to the parent node, which becomes a leaf, after which the computation continues from the parent branch. The internal nodes are not labelled. We can give them a name for practical reasons when we want to refer to them, we will name them with the stack content $\alpha \in \Gamma^*$ which they had when they were leaves. The label of every leaf is a triple (q, w, α), where $q \in Q$ is the current state, $w \in \Sigma^*$ is the remainning input to be read and $\alpha \in \Gamma^*$ is the stack-branch contents.
A configuration of a SAPDA is a directed labeled tree where each internal node is named with a string of the stack alphabet and each leaf with a triple of the form (q, w, α). The computation of a SAPDA is the same as the computation of a PDA, where every computing step is made at only one leaf every time. The only other difference is that here we have an additional step, where a leaf can become an internal node and create leaf nodes when a multiple transition is applied. An accepting computation for a SAPDA on an input w,is when it starts with the single node tree (q_0, w, ϵ) and reaches a configuration of which every leaf node has read its input word and it is in an accept state.
The following examples will help us understand better how a SAPDA works.

Example 3.4.1. *In this example, we will give the formal description of the SAPDA that recognizes the language $\{ w \in \Sigma^* \mid |w|_a = |w|_b = |w|_c\}$.*[6] *Let A be $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where*
$Q = \{q_0, q_0', q_1, q_2, q_3\}$
$\Sigma = \{a, b, c\}$
$\Gamma = \{a, b, c, \#\}$
$F = \{q_3\}$
δ is defined as follows:

[6]We remind that for a word w and a letter a, we denote by $|w|_a$ the number of a's that appear in w.

$$\begin{array}{rcl}
\delta(q_0, \epsilon, \epsilon) & = & \{(q_0', \#)\} \\
\delta(q_0', \epsilon, \epsilon) & = & \{(q_1, \epsilon) \wedge (q_2, \epsilon)\} \\
\delta(q_1, c, \epsilon) & = & \{(q_1, \epsilon)\} \\
\delta(q_1, \sigma, \epsilon) & = & \{(q_1, \sigma)\},\ \sigma \in \{a, b\} \\
\delta(q_1, \sigma', \sigma'') & = & \{(q_1, \epsilon)\},\ \sigma' \neq \sigma''\ \textit{and}\ \sigma', \sigma'' \in \{a, b\} \\
\delta(q_2, a, \epsilon) & = & \{(q_2, \epsilon)\} \\
\delta(q_2, \sigma, \epsilon) & = & \{(q_2, \sigma)\},\ \sigma \in \{b, c\} \\
\delta(q_2, \sigma', \sigma'') & = & \{(q_2, \epsilon)\},\ \sigma' \neq \sigma''\ \textit{and}\ \sigma', \sigma'' \in \{b, c\} \\
\delta(q_1, \epsilon, \#) & = & \{(q_3, \epsilon)\} \\
\delta(q_2, \epsilon, \#) & = & \{(q_3, \epsilon)\}
\end{array}$$

The first step of the computation puts the symbol # for testing later if the stack is empty. After that, the next step of the computation opens two branches, one for verifying that the number of as in the input word equals the number of bs, the state q_1 is responsible for that. The other, for verifying that the number of bs equals to the number of cs, which q_2 is responsible for that. State q_1 pushes into the stack the letters as and bs, or it matches them and deletes both of them if it sees a pair of a,b where one is the reading input symbol and the other is the top of the stack of this branch. State q_2 works similarly. When, states q_1, q_2 turn out to state q_3 means that we have reached the end of the input word and the stack is empty, otherwise the SAPDA halts and it doesn't accept.
Lets see a possible computation of the previous example on the input $abbcccaab$:

$$\begin{gathered}
(q_0, abbcccaab, \epsilon) \triangleright (q_0', abbcccaab, \#) \triangleright (q_1, abbcccaab, \#) \wedge (q_2, abbcccaab, \#) \triangleright \\
(q_1, bbcccaab, a\#) \wedge (q_2, abbcccaab, \#) \triangleright (q_1, bcccaab, \#) \wedge (q_2, abbcccaab, \#) \triangleright \\
(q_1, cccaab, b\#) \wedge (q_2, abbcccaab, \#) \triangleright (q_1, cccaab, b\#) \wedge (q_2, bbcccaab, \#) \triangleright \\
(q_1, cccaab, b\#) \wedge (q_2, bcccaab, b\#) \triangleright (q_1, cccaab, b\#) \wedge (q_2, cccaab, bb\#) \triangleright \\
(q_1, cccaab, b\#) \wedge (q_2, ccaab, b\#) \triangleright (q_1, cccaab, b\#) \wedge (q_2, caab, \#) \triangleright \\
(q_1, cccaab, b\#) \wedge (q_2, aab, c\#) \triangleright (q_1, cccaab, b\#) \wedge (q_2, ab, c\#) \triangleright \\
(q_1, cccaab, b\#) \wedge (q_2, b, c\#) \triangleright (q_1, cccaab, b\#) \wedge (q_2, \epsilon, \#) \triangleright \\
(q_1, cccaab, b\#) \wedge (q_3, \epsilon, \epsilon) \triangleright (q_1, ccaab, b\#) \wedge (q_3, \epsilon, \epsilon) \triangleright (q_1, caab, b\#) \wedge (q_3, \epsilon, \epsilon) \\
\triangleright (q_1, aab, b\#) \wedge (q_3, \epsilon, \epsilon) \triangleright (q_1, ab, \#) \wedge (q_3, \epsilon, \epsilon) \triangleright (q_1, b, a\#) \wedge (q_3, \epsilon, \epsilon) \triangleright \\
(q_1, \epsilon, \#) \wedge (q_3, \epsilon, \epsilon) \triangleright (q_3, \epsilon, \epsilon) \wedge (q_3, \epsilon, \epsilon)
\end{gathered}$$

which is an accepting computation since all branches have reached the end of the input word and all branches are at the accept states. In Figure 3.1 we can see the ninth step of the previous computation of the form of the tree configurations.

Now, we consider the following example of a SAPDA which accepts the interesting non context-free language $\{\, wcw \mid w \in \{a, b\}^* \}$, which we can't even produce by any finite intersection of context-free languages. The example is of particular interest as it shows the model's ability to utilize recursive conjunctive transitions.

Example 3.4.2. *In this example, we will give the formal description of the SAPDA that recognizes the language* $\{\, wcw \mid w \in \{a, b\}^* \}$. *Let* A *be* $(Q, \Sigma, \Gamma, \delta, q_0, F)$, *where*
$Q = \{q_0, q, q', q_e, q_e', q_{ac}\} \cup \{\, q_\sigma^1 \mid \sigma \in \{a, b\}\} \cup \{\, q_\sigma^2 \mid \sigma \in \{a, b\}\}$

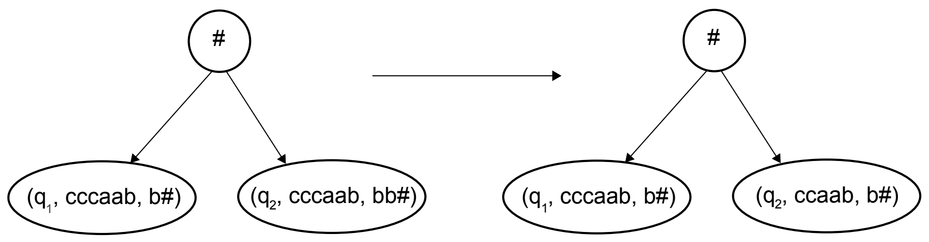

Figure 3.1: In the figure are the 9th and 10th configurations of the computation which we did of Example 3.4.1

$\Sigma = \{a, b, c\}$
$\Gamma = \{d, \#\}$
$F = \{q_{ac}\}$
δ is defined as follows:

$$
\begin{aligned}
\delta(q_0, \epsilon, \epsilon) &= \{(q, \#) \wedge (q', \#)\} \\
\delta(q', \sigma, \epsilon) &= \{(q', d)\},\ \sigma \in \{a, b\} \\
\delta(q', c, \epsilon) &= \{(q_e, \epsilon)\} \\
\delta(q, \sigma, \epsilon) &= \{(q, \epsilon) \wedge (q^1_\sigma, \epsilon)\},\ \sigma \in \{a, b\} \\
\delta(q, c, \epsilon) &= \{(q'_e, \epsilon)\} \\
\delta(q^1_\sigma, \tau, \epsilon) &= \{(q^1_\sigma, d)\},\ \sigma, \tau \in \{a, b\} \\
\delta(q^1_\sigma, c, \epsilon) &= \{(q^2_\sigma, \epsilon)\},\ \sigma \in \{a, b\} \\
\delta(q^2_\sigma, \tau, \epsilon) &= \{(q^2_\sigma, \epsilon)\},\ \sigma, \tau \in \{a, b\}\ \textit{and}\ \sigma \neq \tau \\
\delta(q^2_\sigma, \sigma, \epsilon) &= \{(q^2_\sigma, \epsilon), (q_e, \epsilon)\},\ \sigma \in \{a, b\} \\
\delta(q_e, \sigma, d) &= \{(q_e, \epsilon)\},\ \sigma \in \{a, b\} \\
\delta(q_e, \epsilon, \#) &= \{(q_{ac}, \epsilon)\} \\
\delta(q'_e, \sigma, \epsilon) &= \{(q'_e, \epsilon)\},\ \sigma \in \{a, b\} \\
\delta(q'_e, \epsilon, \#) &= \{(q_{ac}, \epsilon)\}
\end{aligned}
$$

The computation of the SAPDA starts by splitting in two branches. Each branch has two main phases: before and after the sign c is encountered in the input. In one branch, with the state q', it makes sure that the number of letters before c is the same with the number of letters after c. In the first phase of this branch, we count the number of letters before c by putting the symbol d into the stack for each letter that we read at the input. Later, when this branch reaches to the symbol c changes to state q_e. The only function of this state is to check if the letters of the remainning input are the same with the number of ds on the stack. Thus, this branch of the SAPDA accepts inputs of the form wcu, where $w, u \in \{a, b\}^*$ and $|w| = |u|$.

The other branch is more interesting but also more complicated. In the first phase of this branch, each input letter which is read leads to a conjunctive transition that opens two new stack-branches. One new branch continues the recursion, while the second checks that the following condition is met.
Assume σ is the nth letter on the left from the sign c. If so, the new stack branch opened during the transition on σ will verify that the nth letter from the end of the input is also σ. This way, if the computation is accepting, the word will in fact be of the form $wcuw$, where $w, u \in \{a, b\}^*$. To be able to check this property, the branch must know σ and σ's relative position to the sign c. To remember σ, the state of the branch head is q_σ^1, where the 1 superscript denoting that the computation is in the first phase. To find the relative position, the branch adds a sign d to its stack for each input symbol read after the σ, until the c is encountered in the input. Therefore, when the c is read, the number of ds in the stack branch will be the number of letters between σ and the sign c in the first half of the input word.
Once the c is read, the branch perpetuating the recursion ceases to open up new branches, and instead transitions to q_e. All the other branches denote that they have moved to the second phase of the computation by transitioning to states q_σ^2. From this point onward, each branch waits to see the σ encoded in its state in the input. Once it does encounter σ, it can either ignore it and continue to look for another σ in the input, in case there are repetitions in w of the same letter, or it can guess that this is the σ which is the one in the right position and move to state q_e. After transitioning to q_e, if in fact σ was in the right position from the end, the sign # of the stack branch will be exposed exactly when the last input letter is read. At this point, it transits to the accept state. If all branches successfully guess their respective σ symbols then the computation will reach a configuration where all leaf nodes are labelled $(q_{ac}, \epsilon, \epsilon)$, which is an accepting configuration.

3.4.2 Equivalence with conjunctive grammars

In this paragraph we will show that conjunctive grammars and synchronized alternating pushdown automata are equivalent in power. Both are capable of describing the class of conjunctive languages. We will show how to convert any conjunctive grammar into a synchronized alternating pushdown automaton that recognizes the same language and vice versa. Recall that we defined a conjunctive language to be any language that can be described with a conjunctive grammar. Our goal in this paragraph is to show the following theorem, which is the equivalent of Theorem 2.4.1.

Theorem 3.4.1. *A language is conjunctive if and only if some synchronized alternating pushdown automaton recognizes it.*

As we often do in theorems which are 'if and only if', we will split it into the two following propositions.

Proposition 3.4.1. *If a language is conjunctive, then some synchronized alternating pushdown automaton recognizes it.*

Proof. Let L be a conjunctive language, then we know that L has a conjunctive grammar G which generates it. We will convert the conjunctive grammar G into a synchronized alternating pushdown automaton P. The proof of this proposition is similar to the proof of Proposition 2.4.1. The SAPDA P will accept an input w, if and only if the grammar G generates it. It will follow the grammar's rules. In order to do that, we will use the same mechanism with the one which we used on PDAs. Here, we have to notice that each leaf of the SAPDA is like a normal PDA. The only difference is that in the SAPDA model we can choose to split a node, but still each leaf node is like a normal PDA[7]. Again, like in the proof of Proposition 2.4.1, we will have three main states $q_{start}, q_{loop}, q_{accept}$ and ancillary ones. Which the only transition for q_{start}, will be $\delta(q_{start}, \epsilon, \epsilon) = \{(q_{loop}, \epsilon, S\#)\}$ [8] and there are not any transitions which starts from q_{accept}. The stack alphabet will be $\Gamma = \Sigma \cup N \cup \{c, d, \#\}$, where we assume that $c, d, \# \notin \Sigma \cup \Gamma$.
As every leaf is similar with a PDA, the only thing left to explain is what happens when we have to simulate a rule of the form $A \to u_1\& \dots \& u_k$, where $u_i \in (\Sigma \cup \Gamma)^*$ and $i = 1, \dots, k$. A first glimpse of how we will do that simulation is at Figure 3.2. We assume that P wants to simulate a rule of the form $A \to u_1\& \dots \& u_k$, for $k \geq 2$. Then, if the input is w, we will have a factorization of w, $w = w_1w_2$, where $w_1 \in L(A)$ and $w_2 \in L(u')$ if '$Au'\#$' is the whole string on the stack. Then P first pops A off the stack and goes to state p_A. State p_A guesses the length of w_2 and pushes into the stack the string $\underbrace{cc\cdots c}_{|w_2|}d$.
It continues with splitting the node with p_A, to nodes $p_3, \dots, p_{k+2}$ and p_1, without reading the input or changing the stack. Then p_1 guesses the length of w_1 and reads $|w_1|$ symbols from the input without doing something else. Then p_1 splits to two nodes, p_2 and q_{loop}. Node p_2 will have only these transitions $\delta(p_2, \sigma, c) = \{(p_2, \epsilon)\}$ and $\delta(p_2, \epsilon, d) = \{(q_{accept}, \epsilon)\}$, where $\sigma \in \Sigma$. So, state p_2 is responsible for checking if the guess of p_1 was correct. Also, the remaining w_2 continues normally in q_{loop}, after deleting the string '$cc \dots cd$' off the stack. Each leaf node with the state p_{i+2} represents the conjunct u_i, for $i = 1, \dots, k$. So, when p_A is slitted to nodes p_1 and $p_3, \dots p_{k+2}$, each p_{i+2} writes the string u_i on the top of the stack and continues normally by going to state q_{loop}. To sum the above, when P reads on the top of the stack a variable A, and wants to simulate the rule $A \to u_1\& \dots \& u_k$, it guesses a factorization of the input w, $w = w_1w_2$ and secures that A produces w_1. It can do that, by putting $|w_2|$ mark symbols c after A which ensure that A produces the first $|w| - |w_2| = |w_1|$ symbols of the input w. The rest transitions are defined the same as in Proposition 2.4.1. To conclude our proof, we must add to P the transitions $(q_{loop}, \sigma, c) \in \delta(q_{loop}, \epsilon)$, for $\sigma \in \Sigma$, and $(q_{accept}, \epsilon) \in \delta(q_{loop}, \epsilon, d)$.

□

[7]But we can't just split a node when we want to follow a rule with conjuncts. If we do so, we will do the mistake that SAPDA will do a computation on $AC\&BC$ when we want to compute $(A\&B)C$, which are not equal if we notice the rules $A \to a$, $B \to aa$ and $C \to a|aa$. Then, $L((A\&B)C) = \emptyset$ but $L(AC\&BC) = \{aaa\}$.

[8]We can push into the stack two symbols, with the help of an ancillary state.

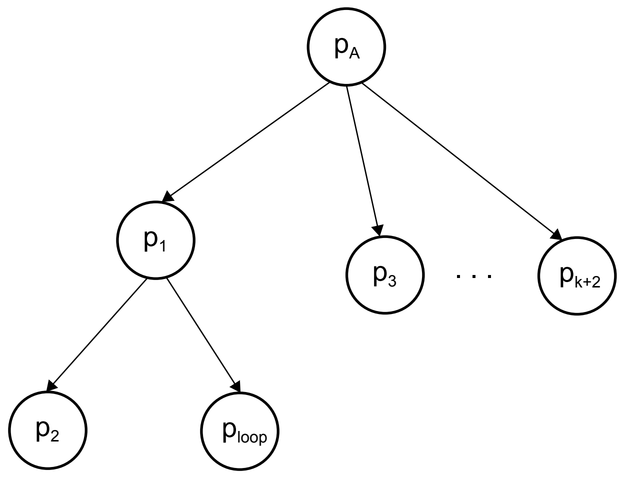

Figure 3.2: The mechanism for simulating a conjunctive rule.

We just showed how to convert a conjunctive grammar into a SAPDA. We will show now, how to convert a SAPDA into a conjunctive grammar. But before we give the next proposition we need to simplify our task. We can assume that every SAPDA P has an equivalence SAPDA P' which satisfies the following three conditions:

- It has a single accept state, 'q_{accept}'.
- It empties its stack before accepting.
- Every transition either pushes a symbol into the stack or pops one off the stack or splits one of its leaf nodes, but it doesn't do any of them at the same time.

Similarly with the PDAs, the first two conditions are shown easily. In the first one, we can put a new state q_{accept} which will be the only terminal state. We change the transition function and from the old terminal states we go to the new one without reading something from the input or from the stack. In the second one, we do the 'trick' to test if the stack is empty as we have already seen. We modify the transition function just before to accept and it goes to a state that empties the stack. Then, it goes to the accept state. We satisfy the third condition by replacing each transition which splits a node and at the same time pops or pushes a symbol into or off the stack with a two transition sequence that first splits and then pops or pushes a symbol. We can do that by introducing new states for each leaf node, which will remember the state that it was to go and the symbol that it was to be popped or pushed. Moreover, we modify the transition function by replacing each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state. We also replace each transition that neither pops nor pushes nor splits, with a transition sequence that pushes then pops an arbitrary stack symbol. Thus, in the rest of the paragraph we assume that every SAPDA satisfies the three previous conditions.

Proposition 3.4.2. *If a synchronized alternating pushdown automaton recognizes a language L, then L is conjunctive.*

Proof. In this proof we need to make a conjunctive grammar G, from a given synchronized alternating pushdown automaton P, which will produce the same language as the one which P accepts. This means, that we want for a word w, which synchronized alternating pushdown automaton P accepts, to be generated by the grammar G.
In order to do that, we will prove something more general. We will make a grammar G, that for every two states p, q and t stack symbol, we will have grammar's variables A^{+t}_{pq}, A^{-t}_{pq} and A^{ϵ}_{pq}. The variables A^{ϵ}_{pq} will generate all the strings that can take P from state p with an empty stack to state q with an empty stack[9]. If the superscript is $+t$, then the variables A^{+t}_{pq} will be generate all strings that can take P from state p with an empty stack to state q with the stack to contain only the symbol t. Similarly, the variable A^{-t}_{pq} will be generate all strings that can take P from state p with the stack to contain only the symbol t, to state q with an empty stack. We have to notice that A^{σ}_{pq}, $\sigma \in \{\epsilon, +t, -t\}$, can also generate strings that take P from p to q, regardless of the stack contents at p, leaving the stack at q in the same condition as it was at p with respect to σ.
Now, we will formally describe the grammar G. The set of the G's variables is $\{A^{\sigma}_{pq} \mid p, q \in Q \text{ and } \sigma \in \Gamma_{\epsilon}\ \}$. The G's start symbol is $A^{\epsilon}_{q_0, q_{accept}}$. We make the G's rules by the following description[10]:

[9]In the model of SAPDA the stack is a tree. So, when we say that an input x can take P from state p with empty stack to state q with an empty stack, we mean that the first and last configuration is the single node tree. The first one is labelled (p, u, ϵ) and the last one is labelled (q, u', ϵ), for some $u, u' \in \Sigma^*$. Even if we had configurations on the computation with multiple node trees, we can reach to the final configuration to be a single node tree after steps based on collapses.

[10]We have to notice that some rules, we do not need to put them here, because they are produced by others. Like the rule $A^{\epsilon}_{pq} \rightarrow aA^{-t}_{rq}$, which can be produced by $A^{+t}_{pq} \longrightarrow a$ and $A^{\epsilon}_{pq} \longrightarrow A^{+t}_{pr}A^{-t}_{rq}$. But, we give all of these rules to help us understand easier the purpose of every rule. Also, we have to keep in mind that the SAPDA P can do only one of pop,push or split at each step of a computation.

$$
\begin{array}{lll}
A^{\epsilon}_{pq} & \longrightarrow & A^{\epsilon}_{pr}A^{\epsilon}_{rq}, \text{ for all } p,r,q \in Q \\
A^{\epsilon}_{pq} & \longrightarrow & A^{+t}_{pr}A^{-t}_{rq}, \text{ for all } p,r,q \in Q \\
A^{+t}_{pq} & \longrightarrow & A^{\epsilon}_{pr}A^{+t}_{rq}, \text{ for all } p,r,q \in Q \\
A^{+t}_{pq} & \longrightarrow & A^{+t}_{pr}A^{\epsilon}_{rq}, \text{ for all } p,r,q \in Q \\
A^{-t}_{pq} & \longrightarrow & A^{\epsilon}_{pr}A^{-t}_{rq}, \text{ for all } p,r,q \in Q \\
A^{-t}_{pq} & \longrightarrow & A^{-t}_{pr}A^{\epsilon}_{rq}, \text{ for all } p,r,q \in Q \\
A^{\epsilon}_{pq} & \longrightarrow & aA^{\epsilon}_{rs}b, \text{ if } (r,t) \in \delta(p,a,\epsilon) \text{ and } (q,\epsilon) \in \delta(s,b,t) \\
A^{\epsilon}_{pq} & \longrightarrow & aA^{-t}_{rq}, \text{ if } (r,t) \in \delta(p,a,\epsilon) \\
A^{\epsilon}_{pq} & \longrightarrow & A^{+t}_{ps}b, \text{ if } (q,\epsilon) \in \delta(s,b,t) \\
A^{\epsilon}_{pq} & \longrightarrow & aA^{\epsilon}_{p_1q}\&\ldots\&aA^{\epsilon}_{p_kq} \text{ if } (p_1,\epsilon) \wedge \ldots \wedge (p_k,\epsilon) \in \delta(p,a,\epsilon) \\
A^{+t}_{pq} & \longrightarrow & aA^{+t}_{p_1q}\&\ldots\&aA^{+t}_{p_kq} \text{ if } (p_1,\epsilon) \wedge \ldots \wedge (p_k,\epsilon) \in \delta(p,a,\epsilon) \\
A^{-t}_{pq} & \longrightarrow & aA^{-t}_{p_1q}\&\ldots\&aA^{-t}_{p_kq} \text{ if } (p_1,\epsilon) \wedge \ldots \wedge (p_k,\epsilon) \in \delta(p,a,\epsilon) \\
A^{\epsilon}_{pp} & \longrightarrow & \epsilon, \text{ for all } p \in Q \\
A^{+t}_{pq} & \longrightarrow & a, \text{ if } (q,t) \in \delta(p,a,\epsilon) \\
A^{-t}_{pq} & \longrightarrow & a, \text{ if } (q,\epsilon) \in \delta(p,a,t)
\end{array}
$$

The only thing that we still have to do, is to explain why our construction works. We do that by proving the following two propositions: *If A^{ϵ}_{pq} generates x, then x can bring P from p with an empty stack to q with an empty stack. Also, the same holds for variables A^{+t}_{pq} and A^{-t}_{pq}, depend of $+$ or $-$ which have only the symbol t at the beginning or at the end of computation.* And: *If x can bring P from p with an empty stack to q with an empty stack, then A^{ϵ}_{pq} generates x. Also, if x can bring P from p with t the only symbol in the stack to state q with empty stack, then A^{-t}_{pq} generates x. Moreover, if x can bring P from p with empty stack to state q with t the only symbol in the stack, then A^{+t}_{pq} generates x.* We will prove both propositions with simultaneous induction on the number of steps in the derivation of x from A^{σ}_{pq}, $\sigma\{\epsilon, +t, -t\}$ and on the number of steps in the computation of P.

For the first proposition we have that, if the derivation has one step, then we have to use a rule which on the right-hand side contains no variables. The only rules in G where no variables occur on the right-hand side are $A^{\epsilon}_{pp} \to \epsilon$, $A^{+t}_{pq} \to a$ and $A^{-t}_{pq} \to b$, where $(q,t) \in \delta(p,a,\epsilon)$ and $(q,\epsilon) \in \delta(p,b,t)$ respectively. Thus, input ϵ takes P from p with an empty stack to p with an empty stack, input a takes P from p with an empty stack to q with t the only stack symbol and input b takes P from p with t the only stack symbol to q with an empty stack. Thus, the proposition holds for one step derivations.

If the derivation is of length of $k+1$, for $k \geq 1$, we assume that the proposition holds for derivations of length of at most k. Suppose that $A^{\epsilon}_{pq} \overset{k+1}{\Longrightarrow} x$. The first step of this derivation is one of the previous rules. We will see the proof only for two of them, similarly we can prove the others. We will see the proofs for the rules, $A^{\epsilon}_{pq} \Rightarrow A^{\epsilon}_{pr}A^{\epsilon}_{rq}$ and $A^{\epsilon}_{pq} \Rightarrow aA^{\epsilon}_{p_1q}\&\ldots\&aA^{\epsilon}_{p_kq}$.

In the first rule, we consider the factorization of x, $x = yz$, where A^{ϵ}_{pr} generates y and A^{ϵ}_{rq} generates z. Because both $A^{\epsilon}_{pr} \overset{*}{\Rightarrow} y$ and $A^{\epsilon}_{rq} \overset{*}{\Rightarrow} z$ are most of k steps, we have from induction hypothesis that y can bring P from p with an empty stack to r with an empty stack and z can bring P from r with an empty stack

to q with an empty stack. Thus, x can bring P from p with an empty stack to q with an empty stack, which it is what we want to show.
In the second rule, the first derivation is $A^{\epsilon}_{pq} \Rightarrow aA^{\epsilon}_{p_1q}\& \dots \&aA^{\epsilon}_{p_kq}$. We consider the factorization of x, $x = ay$. So, we have that each $A^{\epsilon}_{p_iq}$ produce y, for $i = 1, \dots, k$, where each of them are at most k derivations. From the induction hypothesis we have that the input y can bring P from p_i with empty stack to q with empty stack. Thus, each leaf of the computation of P on x ends with (q, ϵ, ϵ)[11], then it collapses to the parent node. Thus, input x can bring P from state p with an empty stack to state q with empty stack, as we want. Similarly we do for the other rules of G.
For the second proposition, we have that if the computation has zero steps, it starts and ends at the same state, say p. In zero step, P only has time to read the empty string, so $x = \epsilon$. By construction, G has the rule $A^{\epsilon}_{pp} \to \epsilon$. We know that P can't split a node and pop or push a symbol into or off the stack in one move. Then in one step, P can only push a symbol into its stack if we want to start with an empty stack and to finish with a symbol into the stack. Then, the only possible step is to read a character of the input and to push a symbol into the stack, then we have $(q, t) \in \delta(p, a, \epsilon)$. From the construction of G, we have the rule $A^{+t}_{pq} \to a$. Similarly for popping a symbol off the stack, we have the rule $A^{-t}_{pq} \to a$. Thus, the proposition holds for zero and one steps.
We assume that the proposition holds for computations of length at most k, where $k \geq 1$. We will prove it for computations of length $k + 1$. First, we suppose that P has a computation wherein x brings p with an empty stack to q with an empty stack in $k + 1$ steps. At the first step of a computation, either P pushes a symbol into the stack , or it splits the root node. We will see each case separately.
We assume that in the first step of the computation, P pushes a symbol into the stack. Then, we have a transition $(r, t) \in \delta(p, a, \epsilon)$, where $w = ay$. Moreover, we have that y can bring P from state r with t the only symbol in the stack to state q with an empty stack, at k steps. From the induction hypothesis we have that $A^{-t}_{rq} \overset{*}{\Rightarrow} y$, then $A^{\epsilon}_{pq} \overset{*}{\Rightarrow} w$ because from the construction of G we have the rule $A^{\epsilon}_{pq} \to aA^{-t}_{rq}$ if $(r, t) \in \delta(p, a, \epsilon)$. The proofs are the same when the computation starts with popping or pushing a symbol, if we want to show that w can bring P from state p with t the only symbol in the stack to state q with an empty stack, we have $A^{-t}_{pq} \overset{*}{\Rightarrow} w$ and if w can bring P from state p with an empty stack to state q with t the only symbol in the stack, we have $A^{+t}_{pq} \overset{*}{\Rightarrow} w$.
We assume now, that the computation splits its root node at the first step. Then, we have a transition $(p_1, \epsilon) \wedge \dots \wedge (p_k, \epsilon) \in \delta(p, a, \epsilon)$, where $w = ay$. Then, each leaf (p_i, y, ϵ) has an computation that accepts y. From the induction hypothesis we have that for each i, $A^{\epsilon}_{p_iq} \overset{*}{\Rightarrow} y$. From the rule, $A^{\epsilon}_{pq} \to aA^{\epsilon}_{p_1q}\& \dots \&aA^{\epsilon}_{p_kq}$, we

[11] Here, we assume that x is the hole input word and the computation on P starts with an empty stack. We have to notice that a similar explanation holds for x to be an initial substring of the input word or the computation on P to start with symbols on the stack. We are speaking for input x, it doesn't matter how the computation will continue, and we don't touch the symbols that may be in the stack initially. So, for each leaf we would have (q, u_1, u_2), where the remaining input u_1 is the same at each leaf as it is the stack content u_2.

have that $A^{\epsilon}_{pq} \overset{*}{\Rightarrow} w$. Similarly we have the proofs for $A^{-t}_{pq} \overset{*}{\Rightarrow} w$ and $A^{+t}_{pq} \overset{*}{\Rightarrow} w$, when the computation starts with splitting its node. $\square$

Chapter 4

Boolean Grammars

Boolean grammars generalize both context-free and conjunctive grammars. Boolean grammars were introduced by A. Okhotin in [15] and appear to posses many interesting properties. The semantics of Boolean grammars was defined later in [16], as it is more complicated than the semantics for context-free grammars or conjunctive grammars. The idea behind the semantics of Boolean grammars came from the area of logic programming. For a further study of Boolean grammars see [10, 15, 16, 17, 18].

4.1 Definition

Definition 4.1.1. *A Boolean grammar is a quadruple $G = (\Sigma, N, P, S)$, in which:*

- *Σ and N are disjoint finite non-empty sets of terminal and non-terminal symbols respectively*
- *P is a finite set of grammar rules, each of the form*

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n$$

  where $A \in N$, $m, n \geq 0$, $m + n \geq 1$ and $\alpha_i, \beta_j \in (\Sigma \cup N)^*$
- *$S \in N$ is a non-terminal designated as the start symbol.*

The semantics of a general Boolean grammar is far more complicated than the one for context-free grammars or conjunctive grammars. We will give semantics for the whole class of Boolean grammars in the next paragraph. For now we only consider a few simple cases. A string can be produced by a rule of a Boolean grammar if it can be produced by all the positive conjuncts of the rule and by none of the negative ones. This interpretation isn't sufficient to give meaning to grammars like $S \to \neg S$.

Example 4.1.1. *The following Boolean grammar generates the language* $\{ a^m b^n c^n \mid m, n \geq 0,\ m \neq n\}$*:*

$$\begin{array}{lcl} S & \longrightarrow & AB\&\neg DC \\ A & \longrightarrow & aA \mid \epsilon \\ B & \longrightarrow & bBc \mid \epsilon \\ C & \longrightarrow & cC \mid \epsilon \\ D & \longrightarrow & aDb \mid \epsilon \end{array}$$

The rules for the non-terminals A, B, C and D are context-free. We have that $L(AB) = \{a^m b^n c^n \mid m, n \in \mathbb{N}\}$ and $L(DC) = \{a^n b^n c^m \mid m, n \in \mathbb{N}\}$. Then $L(AB) \cap \overline{L(DC)} = \{ a^m b^n c^n \mid m, n \geq 0,\ m \neq n\}$.

Example 4.1.2. *The following Boolean grammar generates the language* $\{ww \mid w \in \{a, b\}^*\}$[1]*:*

$$\begin{array}{lcl} S & \longrightarrow & \neg AB\&\neg BA\&C \\ A & \longrightarrow & XAX \mid a \\ B & \longrightarrow & XBX \mid b \\ C & \longrightarrow & XXC \mid \epsilon \\ X & \longrightarrow & a \mid b \end{array}$$

Again, according to the intuitive semantics, the non-terminals A, B, C and X should generate the appropriate context-free languages, and

$$\begin{array}{r} L(A) = \{ uav \mid u, v \in \{a, b\}^*,\ |u| = |v|\} \\ L(B) = \{ ubv \mid u, v \in \{a, b\}^*,\ |u| = |v|\} \\ L(AB) = \{ uavxby \mid u, v, x, y \in \{a, b\}^*,\ |u| = |v|,\ |x| = |y|\} \end{array}$$

Then, $L(AB)$ contains all strings of even length with a mismatched a on the left and b on the right, in any position. Similarly, we have for the set $L(BA)$:

$$L(BA) = \{ ubvxay \mid u, v, x, y \in \{a, b\}^*,\ |u| = |v|,\ |x| = |y|\}$$

Now, $L(BA)$ specifies the mismatch formed by b on the left and a on the right. Then the rule for S specifies the set of strings of even length, from C, without such mismatches. Then, we have:

$$L(S) = \overline{L(AB)} \cap \overline{L(BA)} \cap \{aa, ab, ba, bb\}^* = \{ ww \mid w \in \{a, b\}^*\}$$

To verify this, we have to notice that if from non-terminal A produce the word w_1 and from B the word w_2, we have the following: if $|w_1| = 2i + 1$, the central a is in $i+1$ position of the word, then $|w_2| = 2(k-i-1)+1$ (when $|w_1 w_2| = 2k$) and we also have that the central b is in $(i + 1) + i + (k - i - 1) + 1 = k + i + 1$ position of the word $w_1 w_2$.
Notice that Proposition 3.2.1 can not be generalized to the case of Boolean grammars, as the following example illustrates on alphabet $\{a, b\}$:

[1]We have to remember that this language is the one which we tried to show that it's not conjunctive. Then if this is true, we will know that conjunctive languages is a proper subclass of the Boolean ones.

Example 4.1.3. $\begin{array}{lcl} S & \longrightarrow & \neg A \\ A & \longrightarrow & b \end{array}$

There exists an infinity set of strings of $L(S)$ that do not contain b as a substring (e.g. $a, aa, \ldots$).

4.2 Well-Founded Semantics for Boolean Grammars

In the previous paragraph we presented the definition of Boolean grammars. We now consider their semantics. Let G be the Boolean grammar with the unique rule $S \to \neg S$. What is the meaning of this grammar? For every string w we have the following problem: if $w \in L(G)$ then by the rule $S \to \neg S$ we have that $w \notin L(G)$ and similarly if $w \notin L(G)$ we will have that $w \in L(G)$. So, we reach to the conclusion that the classical definition of a language with just the relation '$\in$' is not enough to define the semantics of Boolean grammars. Then, we introduce the concept of 3-valued languages. Classical (2-valued) languages can also be considered as functions from Σ^* to $\{0, 1\}$, i.e. we have that $w \in L$ iff $L(w) = 1$. Then, we generalize this alternative way of classical definition of languages and we take 3-valued languages as follows[2]:

Definition 4.2.1. *A* 3*-valued language L over an alphabet Σ is a function from Σ^* to $\{0, \frac{1}{2}, 1\}$.*

Then, given a string $w \in \Sigma^*$ we have that $L(w) = 0$ if w doesn't belong to the language L, $L(w) = \frac{1}{2}$ means that we don't know if w belongs to the language L and finally if $L(w) = 1$ then w is an element of the language L. From now on all languages of this paragraph will be 3-valued ones. Since we have a different definition of the languages that we are going to use in this paragraph we have to re-define the basic operations on the 3-valued languages:

Definition 4.2.2. *Let L_1, L_2 be languages over an alphabet Σ. We define the* 3*-valued intersection of the languages L_1, L_2 to be the language $L_1 \cap L_2$ such that:*

$$(L_1 \cap L_2)(w) = \min\{L_1(w), L_2(w)\}$$

Definition 4.2.3. *Let L_1, L_2 be languages over an alphabet Σ. We define the* 3*-valued union of the languages L_1, L_2 to be the language $L_1 \cup L_2$ such that:*

$$(L_1 \cup L_2)(w) = \max\{L_1(w), L_2(w)\}$$

Definition 4.2.4. *Let L be a language over an alphabet Σ. We define the* 3*-valued complement of the language L to be the language $\overline{L}$ such that:*

$$\overline{L}(w) = 1 - L(w)$$

[2]Okhotin, in his lectures [27], has a different but equivalent definition of 3-valued languages. He defines them by a pair of languages (L, L'), where $L \subseteq L'$. The meaning of when a string w belongs to the language is that: If $w \in L$ then we have that w is in the language (L, L'), if $w \in L' - L$ then we don't know and if $w \notin L'$ it doesn't belong to the language.

The difference of two 3-valued languages L_1, L_2 can be defined as:

$$L_1 - L_2 = L_1 \cap \overline{L_2}$$

Definition 4.2.5. *Let $L_1, \ldots, L_n$ be languages over an alphabet Σ. We define the 3-valued concatenation of the languages $L_1, \ldots, L_n$ to be the language L such that:*

$$L(w) = \max_{\substack{(w_1,\ldots,w_n):\\ w=w_1\ldots w_n}} \big(\min_{1\leq i\leq n} L_i(w_i)\big)$$

The concatenation of $L_1, \ldots, L_n$ will be denoted by $L_1 \cdots L_n$.

Definition 4.2.6. *An interpretation I of a Boolean grammar $G = (\Sigma, N, P, S)$ is a function $I : N \to [\Sigma^* \to \{0, \frac{1}{2}, 1\}]$.*[3]

An interpretation I can be recursively extended to apply to expressions that appear as the right-hand sides of Boolean grammar rules:

Definition 4.2.7. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar and I be an interpretation of G. Then I can be extended to become a truth valuation $\hat{I}$ as follows:*

- *For every symbol $\alpha \in \Sigma_\epsilon$ and for all $w \in \Sigma^*$, it is $\hat{I}(\alpha)(w) = 1$ if $w = \alpha$ and 0 otherwise.*
- *For every symbol $A \in N$ and for all $w \in \Sigma^*$, it is $\hat{I}(A)(w) = I(A)(w)$*
- *Let $\alpha = \alpha_1 \ldots \alpha_n$, $n \geq 2$, be a sequence in $(\Sigma \cup N)^*$. Then, for every $w \in \Sigma^*$, it is $\hat{I}(\alpha)(w) = (\hat{I}(\alpha_1) \cdots \hat{I}(\alpha_n))(w)$*
- *Let $\alpha \in (\Sigma \cup N)^*$. Then, for every $w \in \Sigma^*$, $\hat{I}(\neg\alpha)(w) = 1 - \hat{I}(\alpha)(w)$*
- *Let $l_1, \ldots, l_n$ be conjuncts. Then, for every string $w \in \Sigma^*$, $\hat{I}(l_1 \& \ldots \& l_n)(w) = \min\{\hat{I}(l_1)(w), \ldots, \hat{I}(l_n)(w)\}$*

We are now in a position to define the notion of a model of a Boolean grammar:

Definition 4.2.8. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar and I an interpretation of G. Then, I is a model of G if for every rule $A \to l_1 \& \ldots \& l_n$ in P and for every $w \in \Sigma^*$, it is $\hat{I}(A)(w) \geq \hat{I}(l_1 \& \ldots \& l_n)(w)$.*

In the definition of the well-founded model, two orderings on interpretations play a crucial role, as they are taken from the area of logic programming. Given two interpretations, the first ordering (usually called the *standard ordering*) compares their degree of truth:

Definition 4.2.9. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar and I, J be two interpretations of G. Then, we say that $I \preceq J$ if for all $A \in N$ and for all $w \in \Sigma^*$, we have $I(A)(w) \leq J(A)(w)$.*

[3]We remind to the reader that by $[A \to B]$ we denote the set of functions f from A to B.

There is an interpretation, among all the others, of a given Boolean grammar which is the least with respect to the $\preceq$ ordering, we call that interpretation $\perp$ which for all A and all w, $\perp(A)(w) = 0$.
The second ordering, usually called the *Fitting ordering*, compares the degree of information of two interpretations:

Definition 4.2.10. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar and I, J be two interpretations of G. Then, we say that $I \preceq_F J$ if for all $A \in N$ and for all $w \in \Sigma^*$, if $I(A)(w) = 0$ then $J(A)(w) = 0$ and if $I(A)(w) = 1$ then $J(A)(w) = 1$.*

In this ordering ($\preceq_F$), the least interpretation among all others is $\perp_F$ which for all A and all $w \in \Sigma^*$, we have $\perp_F (A)(w) = \frac{1}{2}$.
Given a set U of interpretations, we will write $lub_{\preceq}U$ (respectively $lub_{\preceq_F}U$) for the least upper bound of the members of U under the standard ordering (respectively, the Fitting ordering).

Now, we have reached to a point that we can define the *well-founded semantics* for Boolean grammars. The basic idea here is that the intended model of the grammar is constructed in stages. These stages are related to the levels of negation that the grammar uses. But before we define the well-founded semantics for Boolean grammars, we need one last definition.

Definition 4.2.11. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar, let $\mathscr{I}$ be the set of all 3-valued interpretations of G and let $J \in \mathscr{I}$. The operator $\Theta_J : \mathscr{I} \to \mathscr{I}$ is defined as follows. For every $I \in \mathscr{I}$, for all $A \in N$ and for all $w \in S^*$:*

- $\Theta_J(I)(A)(w) = 1$ *if there is a rule $A \to l_1 \& \ldots \& l_n$ in P such that for every positive l_i it is $\hat{I}(l_i)(w) = 1$ and for every negative l_i it is $\hat{J}(l_i)(w) = 1$*
- $\Theta_J(I)(A)(w) = 0$ *if for every rule $A \to l_1 \& \ldots \& l_n$ in P, either there exists a positive l_i such that $\hat{I}(l_i)(w) = 0$, or there exists a negative l_i such that $\hat{J}(l_i)(w) = 0$*
- $\Theta_J(I)(A)(w) = \frac{1}{2}$, *otherwise.*

In the following theorem we will see that the operator Θ_J is monotonic with respect to the $\preceq$ ordering of interpretations. Moreover, it has a unique least fixed point:

Theorem 4.2.1. *Let G be a Boolean grammar and let J be an interpretation of G. Then, the operator Θ_J is monotonic with respect to the $\preceq$ ordering of interpretations. Moreover, Θ_J has a unique least, with respect to $\preceq$, fixed point $\Theta_J^{\uparrow\omega}$ which is defined as follows:*

$$\begin{array}{lcl} \Theta_J^{\uparrow 0} & = & \perp \\ \Theta_J^{\uparrow n+1} & = & \Theta_J(\Theta_J^{\uparrow n}) \\ \Theta_J^{\uparrow\omega} & = & lub_{\preceq}\{\, \Theta_J^{\uparrow n} \mid n \in \mathbb{N}\} \end{array}$$

The complete proof of the above theorem is given in [16]. Now, we will denote by $\Omega(J)$ the least fixed point of Θ_J. Given a grammar G, we can use the Ω operator to construct a sequence of interpretations whose ω-limit M_G will prove to be a distinguished model of G:

$$\begin{array}{lcl} M_0 & = & \perp_F \\ M_{n+1} & = & \Omega(M_n) \\ M_G & = & lub_{\preceq}\{\, M_n \mid n \in \mathbb{N}\} \end{array}$$

Theorem 4.2.2. *Let G be a Boolean grammar, then M_G is a model of G. Moreover, M_G is the least (with respect to the $\preceq_F$ ordering) fixed point of the operator Ω.*

The previous theorem gives us the uniqueness that we need for defining the well-founded model for a Boolean grammar. For a Boolean grammar G we will call M_G to be its well-founded model. As we mentioned before this way of semantics are form the area of Logic Programming, but we have a difference in our model. This difference is that in general logic programs the construction of the well-founded model may require a transfinite number of iterations which is greater than ω. In other words, the well-founded semantics of logic programs is not computable in the general case. However, in the case of Boolean grammars, the model is constructed in at most ω iterations.
Actually, it can be shown, with a similar reasoning as in [20], that the model M_G is the least model of G according to a syntax-independent relation. Moreover, it can be shown that if we have a Boolean grammar without negations or if we have acyclic negations (that means that there is no non-terminal which can reach to itself after some computational steps through negation) then we will have a 2-valued (values 0 and 1) well founded semantics, which will give values as it will be expected. That means that the well-founded semantics of a Boolean grammar with no negation agrees with the semantics that we gave for conjunctive grammars. The construction of the well-founded model is illustrated by the following example:

Example 4.2.1. *Let G be the grammar given in Example 4.1.2. Then, it is easy to see that $M_G = M_2$, i.e. in order to converge to the well-founded model of G we need exactly two iterations of Ω. More specifically, in $M_1 = \Omega(M_0)$ the denotations of the non-terminals A,B, C and X stabilize (notice that the definitions of these non-terminals are standard context-free rules). However, in order for the denotation of S to stabilize, an additional iteration of Ω is required. Notice that the language produced by this grammar is two-valued.*

Conclusion

In this thesis we have examined four interesting classes of formal languages. The two of them, namely regular and context-free ones, are well-known classes that have been widely studied in the literature [21, 22, 23, 24, 25]. The other two, namely conjunctive and Boolean languages, are more recent ones and appear to possess many interesting properties [1, 3, 15].
For each of the first three classes of formal languages that we have considered, we have also presented a corresponding equivalent formal automaton model. More specifically, as we have seen:

- Regular languages are produced by finite state automata
- Context-free languages are produced by pushdown automata
- Conjunctive languages are produced by synchronized alternating pushdown automata

It appears to be an interesting problem to devise a formal automaton model that captures exactly the class of Boolean languages. A possible idea would be to extend synchronized alternating pushdown automata with negation nodes, but of course this needs to be further investigated.
There are many other research problems regarding conjunctive and boolean languages, which appear to be not only interesting but also very challenging and non trivial. Some of these problems are surveyed in [10]. In the following we present the ones that we believe are the most interesting ones and which, if resolved, will offer a deeper insight to this new area of research:

1. Are conjunctive and Boolean grammars equivalent, or there exist a Boolean language which is not conjunctive?
2. Are conjunctive languages closed under complementation?
3. Is there a language, which is recognized by deterministic linear bounded automata working in time $O(n^2)$, that doesn't produced by a Boolean grammar?
4. Given an arbitrary 3-valued Boolean language L, does there always exist a 2-valued Boolean language L' such that L and L' agree on the set of strings that are assigned the value 1? In other words, are 3-valued Boolean grammars more expressive that 2-valued ones?

Closing, we believe that conjunctive and boolean grammars are very natural and interesting classes of formal grammars that deserve further investigation and development.

Bibliography

[1] A. Okhotin: Conjunctive Grammars. Journal of Automata, Languages and Combinatorics 6(4): 519-535 (2001)

[2] A. Okhotin: Conjunctive Grammars and Systems of Language Equations. Programming and Computer Software 28(5): 243-249 (2002)

[3] A. Okhotin: An overview of conjunctive grammars, Formal Language Theory Column. Bulletin of the EATCS 79: 145-163 (2003)

[4] A. Jeż: Conjunctive Grammars Can Generate Non-regular Unary Languages. Developments in Language Theory 2007: 242-253

[5] A. Jeż, A. Okhotin: Conjunctive Grammars over a Unary Alphabet: Undecidability and Unbounded Growth. Theory Comput. Syst. 46(1): 27-58 (2010)

[6] A. Jeż, A. Okhotin: Complexity of solutions of equations over sets of natural numbers. STACS 2008: 373-384

[7] A. Okhotin, P. Rondogiannis: On the expressive power of univariate equations over sets of natural numbers. IFIP TCS 2008: 215-227

[8] L. Y. Liu and P. Weiner: An Infinite Hierarchy of Intersections of Context-Free Language. Mathematical Systems Theory 7(2): 185-192 (1973)

[9] D. Wotschke: Nondeterminism and Boolean Operations in PDAs. J. Comput. Syst. Sci. 16(3): 456-461 (1978)

[10] A.Okhotin: Nine open problems on conjunctive and Boolean grammars. TUCS Technical Report No 794, Turku Centre for Computer Science, Turku, Finland, November 2006

[11] T. Aizikowitz, M. Kaminski: Conjunctive Grammars and Alternating Pushdown Automata. WoLLIC 2008: 44-55

[12] T. Aizikowitz, M. Kaminski: Linear Conjunctive Grammars and One-turn Synchronized Alternating Pushdown Automata. Formal Grammars 2009: Bordeaux, France, volume 5591 of Lecture Notes in Artificial Intelligence.

[13] J. Gruska: Complexity and Unambiguity of Context-Free Grammars and Languages Information and Control 18(5): 502-519 (1971)

[14] J. Gruska: Some Classifications of Context-Free Languages Information and Control 14(2): 152-179 (1969)

[15] A. Okhotin: Boolean grammars. Inf. Comput. 194(1): 19-48 (2004)

[16] V. Kountouriotis, C. Nomikos, P. Rondogiannis: Well-founded semantics for Boolean grammars. Inf. Comput. 207(9): 945-967 (2009)

[17] A. Okhotin: Unambiguous Boolean grammars. Inf. Comput. 206(9-10): 1234-1247 (2008)

[18] M. Wrona: Stratified Boolean Grammars. MFCS 2005: 801-812

[19] T. C. Przymusinski: Every Logic Program Has a Natural Stratification And an Iterated Least Fixed Point Model. PODS 1989: 11-21

[20] P. Rondogiannis, W. W. Wadge: Minimum model semantics for logic programs with negation-as-failure. ACM Trans. Comput. Log. 6(2): 441-467 (2005)

[21] M. Sipser: Introduction to the Theory of Computation. PWS publishing, 1996

[22] J. E. Hopcroft, J. D. Ullman: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 1979

[23] S. Ginsburg: The Mathematical Theory of Context-Free Languages. McGraw-Hill, New York, 1966

[24] H. R. Lewis, C. H. Papadimitriou: Elements of the Theory of Computation. Prentice-Hall, 1998

[25] Σ. Μποζαπαλίδης: Αυτόματα Μηχανές Γραμματικές.

[26] J. Hromkovic: Theoretical Computer Science. Springer, 2003

[27] A. Okhotin: Lectures of formal grammars course "$http://users.utu.fi/aleokh/formal\_grammars/$", 2009