Synchronized Alternating Pushdown Automata

Tamar Aizikowitz
Synchronized Alternating Pushdown Automata

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Tamar Aizikowitz

Submitted to the Senate of
the Technion — Israel Institute of Technology
Kislev 5770 Haifa November 2010
The research thesis was done under the supervision of Prof. Michael Kamin-
ski in the Computer Science Department.

First and foremost, I would like to thank my advisor, Prof. Michael Kamin-
ski. He was a thoughtful and insightful guide through the technical aspects
of my work, and a source of support through the many trials and tribula-
tions of graduate studies. I will miss our many ”business” coffee breaks and
weekly lunches together.

While I have had the great pleasure and privilege of learning from many
wonderful professors throughout my studies, two have been especially influ-
tential. The first is Prof. Nissim Francez who has played an integral part in
my research from the very beginning. I thank Nissim for his many helpful
comments and recommendations throughout the years. The second is Prof.
Shmuel Zaks, with whom I worked closely on the Automata and Formal
Languages course. I thank Shmuel for his invaluable advice on teaching,
research, and everything in between. I have made many friends during the
course of my graduate studies, all of which have enriched both my academic
and my personal life. In particular, I would like to thank Eitan Yaakobi,
Yaniv Carmeli, Sonya Liberman and Keren Censor for their unfailing friend-
ship and constant encouragement.

My greatest debt of gratitude I owe to my family, and in particular, to
my parents Nava and Jacob Aizikowitz. From a young age they invested in
my education, exposing me to art, culture, music and science. They both
set a very high standard, but managed to give me enough freedom to find
my own way. I thank them and my brother Elad for their endless love and
support.

Last but not least I would like to thank my husband, Sivan Bercovici.
I thank Sivan for his love and his friendship, for inspiring me through his
work ethic and enthusiasm, for supporting me through good times and bad,
and for cooking when I had deadlines. Our years together at the Technion
were an amazing adventure, and I look forward to all the adventures yet to
come.

The generous financial support of the Technion is gratefully acknowledged.
# Contents

Abstract 1

Abbreviations and Notations 2

1 Introduction 3

2 Conjunctive Grammars 7
   2.1 Conjunctive Grammar Model Definition ................. 7
   2.2 Linear Conjunctive Grammars .......................... 10
      2.2.1 LCG Model Definition ............................ 11
      2.2.2 Trellis Automata Model Definition ............... 12

3 Synchronized Alternating Pushdown Automata 14
   3.1 SAPDA Model Definition ................................. 14
   3.2 Equivalence Results .................................... 22
      3.2.1 Construction of an SAPDA from a CG ............... 23
      3.2.2 Construction of a CG from an SAPDA ............... 26

4 One-turn SAPDA and Linear Conjunctive Grammars 31
   4.1 Trace Grammars and Trace Automata ..................... 32
   4.2 One-turn SAPDA Model Definition ....................... 37
   4.3 Equivalence Results .................................... 38
      4.3.1 Constructing a one-turn SAPDA from a Linear CG ... 38
      4.3.2 Constructing a Linear CG from a one-turn SAPDA ... 41

5 Conjunctive Languages 51
   5.1 Placement in the Chomsky Hierarchy ..................... 51
      5.1.1 A Discussion of Mildly Context Sensitive Languages 53
List of Figures

2.1 Computation of Trellis Automaton on $\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$. . . . . . . 12

3.1 Example configuration of an SAPDA. . . . . . . . . . . . . . 16
3.2 Intermediate stage of a computation on $abbeccaaab$. The symbol $^\wedge$ denotes the current position in the input. . . . . . . . . 20
3.3 The configuration of the automaton in Figure 3.2. The denotation of the configuration is $((q_1, aab, b^\perp)(q_2, ecceab, bb^\perp))$. . 20
3.4 Stack tree contents after all branches have read $ab$. . . . . . . . . 22
3.5 Stack tree contents after all branches have read $abba$. . . . . 22

5.1 Relations between the classes of conjunctive languages (CL), linear conjunctive languages (LCL), context-free languages (CFL), and linear languages(LL). . . . . . . . . . . . . . . . 52

6.1 Configurations of the automaton $A_1$ (1), (2), (3), and (4) after reading $aab, aabab, aababb$ and $aababb$, respectively. . . . . . . 66
6.2 Configurations of the automaton $A_2$ (1), (2) and (3), after reading $aababb, aababb$ and $aababb$, respectively. . 66
6.3 Stack structure before merging branches, after merging the two leftmost heads in step 2d, and after applying a transition. 70
6.4 Derivation of full representation from compact one. . . . . . . 72
6.5 Partial construction of the canonical set of item-sets and valid $goto$ function. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
Abstract

Context-free languages combine expressiveness with polynomial parsing, making them very appealing for practical applications. In fact, they are possibly the most widely used class of languages in Computer Science. Thus, models of computation that slightly extend context-free models, without losing parsing efficiency, have great potential for applications in fields such as Programming Languages, Formal Verification, and Computational Linguistics, and are therefore of interest for theoretical research.

Conjunctive Grammars (CG) are an example of such a model. Introduced in 2001, they extend Context-free Grammars by adding explicit intersection rules, while retaining polynomial parsing. CG greatly resembles classical Context-free Grammars, making them more approachable to a wide audience, and a good candidate for practical applications. We present a new automaton model, Synchronized Alternating Pushdown Automata (SAPDA), which is the first automaton counterpart shown for CG.

The SAPDA model is a variation on Alternating Pushdown Automata that uses a limited form of synchronization to create localized parallel computations. The correlation between SAPDA and CG is analogous to the classical correlation between context-free grammars and PDA, making them a natural automaton counterpart for CG. We show that the correlation extends the one-turn sub-family of SAPDA, which is equivalent to the linear sub-family of CG, analogously to the classical equivalence between one-turn PDA and linear context-free grammars.

Furthermore, we define a notion of $LR(0)$ conjunctive grammars, and prove that they are equivalent to deterministic SAPDA. Through this equivalence we develop a linear-time parser for a strong class of languages that strictly includes the boolean closure of classical $LR(0)$ languages, thus extending previous linear-time parsing results.
Abbreviations and Notations

CG — Conjunctive Grammars
LCG — Linear Conjunctive Grammars
PDA — Pushdown Automata
DPDA — Deterministic Pushdown Automata
SAPDA — Synchronized Alternating Pushdown Automata
DSAPDA — Deterministic Synchronized Alternating Pushdown Automata

\(\sigma, \tau, \ldots\) — Terminal Symbols
\(u, v, w, \ldots\) — Terminal Words
\(A, B, \ldots X, Y, \ldots\) — Non-terminal Symbols
\(\alpha, \beta, \ldots\) — Non-Terminal Words
\(A, B, \ldots\) — Conjunctive formulas
Chapter 1

Introduction

Context-free languages lay at the very foundations of Computer Science, proving to be one of the most appealing language classes for practical applications. On the one hand, they are quite expressive, covering such syntactic constructs as necessary, e.g., for mathematical expressions. On the other hand, they are polynomially parsable, making them practical for real world applications. However, research in certain fields has raised a need for computational models that extend context-free models, without losing their computational efficiency.

Conjunctive Grammars (CG) are an example of such a model. Introduced by Okhotin in [14], CG are a generalization of context-free grammars that allow explicit conjunction operations in rules, thereby adding the power of intersection. CG were shown by Okhotin to accept all finite conjunctions of context-free languages, as well as some additional languages. Okhotin proved the languages generated by these grammars to be polynomially parsable [14, 20], making the model practical from a computational standpoint, and therefore, of interest for applications in various fields such as, e.g., programming languages.

Alternating automata models were first introduced by Chandra, Kozen and Stockmeyer in [5]. In these models, computations alternate between existential and universal modes of acceptance. Thus, for a word to be accepted it must meet both disjunctive and conjunctive conditions. In the case of Alternating Finite State Automata and Alternating Turing Machines, the alternating models have been shown to be equivalent in expressive power to their non-alternating counterparts, see [5].
Alternating Pushdown Automata (APDA) were also introduced in [5] and were further explored in [13]. Like Conjunctive Grammars, APDA add the power of conjunction over context-free languages. Therefore, unlike Finite Automata and Turing Machines, here alternation increases the expressiveness of the model. In fact, APDA accept exactly the exponential-time languages, see [5, 13], and are thus too strong to be a counterpart for CG.

A synchronized version of Alternating Finite State Automata was introduced in [26], and further explored and refined in [11]. As with non-synchronized alternating finite state automata, the alternation does not add to the expressive power of the model. However, these automata were found to be useful for reasoning about semi-extended regular expressions.¹

The Visibly Pushdown Automaton model, introduced in [4], added a different type of synchronization to the classical pushdown model. In this model, push and pop stack manipulations are synchronized with the input character read. Visibly pushdown automata have proven to be quite useful in modeling systems with calls and returns. While they are closed under intersection, they are too weak to be a counterpart for CG.

Our first contribution, is the introduction of a new alternating automata model, Synchronized Alternating Pushdown Automata (SAPDA) [1]. SAPDA are a weakened version of APDA that accept intersections of context-free languages. They can also be thought of as a pushdown counterpart for synchronized alternating finite state automata, up to differences in the formalization of the model. In particular, we prove that SAPDA are equivalent to CG², making them the first automaton counterpart shown for CG. The formulation of an automaton counterpart model lends new intuition to Conjunctive Grammars, as well as simplifies some existing proofs.

In [14], Okhotin defined a sub-family of Conjunctive Grammars called Linear Conjunctive Grammars (LCG), analogously to the definition of Linear Grammars as a sub-family of Context-free Grammars. LCG are an interesting sub-family of CG as they have especially efficient parsing algorithms, see [18], making them particularly appealing from a computational standpoint. Also, many of the interesting languages derived by Conjunctive Grammars, can in fact be derived by Linear Conjunctive Grammars. In [21], Okhotin proved that LCG are equivalent to a type of Trellis Automata, [8].

¹Semi-extended regular expressions contain an explicit operator for intersection.
²We call two models equivalent if they accept/generate the same class of languages.
It is a well-known result, due to Ginsburg and Spanier [6], that Linear Grammars are equivalent to one-turn PDA. One-turn PDA are a sub-family of pushdown automata, where in each computation the stack height switches only once from non-decreasing to non-increasing. That is, once a transition replaces the top symbol of the stack with $\epsilon$, all subsequent transitions may write at most one character.

Our second contribution is the introduction of a sub-family of SAPDA, one-turn Synchronized Alternating Pushdown Automata, and the proof that one-turn SAPDA are equivalent to Linear Conjunctive Grammars [2, 3]. The equivalence is analogous to the classical equivalence between one-turn PDA and Linear Grammars. This result greatly strengthens the claim of SAPDA as a natural automaton counterpart for Conjunctive Grammars.

Deterministic context-free languages are a sub-family of context-free languages that can be accepted by a deterministic PDA. In [9], Knuth introduced the notion of $LR(k)$ grammars, and proved their equivalence to deterministic PDA. Through this equivalence, he developed a linear time LR parsing algorithm for deterministic context-free languages, which quickly became the basis of modern-day compilation theory. Furthermore, Knuth proved that $LR(0)$ languages (those that can be parsed with no lookahead) are equivalent to deterministic PDA that accept by empty stack.

In [15], Okhotin presented an extension of Tomita’s Generalized LR parsing algorithm [23] for CG. The algorithm utilizes non-deterministic LR parsing, and works for all conjunctive grammars in polynomial time. When applied to the boolean closure of deterministic context-free languages, the run-time is linear.

Our third contribution is the introduction of a sub-family of SAPDA, Deterministic Synchronized Alternating Pushdown Automata, and a sub-family of CG, $LR(0)$ Conjunctive Grammars. We show that these sub-families are equivalent, analogously to the classical case. Through this equivalence we show a deterministic linear time parsing algorithm for $LR(0)$ conjunctive languages. This class of languages is stronger than classical $LR(0)$, strictly containing all finite intersections of $LR(0)$ languages. Moreover, the algorithm forms a basis for linear parsing of the boolean closure of $LR(0)$ languages. As such, it could prove interesting for potential applications in, e.g., Programming Languages and Natural Language Parsing.

The thesis is organized as follows. In Chapter 2 we recall the definition of
Conjunctive Grammars and their linear sub-family. In Chapter 3 we present our SAPDA model, and prove its equivalence to CG. In Chapter 4 we define the notion of one-turn SADPA and prove their equivalence to LCG. Chapter 5 is an overview of conjunctive languages, specifically their relation to other known language classes, closure properties and decision problems, and in particular, a short survey of parsing algorithms. Chapter 5 concludes with a practical example motivating the exploration of conjunctive languages. In Chapter 6 we present our \( LR(0) \) parsing algorithm for conjunctive languages, which is based on deterministic SAPDA. Finally, Chapter 7 contains some concluding remarks and future directions.
Chapter 2

Conjunctive Grammars

In this chapter we discuss Conjunctive Grammars, which forms the basis for our work. Conjunctive Grammars are an interesting class of grammars as they add intersection over context-free languages, while still maintaining the polynomiality of the languages generated. While CG are by no means the first, or only, extended context-free grammar model, they have the important advantage of resembling the classical well known context-free grammar model. This resemblance makes them accessible to a wide audience that extends beyond formal languages experts, thus making them viable candidates for practical applications.

In Section 2.1 we recall the model definition for Conjunctive Grammars. In Section 2.2, we focus on the sub-family of Linear Conjunctive Grammars and their relation to Trellis Automata.

2.1 Conjunctive Grammar Model Definition

The following definitions are from [14].

Definition 2.1. A Conjunctive Grammar is a quadruple $G = (V, \Sigma, P, S)$, where

- $V$ is a finite set of non-terminal symbols (variables).
- $\Sigma$ is a finite set of terminal symbols disjoint from $V$.
- $S \in V$ is the designated start symbol.

• $P$ is a finite set of rules of the form $A \rightarrow (\alpha_1 \& \cdots \& \alpha_n)$ such that $A \in V$ and $\alpha_i \in (V \cup \Sigma)^*$, $i = 1, \ldots, n$. If $n = 1$ then we write $A \rightarrow \alpha$.

**Definition 2.2.** Conjunctive Formulas over $V \cup \Sigma \cup \{(,),\&\}$ are defined by the following recursion.

• $\epsilon$ is a conjunctive formula.

• Every symbol in $V \cup \Sigma$ is a conjunctive formula.

• If $A$ and $B$ are formulas, then $AB$ is a conjunctive formula.

• If $A_1, \ldots, A_n$ are formulas, then $(A_1 \& \cdots \& A_n)$ is a conjunctive formula.

**Definition 2.3.** Let $A$ be a conjunctive formula such that $A = (A_1 \& \cdots \& A_n)$, $n \geq 2$. We call each $A_i$, $i = 1, \ldots, n$, a conjunct of $A$, and we call $A$ the enclosing formula. If $A_i$ contains no $\&$ signs, then it is called a simple conjunct.

**Definition 2.4.** For a CG $G$, the relation of immediate derivability, $\Rightarrow_G$, on the set of conjunctive formulas is defined as follows.

1. $s_1 A s_2 \Rightarrow_G s_1(\alpha_1 \& \cdots \& \alpha_n)s_2$, for $A \rightarrow (\alpha_1 \& \cdots \& \alpha_n) \in P$;

and

2. $s_1 (w\& \cdots \&w) s_2 \Rightarrow_G s_1 w s_2$, for $w \in \Sigma^*$,

where $s_1, s_2 \in (V \cup \Sigma \cup \{(,),\&\})^*$. As usual, $\Rightarrow_G^*$ is the reflexive and transitive closure of $\Rightarrow_G$.

**Definition 2.5.** The language $L(A)$ of a conjunctive formula $A$ is defined as

$$L(A) = \{w \in \Sigma^* \mid A \Rightarrow_G^* w\}.$$ 

The language $L(G)$ of a conjunctive grammar $G$ is defined as

$$L(G) = L(S) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}.$$ 

We refer to 1 as production and 2 as contraction rules, respectively.

---

1Note that this definition is different from Okhotin’s definition in [14].

2By this definition, conjunctive formulas that are elements of $(V \cup \Sigma)^*$, are not simple conjuncts or enclosing formulas.
In particular, note that a terminal word \( w \) is derived from a conjunctive formula \( (A_1 \& \cdots \& A_n) \) if and only if it is derived from each \( A_i \), \( i = 1, \ldots, n \). Hence, \( (A_1 \& \cdots \& A_n) \) derives the finite intersection of the languages \( L(A_i) \), i.e.,

\[
L((A_1 \& \cdots \& A_n)) = \bigcap_{i=1}^{n} L(A_i) .
\]

**Definition 2.6.** A language \( L \subseteq \Sigma^* \) is a **conjunctive** language if there is a conjunctive grammar \( G \) such that \( L(G) = L \).

**Example 2.7.** ([14, Example 1]) The following conjunctive grammar generates the non-context-free language \( \{a^n b^n c^n : n = 0, 1, \ldots\} \), called the multiple agreement language. \( G = (V, \Sigma, P, S) \), where

- \( V = \{S, A, B, C, D\} \), \( \Sigma = \{a, b, c\} \), and
- \( P \) consists of the following rules.

\[
\begin{align*}
S & \rightarrow (A \& C) \\
A & \rightarrow aA \mid B \\
B & \rightarrow bBc \mid \epsilon \\
C & \rightarrowCc \mid D \\
D & \rightarrow aDb \mid \epsilon
\end{align*}
\]

The intuition of the above derivation rules is as follows.

\[
L(A) = \{a^m b^n c^n : m, n = 0, 1, \ldots\} ,
\]

\[
L(C) = \{a^m b^m c^n : m, n = 0, 1, \ldots\} ,
\]

and

\[
L(G) = L(C) \cap L(A) = \{a^n b^n c^n : n = 0, 1, \ldots\} .
\]

For example, the word \( aabbcc \) can be derived as follows.

\[
\begin{align*}
S & \Rightarrow (A \& C) \Rightarrow (AA \& C) \Rightarrow (aA \& C) \Rightarrow (aB \& C) \Rightarrow (aBc \& C) \\
& \Rightarrow (aabbBc \& C) \Rightarrow (aabb & C) \Rightarrow (aBc \& C) \\
& \Rightarrow (aabb & Cc) \Rightarrow (aBc \& Dcc) \Rightarrow (aBc \& aDbcc) \\
& \Rightarrow (aBc \& aDbcc) \Rightarrow (aBc \& aBc) \Rightarrow aBc
\end{align*}
\]
Example 2.7 demonstrates how conjunctive grammars can easily derive any finite conjunction of context-free languages. Following is an especially interesting example, due to Okhotin, of a conjunctive grammar that uses recursive conjunctions to derive a language that cannot be obtained by a finite conjunction of context-free languages.

**Example 2.8.** ([14, Example 2]) The following conjunctive grammar derives the non-context-free language \{w$w \mid w \in \{a, b\}^*\}, called *reduplication* with a center marker. \( G = (V, \Sigma, P, S) \), where

- \( V = \{S, A, B, C, D, E\} \), \( \Sigma = \{a, b, \$\} \), and
- \( P \) consists of the following derivation rules.

\[
S \rightarrow (C & D) \\
C \rightarrow aCa \mid aCb \mid bCa \mid bCb \mid \$ \\
D \rightarrow (aA & aD) \mid (bB & bD) \mid $E \\
A \rightarrow aAa \mid aAb \mid bAa \mid bAb \mid $Ea \\
B \rightarrow aBa \mid aBb \mid bBa \mid bBb \mid $Eb \\
E \rightarrow aE \mid bE \mid \epsilon
\]

The non-terminal \( C \) verifies that the lengths of the words before and after the center marker \( \$ \) are equal. The non-terminal \( D \) generates the language \( \{w$uw \mid u, w \in \{a, b\}\} \). The grammar languages is the intersection of these two languages, and is therefore, in fact, the reduplication with a center marker language. For a more detailed description, see [14, Example 2]. In Section 3.1, we construct an SAPDA for the reduplication with a center marker language, see Example 3.8.

**Remark 2.9.** It is an open question whether the reduplication language *without* a center marker, i.e., \( \{ww \mid w \in \Sigma^*\} \), is conjunctive.

### 2.2 Linear Conjunctive Grammars

Okhotin introduced in [14] a sub-family of conjunctive grammars called *Linear Conjunctive Grammars* (LCG). The definition of LCG is analogous to the definition of Linear Grammars as a sub-family of Context-free Grammars. In [21] Okhotin proved that LCG are equivalent to a type of automata.
called Trellis Automata. This equivalence enabled several simplified proofs of closure properties for LCG. However, there is no known extension of Trellis automata that are equivalent to full CG.

### 2.2.1 LCG Model Definition

**Definition 2.10.** A conjunctive grammar $G = (V, \Sigma, P, S)$ is said to be **linear** if all rules in $P$ are in one of the following forms.

- $A \rightarrow (u_1B_1v_1 \& \cdots \& u_nB_nv_n); u_i, v_i \in \Sigma^*$ and $A, B_i \in V$, or
- $A \rightarrow w; w \in \Sigma^*$, and $A \in V$.

**Definition 2.11.** A language $L \subseteq \Sigma^*$ is called a **linear conjunctive** language if there exists a linear conjunctive grammar $G$ such that $L(G) = L$.

Several interesting languages can be generated by LCGs. In particular, the grammars in Examples 2.7 and 2.8 are linear. Another example of a classical non-context-free language that is linear conjunctive is the following.

**Example 2.12.** The following linear conjunctive grammar derives the non-context-free **cross-agreement** language $\{a^nb^nc^n d^m | n, m \in \mathbb{N}\}$. $G = (V, T, P, S)$ where:

- $V = \{S, A, B, C, D, X, Y\}$, $T = \{a, b, c, d\}$,
- $P$ contains the following derivation rules:
  
  $S \rightarrow (A \& D)$
  $A \rightarrow aA \mid X$ ; $D \rightarrow Dd \mid Y$
  $X \rightarrow bXd \mid C$ ; $Y \rightarrow aYe \mid B$
  $C \rightarrow cC \mid \epsilon$ ; $B \rightarrow bB \mid \epsilon$

The non-terminal $A$ derives words of the form $a^ib^nc^jd^m$ for $m, i, j \in \mathbb{N}$, while $D$ derives words of the form $a^nb^ic^jd^i$ for $n, i, j \in \mathbb{N}$. Therefore, the conjunction of the two generates the cross-agreement language.
2.2.2 Trellis Automata Model Definition

Okhotin proved in [21] that LCG are equivalent to Trellis Automata. Triangular (real-time, homogenous) Trellis Automata, which we will refer to just as Trellis Automata, are a particular case of Systolic Trellis Automata [8], in which the connections between nodes form the shape of a triangle, see Figure 2.1. The input string is loaded from the bottom, and the node values of each level are computed based on the values of the two ancestors in the level below. Acceptance is determined by the value of the topmost node. The following definition of Trellis Automata is taken from [22], and is an extension of the standard definition that can handle the case of $\epsilon$.

**Definition 2.13.** A Trellis Automaton is a tuple $A = (\Sigma, Q, I, \delta, F, e)$ where:

- $\Sigma$ is the input alphabet
- $Q$ is a finite set of states
- $I : \Sigma \rightarrow Q$ sets the initial state assignments (for the bottom level)
- $\delta : Q \times Q \rightarrow Q$ is the transition function
- $F \subseteq Q$ is a set of accepting states (relevant for the top level only)
- $e \in \{0, 1\}$ is a bit that determines whether $\epsilon$ is in the language or not
Definition 2.14. Given an input string $\sigma_1 \cdots \sigma_n \in \Sigma^+$, we label each node of the trellis automaton by the substring $\sigma_i \cdots \sigma_j$, which corresponds to the sub-tree of the node (with arrows reversed). The value (state) of a node $w$, $\Delta(w) \in Q$, is determined as follows:

- $\Delta(\sigma_i) = I(\sigma_i)$
- $\Delta(\sigma_i \cdots \sigma_j) = \delta(\Delta(\sigma_i \cdots \sigma_{j-1}), \Delta(\sigma_{i+1} \cdots \sigma_j))$

The language of a trellis automaton is defined as follows:

$$L(A) = \{ w \in \Sigma^+ \mid \Delta(w) \in F \} \cup \{ \epsilon \mid e = 1 \}.$$
Chapter 3

Synchronized Alternating Pushdown Automata

Since their introduction in 2001, many interesting papers have been published on Conjunctive Grammars. However, no automaton counterpart had been introduced. Our first contribution is to suggest a counterpart automaton model for Conjunctive Grammars, [1]. We define a class of automata called Synchronized Alternating Pushdown Automata (SAPDA) as a variation on the standard PDA model.

An automaton model is an important component of the theoretical framework for conjunctive languages, and is therefore a significant addition to the field. Moreover, as we will see in the following chapters, the automaton model lends additional intuition and insight to the language class, provides alternate (and in some cases simplified) proofs, and perhaps most importantly, forms the basis of an LR parsing algorithm for a strong class of languages.

In Section 3.1 we present the formal definition of an SAPDA, and in Section 3.2 we prove that they accept exactly the conjunctive languages, and are therefore equivalent to CG.

3.1 SAPDA Model Definition

In the following section we present a development of the SAPDA model definition that was presented in [1]. SAPDA are an extension of PDA,
and similarly to Alternating Pushdown Automata [5, 13], SAPDA have the power of both existential and universal choice. Instead of basing our definition on existential and universal state sets, as presented in [5], we use a different formulation of alternation. In our model, transitions are made to a conjunction of states. The model is non-deterministic, therefore, several different conjunctions of states may be possible from a given configuration.\footnote{This type of formulation is standard in the field of Formal Verification, e.g., see [10].} If all conjunctions are of one state only, the automaton is a standard PDA.

The stack memory of an SAPDA is a tree. Each leaf has a processing head that 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.\footnote{This is similar to the concept of a transition from a universal state in the standard formulation of alternating automata, as all branches must accept.} The branches process the input independently, however sibling branches must empty synchronously, after which the computation continues from the parent branch.

**Definition 3.1.** A synchronized alternating pushdown automaton is a tuple $A = (Q, \Sigma, \Gamma, \delta, q_0, \bot)$, where $\delta$ is a function that assigns to each element of $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ a finite subset of

$$\{(q_1, \alpha_1) \land \cdots \land (q_k, \alpha_n) \mid n = 1, 2, \ldots, q_i \in Q \text{ and } \alpha_i \in \Gamma^*, i = 1, \ldots, n\}.$$  

Everything else is defined as in the standard PDA model. Namely,

- $Q$ is a finite set of states,
- $\Sigma$ and $\Gamma$ are the input and the stack alphabets, respectively,
- $q_0 \in Q$ is the initial state, and
- $\bot \in \Gamma$ is the initial stack symbol,

see, e.g., [7, pp. 107–112].

We describe the current stage of the automaton computation as a labeled tree. The tree encodes the stack contents, the current states of the stack-branches, and the remaining input to be read for each stack-branch. States and remaining inputs are saved in leaves only, as these encode the stack-branches currently processed.
Definition 3.2. A \textit{configuration} of an SAPDA is a labeled tree. Each internal node is labeled $\alpha \in \Gamma^*$ denoting the stack-branch contents, and each leaf node is labeled $(q, w, \alpha)$, where

- $q \in Q$ is the current branch state,
- $w \in \Sigma^*$ is the remaining input to be read of the branch, and
- $\alpha \in \Gamma^*$ is the branch contents.

Each node in an automaton configuration describes a part of the stack called a \textit{branch}. The branches that correspond to the leaf nodes are called \textit{active} branches, as they are the branches currently being processed. We define the following syntax for denoting automaton configurations.

Definition 3.3. Let $T$ be a tree configuration of an SAPDA automaton. We define the \textit{denotation}, $d(T)$, of $T$ by the following recursion.

- If $T$ consists of a single node labeled $(q, w, \gamma)$ then $d(T) = (q, w, \gamma)$.
- If $T$ consists of a root node labeled $\gamma$ with $n$ sub-trees $T_1, \cdots, T_n$ then $d(T) = (d(T_1) \land \cdots \land d(T_n))\gamma$.

For example, Figure 3.1 shows a configuration of an SAPDA, and

$((q_1, aab, B\bot) \land (q_2, cccaab, BB\bot))AAB$

is the denotation of the configuration. Henceforth, we use \textit{configuration} to refer both to the configuration tree and to the denotation of the configuration.

Figure 3.1: Example configuration of an SAPDA.
As all active branches in an automaton are independent, the order in which they are processed is irrelevant. Examples of possible modes of computation are processing the branches in a random order, processing in a specified order, e.g., left-to-right, or processing them in parallel. Following is a definition of the criteria any valid computational model must meet.

**Definition 3.4.** A *mode of computation* for an SAPDA is the order in which branches are processed during the computation. A mode of computation is *valid* if the following hold.

- A transition can be applied only to an active branch.
- If a branch empties, it cannot be chosen for the next transition (because it has no top symbol).
- If all sibling branches are empty, and each branch emptied with the *same* remaining input (i.e., after processing the same portion of the input) and with the same state, i.e., they are synchronized, the branches are collapsed back to the parent branch.

Note that the branches do not necessarily need to read the input together, but they must synchronize on the next step of the computation, i.e., the next state and the current position in the input, before they can be collapsed. To match the grammar derivations, we consider a mode of computation where the next branch to be processed can be any active branch.

**Definition 3.5.** Let $A = (Q, \Sigma, \Gamma, q_0, \delta, \perp)$ be an SAPDA. Be define the relation *yields* on the configuration denotations of $A$, denoted $\vdash_A$ as follows.

1. $s_1 (q, \sigma w, X \gamma) \mathrel{\vdash} s_2 \mathrel{\vdash} s_1 ((q_1, w, \gamma_1) \land \cdots \land (q_n, w, \gamma_n)) \gamma s_2$
   for $(q_1, \gamma_1) \land \cdots \land (q_n, \gamma_n) \in \delta(q, \sigma, X)$, and
2. $s_1 ((q, w, \epsilon) \land \cdots \land (q, w, \epsilon)) \gamma s_2 \mathrel{\vdash} s_1 (q, w, \gamma) s_2$,

where $\gamma \in \Gamma^*$, and $s_1, s_2 \in (\{ (, ) , \land , , \} \cup Q \cup \Gamma \cup \Sigma)^*$. We refer to (1) as the application of a transition, and to (2) as the collapsing of sibling branches. As usual, we denote by $\vdash_A^\ast$ the reflexive and transitive closure of $\vdash_A$.
For example, if \((q_3, AB) \land (q_4, AA) \in \delta(q_1, a, C)\) then 
\[((q_1, ab, CD) \land (q_2, b, D))EF \vdash \langle((q_3, b, AB) \land (q_4, b, AA)D) \land (q_2, b, D)\rangle EF\),
and
\[((q, ab, e) \land (q, ab, e))AB \land (p, b, BB)) \vdash \langle(q, ab, AB) \land (p, b, BB)\rangle\).

**Definition 3.6.** Let \(A\) be an SAPDA and let \(w \in \Sigma^*\).

- The *initial* configuration of \(A\) on \(w\) is the configuration \((q_0, w, \bot)\).
- An *accepting* configuration of \(A\) is a configuration \((q, \epsilon, \epsilon)\) for some \(q \in Q\).
- A *computation* of \(A\) on \(w\) is a sequence of configurations \(T_0, \ldots, T_n\), where
  - \(d(T_0)\) is the initial configuration,
  - \(d(T_{i-1}) \vdash_A d(T_i)\) for \(i = 1, \ldots, n\), and
  - all the leaves in \(T_n\) are labeled \((q, \epsilon, \gamma)\), i.e., all branches have finished reading the input.
- An *accepting* computation of \(A\) on \(w\) is a computation whose last configuration \(T_n\) is accepting.

The language \(L(A)\) of \(A\) is the set of all \(w \in \Sigma^*\) such that \(A\) has an accepting computation on \(w\).\(^3\)

**Example 3.7.** The SAPDA \(A = (Q, \Sigma, \Gamma, \delta, q_0, \bot)\) defined below accepts the non-context-free language
\[
\{w : |w|_a = |w|_b = |w|_c\}.
\]

- \(Q = \{q_0, q_1, q_2\}\),
- \(\Sigma = \{a, b, c\}\).

\(^3\)Alternatively, one can extend the definition of \(A\) with a set of *accepting* states \(F \subseteq Q\), and define collapsing and acceptance by accepting states, similarly to the classical definition. It can readily be seen that such an extension results in an equivalent model of computation.

18
• $\Gamma = \{\bot, a, b, c\}$, and

• $\delta$ is defined as follows.
  
  $\delta(q_0, \epsilon, \bot) = \{(q_1, \bot) \land (q_2, \bot)\}$,
  $\delta(q_1, \sigma, \bot) = \{(q_1, \sigma\bot)\}, \sigma \in \{a, b\},$
  $\delta(q_2, \sigma, \bot) = \{(q_2, \sigma\bot)\}, \sigma \in \{b, c\},$
  $\delta(q_1, \sigma, \sigma) = \{(q_1, \sigma\sigma)\}, \sigma \in \{a, b\},$
  $\delta(q_2, \sigma, \sigma) = \{(q_2, \sigma\sigma)\}, \sigma \in \{b, c\},$
  $\delta(q_1, \sigma', \sigma'') = \{(q_1, \epsilon)\}, (\sigma', \sigma'') \in \{(a, b), (b, a)\},$
  $\delta(q_2, \sigma', \sigma'') = \{(q_2, \epsilon)\}, (\sigma', \sigma'') \in \{(b, c), (c, b)\},$
  $\delta(q_1, c, X) = \{(q_1, X)\}, X \in \{\bot, a, b\},$
  $\delta(q_2, a, X) = \{(q_2, X)\}, X \in \{\bot, b, c\},$ and
  $\delta(q_i, \epsilon, \bot) = \{(q_0, \epsilon)\}, i = 1, 2.$

The first step of the computation opens two branches, one for verifying that the number of $a$s in the input word equals to the number of $b$s, and the other for verifying that the number of $b$s equals to the number of $c$s. If both branches manage to empty their stack then the word is accepted.

Figure 3.2 shows the contents of the stack tree at an intermediate stage of a computation on the word $abbcccaab$. The left branch has read $abbccc$ and indicates that one more $b$s than $a$s have been read, while the right branch has read $abb$ and indicates that two more $b$s than $c$s have been read. Figure 3.3 shows the configuration corresponding the above computation stage of the automaton.

We now consider the following example of an SAPDA that accepts the non-context-free language $\{w\$uw : w, u \in \{a, b\}\}$. Note that the intersection of this language with $\{u$v : u, v \in \{a, b\} \land |u| = |v|\}$ is the reduplication with a center marker language. As the latter language is context-free, and SAPDA are closed under intersection, the construction can easily be modified to accept the reduplication language.

The example is of particular interest as it showcases the model’s ability to utilize recursive conjunctive transitions, allowing it to accept languages that are not finite intersections of context-free languages. Moreover, the example gives additional intuition towards understanding Okhotin’s grammar.
Figure 3.2: Intermediate stage of a computation on $abbcccaab$. The symbol $\hat{\cdot}$ denotes the current position in the input.

Figure 3.3: The configuration of the automaton in Figure 3.2. The denotation of the configuration is $((q_1, aab, b\perp), (q_2, cccaab, bb\perp))$.

for the reduplication language as presented in Example 2.8. The following automaton accepts the language derived by the non-terminal $D$ in the grammar.

**Example 3.8.** (Cf. Example 2.8) The SAPDA $A = (Q, \Sigma, \Gamma, \delta, q_0, \perp)$ defined below accepts the non-context-free language

$$\{ w$uw : w, u \in \{a,b\}^* \} .$$

- $Q = \{ q_0, q_w, q_e \} \cup \{ q^1_\sigma : \sigma \in \{a,b\} \} \cup \{ q^2_\sigma : \sigma \in \{a,b\} \}$,
- $\Sigma = \{a,b,\$$\}$,
- $\Gamma = \{\perp, \#\}$, and
- $\delta$ is defined as follows.

1. $\delta(q_0, \sigma, \perp) = \{(q^1_\sigma, \perp), (q_0, \perp)\}, \sigma \in \{a,b\}$
2. $\delta(q^1_\sigma, \tau, X) = \{(q^2_\sigma, \#X)\}, \sigma, \tau \in \{a,b\}, X \in \Gamma,$
3. $\delta(q_0, \$$, \perp) = \{(q_w, \perp)\},$
4. $\delta(q_w, \sigma, \perp) = \{(q_w, \perp), (q_e, \epsilon)\}, \sigma \in \{a,b\},$
5. $\delta(q^1_\sigma, \$$, X) = \{(q^2_\sigma, X)\}, \sigma \in \{a,b\}, X \in \Gamma,$
6. $\delta(q^2_\sigma, \tau, X) = \{(q^2_\sigma, X)\}, \sigma, \tau \in \{a,b\} : \sigma \neq \tau, X \in \Gamma,$
7. $\delta(q^2_\sigma, \sigma, X) = \{(q^2_\sigma, X), (q_e, X)\}, \sigma \in \{a,b\}, X \in \Gamma,$
The computations of the automaton have two main phases: before and after
the $ sign is encountered in the input. In the first phase, each input letter
$\sigma$ that is read leads to a conjunctive transition (transition 1) that opens
two new stack-branches. One new branch continues the recursion, while the
second checks that the following condition is met.

Assume $\sigma$ is the $n$-th letter from the first $\$$ sign. If so, the new stack
branch opened during the transition on $\sigma$ will verify that the $n$-th letter from
the end of the input is also $\sigma$. This way, if the computation is accepting, the
word will in fact be of the form $w\$w$. To be able to check this property,
the branch must know $\sigma$ and $\sigma$’s relative position - $n$ - to the $\$$ sign. To
“remember” $\sigma$, the state of the branch head is $q^1_\sigma$ (the 1 superscript denoting
that the computation is in the first phase). To find $n$, the branch adds a $\#$
sign to its stack for each input character read (transition 2), until the $\$$ is
encountered in the input. Therefore, when the $\$$ is read, the number of $\#$s
in the stack branch will be the number of letters between $\sigma$ and the $\$$ sign
in the first half of the input word.

Once the $\$$ is read, the branch perpetuating the recursion ceases to open
new branches, and transitions to state $q_w$ (transition 3). From this state it
can read the characters left till the end of the word, and then empty its stack
(transition 4). All the other branches denote that they have moved to the
second phase of the computation by transitioning to states $q^2_\sigma$ (transition
5). From this point onward, each branch “waits” to see the $\sigma$ encoded in
its state in the input (transition 6). Once it does encounter $\sigma$, it can either
ignore it and continue to look for another $\sigma$ in the input (in case there are
repetitions in $w$ of the same letter), or it can “guess” that this is the $\sigma$ that
is $n$ letters from the end of the input, and move to state $q_e$ (transition 7).

After transitioning to $q_e$, one $\#$ is emptied from the stack for every input
character read. If in fact $\sigma$ was the right number of letters from the end,
the $\perp$ sign of the stack branch will be exposed exactly when the last input
letter is read. At this point, an $\epsilon$-transition that empties the stack branch
is applied (transition 9).

If all branches successfully “guess” their respective $\sigma$ symbols then the
computation will reach a configuration where all leaf nodes are labeled
(q_e, \epsilon, \epsilon). From here, successive branch collapsing steps can be applied until an accepting configuration is reached.

Consider a computation on the word $abb\$abb$. Figure 3.4 shows the contents of the stack tree after all branches have read the prefix $ab$. The rightmost branch is the branch perpetuating the recursion. The leftmost branch remembers seeing $a$ in the input, and has since counted one letter. The middle branch remembers seeing $b$ in the input, and has not yet counted any letters.

Figure 3.5 shows the contents of the stack tree after all branches have read the prefix $abb\$bab$. The rightmost branch, has stopped perpetuating the recursion, transitioned to $q_e$, and emptied its stack. The leftmost branch correctly “guessed” that the $a$ read was the $a$ it was looking for. Subsequently, it transitioned to $q_e$ and removed one # from its stack for the $b$ that was read afterwards. The second branch from the left correctly ignored the first $b$ after the $\$$ sign, and only transitioned to $q_e$ after reading the second $b$. The second branch from the right is still waiting to find the correct $b$, and is therefore still in state $q_{b2}$.

3.2 Equivalence Results

In this section we show that SAPDA are in fact an automaton counterpart for Conjunctive Grammars, i.e., they accept exactly the class of conjunctive languages. The proofs demonstrate a correlation between the models.
that is analogous to the correlation between classical PDA and context-free grammars.

**Theorem 3.9.** A language is accepted by an SAPDA if and only if it is generated by a CG.

The proof of the theorem is an extension of the classical one, see, e.g., [7, pp. 115–119].

### 3.2.1 Construction of an SAPDA from a CG

We begin with a proof of the “if” part of Theorem 3.9. To do so, we first need to define a notion of leftmost derivations for conjunctive grammars.

**Definition 3.10.** Leftmost conjunctive formulas and leftmost derivations are defined by the following simultaneous recursion.

- The elements of $(V \cup \Sigma)^+$ are leftmost (conjunctive) formulas. One-step derivations from them in which a rule is applied to their leftmost variable are leftmost derivations.
- Let $A$ be a leftmost formula. Then a one-step leftmost derivation from $A$ is either contraction or a rule applied to the leftmost variable of a simple conjunct of $A$, and yields a leftmost formula.

The one-step leftmost derivation relation is denoted $\Rightarrow_L$, and, as usual, its reflexive and transitive closure is denoted $\Rightarrow_L^*$.\(^4\)

For example, the following is a leftmost derivation

$$S \Rightarrow (AB\&CD) \Rightarrow (AB\&cD)$$

while, the next derivation is not leftmost

$$S \Rightarrow (AB\&CD) \Rightarrow (Ab\&CD)$$

as the rule is not applied to the leftmost variable in a simple conjunct.

Similarly to context-free grammars, in CGs, the order in which independent derivation rules are applied, does not affect the derived word. Therefore, we have the following lemma.

\(^4\)Note that leftmost formulas are defined with respect to the underlying conjunctive grammar.
Lemma 3.11. Let $\alpha \in (V \cup \Sigma)^*$ and $w \in \Sigma^*$ be such that $\alpha \Rightarrow^* w$. Then $\alpha \Rightarrow_L^* w$.

For the purposes of our proof we assume that grammars do not contain $\epsilon$-rules. An $\epsilon$-rule is a rule of the form $A \rightarrow \cdots \epsilon \cdots$. Okhotin proved in [14] that it is possible to remove such rules from the grammar, with the exception of an $\epsilon$-rule from the start symbol in the case where $\epsilon$ is in the grammar language. We also assume that the start symbol does not appear in the right-hand side of any rule. This can be achieved by augmenting the grammar with a new start symbol $S'$, and a new rule $S' \rightarrow S$, as is done in the classical case.

Let $G = (V, \Sigma, P, S)$ be a conjunctive grammar. Consider the single-state SAPDA $A_G = (\Sigma, \Gamma, \delta, \bot)$,\(^5\) where

- $\Gamma = V \cup \Sigma$ and $\bot = S$,
- $\delta(\epsilon, A) = \{\alpha_1 \land \cdots \land \alpha_n \mid A \rightarrow (\alpha_1 \& \cdots \& \alpha_n) \in P\}$,\(^6\) and
- $\delta(\sigma, \sigma) = \{\epsilon\}$.

We shall prove that $L(A_G) = L(G)$. The proof is an extension of the classical one, see, e.g., [7, Theorem 5.3, pp. 115–116]. In the classical proof, a correlation is shown between the input and stack contents of the automaton and the leftmost sentential forms of the grammar. As the construction is a conservative extension, the same correlation holds, as long as no conjunctive transitions are applied. Therefore, we have the following lemma.

Lemma 3.12. Let $\gamma, \alpha \in (V \cup \Sigma)^*$, and $x \in \Sigma^*$, then

$$\gamma \Rightarrow_L^* x \alpha \quad \text{if and only if} \quad (x, \gamma) \vdash^* (\epsilon, \alpha)$$

if no conjunctive rules were used in the derivation, and no conjunctive transitions were used in the computation.

The proof of the “if” part of Theorem 3.9 is be based on the following proposition.

---

\(^5\)We omit the state component of both $A_G$ and $\delta$.

\(^6\)Since $G$ does not contain $\epsilon$-rules, $\alpha_1, \ldots, \alpha_n \in \Gamma^*$. 

24
Proposition 3.13. Let $\gamma \in (V \cup \Sigma)^*$, and $w \in \Sigma^*$, then

$$\gamma \Rightarrow_L^* w \quad \text{if and only if} \quad (w, \gamma) \vdash^* (\epsilon, \epsilon).$$

We begin with the proof of the “only if” part of the proposition.

Proof. Assume that $\gamma \neq w$, otherwise the claim is trivial. The proof is by induction on the number of conjunctive rules applied in the derivation.

**Basis:** If no conjunctive rules are applied in the derivation, then the claim follows immediately from Lemma 3.12.

**Induction Step:** Assume the claim holds for derivations with up to $i$ applications of conjunctive rules. Let $\gamma \Rightarrow_L^* w$ be a derivation with $i+1$ applications of conjunctive transitions. Therefore, the derivation is of the form

$$\gamma \Rightarrow_L^* xX\alpha \Rightarrow_L x(\alpha_1 \& \cdots \& \alpha_n)\alpha \Rightarrow_L^* x(y\&\cdots\&y)\alpha \Rightarrow_L xy\alpha \Rightarrow_L^* xyz,$$

where no conjunctive rules were used in the derivation $\gamma \Rightarrow_L^* xX\alpha$, and $w = xyz$.

By Lemma 3.12, $(xyz, \gamma) \vdash^* (yz, X\alpha)$. By the construction of $A_G$, and the fact that $X \rightarrow (\alpha_1 \& \cdots \& \alpha_n) \in P$ we know that $\alpha_1 \land \cdots \land \alpha_n \in \delta(\epsilon, X)$.

Each of the derivations $\alpha_j \Rightarrow_L^* y$ and $\alpha \Rightarrow_L^* z$ have at most $i$ applications of conjunctive rules. Therefore, by the induction hypothesis, $(y, \alpha_i) \vdash^* (\epsilon, \epsilon)$ and $(z, \alpha) \vdash^* (\epsilon, \epsilon)$. It follows that

$$(xyz, \gamma) \vdash^* (yz, X\alpha) \vdash^* ((yz, \alpha_1) \land \cdots \land (yz, \alpha_n))\alpha \vdash^* ((z, \epsilon) \land \cdots \land (z, \epsilon))\alpha \vdash^* (z, \alpha) \vdash^* (\epsilon, \epsilon)$$

and we have our claim. \qed

We proceed to prove the “if” part of the proposition.

Proof. Assume that $\gamma \neq w$, otherwise the claim is trivial. The proof is by induction on the number of conjunctive transitions applied in the computation.

**Basis:** If no conjunctive transitions are applied in the computation, then the claim follows immediately from Lemma 3.12.
**Induction Step:** Assume the claim holds for computations with up to $i$ conjunctive transitions. Let $(w, \gamma) \vdash^* (\epsilon, \epsilon)$ be a computation with $i + 1$ conjunctive transitions. Therefore, the computation is of the form

$$(xyz, \gamma) \vdash^* (yz, X\alpha)$$

$$(yz, \alpha_1) \land \cdots \land (yz, \alpha_n) \alpha$$

$$(z, \epsilon) \land \cdots \land (z, \epsilon) \alpha$$

$$(z, \alpha) \vdash^* (\epsilon, \epsilon)$$

where $w = xyz$ and no conjunctive transitions were applied in the computation $(xyz, \gamma) \vdash^* (yz, X\alpha)$.

By Lemma 3.12, $\gamma \Rightarrow_L x\alpha$. By the construction of $A_G$, and the fact that $\alpha_1 \land \cdots \land \alpha_n \in \delta(\epsilon, X)$, we know that $X \rightarrow (\alpha_1 \& \cdots \& \alpha_n) \in P$.

Each of the computations $(y, \alpha_i) \vdash^* (\epsilon, \epsilon)$ and $(z, \alpha) \vdash^* (\epsilon, \epsilon)$ have at most $i$ applications of conjunctive transitions. Therefore, by the induction hypothesis, $\alpha_j \Rightarrow_L y$ and $\alpha \Rightarrow^*_L z$. It follows that

$$\gamma \Rightarrow_L x\alpha \Rightarrow_L x(\alpha_1 \& \cdots \& \alpha_n) \alpha \Rightarrow_L x(y \& \cdots \& y) \alpha \Rightarrow_L xy \alpha \Rightarrow_L xyz$$

and we have our claim.

Now we can prove the “if” part of Theorem 3.9.

**Proof.** Let $G$ be a conjunctive grammar, and let $A_G$ be the SAPDA constructed as above. Let $w \in L(G)$, i.e., $S \Rightarrow_L^* w$. By Proposition 3.13, this is if and only if $(w, S) \vdash^* (\epsilon, \epsilon)$, which means that $w \in L(A_G)$. Therefore, $L(G) = L(A_G)$, and the proof of the “if” part of the theorem is complete.

### 3.2.2 Construction of a CG from an SAPDA

We expand upon the classical construction of a context-free grammar from a PDA, see e.g., see, e.g., [7, Theorem 5.4, pp. 116–119]. To simplify the construction and proof, we assume that conjunctive transitions are all $\epsilon$ transitions, and that they write exactly one symbol into each stack branch, i.e., they are of the form $(q_1, X_1) \land \cdots \land (q_n, X_n) \in \delta(q, \epsilon, X)$. This assumption can easily be shown not to be limiting.
Let $A = (Q, \Sigma, \Gamma, \delta, q_0, \perp)$ be an SAPDA. Consider the CG $G_A = (V, \Sigma, P, S)$, where

- $V = Q \times \Gamma \times Q \cup \{S\}$, \(^7\) and
- $P$ contains the following rules.
  - $S \rightarrow [q_0, \perp, p]$, for all $p \in Q$,
  - $[q, X, p] \rightarrow \sigma[q_1, Y_1, q_2, \ldots, q_n, Y_n, p]$, for all transitions $(q_1, Y_1, \ldots, Y_n) \in \delta(q, \sigma, X)$, and for all choices of $q_1, \ldots, q_n \in Q$, and
  - $[q, X, p] \rightarrow ([q_1, Y_1, p] \& \cdots \& [q_n, Y_n, p])$, for all transitions $(q_1, Y_1, \ldots, Y_n) \in \delta(q, \epsilon, X)$.

We shall prove that $L(A_G) = L(G)$. In the classical proof, a correlation is shown between the input and stack contents of the automaton and the leftmost sentential forms of the grammar. As the construction is a conservative extension, the same correlation holds, as long as no conjunctive transitions are applied. Therefore, we have the following lemma.

**Lemma 3.14.** Let $q, p \in Q$, $X, X_1, \ldots, X_m \in \Gamma^*$ and $x \in \Sigma^*$, then

$$(q, x, X) \vdash^* (q_1, \epsilon, X_1 \cdots X_m)$$

if and only if there exist $q_1, \ldots, q_m \in Q$ such that

$$[q, X, p] \Rightarrow^* x[q_1, X_1, q_2, \ldots, q_m, X_m, p] \ ,$$

if no conjunctive rules were used in the derivation, and no conjunctive transitions were used in the computation.

The proof of the “if only” part of Theorem 3.9 is based on the following proposition.

**Proposition 3.15.** Let $q, p \in Q$, $X \in \Gamma^*$ and $w \in \Sigma^*$, then

$$[q, X, p] \Rightarrow^*_L w \ \text{if and only if} \ \ (q, w, X) \vdash^* (p, \epsilon, \epsilon) .$$

We begin with the proof of the “only if” part of the proposition.

\(^7\)Renaming $S$, if necessary, we may assume that $S \not\in Q \times \Gamma \times Q$. 

27
Proof. The proof is by induction on the number of conjunctive rules applied in the derivation.

Basis: If no conjunctive rules are applied in the derivation, then the claim follows immediately from Lemma 3.14.

Induction Step: Assume the claim holds for derivations with up to \( i \) applications of conjunctive rules. Let \([q, X, p] \Rightarrow^*_L w\) be a derivation with \( i + 1 \) applications of conjunctive transitions. Therefore, the derivation is of the form

\[
[q, X, p] \Rightarrow^*_L x[r, Y, q_1][q_1, X_1, q_2] \cdots [q_m, X_m, p]
\]

where no conjunctive rules were used in the derivation

\[
[q, X, p] \Rightarrow^*_L x[r, Y, q_1][q_1, X_1, q_2] \cdots [q_m, X_m, p],
\]

and \( w = xyz_1 \cdots z_m \).

By Lemma 3.14, \((q, x, X) \vdash^* (r, \epsilon, YX_1 \cdots X_m)\). By the construction of \( G_A \), and the fact that \([r, Y, q_1] \rightarrow ([p_1, Y_1, q_1] \& \cdots \& [p_n, Y_n, q_1]) \in P\) we know that \((p_1, Y_1) \land \cdots \land (p_n, Y_n) \in \delta(r, \epsilon, Y)\).

Each of the derivations \([p_i, Y_i, q_1] \Rightarrow^*_L y\) have at most \( i \) applications of conjunctive rules. Therefore, by the induction hypothesis, \((p_i, y, Y_i) \vdash^* (q_1, \epsilon, \epsilon)\). Similarly, each of the derivations \([q_i, X_i, q_{i+1}] \Rightarrow^*_L z_i\) and the derivation \([q_m, X_m, p] \Rightarrow^*_L z_m\) have at most \( i \) applications of conjunctive rules, and therefore, by the induction hypothesis, \((q_i, z_i, X_i) \vdash (q_{i+1}, \epsilon, \epsilon)\) and \((q_m, z_m, X_m) \vdash^* (p, \epsilon, \epsilon)\). It follows that

\[
(q, xyz_1 \cdots z_m, X) \vdash^* (r, yz_1 \cdots z_m, YX_1 \cdots X_m)
\]

\[
\vdash ((p_1, yz_1 \cdots z_m, Y_1) \land \cdots \land (p_n, yz_1 \cdots z_m, Y_n))X_1 \cdots X_m
\]

\[
\vdash^* ((q_1, z_1 \cdots z_m, \epsilon) \land \cdots \land (q_1, z_1 \cdots z_m, \epsilon))X_1 \cdots X_m
\]

\[
\vdash (q_1, z_1 \cdots z_m, X_1 \cdots X_m)
\]

\[
\vdash^* (q_2, z_2 \cdots z_m, X_2 \cdots X_m) \vdash^* \cdots \vdash^* (p, \epsilon, \epsilon)
\]

and we have our claim. \(\square\)
We proceed to the proof of the "if" part of the proposition.

**Proof.** The proof is by induction on the number of conjunctive transitions applied in the computation.

**Basis:** If no conjunctive transitions are applied in the computation, then the claim follows immediately from Lemma 3.14.

**Induction Step:** Assume the claim holds for computations with up to \(i\) conjunctive transitions. Let \((q, w, X) \vdash^* (p, \epsilon, \epsilon)\) be a computation with \(i + 1\) conjunctive transitions. Therefore, the computation is of the form

\[
(q, xyz_1 \cdots z_m, X) \vdash^* (r, yz_1 \cdots z_m, YX_1 \cdots X_m)
\]

where \(w = xyz_1 \cdots z_m\), no conjunctive transitions were applied in the first part of the computation, and after reading each \(z_i\), is exposed as the top symbol in the stack \(X_{i+1}\) for the first time (similarly to the proof in the classical case), i.e., \((q_i, z_i, X_i) \vdash^* (q_{i+1}, \epsilon)\) and \((q_m, z_m, X_m) \vdash^* (p, \epsilon, \epsilon)\).

By Lemma 3.14, \([q, X, p] \Rightarrow_L^* x[r, Y, q_1][q_1, X_1, q_2] \cdots [q_m, X_m, p]\). By the construction of \(G_A\), and the fact that \((p_1, Y_1) \land \cdots \land (p_n, Y_n) \in \delta(r, \epsilon, Y)\), we know that \([r, Y, q_1] \rightarrow ([p_1, Y_1, q_1] \land \cdots \land [p_n, Y_n, q_1]) \in P\).

Each of the computations \((p_j, y, Y_j) \vdash^* (q_j, \epsilon, \epsilon)\) have at most \(i\) applications of conjunctive transitions. Therefore, by the induction hypothesis, \([p_j, Y_j, q_1] \Rightarrow_L^* y\). Similarly, each of the computations \((q_i, z_i, X_i) \vdash^* (q_{i+1}, \epsilon, \epsilon)\) and the computation \((q_m, z_m, X_m) \vdash^* (p, \epsilon, \epsilon)\) have at most \(i\) applications of conjunctive transitions, and therefore, by the induction hypothesis, \([q_i, X_i, q_{i+1}] \Rightarrow_L^* z_i\) and \([q_m, X_m, p] \Rightarrow_L^* z_m\). It follows that

\[
[q, X, p] \Rightarrow_L^* x[r, Y, q_1][q_1, X_1, q_2] \cdots [q_m, X_m, p]
\]

\[
\Rightarrow_L x([p_1, Y_1, q_1] \land \cdots \land [p_n, Y_n, q_1])[q_1, X_1, q_2] \cdots [q_m, X_m, p]
\]

\[
\Rightarrow_L^* x[q_1, X_1, q_2] \cdots [q_m, X_m, p]
\]

\[
\Rightarrow_L xy[q_1, X_1, q_2] \cdots [q_m, X_m, p] \Rightarrow_L^* xyz_1 \cdots z_m,
\]

and we have our claim.
Now we can prove the “if only” part of Theorem 3.9.

**Proof.** Let $A$ be an SAPDA, and let $G_A$ be the conjunctive grammar constructed as above. Let $w \in L(A)$, i.e., $(q_0, w, \bot) \vdash^* (p, \epsilon, \epsilon)$. By Proposition 3.15, this is if and only if $[q_0, \bot, p] \Rightarrow^*_{L} w$. Therefore, $S \Rightarrow^*_{L} w$, implying $w \in L(A_G)$, and vice versa. It follows that $L(A) = L(G_A)$, and the proof of the “only if” part of the theorem is complete. □

From Theorem 3.9, and the proof of its “only if” part we obtain the following immediate corollary.

**Corollary 3.16.** For every SAPDA there is a single-state SAPDA that accepts the same language.

In other words, allowing SAPDA to have any finite number of states does not make the computation model stronger than that of one-state SAPDA.
Chapter 4

One-turn SAPDA and Linear Conjunctive Grammars

It is a well-known result, due to Ginsburg and Spanier [6], that linear grammars are equivalent to one-turn PDA. One-turn PDA are a sub-family of pushdown automata, where in each computation the stack height switches only once from non-decreasing to non-increasing. That is, once a transition replaces the top symbol of the stack with $\epsilon$, all subsequent transitions may write at most one character.

We define a similar notion of one-turn SAPDA, where each stack branch can make only one turn during the course of a computation, [3, 2]. As in the classical case, we show that one-turn SAPDA are equivalent to LCG. Based on Okhotin’s proof that LCG are equivalent to Trellis Automata [21], we deduce that one-turn SAPDA are also equivalent to Trellis Automata. In contrast to the equivalence with Trellis Automata, the correlation between one-turn SAPDA and LCG reflects the classical correlation between linear grammars and one-turn PDA.

In Section 4.1 we explore a new way of looking at CG and SAPDA that will assist us in defining one-turn SAPDA. In Section 4.2 we define the one-turn SAPDA model, and in Section 4.3 we prove its equivalence to LCG.
4.1 Trace Grammars and Trace Automata

We explore a new way of looking at CG and SAPDA, which will prove extremely useful. We remove the conjunctions from the models, essentially “flattening” them back into their context-free counterparts. While the language obviously changes, many of the structural qualities of the derivations and computations remain, making this a useful tool for formal definitions and proofs that relate to linear qualities of the models that are maintained after flattening.

**Definition 4.1.** Let \( G = (V, T, P, S) \) be a conjunctive grammar. The trace grammar of \( G \) is a context-free grammar \( G_T = (V, T, P_T, S) \) where \( P_T \) is defined as follows:

1. \( X \rightarrow \alpha \in P_T \) for all \( X \rightarrow \alpha \in P \).
2. \( X \rightarrow \alpha_i \in P_T \) for all \( X \rightarrow (\alpha_i \& \cdots \& \alpha_n) \in P \) and \( i = 1, \ldots, n \).

Rules of type 2 are called projections of the original conjunctive rules they were obtained from. Applications of these rules are referred to as conjunct selections.

**Definition 4.2.** Let \( G \) be a conjunctive grammar, and let \( G_T \) be its trace grammar. Any derivation in \( G_T \) is called a trace derivation of \( G \). We denote trace derivations by \( \Rightarrow_T \) and \( \Rightarrow^*_T \).

**Definition 4.3.** Let \( G \) be a conjunctive grammar, and let \( G_T \) be its trace grammar. Let \( \alpha \Rightarrow^* \beta \) be a derivation in \( G \), and let \( \alpha \Rightarrow^*_T \beta \) be a trace derivation in \( G_T \). We say that the trace derivation is a projection of the full derivation if one of the following holds.

- The derivation \( \alpha \Rightarrow^* \beta \) has no applications of conjunctive rules, and it is identical to the trace.
- The derivation \( \alpha \Rightarrow^* \beta \) is of the form

\[
\alpha \Rightarrow^* \gamma_1 X \gamma_2 \Rightarrow \gamma_1 (\alpha_1 \& \cdots \& \alpha_n) \gamma_2 \Rightarrow \gamma_3 (B_1 \& \cdots \& B_n) \gamma_4 ,
\]

where \( \gamma_1 \Rightarrow^* \gamma_3, \gamma_2 \Rightarrow^* \gamma_4, \) and \( \alpha_i \Rightarrow^* B_i, i = 1, \ldots, n; \) and, \( \alpha \Rightarrow^*_T \beta \)

is of the form

\[
\alpha \Rightarrow^*_T \gamma_1' X \gamma_2' \Rightarrow_T \gamma_1' \alpha_i \gamma_2' \Rightarrow^*_T \gamma_3' \beta \gamma_4' = \beta
\]

32
for some $i = 1, \ldots, n$, where $\alpha \Rightarrow^* \gamma_1' X \gamma_2'$, $\gamma_1' \Rightarrow_T^* \gamma_3'$, $\gamma_2' \Rightarrow_T^* \gamma_4'$, and $\alpha_i \Rightarrow_T^* \beta_i$ are trace projections of $\alpha \Rightarrow^* \gamma_1 X \gamma_2$, $\gamma_1 \Rightarrow^* \gamma_3$, $\gamma_2 \Rightarrow^* \gamma_4$, and $\alpha_i \Rightarrow^* B_i$, respectively.

The derivation $\alpha \Rightarrow^* B$ is of the form

$\alpha \Rightarrow^* \gamma_1 X \gamma_2 \Rightarrow \gamma_1 (\alpha_1 \& \cdots \& \alpha_n) \gamma_2 \Rightarrow^* \gamma_3 (y \& \cdots \& y) \gamma_4 \Rightarrow \gamma_3 y \gamma_4 = \beta$, 

where $\gamma_1 \Rightarrow^* \gamma_3$, $\gamma_2 \Rightarrow^* \gamma_4$, and $\alpha_i \Rightarrow^* y$, $i = 1, \ldots, n$; and, up to a permutation of derivation steps, $\alpha \Rightarrow_T^* \beta$ is of the form

$\alpha \Rightarrow_T^* \gamma_1' X \gamma_2' \Rightarrow_T \gamma_1' \alpha_i \gamma_2' \Rightarrow_T^* \gamma_3 y \gamma_4'$

for some $i = 1, \ldots, n$, where $\alpha \Rightarrow_T^* \gamma_1' X \gamma_2'$, $\gamma_1' \Rightarrow_T^* \gamma_3'$, $\gamma_2' \Rightarrow_T^* \gamma_4'$, and $\alpha_i \Rightarrow_T^* y$ are trace projections of $\alpha \Rightarrow^* \gamma_1 X \gamma_2$, $\gamma_1 \Rightarrow^* \gamma_3$, $\gamma_2 \Rightarrow^* \gamma_4$, and $\alpha_i \Rightarrow^* y$, respectively.

A projection of a conjunctive derivation is a trace that follows one of the possible conjunctive paths of the derivation. The conjunct selections in the projections match the conjunctive rules applied along the path of the full derivation, and the other “regular” rules applied in the projection match the non-conjunctive rules applied in the path.

**Definition 4.4.** If two trace derivations are both projections of the same full derivation, we say that they are *sibling* traces.

For example, consider the conjunctive grammar from Example 2.7 in Section 2.1. The following is a conjunctive derivation of the word $abc$,

$S \Rightarrow (A \& C) \Rightarrow (aA \& C) \Rightarrow (aB \& C) \Rightarrow (abBc \& C) \Rightarrow (abc \& C) \Rightarrow (abc \& Cc) \Rightarrow (abc \& Dc) \Rightarrow (abc \& aDbc) \Rightarrow (abc \& abc) \Rightarrow abc$, 

and below is a trace that is a projection of the derivation

$S \Rightarrow A \Rightarrow aA \Rightarrow aB \Rightarrow abBc \Rightarrow abc$.

The following is a different projection, and therefore a sibling of the previous trace.

$S \Rightarrow C \Rightarrow Cc \Rightarrow Dc \Rightarrow aDbc \Rightarrow abc$.
From the definition of sibling traces, we have the following immediate lemma, which formalizes the structural relationship between sibling trace derivations.

**Lemma 4.5.** Let \( \alpha \Rightarrow^* T \beta_1 \) and \( \alpha \Rightarrow^* T \beta_2 \) be two sibling traces. Then there exists a sentential form \( \gamma \) such that

- the first trace is of the form \( \alpha \Rightarrow^* T \gamma \Rightarrow T \gamma_1 \Rightarrow^* T \beta_1 \),
- the second trace is of the form \( \alpha \Rightarrow^* T \gamma \Rightarrow T \gamma_2 \Rightarrow^* T \beta_2 \),
- the derivation of \( \gamma \) from \( \alpha \) is identical in both traces, including conjunct selections, and
- the rules used for \( \gamma \Rightarrow T \gamma_1 \) in the first trace and \( \gamma \Rightarrow T \gamma_2 \) in the second trace are both different projections of the same conjunctive rule.

Analogously, we define a “flattened” SAPDA.

**Definition 4.6.** Let \( A = (Q, \Sigma, \Gamma, q_0, \bot, \delta) \) be an SAPDA. The trace automaton of \( A \) is a classical pushdown automaton \( A_T = (Q, \Sigma, \Gamma, q_0, \bot, \delta_T) \) where \( \delta_T \) is defined as follows:

1. \((p, \alpha) \in \delta_T(q, a, X)\) for all \((p, \alpha) \in \delta(q, a, X)\).
2. \((p_i, \alpha_i) \in \delta_T(q, a, X), \) for all \((p_1, \alpha_1) \land \cdots \land (p_n, \alpha_n) \in \delta(q, a, X)\) and \(i = 1, \ldots, n\).

Transitions of type 2 are called *projections* of the original conjunctive transitions they were obtained from. Applications of these transitions are referred to as *branch selections*.

**Definition 4.7.** Let \( A \) be as SAPDA, and let \( A_T \) be its trace automaton. Any computation of \( A_T \) is called a *trace* computation of \( A \). We denote trace computations by \( \vdash_T \) and \( \vdash^*_T \).

**Definition 4.8.** Let \( A \) be an SAPDA and let \( A_T \) be its trace automaton. Let \((q, w, \alpha) \vdash^* d(T)\) be a computation on \( w \) in \( A \), and let \((q, w, \alpha) \vdash^*_T (p, \epsilon, \beta)\) be a trace computation on \( w \) in \( A_T \). We say that the trace computation is a *projection* of the full computation, if one of the following holds.

- There are no conjunctive transitions in the computation \((q, w, \alpha) \vdash^* d(T)\), and it is identical to the trace.

---

34
• The computation \((q, w, \alpha) \vdash^* d(T)\) is of the form
\[
(q, w, \alpha) \vdash^* (q', \sigma v, X\gamma) \]
\[
\vdash ((q_1, v, \alpha_1) \land \cdots \land (q_n, v, \alpha_n))\gamma
\]
\[
\vdash^* (d(T_1) \land \cdots \land d(T_n))\gamma ,
\]
where \(q_i, v, \alpha_i) \vdash^* d(T_i),\ i = 1, \ldots, n;\) and \((q, w, \alpha) \vdash^*_T (p, \epsilon, \beta)\) is of the form
\[
(q, u\sigma v, \alpha) \vdash^*_T (q', \sigma v, X\gamma) \vdash_T (q_i, v, \alpha_i)\gamma \vdash^*_T (p, \epsilon, \beta')\gamma = (p, \epsilon, \beta) ,
\]
for some \(1 \leq i \leq n,\) where \((q, u\sigma v, \alpha) \vdash^*_T (q', \sigma v, X\gamma)\) and \((q_i, v, \alpha_i) \vdash^*_T (p, \epsilon, \beta')\) are trace projections of \((q, u\sigma v, \alpha) \vdash^* (q', \sigma v, X\gamma)\) and \((q_i, v, \alpha_i) \vdash^* d(T_i),\) respectively.

• The computation \((q, w, \alpha) \vdash^* d(T)\) is of the form
\[
(q, w, \alpha) \vdash^* (q', \sigma vy, X\gamma)
\]
\[
\vdash ((q_1, vy, \alpha_1) \land \cdots \land (q_n, vy, \alpha_n))\gamma
\]
\[
\vdash^* ((p', y, \epsilon) \land \cdots \land (p', y, \epsilon))\gamma
\]
\[
\vdash (p', y, \gamma) \vdash^*_T (p, \epsilon, \beta) ,
\]
where \((q_i, v, \alpha_i) \vdash^* (p', y, \epsilon),\ i = 1, \ldots, n;\) and \((q, w, \alpha) \vdash^*_T (p, \epsilon, \beta)\) is of the form
\[
(q, u\sigma vy, \alpha) \vdash^*_T (q', \sigma vy, X\gamma) \vdash_T (q_i, vy, \alpha_i)\gamma \vdash^*_T (p', y, \gamma) \vdash^*_T (p, \epsilon, \beta) ,
\]
for some \(1 \leq i \leq n,\) where \((q, w, \alpha) \vdash^*_T (q', \sigma vy, X\gamma),\) \((q_i, vy, \alpha_i) \vdash^*_T (p', y, \gamma),\) and \((p', y, \gamma) \vdash^*_T (p, \epsilon, \beta)\) are trace projections of \((q, w, \alpha) \vdash^* (q', \sigma vy, X\gamma),\) \((q_i, vy, \alpha_i) \vdash^* (p', y, \gamma),\) and \((p', y, \gamma) \vdash^* (p, \epsilon, \beta),\) respectively.

A projection of a full SAPDA computation is a trace that follows one of the branches of the computation. Every branch selection of the projection matches a conjunctive transition applied to the branch, and every regular transition matches a non-conjunctive transition applied to the branch.

35
Definition 4.9. If two trace computations are both projections of the same full computation, we say that they are sibling traces.

From the definition of sibling traces, we have the following immediate lemma, which formalizes the structural relationship between sibling trace computations.

Lemma 4.10. Let \((q, w, \alpha) \vdash_T^* (p_1, \epsilon, \beta_1)\) and \((q, w, \alpha) \vdash_T^* (p_2, \epsilon, \beta_2)\) be two sibling traces. Then there exists a configuration \((r, \sigma v, \gamma)\) such that

- the first trace is of the form
  \[(q, u\sigma v, \alpha) \vdash_T^* (r, \sigma v, \gamma) \vdash_T (s_1, v, \gamma_1) \vdash_T^* (p_1, \epsilon, \beta_1),\]

- the second trace is of the form
  \[(q, u\sigma v, \alpha) \vdash_T^* (r, \sigma v, \gamma) \vdash_T (s_2, v, \gamma_2) \vdash_T^* (p_2, \epsilon, \beta_2),\]

- the computation on \(v\) is identical in both traces, including branch selections, and

- the transitions used for \((r, \sigma v, \gamma) \vdash_T (s_1, v, \gamma_1)\) in the first trace and \((r, \sigma v, \gamma) \vdash_T (s_2, v, \gamma_2)\) in the second trace are both different projections of the same conjunctive transition.

The formalization of trace grammars and trace automata will prove to be quite useful as a way to define or prove certain types of claims regarding the full conjunctive models. For example, the following proposition shows an alternative definition for leftmost derivations, which is equivalent to Definition 3.10.

Proposition 4.11. A conjunctive derivation is leftmost if and only if all its projections are leftmost derivations in the classical sense.

Another example of an equivalent definition using traces is presented in the following proposition.

Proposition 4.12. Let \(G\) be a conjunctive grammar, and let \(G_T\) be its trace grammar. \(G\) is linear if and only if its trace \(G_T\) is a linear grammar in the classical sense.
Moreover, consider the construction presented in Subsection 3.2.1. By applying the classical construction to the trace grammar $G_T$ we obtain the trace automaton $A_{G_T}$. Lemma 3.12 follows immediately. Similarly, for the construction presented in Subsection 3.2.2, by applying the classical construction to $A_T$ we obtain $G_{A_T}$.

4.2 One-turn SAPDA Model Definition

To define a one-turn SAPDA, we expand upon the classical definition of a one-turn PDA. Let us recall the formal definition of a one-turn PDA.

**Definition 4.13.** Let $(q, \sigma w, X\alpha) \vdash (p, w, \gamma\alpha)$ be a transition of a classical PDA.

- We say that the transition is **increasing** if $|\gamma| > 1$.
- We say that the transition is **decreasing** if $\gamma = \epsilon$.
- We say that the transition is **non-decreasing** if it is increasing, or if $|\gamma| = 1$.
- We say that the transition is **non-increasing** if it is decreasing, or if $|\gamma| = 1$.

**Definition 4.14.** Let $A = (Q, \Sigma, \Gamma, q_0, \bot, \delta)$ be a classical pushdown automaton. We say that $A$ is **one-turn** if each computation of $A$ is of one of the following two forms

1. $(q_0, xy, \bot) \vdash^* (q, y, \alpha)$ s.t all the transitions are non-decreasing.\(^1\)

2. $(q_0, x\sigma yz, \bot) \vdash^* (q, \sigma yz, X\alpha) \vdash (p, yz, \alpha) \vdash^* (r, z, \beta)$ such that all the transitions in $(q_0, x\sigma yz, \bot) \vdash^* (q, \sigma yz, X\alpha)$ are non-decreasing, and all the transitions in $(p, yz, \alpha) \vdash^* (r, z, \beta)$ are non-increasing.

Similarly, a one-turn SAPDA is an SAPDA where in any given computation, all stack branches make exactly one turn. As this criteria speaks of branches individually, the formal definition can be based on traces.\(^2\)

\(^1\)Note that computations of type (1) cannot be accepting.
\(^2\)The definition for one-turn SAPDA presented in [3, 2] is a direct definition on the conjunctive model. Both definitions are equivalent.
**Definition 4.15.** An SAPDA $A = (Q, \Sigma, \Gamma, q_0, \bot, \delta)$ is one-turn if its trace automaton $A_T$ is a one-turn PDA.

**Lemma 4.16.** If $A$ is a one-turn SAPDA then in all computations of $A$, each branch of in the computation makes at most one-turn.

**Proof.** Assume there is a computation where a branch makes more that one turn. It follows that the projection matching that branch will also make more than one turn. However, the projection is a computation in the trace automaton $A_T$ in contradiction to $A_T$ being one-turn. \qed

Note that the automaton from Example 3.8 is one-turn, while the automaton from Example 3.7 is not.

### 4.3 Equivalence Results

Similarly to the context-free case, one-turn SAPDA and LCG are equivalent.

**Theorem 4.17.** A language is accepted by a one-turn SAPDA if and only if it is generated by an LCG.

The proof of the “if” part of the theorem is presented in Subsection 4.3.1, and the proof of its “only if” part is presented in Subsection 4.3.2.

We precede the proof of Theorem 4.17 with the following immediate corollary.

**Corollary 4.18.** One-turn SAPDA are equivalent to Trellis Automata.

While both computational models employ a form of parallel processing, their behavior is quite different. To better understand the relationship between the models, see the proof of equivalence between LCG and Trellis Automata in [21]. As LCG and one-turn SAPDA are closely related, the equivalence proof provides intuition on the relation with one-turn SAPDA as well.

#### 4.3.1 Constructing a one-turn SAPDA from a Linear CG

For the purposes of our proof we assume that grammars do not contain $\epsilon$-rules, and that the start symbol does not appear in the right-hand side of any rule.
Let $G = (V, \Sigma, P, S)$ be a linear conjunctive grammar. Consider the SAPDA $A_G = (Q, \Sigma, \Gamma, q_0, \perp, \delta)$, where

- $\Gamma = V \cup \Sigma$ and $\perp = S$,
- $Q = \{q_u | \text{for } X \rightarrow (\cdots \& uYv \& \cdots) \in P \text{ and } z \in \Sigma^*, \ u = zu^\prime\}$,
- $q_0 = q_e$, and
- $\delta$ is defined as follows.

1. $\delta(q_e, \epsilon, X)$ is the union of
   a. $\{(q_e, w) | X \rightarrow w \in P\}$ and
   b. $\{(q_u, Y_1 v_1) \& \cdots \& (q_u, Y_k v_k) | X \rightarrow (u_1 Y_1 v_1 \& \cdots \& u_k Y_k v_k) \in P\}$,
2. $\delta(q_{u}, \sigma, X) = \{(q_u, X)\}$,
3. $\delta(q_e, \sigma, \sigma) = \{(q_e, \epsilon)\}$

The construction of $A_G$ is very similar to the construction of an SAPDA from a general CG as presented in Subsection 3.2.1. The construction is modified so as to result in a one-turn automaton. For example, transitions, such as transition 2, are used to avoid removing symbols from the stack in the increasing phase of the computation. Essentially, the symbols from $\Sigma$ that should be above the top-most variable in the stack-branch, are encoded by the state of the branch. Clearly, the language of $A_G$ as defined here is the same as the language obtained through the general construction, and therefore, $L(A_G) = L(G)$.

It remains to be shown that $A_G$ is a one-turn SAPDA. By Definition 4.15, this means that we need to prove that $A_{G_T}$ is a one-turn automaton. The proof is based on the following lemmas.

**Lemma 4.19.** Let $(q_0, uv, S) \vdash^* (q, v, \alpha)$ be a computation of the trace automaton $A_{G_T}$. If the transition 1.a was not used in the computation, then $\alpha = Xy$ for some $X \in V$ and $y \in \Sigma^*$. Moreover, all transitions in the computation are non-decreasing.

---

3That is, $u'$ is a suffix of $u$. 39
Proof. We prove the claim by induction on the length of the computation.

Basis: The claim trivially holds for zero-length computations.

Induction Step: Assume the claim holds for computations of length $i$. Let $(q_e, u\sigma v, S) \vdash_{i+1} (p, v, \alpha)$. As the induction hypothesis holds for the first $i$ steps of the computation, it follows that

$$(q_e, u\sigma v, S) \vdash^i (q, \sigma v, Xy) \vdash (p, v, \alpha).$$

We consider the options for the last transition in the computation.

- If $q = q_e$ then $\sigma = \epsilon$, and the transition must be of type 1.b. Therefore, $\alpha = Yz_y$, and the transition is non-decreasing.
- If $q = q_{\sigma z}$ then the transition must be of type 2. It follows that $\alpha = Xy$, and the transition is non-decreasing.

Therefore, in either case, we have our claim.

Lemma 4.20. If $(q_e, uvw, S) \vdash^* (q_e, vw, Xy) \vdash (q_e, vw, xy) \vdash^* (p, w, \alpha)$ is a computation of the trace automaton $A_{GT}$ such that $(q_e, vw, Xy) \vdash (q_e, vw, xy)$ is the first application of transition 1.a, then the following holds.

- $p = q_e$,
- $\alpha = z \in \Sigma^*$,
- all the transitions in $(q_e, vw, xy) \vdash^* (p, w, \beta)$ are of type 3 (and are therefore decreasing).

Proof. We prove the claim by induction on the length of the derivation $(q_e, vw, xy) \vdash^* (p, w, \alpha)$.

Basis: The claim trivially holds for zero-length computations.

Induction Step: Assume the claim holds for computations of length $i$. Let $(q_e, v\sigma w, xy) \vdash_{i+1} (p, w, \alpha)$. As the induction hypothesis holds for the first $i$ steps of the computation, it follows that

$$(q_e, v\sigma w, xy) \vdash^i (q_e, \sigma w, tz) \vdash (p, w, \alpha).$$
As \( \tau \in \Sigma \), the last transition in the computation can only be of type 3. Therefore, \( \tau = \sigma, p = q_e, \alpha = z \), the transition is decreasing, and we have our claim. \( \square \)

Now we can prove that \( A_G \) is a one-turn automaton.

**Proof.** We consider possible computations of the trace automaton \( A_{G_T} \). If transition 1.a is not used, then by Lemma 4.19, all steps in the computation are non-decreasing, and it is of type (1) from Definition 4.14. Otherwise, there must be at least one application of the transition 1.a., and by Lemma 4.20, the computation is of type (2) from Definition 4.14. By Definition 4.14, the trace automaton \( A_{G_T} \) is a one-turn PDA. Therefore, by Definition 4.15, \( A_G \) is also one-turn. \( \square \)

This completes the proof of the “if” direction of Theorem 4.17.

### 4.3.2 Constructing a Linear CG from a one-turn SAPDA

The proof is a variation on a similar proof for the classical case presented in [6]. For the purposes of simplification, we make several assumptions regarding the structure of one-turn SAPDA, namely that they are single-state and that their transitions write at most two symbols at a time. We prove that these assumptions are not limiting in the following two lemmas.

**Lemma 4.21.** For each one-turn SAPDA there is an equivalent single-state one-turn SAPDA.

**Proof.** Let \( A = (Q, \Sigma, \Gamma, q_0, \bot, \delta) \) be an SAPDA. Consider the single-state SAPDA \( A' = (\Sigma, \Gamma', \bot', \delta') \), where

1. \( \Gamma' = Q \times \Gamma \times Q \cup \{ \bot' \} \), and
2. \( \delta' \) is defined as follows.
   - \( \delta'(\epsilon, \bot') = \{ [q_0, \bot, q] \mid q \in Q \} \), and
   - for all \( \sigma \in \Sigma \cup \{ \epsilon \} \) and all \( [q, X, p] \in \Gamma' \), \( \delta'(\sigma, [q, X, p]) \) consists of \( \epsilon \), if \( (p, \epsilon) \in \delta(q, \sigma, X) \), and all conjunctions of the form

\[ \text{Renaming } \bot', \text{ if necessary, we may assume that } \bot' \not\in Q \times \Gamma \times Q. \]
\[ [p_1, Y_{1,1}, q_{1,2}] \cdots [q_{1,m_1}, Y_{1,m_1}, p] \land \cdots \land [p_n, Y_{n,1}, q_{n,2}] \cdots [q_{n,m_n}, Y_{n,m_n}, p], \]

such that \( q_{i,j} \in Q, i = 1, \ldots, n, i = 1, \ldots, m_i, \) and

\[(p_1, Y_{1,1} \cdots Y_{1,m_1}) \land \cdots \land (p_n, Y_{n,1} \cdots Y_{n,m_n}) \in \delta(q, \sigma, X).\]

The stack symbol \([q, X, p]\) represents the current state \(q\) and the current top stack symbol \(X\). The state \(p\) needs to be saved so that, when \(X\) is erased from a stack by a computation of \(A\), \(A'\) can determine whether the computation was legal.

Namely, a symbol \([q, X, p]\) is erased from the stack if and only if there is a series of transitions in \(A\) starting at state \(q\), ending at state \(p\), and emptying \(X\). If the series consists of one step only, then this means that there is a direct transition from \(q\) to \(p\) emptying \(X\). When two symbols are on top of the other in the stack, they are always of the form \([q_1, X_1, p]\)\([p, X_2, q_2]\).

Therefore, if \([q_1, X_1, p]\) is emptied by a transition of \(A\) from \(q_1\) to \(p\), then the next exposed symbol \([p, X_2, q_2]\) does in fact show \(p\) to be the current state of \(A\) and thus the correctness is maintained.

The above intuitive explanation is formalized by the claim that, for all \([q, X, p] \in \Gamma'\) and all \(w \in \Sigma^*\),

\[(q, w, X) \vdash^*_\delta (p, \epsilon, \epsilon) \text{ if and only if } (w, [q, X, p]) \vdash^*_{A'} (\epsilon, \epsilon).\]

We prove the “only if” part of the claim by induction on the number of computation steps of \(A\).

**Basis:** If \((q, w, X) \vdash_A (p, \epsilon, \epsilon)\), then \(w = \sigma \in \Sigma \cup \{\epsilon\}\) and \((p, \epsilon) \in \delta(q, \sigma, X)\). By the definition of \(\delta'\), \(\epsilon \in \delta'(\sigma, [q, X, p])\), implying \((w, [q, X, p]) \vdash_{A'} (\epsilon, \epsilon)\).

**Induction step:** Assume the claim holds for all computations of \(A\) shorter than \(i\) and let \((q, \sigma w, X) \vdash_i^A (p, \epsilon, \epsilon)\), where \(\sigma \in \Sigma \cup \{\epsilon\}\). Therefore, the computation is of the form

\[(q, \sigma w, X) \vdash ((p_1, w, Y_{1,1} \cdots Y_{1,m_1}) \land \cdots \land (p_n, w, Y_{n,1} \cdots Y_{n,m_n})) \vdash^* ((p, \epsilon, \epsilon) \land \cdots \land (p, \epsilon, \epsilon)) \vdash (p, \epsilon, \epsilon).

For each sub-computation \((p_j, w, Y_{j,1} \cdots Y_{j,m_j}) \vdash^* (p, \epsilon, \epsilon)\) there exist states \(q_{j,1}, \ldots, q_{j,m_j+1}, q_{j,1} = p_j, q_{j,m_j} = p\), such that for each \(j = 1, \ldots, m_j\),
the sub-computation is comprised of sub-computations of the form

\[(q_{j,k}, w_{j,k} \cdots w_{j,m_j}, Y_{j,k} \cdots Y_{j,m_j}) \vdash^* A (q_{j,k+1}, w_{j,k+1} \cdots w_{j,m_j}, Y_{j,k+1} \cdots Y_{j,m_j})\]

where \(Y_{j,k+1}\) is exposed as the top symbol of the stack branch only at the last step of the computation. Therefore,

\[(q_{j,k}, w_{j,k}, Y_{j,k}) \vdash^*_A (q_{j,k+1}, \epsilon, \epsilon)\]

Each of these sub-computations is shorter than \(i\). Hence, by the induction hypothesis, \((w_{j,k}, [q_{j,k}, Y_{j,k}, q_{j,k}]]) \vdash^*_A (\epsilon, \epsilon)\), implying

\[(w_{j,1} \cdots w_{j,m_j}, [p_1, Y_{j,1}, q_{j,2}] \cdots [q_{j,m_j}, Y_{j,m_j}, q_{m_j+1}]) \vdash^*_A (\epsilon, \epsilon)\]

for \(w_{j,1} \cdots w_{j,m_j} = w\). Thus, starting with the transition

\[[p_1, Y_{1,1}, q_{1,2}] \cdots [q_{1,m_1}, Y_{1,m_1}, p] \land \cdots \land [p_n, Y_{n,1}, q_{n,2}] \cdots [q_{n,m_n}, Y_{n,m_n}, p]\]

from \((\sigma w, [q, X, p])\), we obtain the desired computation

\[(\sigma w, [q, X, p]) \vdash^*_A (\epsilon, \epsilon)\].

We prove the “if” part of the claim by induction on the number of computation steps of \(A'\).

**Basis:** If \((w, [q, X, p]) \vdash A' (\epsilon, \epsilon)\), then, \(w = \sigma \in \Sigma \cup\{\epsilon\}\) and \(\epsilon \in \delta'(\sigma, [q, X, p])\). Therefore, \((p, \epsilon) \in \delta(q, \sigma, X)\), implying \((q, w, X) \vdash A (p, \epsilon, \epsilon)\).

**Induction step:** Assume the claim holds for all computations of \(A'\) shorter than \(i\) and let \((\sigma w, [q, X, p]) \vdash^*_A (\epsilon, \epsilon)\). Therefore, the computation is of the form

\[(\sigma w, [q, X, p]) \vdash^*_A ((w, [p_1, Y_{1,1}, q_{1,2}] \cdots [q_{1,m_1}, Y_{1,m_1}, p]) \land \cdots \land (w, [p_n, Y_{n,1}, q_{n,2}] \cdots [q_{n,m_n}, Y_{n,m_n}, p]) \vdash^*_A ((\epsilon, \epsilon) \land \cdots \land (\epsilon, \epsilon)) \vdash A' (\epsilon, \epsilon)\).

For each sub-computation \((w, [p_j, Y_{j,1}, q_{j,2}] \cdots [q_{j,m_j}, Y_{j,m_j}, p]) \vdash^*_A (\epsilon, \epsilon)\)

there exist states \(q_{j,1}, \ldots, q_{j,m_j+1}\), \(q_{j,1} = p_j\), such that for each \(j = 1, \ldots, m_j\),

the sub-computation is comprised of sub-computations of the form

\[(w_{j,k} \cdots w_{j,m_j}, [q_{j,k}, Y_{j,k}, q_{j,k+1}] \cdots [q_{j,m_j}, Y_{j,m_j}, q_{j,m_j+1}])\]
\[ \vdash_{A'}^* (w_{j,k+1} \cdots w_{j,m_j}, [q_{j,k+1}, Y_{j,k+1}, q_{j,k+2}] \cdots [q_{j,m_j}, Y_{j,m_j}, q_{j,m_j+1}]), \]

where \([q_{j,k+1}, Y_{j,k+1}, q_{j,k+2}]\) is exposed as the top symbol of the stack branch only at the last step of the computation. Therefore,

\[ (w_{j,k}, [q_{j,k}, Y_{j,k}, q_{j,k+1}]) \vdash_{A'}^* (\epsilon, \epsilon). \]

Each of these sub-computations is shorter than \(i\). Hence, by the induction hypothesis, \((q_{j,k}, w_{j,k}, Y_{j,k}) \vdash_{A}^* (q_{j,k+1}, \epsilon, \epsilon)\). Therefore, for \(w_{j,1} \cdots w_{j,m_j} = w\),

\[ (w, p_1, Y_{j,1} \cdots Y_{j,m_j}) \vdash_{A}^* (p, \epsilon, \epsilon). \]

Thus, starting with the transition

\[ (p_1, Y_{1,1} \cdots Y_{1,m_1}) \land \cdots \land (p_k, Y_{k,1} \cdots Y_{k,m_k}) \in \delta(q, \sigma, X) \]

from \((q, \sigma w, X)\), we obtain the desired computation

\[ (q, \sigma w, X) \vdash_{A}^* (p, \epsilon, \epsilon). \]

It follows that \(L(A') = L(A)\). Moreover, the proof of the claim shows that each computation of \(A'\) from the second step\(^5\) corresponds step by step to a accepting computation of \(A\) and vice versa, and at each corresponding step of corresponding computations, the stack height in both \(A'\) and \(A\) is the same. Therefore, as \(A\) is a one-turn SAPDA, so is \(A'\).

\(\square\)

**Definition 4.22.** Let \(A = (\Sigma, \Gamma, \bot, \delta)\) be a single-state SAPDA. We say that \(A\) is **bounded** if for all \(\sigma \in \Sigma \cup \{\epsilon\}\) and all \(X \in \Gamma\) the following holds.

- For every \(\alpha \in \delta(\sigma, X)\), \(|\alpha| \leq 2\).
- For every \(\alpha_1 \land \cdots \land \alpha_n \in \delta(\sigma, X)\), \(n \geq 2\), \(|\alpha_i| = 1, i = 1, \ldots, n\).

**Lemma 4.23.** *Every single-state one-turn SAPDA is equivalent to a bounded single-state one-turn SAPDA.*

**Proof.** The proof idea is quite standard. We construct an equivalent automaton \(A'\), where all words \(\alpha \in \Gamma^*\) written to the stack are of length two

\(^5\)This is because the first step of a computation of \(A'\) is \((w, \bot') \vdash (w, [q_0, \bot, q]),\) for some \(q \in Q\).
or shorter. For this we encode “long” words with special stack symbols, and then repetitively write the word to the stack, each time adding another symbol.

Let \( A = (\Sigma, \Gamma, \bot, \delta) \) be a single-state one-turn SAPDA, and let the subsets \( B_1 \) and \( B_2 \) of \( \Gamma^* \) be as follows.

- \( B_1 = \{ \alpha \mid \text{for some } \sigma \in \Sigma \cup \{\epsilon\} \text{ and } X \in \Gamma, (\cdots \alpha \land \cdots) \in \delta(\sigma, X) \} \).
  That is, \( B_1 \) is the set of all words over \( \Gamma \) that appear as conjuncts in the transitions of \( \delta \).

- \( B_2 = \{ \alpha \mid \text{and for some } \beta, \alpha \beta \in B_1 \} \).
  That is, \( B_2 \) is the set of all prefixes of the words in \( B_1 \).

Consider the SAPDA \( A' = (\Sigma, \Gamma', \bot, \delta') \), where

- \( \Gamma' = \Gamma \cup \{ \alpha \mid \alpha \in B_2 \} \), and

- \( \delta' \) is defined as follows.

  1. For all \( \sigma \in \Sigma \cup \{\epsilon\} \) and \( X \in \Gamma \),
     \[ \delta'(\sigma, X) = \{ \alpha_1 \land \cdots \land \alpha_n \mid \alpha_1 \land \cdots \land \alpha_n \in \delta(q, X) \} \],
  2. for all \( \alpha X \in B_2 \), such that \( \alpha \neq \epsilon \), \( \delta'(\epsilon, \alpha X) = \{ \alpha X \} \),
  3. for all \( X \in \Gamma \), \( \delta'(\epsilon, X) = \{ X \} \), and
  4. \( \delta'(\epsilon, \epsilon) = \{ \epsilon \} \).

That is, when \( A \) writes a word \( \alpha \) to the stack, \( A' \) writes \( \alpha \) to the stack instead. Subsequent transitions can only be of type (2), (3), or (4), ultimately replacing \( \alpha \) with \( \alpha \) in the stack, as \( A \) would have written in the first place. Thus, it is easily seen that \( A \) and \( A' \) are equivalent.

We also have the following correspondence between computation steps of the trace automaton \( A_T \) and \( A'_T \).

- Each non-changing computation step of \( A_T \) “is replaced” with two non-changing computation steps of \( A'_T \), where the first is of type (1) and the second is of type (3).

- Each decreasing computation step of \( A_T \) “is replaced” in \( A'_T \) with a non-changing computation step of type (3) followed by a decreasing computation step of type (4).
• Each increasing computation step of $A_T$ based on writing some $\alpha$ to
the stack “is replaced” in $A'_T$ with non-changing computation step
of type (1) followed by a number of increasing computation steps of
type (2).

Consequently, since $A$ is a one-turn SAPDA, $A_T$ is a one-turn PDA and
therefore, $A'_T$ is also a one-turn PDA. It follows that $A'$ is a one-turn SAPDA
as well.

We can now proceed to the proof of the “if” part of Theorem 4.17. Let
$A$ be a one-turn SAPDA. By Lemmas 4.21 and 4.23, we may assume that
$A$ is single-state and bounded.

So, let $A = (\Sigma, \Gamma, \perp, \delta)$. Consider the linear conjunctive grammar
$G_A = (V, \Sigma, P, S)$, where

- $V = \Gamma \times (\Gamma \cup \{\epsilon\})$,
- $S = [\perp, \epsilon]$, and
- $P$ is the union of the following sets of rules, for all $\sigma \in \Sigma \cup \{\epsilon\}$ and all $X, Y, Z \in \Gamma$.

1. $\{[X, Y] \rightarrow \sigma[Z, \epsilon] \mid ZY \in \delta(\sigma, X)\}$,
2. $\{[X, Y] \rightarrow \sigma[Z, Y] \mid Z \in \delta(\sigma, X)\}$,
3. $\{[X, \epsilon] \rightarrow \sigma[Z, \epsilon] \mid Z \in \delta(\sigma, X)\}$,
4. $\{[X, \epsilon] \rightarrow (\sigma[X_1, \epsilon] \& \cdots \& \sigma[X_n, \epsilon]) \mid
   n \geq 2 \text{ and } X_1 \land \cdots \land X_n \in \delta(\sigma, X)\}$,\(^6\)
5. $\{[X, \epsilon] \rightarrow \sigma \mid \epsilon \in \delta(\sigma, X)\}$,
6. $\{[X, Y] \rightarrow [X, Z]\sigma \mid Y \in \delta(\sigma, Z)\}$, and
7. $\{[X, \epsilon] \rightarrow [X, Z]\sigma \mid \epsilon \in \delta(\sigma, Z)\}$.

The grammar variables $[X, Y]$ correspond to zero- and one-turn computations starting with $X$ in the stack and ending with $Y$ in the stack. In
particular, any word derived from $[\perp, \epsilon]$ is a word with a one-turn emptying
computation of $A$.

The various types of rules defined, correspond to the different types of
transitions of the automaton.

\(^6\)Since $A$ is bounded, $X_i \neq \epsilon, i = 1, \ldots, n$. 46
• Rules of type 1 correspond to increasing computation steps,
• rules of type 2, 3 and 6 correspond to non-changing computation steps,
• rules of type 4 correspond to conjunctive transitions,
• rules of type 5 correspond to the turn step, and
• rules of type 7 correspond to decreasing computation steps.

The correctness of the construction follows from Proposition 4.24 below.

**Proposition 4.24.** For every \( X \in \Gamma, \ Y \in \Gamma \cup \{\epsilon\} \), and \( w \in \Sigma^* \), \([X, Y] \Rightarrow^* w\) if and only if \((w, X) \vdash^* (\epsilon, Y)\) such that all projections of the computation have exactly one turn.

**Proof.** We start with the proof of the “only if” part of the proposition, which is by induction on the length of the derivation in \( G_A \).

**Basis:** Let \( w \in \Sigma^* \) be such that \([X, Y] \Rightarrow w\). The only rules that derive a terminal word directly are rules of type 5. Therefore, \( w = \sigma \in \Sigma \cup \{\epsilon\} \) and \( \epsilon \in \delta(\sigma, X) \), implying \((\sigma, X) \vdash (\epsilon, \epsilon)\).

**Induction step:** Assume the “only if” part of the proposition holds for all derivations of \( G_A \) of length up to \( i \), and let \( w \in \Sigma^* \) be such that \([X, Y] \Rightarrow^{i+1} w\). We consider the various cases based on the first rule applied in this derivation. Note that it cannot be of type 5, as that would halt the derivation.

• If the first rule applied is of type 1, then for some \( \sigma \in \Sigma \cup \{\epsilon\} \) and some \( u \in \Sigma^* \), \( w = \sigma u \),

\[
[X, Y] \Rightarrow \sigma [Z, \epsilon] \Rightarrow^* \sigma u .
\]

By the definition of rules of type 1, \( ZY \in \delta(\sigma, X) \), and, by the induction hypothesis, \((u, Z) \vdash^* (\epsilon, \epsilon)\). Therefore,

\[
(\sigma u, X) \vdash (u, ZY) \vdash^* (\epsilon, Y) .
\]

As we are adding a first step that is increasing, the number of turns in any projection will still be one.
• If the first rule applied is of type 2 or 3, then for some \( \sigma \in \Sigma \cup \{\epsilon\} \) and some \( u \in \Sigma^* \), \( w = \sigma u \),

\[
[X, Y] \Rightarrow \sigma[Z, Y] \Rightarrow^n \sigma u .
\]

By the definition of rules of type 2 and 3, \( Z \in \delta(\sigma, X) \), and, by the induction hypothesis, \( (u, Z) \vdash^* (\epsilon, Y) \). Therefore,

\[
(\sigma u, X) \vdash (u, Z) \vdash^* (\epsilon, Y) .
\]

As we are adding a first step that is non-decreasing, the number of turns in any projection will still be one.

• If the first rule applied is of type 4, then \( Y = \epsilon \), for some \( \sigma \in \Sigma \) and some \( u \in \Sigma^* \), \( w = \sigma u \)

\[
[X, \epsilon] \Rightarrow (\sigma[X_1, \epsilon] \land \cdots \land \sigma[X_n, \epsilon]) \Rightarrow^{i-1} (\sigma u \land \cdots \land \sigma u) \Rightarrow \sigma u ,
\]

where \( n \geq 2 \). Thus, \( [X_j, \epsilon] \Rightarrow^i u \), \( j = 1, \ldots, n \), and, by the induction hypothesis, \( (u, X_j) \vdash^* (\epsilon, \epsilon) \), \( j = 1, \ldots, n \). By the definition of rules of type 4, \( X_1 \land \cdots \land X_n \in \delta(\sigma, X) \), which allows us to construct the following computation of \( A \) on \( w \).

\[
(\sigma u, X) \vdash ((u, X_1) \land \cdots \land (u, X_n)) \vdash^* ((\epsilon, \epsilon) \land \cdots \land (\epsilon, \epsilon)) \vdash (\epsilon, \epsilon) .
\]

In terms of the projections of the computation, each project now starts with a non-decreasing transition, and therefore the number of turns must be one.

• If the first rule applied is of type 6 or 7, then for some \( u \in \Sigma^* \) and some \( \sigma \in \Sigma \cup \{\epsilon\} \), \( w = u\sigma \),

\[
[X, Y] \Rightarrow [X, Z] \Rightarrow^i u\sigma .
\]

By the definition of rules of type 6 and 7, \( Y \in \delta(\sigma, Z) \), and by the induction hypothesis, \( (u, X) \vdash^* (\epsilon, Z) \). Therefore,

\[
(u\sigma, X) \vdash^* (\sigma, Z) \vdash (\epsilon, Y) .
\]
Here we have added a decreasing step at the end of the computation, and therefore the number of turns in each projection remains one.

The proof of the “if” part of the proposition is by induction on the length of the computation of $A$.

**Basis:** Let $w$ be a terminal word such that $(w, X) \vdash (\epsilon, Y)$. Since there is a turn in the projections of the computation, $Y = \epsilon$. Therefore, by the definition of rules of type 5, $[X, \epsilon] \Rightarrow \sigma$.

**Induction step:** Assume the “if” part of the proposition holds for all computations of $A$ of length up to $i$ and let $w \in \Sigma^*$ be such that $(w, X) \vdash^{i+1} (\epsilon, y)$ with exactly one turn in all projections of the computation. We consider the various cases based on the first transition applied in this computation. Note that in the first computation step, $A$ must write at least one symbol to the stack. Otherwise it will be empty after the first computation step, which contradicts the assumption that the computation is of $i + 1$ steps for $i \geq 1$.

- If the first step of the computation writes one symbol to the stack, then for some $\sigma \in \Sigma \cup \{\epsilon\}$ and some $u \in \Sigma^*$, $w = \sigma u$,

  $$(\sigma u, X) \vdash (u, Z) \vdash (u, Z) \vdash^{i} (\epsilon, Y) .$$

  Since projections of $(\sigma u, X) \vdash^{i+1} (\epsilon, Y)$ have exactly-one-turn and $(\sigma u, X) \vdash (u, Z)$ is non-changing, projections of $(u, Z) \vdash^{i} (\epsilon, Y)$ must have exactly-one-turn as well. By the definition of rules of type 2 or 3, $[X, Y] \rightarrow \sigma[Z, Y]$, and by the induction hypothesis, $[Z, Y] \Rightarrow^* u$. Therefore, $[X, Y] \Rightarrow \sigma[Z, Y] \Rightarrow^* \sigma u$.

- If the first step of the computation writes two symbols to the stack, then there are two possible cases for the corresponding transitions.

  - If $X$ is replaced with $ZY$, then for some $\sigma \in \Sigma \cup \{\epsilon\}$ and some $u \in \Sigma^*$, $w = \sigma u$,

    $$(\sigma u, X) \vdash (u, ZY) \vdash (u, ZY) \vdash^{i} (\epsilon, Y) ,$$

    implying $(u, Z) \vdash^{i} (\epsilon, \epsilon)$. Since projections of $(\sigma u, X) \vdash^{i+1} (\epsilon, Y)$ have exactly-one-turn and $(\sigma u, X) \vdash (u, ZY)$ is increasing, pro-
jections of \((u, ZY)\) \(\vdash^i (\epsilon, Y)\) must have exactly one-turn as well. By the definition of rules of type 1, \([X, Y] \rightarrow \sigma[Z, \epsilon]\), and, by the induction hypothesis applied to \((u, Z) \vdash^i (\epsilon, \epsilon)\), \([Z, \epsilon] \Rightarrow^* u\). Therefore,

\[ [X, Y] \Rightarrow \sigma[Z, \epsilon] \Rightarrow^* \sigma u \]

- If \(X\) is replaced with \(ZZ'\), where \(Z' \neq Y\), then for some \(u \in \Sigma^*\) and some \(\sigma \in \Sigma \cup \{\epsilon\}\), \(w = u\sigma\)

\[(u\sigma, X) \vdash^i (\sigma, Z'')\]

for some \(Z'' \in \Gamma\), implying \(Y \in \delta(\sigma, Z'')\). Since projections of \((\sigma u, X) \vdash^i \ (\epsilon, Y)\) have exactly one-turn and in \((u\sigma, X) \vdash^i \ (\sigma, Z'')\), \(Z\) has been emptied from the stack, projections of \((u\sigma, X) \vdash^n \ (\sigma, Z'')\) have exactly one-turn as well. By the induction hypothesis, \([X, Z''] \Rightarrow^* u\) and, by the definition of rules of type 6 and 7, \([X, Y] \Rightarrow [X, Z'']\sigma\). Therefore, \([X, Y] \Rightarrow [X, Z'']\sigma \Rightarrow^* u\sigma\).

- If the first step of the computation is a conjunctive transition \(X_1 \wedge \cdots \wedge X_n \in \delta(\sigma, X)\), then \(w = \sigma u\), where \(\sigma \in \Sigma \cup \{\epsilon\}\) and \(u \in \Sigma^*\). Therefore, the computation is of the form

\[(\sigma u, X) \vdash ((u, X_1) \wedge \cdots \wedge (u, X_n)) \Rightarrow^* ((\epsilon, \epsilon) \wedge \cdots \wedge (\epsilon, \epsilon)) \vdash (\epsilon, \epsilon)\]

Therefore, \((u, X_j) \vdash^{\leq i} (\epsilon, \epsilon), j = 1, \ldots, n\). Since projections of the computation \((\sigma u, X) \vdash^{i+1} (\epsilon, Y)\) have exactly one turn, and the first step in these projections is non-decreasing, it follows that projections of the sub-computations \((u, X_j) \vdash^i (\epsilon, \epsilon)\) have exactly one turn as well. By the induction hypothesis, for \(j = 1, \ldots, n\), \([X_j, \epsilon] \Rightarrow^* u\) and by the definition of rules of type 4, \([X, \epsilon] \rightarrow (\sigma[X_1, \epsilon] \& \cdots \& \sigma[X_n, \epsilon])\). Therefore, combining the above derivations, we obtain

\[[X, \epsilon] \Rightarrow (\sigma[X_1, \epsilon] \& \cdots \& \sigma[X_n, \epsilon]) \Rightarrow^* (\sigma u \& \cdots \& \sigma u) \Rightarrow \sigma u\]

Therefore, in all cases, we have our claim.

This completes our proof of the “if” part of Theorem 4.17.
Chapter 5

Conjunctive Languages

In the following chapter we explore various characteristics of the class of conjunctive languages. In Section 5.1, we explore the generative power of conjunctive grammars, comparing the class of conjunctive languages to other classes in the Chomsky hierarchy. In particular, we discuss the relation of conjunctive languages to the class of mildly context sensitive languages. In Section 5.2 we present the known closure properties for the class of conjunctive languages as well as the open questions remaining. We also present a new and much simplified proof for closure under inverse homomorphism, which is based on SAPDA. In Section 5.3 we discuss the complexity of various decision problems for conjunctive languages, and, in particular, we focus on known parsing algorithms for both conjunctive and linear conjunctive languages. Finally, in Section 5.4, we present a motivating example for the importance of conjunctive languages from the field of Compilation and Programming Languages.

5.1 Placement in the Chomsky Hierarchy

In this section we discuss the relation of the class of conjunctive languages to other well known language classes from the Chomsky Hierarchy. As conjunctive languages are polynomially parsable, see Section 5.3.1, they are contained in the polynomial-time languages. However, there is no known general method for proving that a language is not conjunctive, and therefore the exact placing in the hierarchy is not clear.
Clearly, the class of finite intersections of context-free languages is included in the class of conjunctive languages. The inclusion is proper, as demonstrated by the language \( \{ w\$w \mid w \in \Sigma^* \} \), which is conjunctive by Examples 2.8 and 3.8, but is known not to be the finite intersection of context-free languages. Similarly, the class of finite intersections of linear languages is included in the class of linear conjunctive languages. As \( \{ w\$w \mid w \in \Sigma^* \} \) is linear conjunctive, it demonstrates that the inclusion in this case is also proper. The language \( \{ a^n b^n c^n \mid n \geq 1 \} \) from Example 2.7 is both a finite intersection of context-free languages and linear conjunctive but is not context-free. In Okhotin’s overview paper [22], he demonstrates that there is a language that is both linear conjunctive and context-free but not linear, and that there is a conjunctive language that is not linear conjunctive and not context-free. Therefore, the known relationships between these language classes are as depicted in Figure 5.1.

It is a well known fact that context-free languages over unary alphabets are context-free. In [19] Okhotin proved that the same holds for linear
conjunctive languages. In 2006, Okhotin posted a technical report with nine open questions regarding conjunctive grammars. One of them was whether general conjunctive grammars over unary alphabets could generate non-regular languages. The conjecture was that they should not be able to do so. In 2007, Artur Jez proved that the conjecture was, in fact, incorrect, by showing a conjunctive grammar for the language \( \{a^{4n} \mid n \geq 0\} \).

5.1.1 A Discussion of Mildly Context Sensitive Languages

The field of computational linguistics focuses on defining a computational model for natural languages. Originally, context-free languages were considered, and many natural language models are in fact models for context-free languages. However, certain natural language structures cannot be expressed in context free languages. This led to an interest in a slightly wider class of languages that came to be known as mildly context-sensitive languages (MCSL). Several formalisms for grammar specification are known to converge to this class [24].

Mildly context sensitive languages are loosely categorized as having the following properties:

1. They contain the context-free languages
2. They contain such languages as multiple-agreement, cross-agreement and reduplication
3. They are polynomially parsable
4. They are semi-linear\(^1\)

It is clear that there is a strong relation between the class of conjunctive languages and the class of mildly context sensitive languages. The first criterion of MCSL is obviously met, as both CG and SAPDA contain their context free counterparts, CFG and PDA respectively. The third criterion is also met by Okhotin’s proof that CG membership is polynomial.

Multiple-agreement and cross-agreement are covered as shown in Examples 2.7, 2.12 respectively. Reduplication with a center marker is shown in Example 2.8.

\(^1\)A language \( L \) is \textit{semi-linear} if \( \{|w| \mid w \in L\} \) is a finite union of sets of integers of the form \( \{l + im \mid i = 0, 1, \ldots\} \), \( l, m \geq 0 \).
Surprisingly, it is the fourth criterion of semi-linearity, which is not met, as demonstrated, e.g., by the following example due to Okhotin [19].

Example 5.1. The following linear conjunctive grammar derives the non-context-free language \( \{ ba^2ba^4 \cdots ba^{2n} b | n \in \mathbb{N} \} \). \( G = (V, T, P, S) \) where:

- \( V = \{ S, A, B, C, D, U, V \} \), \( T = \{ a, b \} \),
- \( P \) contains the following derivation rules:
  
  \[
  S \rightarrow (U \& V) | b \\
  U \rightarrow Ua | Ub | b ;; V \rightarrow Ab | (B \& D) \\
  A \rightarrow aA | a ;; B \rightarrow Ba | Bb | Cb \\
  C \rightarrow aCa | baa ;; D \rightarrow aD | bV
  \]

For more details regarding this example see [19, Example 2].

The language generated in Example 5.1 has super-linear growth, which means that CG and SAPDA accept languages that are not mildly context-sensitive. In this respect, it may be that the CG and SAPDA models are stronger than needed for natural language processing.

5.2 Closure Properties

Conjunctive languages can easily be shown to be closed under union, intersection, concatenation, and Kleene star. The proofs are straightforward using either CG or SAPDA. Okhotin showed in [14] that conjunctive languages are not closed under homomorphism, prefix, suffix and substring. Two main questions remain open: whether conjunctive languages are closed under complement, and whether they are closed under \( \epsilon \)-free homomorphism.

Linear conjunctive languages are also closed under union, intersection and Kleene start, by the same proofs as for the general case. Interestingly, linear conjunctive languages are closed under complement. Okhotin proved this using linear conjunctive grammars in [19]. It is also derivable from the fact that LCG are equivalent to trellis automata [21], and from the fact that trellis automata are closed under complement.

In [19], Okhotin also proved that linear conjunctive languages are also closed under a limited form of concatenation. First Okhotin proved that
\( L_1 \cdot L_2 \) is linear conjunctive if \( L_1 \) is linear conjunctive and \( L_2 \) is regular, or vice versa. This can easily be proven using one-turn SAPDA, because when an SAPDA simulates a finite state automaton it makes no use of its stack, and therefore adding such a simulation in the beginning or the end of a computation will not change the number of turns. Then, Okhotin expanded this result to concatenation through a center marker, i.e., \( L_1 \$ L_2 \) for two linear conjunctive languages and a center marker \( \$ \) not from the alphabet, and also for \( L_1 \cdot L_2 \) where both languages are linear conjunctive, but over different alphabets. In both these cases, the factorization of the word is clear, and therefore, by a simple intersection, e.g., \( L_1 \$ \Sigma^* \cap \Sigma^* \$ L_2 \), we reduce the problem to the previous case. Therefore, Okhotin generalized the result to closure under concatenation where there is only one factorization for any given word in the resulting language.

Both general conjunctive languages and linear conjunctive languages are closed under inverse homomorphism. We elaborate about this closure property and its proof in the following section.

5.2.1 A Simple Proof for Closure Under Inverse Homomorphism

In 2003 Okhotin published a technical report proving that conjunctive languages are closed under inverse homomorphism, see [17]. The proof is a direct proof using conjunctive grammars, and is very involved. The proof of closure under inverse homomorphism of linear conjunctive languages was presented by Okhotin in [21], and is based on their equivalence to trellis automata.

The classical proof of closure under inverse homomorphism for context-free languages utilizes PDA, as they are a more convenient model for this case. Therefore, it stands to reason that proving this closure property using SAPDA would be simpler. As we show in this section, that is in fact the case. The proof is a straightforward extension of the classical one, see e.g. [7, Theorem 6.3, pp 132–134], and is much simpler than the grammar based one. Moreover, the same proof holds for both general conjunctive languages, and their linear subset.

Before presenting the proof, we recall the definitions of homomorphisms and inverse homomorphisms.
Definition 5.2. Let $\Sigma, \Delta$ be two finite alphabets. A mapping of the form $h : \Sigma \rightarrow \Delta^*$ is called a homomorphism. We extend homomorphisms to words by the following recursion.

- $h(\epsilon) = \epsilon$
- $h(\sigma w) = h(\sigma) \cdot h(w)$

The homomorphic image of a language $L$ is defined by $h(L) = \{ h(w) \mid w \in L \}$.

Definition 5.3. Let $h : \Sigma \rightarrow \Delta^*$ be a homomorphism. The inverse homomorphism of $h$ is a mapping $h^{-1} : \Delta^* \rightarrow 2^{\Sigma^*}$ where

$$ h^{-1}(w) = \{ x \in \Sigma^* \mid h(x) = w \} . $$

The inverse homomorphic image of a language $L$ is defined by $h^{-1}(L) = \bigcup_{w \in L} h^{-1}(w)$. An alternative equivalent definition is

$$ h^{-1}(L) = \{ w \in \Sigma^* \mid h(w) \in L \} . $$

We now present the proof of closure under inverse homomorphism. The construction we use is the same construction used in the classical proof. Let $A = (Q, \Delta, \Gamma, q_0, \bot, \delta)$ be an SAPDA, and let $h : \Sigma \rightarrow \Delta^*$ be a homomorphism. Let

$$ U = \{ u \in \Delta^* \mid u \text{ is a suffix of } h(\sigma) \text{ for some } \sigma \in \Sigma \} . $$

Note that $U$ is a finite set.

The idea is to simulate the computation of $A$ on some word $h(w)$. To do so, we add a buffer to each state that can contain suffixes of $h(\sigma)$ for $\sigma \in \Sigma$ (this is the set $U$). For each character $\sigma$ read from the input, we store $h(\sigma)$ in the buffer, and simulate $A$ on the contents of the buffer. When the buffer is empty, we read the next input symbol and start again. As the buffer is attached to the state, when a conjunctive transition is applied, each branch receives its own buffer where it simulates the relevant branch in $A$’s computation. Sibling branches in $A$ can collapse only if they are synchronized both in state and in the next input symbol. Therefore, in our
simulation, we will have synchronization of both the state and the buffer contents, and the branches can be collapsed as well.

We define an SAPDA \( A' = (Q', \Sigma, \Gamma', q'_0, \perp', \delta') \), where

- \( Q' = \{ [q, u] \mid q \in Q \text{ and } u \in U \} \cup \{ q'_0 \} \),
- \( \Gamma' = \Gamma \cup \{ \perp' \} \), and
- \( \delta' \) is defined as follows, where \( X \in \Gamma, \sigma \in \Sigma, \) and \( \tau \in \Delta \cup \{ \epsilon \} \).

1. \( \delta'(q'_0, \epsilon, \perp') = \{ ([q_0, \epsilon], \perp \perp') \} \)
2. \( \delta'([q, \tau u], \epsilon, X) = \{ ([p_1, u], \alpha_1) \wedge \cdots \wedge ([p_n, u], \alpha_n) \mid \)
   \( (p_1, \alpha) \wedge \cdots \wedge (p_n, \alpha_n) \in \delta(q, \tau, X) \} \)
3. \( \delta'([q, \epsilon], \sigma, X) = \{ ([q, h(\sigma)], X) \} \)
4. \( \delta'([q, \epsilon], \epsilon, \perp') = \{ ([q, \epsilon], \epsilon) \} \)

Clearly, for any input \( w \), \( A' \) simulates the computation of \( A \) on \( h(w) \).

In particular, let \( h(w\sigma z) = xuvy \) where \( h(\sigma) = uv \). Then, \( (q_0, xuvy, \perp) \vdash^*_A T \), where the remaining input in all branches of \( T \) is \( vy \), if and only if \( (q'_0, w\sigma z, \perp') \vdash^*_{A'} T' \) where there remaining input in all branches of \( T' \) is \( z \), and \( T' \) can be obtained from \( T \) as follows.

- All internal nodes of \( T \) and \( T' \) are identical.
- Every leaf node of the form \( (q, vy, \alpha) \) in \( T \) corresponds to a leaf node in \( T' \) labeled \( ([q, v], y, \alpha) \).

Therefore,

\( (q_0, w, \perp) \vdash^*_A (p, \epsilon, \epsilon) \) if and only if \( (q'_0, w, \perp') \vdash^*_{A'} ([p, \epsilon], \epsilon, \epsilon) \),

and so \( w \in L(A') \) if and only if \( h(w) \in L(A) \), and we have closure under inverse homomorphism.

Note that in the construction, the stack contents of \( A' \) is identical to that of \( A \), except for the \( \perp' \) symbol that is inserted in the first transition, and removed in the last. Therefore, if \( A \) is one-turn, \( A' \) is also one-turn.

It follows that by the same proof that linear conjunctive languages are also closed under inverse homomorphism.

57
5.3 Decision Problems

Many classical decision problems are known to be undecidable for the class of finite intersections of context-free languages. As this class is included in the class of conjunctive languages, these same undecidability results carry over. In particular, the problems of emptiness, finiteness, equivalence, inclusion, and regularity are undecidable for conjunctive languages.

The redeeming feature of the language class, is that membership, or parsing, is polynomial. This is what makes the class viable for use in practical applications. In the following section we give an overview of existing parsing methods for conjunctive languages and their linear subset. In the next chapter, we present our new $LR(0)$ parsing algorithm.

5.3.1 Parsing

Several of the classical parsing algorithms for context-free languages have been modified to deal with the class of conjunctive languages, generally maintaining the same space and time complexity. In his original paper from 2001 [14], Okhotin presented an extension of the CYK algorithm. The algorithm can be applied to conjunctive grammars in binary normal form, which is a straightforward extension of the classical Chomsky normal form. Like in the classical case, a matrix is populated with potential sub-computations that ultimately combine to show a full computation on a given word. Also like in the classical case, the time complexity is $O(n^3)$ and the space complexity is $O(n^2)$. When applied to linear conjunctive grammars, the time complexity is $O(n^2)$ and the space complexity is $O(n)$.

In 2002, Okhotin presented a notion of $LL(k)$ parsing for conjunctive grammars [16]. The method is an extension of classical $LL(k)$ parsing, in which the parser attempts to construct a leftmost derivation of the grammar, while looking $k$ characters ahead in the input. To do so, the parser utilizes a tree-structured stack, instead of the standard PDA used in the classical construction. In general, the algorithm is exponential. However, it runs in linear time for the intersection closure of the classical $LL(k)$ languages.

Also in 2002, Okhotin presented a generalized $LR$ parsing algorithm, [15]. The algorithm is an extension of Tomita’s Generalized $LR$ parsing algorithm [23] for CG. As opposed to classical $LR$ parsing, Tomita’s generalized version uses a graph-structured stack to process various possible derivations,
and is applicable to all context-free grammars. In general, it runs in polynomial time, but is linear for the classical $LR$ languages. Okhotin’s extension uses a similar graph-structured pushdown to process both non-deterministic choice, as well as conjunctive transitions. The algorithm works for all conjunctive grammars in polynomial time, and is linear when applied to the boolean closure of classical $LR$ languages.

Note that both Okhotin’s $LL$ parsing and his generalized $LR$ parsing use a type of specialized graph- or tree-structured pushdown data-structure. Essentially, both the structures are essentially particular cases of our SAPDA. It makes sense that this is the case, as the classical counterparts of these algorithms were based on PDA. We believe that our SAPDA can give a unified formal approach to these algorithms. A particular case is demonstrated in Chapter 6, where we present an SAPDA-based $LR(0)$ parsing algorithm for conjunctive languages.

5.4 A Motivating Example

In this section we give a simple example of how conjunctive languages could impact the field of Compilation and Programming Languages. One of the main practical uses of context-free languages, has been to define the specifications of programming languages. The reason for this is that context-free language can be parsed quickly and efficiently, making the compilation process effective. However, real-world programming languages are not strictly context-free. While their basic syntax can in fact be defined by a context-free grammars, many additional criteria are not. For example, checking that variables have been defined before they are referenced in the program is inherently not context-free, as it greatly resembles the reduplication language \{w$w \mid w \in \Sigma^*\}.

In the following we present a partial definition of a toy programming language $PrintVars$. A program in $PrintVars$ has three parts.

1. The word $VARS$ followed by a definition of variables.
2. The word $VALS$ followed by an assignment of values to the variables.
3. The word $PRNT$ followed by a list of variables to be printed.
We say that a PrintVars program is well-formed if it meets the following criteria.

1. It has the correct structure (syntax).
2. All used variables are defined.
3. All defined variables are used (i.e., are printed).
4. All defined variables are assigned a value.
5. All variables assigned a value are defined.

The following is a well-formed PrintVars program.

```
VARS a, b, c
VALS b = 2, a = 1, c = 3
PRNT b, a, a, c, b
```

The output of the above program is: 2 1 1 3 2.

Consider the definition of a well-formed PrintVars program. While the first criterion is easily defined by a context-free grammar, criteria (2) – (5) are not context-free. Therefore, if the compiler is based on context-free parsing, it must perform two steps. First, it must check that the program is syntactically correct by parsing it using a context-free grammar describing criterion (1). Then it must perform a semantic check of criteria (2) – (5). However, criteria (2) – (5) are not really semantic, but rather syntactic. The only reason that they are not verified during parsing is because they are not context-free.

As mentioned before, criteria (2) – (5) highly resemble the language \{w$w \mid w \in \Sigma^*\}. By Example 2.8, this is a conjunctive language, and in fact, criteria (2) – (5) can be described by a conjunctive grammar. Consider the following partial specification of PrintVars using a conjunctive grammar.

\[
S \rightarrow (S_1 \& S_2 \& S_3 \& S_4 \& S_5) \\
S_1 \rightarrow \text{vars vals prnt} \\
\text{vars} \rightarrow \text{VARS...} \\
\text{vals} \rightarrow \text{VALS...} \\
\text{prnt} \rightarrow \text{PRNT...} \\
S_2 \rightarrow \text{VARS checkS}_2
\]
The variables $S_i$ each derive the language meeting criterion $i$. Therefore, $S$ derives only well-formed programs. We discuss the rules of $S_2$. $S_2$ derives the language where all variables defined in the \texttt{VARS} section are eventually used in the \texttt{PRNT} section. The variable $\text{check}_S$ verifies that this criterion is met. Under the simplifying assumption that variable names are letters from the English alphabet, $\text{check}_S$ considers the case where the first variable is $a$, or $b$, etc..., making sure that this variable name appears in the \texttt{PRNT} section. Then it recursively checks the second variable, etc.

If this grammar were to be used for the parsing phase in a compiler, then all the specified criteria would be checked in one pass of the parser, without the need for additional semantic checks. Of course, many similar scenarios where conjunctive grammars can enrich syntactic program specifications are possible, making them an interesting candidate for practical applications in this field.
Chapter 6

Deterministic SAPDA and $LR(0)$ Grammars

Many of the classical context-free parsing algorithms have been extended and adapted to conjunctive languages, while maintaining both the spirit and their complexity. One major gap is the lack of $LR$ parsing algorithms. In [9], Knuth presented a linear time $LR$ parsing algorithm for a strong sub-family of context-free languages, which quickly became the basis of modern-day compilation theory. As such, the development of $LR$ parsing algorithms for a strong sub-family of conjunctive languages is an important steps towards positioning them as a language class with practical use.

Knuth based his $LR$ parsing algorithm on a sub-class of context-free grammars he introduced, called $LR(k)$ grammars. He showed that these grammars are equivalent to deterministic PDA, and therefore generate the deterministic context-free languages. The linear-time parser he developed was strongly based on the equivalence, and was in fact a deterministic PDA. Furthermore, Knuth proved that $LR(0)$ languages (those that can be parsed with no lookahead) are equivalent to deterministic PDA that accept by empty stack.

As PDA played an important role in the classical $LR$ parsing algorithm, our SAPDA is a good starting point for an extended $LR$ parsing algorithm for conjunctive languages. Our third contribution is to introduce a sub-family of SAPDA, $Deterministic$ $Synchronized$ $Alternating$ $Pushdown$ $Automata$ (DSAPDA), and a sub-family of CG, $LR(0)$ Conjunctive Grammars.
We show that these sub-families are equivalent, analogously to the classical case. Furthermore, we present a sophisticated and efficient implementation of DSAPDA, which, in particular, forms the basis of a deterministic linear time parsing algorithm $LR(0)$ conjunctive languages. This class of languages properly contains the classical $LR(0)$, thus expanding upon the classical results. As such, it could prove interesting for potential applications in, e.g., Programming Languages and Natural Language Parsing.

In Section 6.1 we introduce deterministic SAPDA as a sub-family of general SAPDA, and in Section 6.2 we prove that the membership problem for DSAPDA is decidable in linear time. In Section 6.3 we introduce $LR(0)$ conjunctive grammars as a sub-family of general CG, and present a deterministic SAPDA based parser. In Section 6.4 we construct an $LR(0)$ conjunctive grammar from a deterministic SAPDA, proving that the two models are equivalent.

### 6.1 Deterministic SAPDA Model Definition

We define the notion of a deterministic SAPDA analogously to the classical definition of a deterministic PDA.

**Definition 6.1.** An SAPDA $A = (Q, \Sigma, \Gamma, q_0, \delta, \bot)$ is deterministic if

- If $\delta(q, \sigma, X) \neq \emptyset$ for some $\sigma \in \Sigma$, then $\delta(q, \epsilon, X) = \emptyset$.
- For all $q \in Q, \sigma \in \Sigma \cup \{\epsilon\}, X \in \Gamma : |\delta(q, \sigma, X)| \leq 1$.

The first condition prevents a choice between reading the next input symbol and making an $\epsilon$-transition, while the second condition prevents a choice on the same input or a choice of $\epsilon$-transitions. Intuitively, this means that a deterministic SAPDA has at most one computation on any given input word.\(^2\)

Consider the following non-context-free language.

$$L_{inf} = \{ a^{i_1} b a^{i_2} b^2 \cdots a^{i_n} b^n \ \& \ ba^{i_1} ba^{i_2} \cdots ba^{i_n} \ \& \ n \geq 1 \ \& \ i_1, \ldots, i_n \geq 1 \}.$$\(^3\)

\(^1\)Note that SAPDA accept by empty stack.

\(^2\)Of course, up to permutations on the order in which the branches were chosen for transitions.
In the following example, we construct a DSAPDA that accepts the language $L_{inf}$.

**Example 6.2.** To build an automaton for $L_{inf}$ we construct two automata, each in charge of a specific aspect of the language. The first automaton, $A_1 = (Q_1, \Sigma, \Gamma, 0_1, \bot, \delta_1)$, checks that the series of $b$s before the first $\$" sign starts at 1 and increases by 1 at each step. The second automaton, $A_2 = (Q_2, \Sigma, \Gamma_2, 0_2, \bot, \delta_2)$, checks that the numbers of $a$s before and after the first $\$ match up appropriately. Assuming these two automata are constructed properly, we can define a DSAPDA for $L_{inf}$ as follows.

Let $A = (Q, \Sigma, \Gamma, r_0, \bot, \delta)$ where

- $Q = Q_1 \cup Q_2 \cup r_0$,
- $\Sigma = \{a, b, \$\}$,
- $\Gamma = \Gamma_1 \cup \Gamma_2$, and
- $\delta = \delta_1 \cup \delta_2 \cup \{(p_0, \bot) \land (q_0, \bot) \in \delta(r_0, \epsilon, \bot)\}$.

The DSAPDA $A_1$ is defined as follows. $Q_1 = \{p_0, p_0', p_0'', p_1, p_1', p_2, p_2', p_3, p_4, q_e, q_b\}$, $\Gamma_1 = \{b, \bot\}$,

1. $\delta_1(p_0, \epsilon, \bot) = (p_0, \bot) \land (p_0', \bot)$
2. $\delta_1(p_0, a, \bot) = (p_0, \bot)$
3. $\delta_1(p_0, b, \bot) = (p_0, b)$
4. $\delta_1(p_0, a, b) = (p_3, \bot)$
5. $\delta_1(p_0, \$, b) = (p_e, \bot)$
6. $\delta_1(p_0', a, \bot) = (p_0', \bot)$
7. $\delta_1(p_0', b, \bot) = (p_1, b\bot) \land (p_0'', \bot)$
8. $\delta_1(p_0'', a, \bot) = (p_0', \bot)$
9. $\delta_1(p_1, \bot, \bot) = (p_1, bb)$
10. $\delta_1(p_1', a, b) = (p_1', b)$
11. $\delta_1(p_1', b, b) = (p_1', b)$
12. $\delta_1(p_1', a, b) = (p_1', b)$
13. $\delta_1(p_1', b, b) = (p_1', b)$
14. $\delta_1(p_2, \bot, \bot) = (p_2, \bot)$
15. $\delta_1(p_2', a, \bot) = (p_3, \bot)$
16. $\delta_1(p_2', a, b) = (p_3, \bot)$
17. $\delta_1(p_2', b, b) = (p_3, \bot)$
18. $\delta_1(p_2', a, b) = (p_3, \bot)$
19. $\delta_1(p_2', b, b) = (p_3, \bot)$
20. $\delta_1(q_e, \bot, \bot) = (q_e, \bot)$
21. $\delta_1(q_e, a, \bot) = (q_e, \bot)$
22. $\delta_1(q_e, a, \bot) = (q_e, \bot)$
23. $\delta_1(q_e, b, \bot) = (q_e, \bot)$
24. $\delta_1(q_e, b, \bot) = (q_e, \bot)$
25. $\delta_1(q_e, \bot, \bot) = (q_e, \bot)$
26. $\delta_1(q_e, \bot, \bot) = (q_e, \bot)$
27. $\delta_1(q_e, \$, b) = (q_e, \epsilon)$

Transition 1 starts a left branch that verifies the first series of $b$s is of length one. The right branch continues as follows. Every time the automaton reads the first $b$ in a series, it recursively opens two branches (transition
Throughout a computation, each branch behaves like a classical deterministic PDA. Furthermore, because the acceptance is by empty stack, each branch has the prefix property, as it must know when to empty its stack, without information regarding any sibling branches it may have. However,

Remark 6.3. Throughout a computation, each branch behaves like a classical deterministic PDA. Furthermore, because the acceptance is by empty stack,
Figure 6.1: Configurations of the automaton $A_1$ (1), (2), (3), and (4) after reading $aab$, $aabab$, $aababb$ and $aababb\$", respectively.

Figure 6.2: Configurations of the automaton $A_2$ (1), (2) and (3), after reading $aababb$, $aababb\$ba$ and $aababb\$baaba\$, respectively.
note that the trace automaton is not necessarily deterministic as conjunctive transitions become non-deterministic ones after they are “flattened”.

6.2 Linear Membership for DSAPDA

In this section, we show that the membership problem for DSAPDA is decidable in linear time.

Remark 6.4. For the purposes of our discussion, we assume the automaton does not have an infinite series of \( \epsilon \) transitions. As in the classical case, this assumption is not limiting (see [7, Lemma 10.3, p 236]), as \( \epsilon \)-loops can be detected. In particular, \( \epsilon \)-loops in the automaton definition manifest as \( \epsilon \)-loops in the trace automaton, and can therefore be detected in the usual manner.

We consider an implementation model where the computation proceeds in rounds such that in each round, every branch takes one step, i.e., the branches read the input synchronously. Note that this model is equivalent to the one with asynchronous reading.\(^3\) The computation of the automaton is implemented by the following pseudo-code.

1. Set the automaton to the initial configuration.

2. While TRUE
   
   (a) For all sets of sibling branches, do
       
       i. If all sibling branches in the set are empty, then
           A. If they are synchronized (in the same state), then collapse them.
           B. Else, return fail // Computation stuck
       
   (b) If the stack is empty, then
       
       i. If at end of input, return accept. // Accepting configuration
       ii. Else, return fail. // Computation stuck
   
   (c) If there exists a branch-head that has an epsilon transition, then
       
       i. For each branch-head,

\(^3\)Note that this is synchronization in the reading of the input, and is not the synchronization that we require before collapsing branches.
A. If an epsilon transition is applicable then apply it.
B. Else, do nothing.

(d) Else, if there exists a current input symbol $\sigma$, then
   i. For each branch-head
      A. If the branch has a transition on $\sigma$ then apply it.
      B. Else, return fail. // Computation stuck
   ii. Advance the input reader.

To proceed, we shall need the following notation. Let $A = (Q, \Sigma, \Gamma, q_0, \bot, \delta)$ be a DSAPDA.

- We denote by $N_A$ the maximal number of branches opened in a single transition, and
- we denote by $M_A$ the total number of different configurations possible for a branch head, i.e., $M_A = |Q| \times (|\Gamma| \cup \{\epsilon\})$.

Consider a single stack-branch. As it behaves exactly like a standard deterministic PDA without $\epsilon$-loops, it performs a linear number of steps in the input length. At each step, at most $N_A$ new branches are opened from each existing branch. Therefore, the total number of branches is $O(N_A^n)$, where $n$ is the number of input letters read. It follows that a DSAPDA can perform an exponential number of steps in the length of the input.

To achieve linear-time membership for DSAPDA languages, we must circumvent the potentially exponential number of stack branches that the automaton can open. To do so, we require the following immediate lemmas.

**Lemma 6.5.** During the computation, at any given time, there can be at most $M_A$ different state and stack-symbol configurations among the heads of the stack-branches of the automaton.

**Lemma 6.6.** If two branches have the same stack-head configuration, then they behave identically on the same input, as long as the stack height does not dip below the initial height of the head.$^4$

By these two lemmas, we do not need the full exponential power of SAPDA to decide membership for DSAPDA. The core concept of the implementation is to execute the minimal number of branches necessary (at

$^4$This lemma stems from the fact that the automaton is deterministic.
most $M_A$). When a number of branch heads have the same configuration, they are combined to be one head. The computation then continues on the merged branch, as long as its stack is not empty. Once the merged branch empties, the computation continues on the original branches. Thus, at any given time, at most $M_A$ branch heads are necessary, and we achieve linear time. Note that this implementation yields a DAG structure rather than a tree.

To support this new data structure, we modify our computation algorithm as follows:

1. Set the automaton to the initial configuration.

2. While **TRUE**
   
   (a) For all sets of sibling branches, do
      
      i. If all sibling branches in the set are empty, then
         
         A. If they are synchronized (in the same state), then collapse them.
         
         B. Else, return **fail** // Computation stuck
   
   (b) For all empty merged branches, do
      
      i. Remove the merged head, and save its state $q$.
      
      ii. Create new heads for each of the previously merged branches.
      
      iii. Set the states of the new heads to be the state $q$.
   
   (c) Group branch-heads by configuration.
   
   (d) For every set of branch-heads with the same configuration, do
      
      i. Remove the heads, and save their state $q$ and symbol $X$.
      
      ii. Create a new merged head with state $q$ and symbol $X$.
      
      iii. Create a pointer from the new head to all merged branches.
   
   (e) If the stack is empty, then
      
      i. If at end of input, return **accept**. // Accepting configuration
      
      ii. Else, return **fail**. // Computation stuck
   
   (f) If there exists a branch-head that has an epsilon transition, then
      
      i. For each branch-head,
If an epsilon transition is applicable then apply it.\(^5\)

B. Else, do nothing.

(g) Else, if there exists a current input symbol \(\sigma\), then

i. For each branch-head

A. If the branch has a transition on \(\sigma\) then apply it.

B. Else, return \texttt{fail}. // Computation stuck

ii. Advance the input reader.

See Figure 6.3 for an example.

**Theorem 6.7.** The membership problem for DSAPDA is decidable in linear time.

**Proof.** After step 2d, there are at most \(M_A\) different branch heads. Therefore, steps 2(f) and 2(g) are only applied to a bounded number of branches. Moreover, at the end of each iteration, there are at most \(M_A N_A\) branch heads, so steps 2(a) and 2(b) are applied to a bounded number of heads.

Therefore, steps 2(a), 2(b), 2(e), and 2(f) are bounded in each iteration, and linear over the course of the entire computation. We now consider steps 2(c) and 2(d). For any given input character \(\sigma\), we consider the number of stack symbols that can be inserted into the stack as a result of the branches reading \(\sigma\). This number is derived from the maximal number of symbols that can be written to the stack in a given transition, the maximal number of \(\epsilon\) loops.

\(^5\)Recall that we assume the automaton has no \(\epsilon\)-loops.
transitions that can occur after a character is read (recall that we assume this is bounded), and the number of active branches in steps 2(f) and 2(g). As all these are bounded, for any given $\sigma$, the number of symbols is at most some constant $K_A$. Therefore, the number of symbols inserted to the stack during a computation on a word of length $n$ is at most $K_A n$.

Each symbol inserted, at some point is exposed as the top symbol in its stack branch. Subsequently, in steps 2(c) and 2(d), it is considered for grouping, and immediately after, in step 2(f) or 2(g), it is removed (as the top stack symbol is always removed in any given transition). Therefore, over the course of the entire computation, steps 2(c) and 2(d) consider each stack symbol exactly once, meaning that their execution time is on the order of $K_A n$.

It follows that, all in all, the algorithm yields linear-time execution. \qed

Remark 6.8. There is a one-to-one correlation between the full configuration of a DSAPDA and its above compact representation. The full representation can be derived from the compact one by performing the following steps.

1. Scan the stack-structure from bottom to top, i.e., starting at the root.

2. Whenever a merged node $v$ is encountered, do the following. Let $u_1, \ldots, u_k$ be the top nodes of the branches $v$ points to (the branches merging a $v$).
   (a) Make $k$ copies of $v, v_1, \ldots, v_k$, and have each $v_i$ point to $u_i$.
   (b) For every node $u$ that points at $v$, have $u$ point at all $v_1, \ldots, v_k$.
   (c) Remove node $v$.

See Figure 6.4 for an example.

6.3 LR(0) Conjunctive Grammars

In this section, we extend the classical notion of an LR(0) grammar (see [7, pp. 248–252]) to CG. First, we define the notion of a rightmost derivation.

Definition 6.9 (Cf. Definition 3.10). Rightmost conjunctive formulas and rightmost derivations are defined by the following simultaneous recursion.
The elements of \((V \cup \Sigma)^+\) are rightmost (conjunctive) formulas. One-step derivations from them in which a rule is applied to their rightmost variable are rightmost derivations.

Let \(\mathcal{A}\) be a rightmost formula. Then a one-step rightmost derivation from \(\mathcal{A}\) is either contraction or a rule applied to the rightmost variable of a simple conjunct of \(\mathcal{A}\).

The one-step rightmost derivation relation is denoted \(\Rightarrow_R\), and, as usual, its reflexive and transitive closure is denoted \(\Rightarrow^*_R\).

As with leftmost derivations, we can also define rightmost derivations using traces.

**Proposition 6.10** (Cf. Proposition 4.11). A derivation \(X \Rightarrow^* \mathcal{A}\) is a rightmost derivation if all its projections are rightmost derivations in the classical sense.

For simplification purposes, we define an augmented form of conjunctive grammars. Note that the augmentation process below is linear in the number of conjunctive rules in the grammar.
Definition 6.11. Given a conjunctive grammar \( G = (V, \Sigma, P, S) \), we define an augmented grammar \( G' \) by adding the following variables and rules.

- We add a new start symbol \( S' \), and add a rule \( S' \rightarrow S \).
- Let \( n \) be the maximal number of conjuncts in a rule of \( P \). Let \( m \) be the number of conjunctive rules in \( P \). For every rule \( X \rightarrow (\alpha_1 \& \cdots \& \alpha_k) \in P \) that is the \( j \)-th conjunctive rule in \( P \) (according to some rule enumeration), we do the following.
  - We add new variables \( S^j_1, \ldots, S^j_n \) and \( S^j \),
  - replace the rule with \( X \rightarrow S^j \),
  - add the rule \( S^j \rightarrow (S^j_1 \& \cdots \& S^j_n) \), and
  - add the rules \( S^j_i \rightarrow \alpha_i \) for \( i = 1, \ldots, k \) and \( S^j_i \rightarrow \alpha_k \) for \( i = k+1, \ldots, n \).

Clearly, for any conjunctive grammar \( G \) and its augmentation \( G' \), \( L(G') = L(G) \). Henceforth, we only consider augmented grammars.

Example 6.12. The following CG, \( G = (V, \Sigma, P, S) \), is a very simple augmented grammar, which derives the regular language \( \{ab\} \cup \{a^n$ \mid n \geq 1\}.^6

- \( V = \{S', S, A, B, S^1_1, S^1_2\} \), and
- \( P \) consists of the following rules:
  \[
  S' \rightarrow S \mid S \rightarrow ab \mid S^1_1 \mid S^1_2 \mid (S^1_1 \& S^1_2) \mid S^1_1 \rightarrow aaA \mid S^1_2 \rightarrow aabB \mid A \rightarrow aaA \mid $ \mid B \rightarrow aabB \mid $ \]

Now we proceed to define the basic building blocks of \( LR \) grammars, e.g., items, viable prefixes, and valid prefixes. We define these by applying the classical definitions to the trace grammar.

Definition 6.13. Given a conjunctive grammar \( G = (V, \Sigma, P, S) \), the set of \( LR(0) \) items of \( G \) is the set of classical \( LR(0) \) items of the trace grammar \( G_T \), i.e.,

- \( X \rightarrow \alpha \cdot \beta \) is an \( LR(0) \) item if \( X \rightarrow \alpha \beta \in P \), and

^6Obviously, the full power of conjunctive grammars is not necessary here. This simple example was chosen for illustration purposes only.
• all \( S^j \to S^j_i \) and \( S^j \to S^j_i \cdot \) such that \( S^j \to (S^j_1 \& \cdots \& S^j_n) \in P \) are \( LR(0) \) items.

**Definition 6.14.** We say that \( \gamma \in (V \cup \Sigma)^* \) is a viable prefix of \( G \) if it is a viable prefix of \( G_T \), i.e., if there is a trace derivation of \( G_T \) of the form \( S \xrightarrow{\star_{TR}} \delta Xw \xrightarrow{\star_{TR}} \delta \beta w \) and \( \gamma \) is a prefix of \( \delta \beta \).

**Definition 6.15.** We say an item \( X \to \alpha \cdot \beta \) is valid for a viable prefix \( \gamma \) if there is a trace derivation \( S \xrightarrow{\star_{TR}} \delta Xw \xrightarrow{\star_{TR}} \delta \alpha \beta w \) for some \( X \to \alpha \beta \in P \), and \( \gamma = \delta \alpha \).

**Example 6.16.** Consider the augmented grammar \( G \) from Example 6.12. The derivation \( S \Rightarrow S \Rightarrow S_1 \Rightarrow S_1 \Rightarrow aaA \Rightarrow aaaaA \) is a trace derivation of \( G \). Therefore, all prefixes of \( aaaaA \) are viable prefixes of \( G \). It follows that the item \( A \to a \cdot aA \) is valid for the viable prefix \( aaa \), and the item \( A \to aa \cdot A \) is valid for the viable prefix \( aaaa \).

We proceed with the definition of a deterministic SAPDA that acts as an \( LR \) parser for conjunctive languages. To do so, we need to define a canonical set of item-sets and two functions, \( action \) and \( goto \), which together make up the parsing table for a given grammar. As in the classical case, the \( goto \) function recognizes sets of valid items for viable prefixes. However, we cannot simply use the classical definition of a \( goto \) function because valid items from sibling traces could cause conflicts. To avoid this problem, we define a \( goto \) function that separates between items originating from different sibling traces. As there is more than one way to do so, instead of one \( goto \) function (as in the classical case), there is a class of valid \( goto \) functions, of which any one function can be used.

The \( action \) function decides which step the automaton should take (\( shift, reduce, split, accept, error \)), based on the set of valid items supplied by \( goto \). The main difference from the classical case is that when an item of the form \( X \to \cdot S^j \) is valid (i.e., a conjunctive rule can be applied), the deterministic SAPDA makes a conjunctive transition such that each branch processes one of the possible conjuncts \( S^j_1, \ldots, S^j_n \) (this is the \( split \) action), based on the separation defined by the \( goto \) function.

We begin with the definition of a valid \( goto \) function. A \( goto \) function receives a set of items \( I \) and a symbol \( X \in V \cup \Sigma \cup \{\epsilon\} \).\(^7\) The function is

\(^7\)Note that in the classical \( goto \) function, \( X \in V \cup \Sigma \).
applied to two types of item sets: regular sets and split sets. Regular set are sets of items that do not need to be separated. Therefore, when a goto function $g$ is applied to regular sets, it behaves exactly as in the classical case, i.e., if $I$ is the set of valid items for some viable prefix $\gamma$ and $X \neq \epsilon$ then $g(I, X)$ is the set of valid items for viable prefix $\gamma X$.

Split sets are sets that call for a separation of items in the next step, to avoid conflicts from sibling traces. A split set always contains an item of the form $X \rightarrow \cdot S_j$. When a goto function is applied to a split set, it only has $\epsilon$-transitions. Namely, it has $n \epsilon$-transitions to $n$ item-sets, each containing exactly one of the $S_j \rightarrow \cdot S_{j_1}$ items, thus separating items from sibling traces. These transitions correlate with the conjunctive transitions in the deterministic SAPDA parser. To accommodate for these multiple transitions, the goto function maps to sets of item-sets.

**Definition 6.17.** Let $I$ be a set of LR(0) items. We define the item-closure of $I$, denoted $[I]$ as the smallest set of items such that:

- $I \subseteq [I]$, and
- if $X \rightarrow \delta \cdot Y \alpha \in [I]$ and $Y \neq S_j$, then $Y \rightarrow \beta \in [I]$ for all $Y \rightarrow \beta \in P$.

Note that this definition is the same as the classical item-closure definition, except that items of the form $X \rightarrow \cdot S_j$ are not expanded. This is because the expansions of the items need to be separated, and they are therefore expanded in a separate step.

**Definition 6.18.** A set of items $I$ is split if it contains an item of the form $X \rightarrow \cdot S_j$, yet it does not contain the items $S_j \rightarrow \cdot S_{j_1}$. In this case, we also say that $S_j$ is split in $I$. If an item set is not split, it is called regular.

Note that by Definitions 6.17 and 6.18, the item-closure of a regular item-set corresponds to the classical item-closure definition.

**Definition 6.19.** A function $g$ is a valid goto function if for each regular item-set $I$, split item-set $J$, and symbol $X \in V \cup \Sigma$, the following holds.

- $g(I, X) = \{ [[Z \rightarrow \alpha X \cdot \beta \mid Z \rightarrow \alpha \cdot X \beta \in I]] \}$, and $g(I, \epsilon) = \{I\}$.
- $g(J, X) = \emptyset$, and $g(J, \epsilon) = \{[J_1], \ldots, [J_n]\}$ where $J_1, \ldots, J_n$ are minimal item-sets such that
\(- J \subseteq J_k \) for \( k = 1, \ldots, n \), and
\(- \) for each \( j \) such that \( S^j \) is split in \( J \), and for each \( i = 1, \ldots, n \), there exists \( 1 \leq k \leq n \) such that \( S^j \rightarrow \cdot S^j_i \in J_k \), and for no \( i' \neq i \), \( S^j \rightarrow \cdot S^j_i \in J_k \).

Note that in transitions from split-sets, exactly one \( S^j \rightarrow \cdot S^j_i \) item from each conjunctive rule in \( J \) appears in each resulting item-set. In particular, if \( J \) contains only one item of the form \( X \rightarrow \cdot S^j \), then for \( k = 1, \ldots, n \), \( J_k = J \cup \{ \{ S^j \rightarrow \cdot S^j_{i_k} \} \} \), for some \( 1 \leq i_k \leq n \). Furthermore, note that when applied to regular sets, a valid goto function behaves exactly like the classical one.

Example 6.20. Continuing our discussion of the grammar \( G \) from Example 6.12, consider the item-set \( I_0 = \{ S' \rightarrow \cdot S, S \rightarrow \cdot ab, S \rightarrow \cdot S^1 \} \). Note that \( I_0 \) is split. Therefore, a valid goto function can be defined such that \( g(I_0, \epsilon) = \{ I_1, I_2 \} \) where \( I_1 = I_0 \cup \{ S^1 \rightarrow \cdot S^1_1, S^1_1 \rightarrow \cdot aaA \} \), and \( I_2 = I_0 \cup \{ S^1 \rightarrow \cdot S^2_1, S^2_1 \rightarrow \cdot aaaB \} \). The sets \( I_1 \) and \( I_2 \) are regular. Therefore, e.g., as in the classical case, \( g(I_1, a) = \{ S \rightarrow a \cdot b, S^1_1 \rightarrow a \cdot aA \} \).

We define the set of canonical item-sets of a conjunctive grammar, with respect to a valid goto function.

Definition 6.21. Let \( G = (V, \Sigma, P, S) \) be a conjunctive grammar and \( g \) a valid goto function. We define the canonical collection of item-sets of \( G \) with respect to \( g \) to be the smallest set \( C_g \) such that
\[ \bullet \quad [\{ S' \rightarrow \cdot S \}] \in C_g, \quad \text{and} \]
\[ \bullet \quad \text{if } I \in C_g \text{ and } X \in V \cup \Sigma \cup \{ \epsilon \}, \text{then for all } J \in g(I, X), J \in C_g. \]

We denote the item-set \( [\{ S' \rightarrow \cdot S \}] \in C_g \) by \( I_0 \).

We now proceed to define the notion of an \( LR(0) \) grammar.

Definition 6.22. We say that an item-set \( I \) is conflict free if the following holds.
\[ \bullet \quad \text{If there is an item } X \rightarrow \alpha \cdot \text{ in } I, \text{ then there is no other item } Y \rightarrow \beta \cdot, \quad Y \neq X \text{ or } \alpha \neq \beta, \quad \text{in } I. \]
• If there is an item \( X \rightarrow \alpha \cdot \) in \( I \), then there is no item \( Y \rightarrow \beta \cdot \sigma \gamma, \sigma \in \Sigma, \) in \( I \).

The first is called a reduce-reduce conflict, and the second is called a shift-reduce conflict.

**Definition 6.23.** A conjunctive grammar \( G = (V, \Sigma, P, S) \) is \( LR(0) \) if there is a valid \( goto \) function \( g \) for which \( C_g \) is conflict free.

**Remark 6.24.** Note that finding a conflict free grouping is a pre-processing step, and, therefore, does not impact the run-time of the parsing algorithm. Moreover, in the next section, we will see that for every \( LR(0) \) grammar, there exists an equivalent \( LR(0) \) grammar, where any choice of grouping is guaranteed not to cause conflicts.

Let \( g \) be a valid \( goto \) function for a CG \( G \), and let \( C_g \) be the resulting canonical set of item-sets. Together, \( g \) and \( C_g \) define a finite-state automaton where \( g \) is the transition function and \( C_g \) is the set of states. The automaton is non-deterministic, as \( g \) has \( \epsilon \)-transitions. Let \( \hat{g} \) be the standard extension of a non-deterministic transition function to strings and sets of states, see [7, pp. 24–25]. Then, for a set of item-sets \( Q \subseteq C_g \) and a string \( \gamma \in (V \cup \Sigma)^* \), \( \hat{g}(Q, \gamma) \) contains all the item-sets \( J \) reachable from some set \( I \in Q \) by reading the string \( \gamma \).

Figure 6.5 describes a partial construction of the canonical set of item-sets and a valid \( goto \) function \( g \) for the grammar \( G \) from Example 6.12. Note that the first set, \( I_0 \) is split. Therefore, \( g(I_0, \epsilon) = \{I_1, I_2\} \) where \( I_1 \) and \( I_2 \) each contain one of the items \( S \rightarrow \alpha \cdot \), \( S_1 \rightarrow \alpha \cdot \). Furthermore, we can see that, e.g.,

\[
\hat{g}(\{I_0\}, a) = \{I_7, I_9\} = \{\{S \rightarrow a \cdot b, S_1 \rightarrow a \cdot aA\}, \{S \rightarrow a \cdot b, S_2 \rightarrow a \cdot aB\}\},
\]

and

\[
\hat{g}(\{I_0\}, ab) = I_8 = \{S \rightarrow ab \cdot \}.
\]

In the classical construction of an \( LR(0) \) item automaton, \( \text{goto}(I_0, \gamma) \) contains all the valid items for the viable prefix \( \gamma \). We show that this also holds for a valid \( goto \) function of a conjunctive grammar. To do so, we show that for a conjunctive grammar \( G \) and its trace grammar \( G_T \) the
Figure 6.5: Partial construction of the canonical set of item-sets and valid goto function.
classical LR(0) item-set automaton constructed for \( G_T \), is the deterministic counterpart of our new item-set automaton construction applied to \( G \). Therefore, the classical item-set automaton’s capability to find valid items for viable prefixes, is translated to the new construction as well.

Let \( G \) be a conjunctive grammar, and let \( g \) be a valid goto function for \( G \). Recall that \( C_g \) is the canonical set of item-sets for \( g \), and \( I_0 \) is the initial set. We denote by \( A_g \), the non-deterministic finite-state automaton defined by \( g \) and \( C_g \). Consider the trace grammar \( G_T \). Let \( f \) be the classically defined goto function for the grammar \( G_T \), and let \( D \) be the canonical set of item-sets, where the initial set is denoted by \( J_0 \). We denote by \( A_f \) the deterministic finite-state automaton defined by \( f \) and \( D \).

The following proposition describes the relation between \( A_g \) and \( A_f \). The proposition follows immediately from the definitions of \( A_g \), \( C_g \), \( A_f \), and \( D \).

**Proposition 6.25.** The automaton \( A_f \) is the deterministic automaton obtained by applying \( \epsilon \)-transition removal and and the powerset construction to \( A_g \).

The proof of the proposition is as follows. Recall that the powerset construction converts a non-deterministic automaton into a deterministic one, see e.g., [7, Proof of Theorem 2.1, pp 22 – 23]. Note that by Definition 6.13, the LR(0) items for \( G \) and \( G_T \) are the same. Also, for any regular item-set \( I \), and any \( X \in V \cup \Sigma \cup \{ \epsilon \} \), \( f(I, X) = g(I, X) \). Let \( I \) be a split item-set in \( C_g \). After the application of \( \epsilon \)-removal and the powerset construction, \( I \) is grouped together with all the states \( J \) such that \( J \in \hat{g}(I, \epsilon) \), which is exactly the classical item-closure of \( I \), and therefore, a state in \( D \).

From Proposition 6.25, we can deduce the following important corollary.

**Corollary 6.26.** Let \( \gamma \in (V \cup \Sigma)^* \) be a viable prefix. Then,

\[
f(J_0, \gamma) = \bigcup \hat{g}(I_0, \gamma).^8
\]

We now formally describe the ability of a valid goto function to find valid items for viable prefixes.

**Lemma 6.27.** Let \( \gamma \) be a viable prefix of \( G \) and let \( g \) be a valid goto function. Then \( \bigcup \hat{g}(\{I_0\}, \gamma) \) contains an item \( X \rightarrow \alpha \cdot \beta \) if and only if \( X \rightarrow \alpha \cdot \beta \) is valid for \( \gamma \).

---

^8Recall that \( \hat{g} \) results in a set of item-sets, whereas \( f \) results in a set of items.
Proof. By the classical construction, $X \rightarrow \alpha \cdot \beta \in f(J_0, \gamma)$ if and only if $A \rightarrow \alpha \cdot \beta$ is valid for $\gamma$ in $G_T$, see e.g., [7, Theorem 10.9, p. 251]. Therefore, by Proposition 6.25, $X \rightarrow \alpha \cdot \beta \in \bigcup \hat{g}(I_0, \gamma)$ if and only if $A \rightarrow \alpha \cdot \beta$ is valid for $\gamma$ in $G_T$. By Definition 6.13, validity in $G_T$ is the same as validity in $G$. This completes our proof. 

We now define a deterministic SAPDA that recognizes the language of an LR(0) grammar $G$. For each branch, the automaton writes grammar symbols and item-sets from the canonical set of item-sets to its stack, thus keeping track of valid items for the viable prefix it has reduced so far. The transitions of the automaton are determined by the action function, and a valid goto function $g$. The initial symbol in the stack is the item-set $I_0$. The action function receives a current item-set from the top of the stack $I$, and the next symbol from the input $\sigma \in \Sigma$ (if such a symbol exists), and returns the following:

1. If $I$ contains the item $S' \rightarrow S \cdot$, the stack is emptied (accept).

2. If $I$ is regular and contains an item $X \rightarrow \alpha \cdot \sigma \beta$ then $\sigma$ is shifted onto the stack, and $g(I, \sigma)$ is placed above it (shift). Note that $g(I, \sigma)$ returns a single item-set as a non-epsilon transition is applied.

3. If $I$ is regular and contains an item $X \rightarrow \alpha \cdot$, then the symbols of $\alpha$ and the padding item-sets are removed from the stack, revealing some item set $J$ at the top of the stack. Now, $X$ is written to the stack above $J$, and then $g(J, X)$ is written on top of that (reduce).

4. If $I$ is a split set, then $I$ is removed from the stack, $n$ new branches are opened, and the $n$ $g(I, \epsilon)$ item sets are put into these branches, one for each branch (split).

Note that transition 2 is a transition on $\sigma$, whereas transitions 1, 3 and 4 are $\epsilon$ transitions. In the case of $\epsilon$ transitions, the next input character is not read, and is supplied to the subsequent action transition.

Remark 6.28. To simplify our description of the automaton computations, we assume that no non-trivial chain of $\epsilon$-transitions exists, i.e., if $I$ is a split set and $J \in g(I, \epsilon)$ then $J$ is regular. A non-trivial chain of split item-sets occurs if there is an item $X \rightarrow \cdot S^j$ in a split set, such that there is an item
of the form $Y \rightarrow \cdot S^k$ in the expansion $\{S^j \rightarrow \cdot S^j_i\}$. However, this situation can easily be circumvented by making the following modification. Take the grammar rule that derives $S^k$, $Y \rightarrow S^k$, and simply add a new variable $Y \rightarrow E S^k$ with a single rule $E \rightarrow \epsilon$. Clearly, the grammar language remains unchanged, and the problematic scenario is no longer possible. Moreover, as the change does not add any new derivations, the grammar remains $LR(0)$. Henceforth, we assume that all valid goto functions do not have $\epsilon$-chains.

We now proceed to show that the automaton does, in fact, accept exactly the language of the grammar. As in the classical case, the automaton attempts to construct a rightmost derivation of the input. To do so, it stores the (tree) prefix of the derivation that it has managed to reduce so far in the stack. At each point, the top symbols of the branches in the stack are the item-sets obtained by applying the goto function to this prefix. By Corollary 6.26, these are exactly the valid items for the prefix, and therefore, they are the candidate derivation rules that can be added next to the rightmost derivation. The automaton continues to shift input symbols onto the stack, until the set of valid items contains an item of the form $X \rightarrow \alpha \cdot$. This signals that $X \rightarrow \alpha$ is the correct choice, and the symbols of $\alpha$ on the stack are replaced with $X$, thus simulating the reduction. Because the grammar is $LR(0)$, the item-sets are guaranteed to be conflict free, and therefore the automaton is well defined, i.e., it only has one valid transition defined for any given configuration.

To prove that the rightmost derivation the automaton constructs is correct, consider the following. When an item of the form $X \rightarrow \alpha \cdot$ is valid for a viable prefix $\gamma$, reducing $\alpha$ to $X$ is a valid continuation of a rightmost derivation. Because the grammar is $LR(0)$, the canonical set of item-sets is conflict free. Therefore, there are two possibilities. Either $X \rightarrow \alpha \cdot$ is the only valid item for $\gamma$, or $\alpha = \epsilon$ and there is no other valid item for $\gamma$ with a terminal symbol to the right of the dot. It follows that $X \rightarrow \alpha$ is the correct reduction to apply, regardless of the rest of the input. Moreover, reductions such as these are applied as soon as they are valid, because whenever a set of items is written to the stack, it is immediately examined in the next step of the computation. Therefore, if an item of the form $X \rightarrow \alpha \cdot$ is valid, the next step is in fact a reduce step based on $X \rightarrow \alpha$.

The main difference from the classical construction, is of course the split transitions. Consider the case where the top item-set $I$ is split. It
follows that the item-set contains zero or more items of the form \( X \rightarrow \alpha \cdot \beta \), where \( \alpha \beta \) is not an \( S_j \) variable, as well as at least one item of the form \( Y \rightarrow \cdot S_j^j \). After the split transition, \( n \) new branches are created, each with a set of items containing all the items from the original item-set \( I \), as well as the item-set closure of one of the \( S_j^j \rightarrow \cdot S_j^j \).

For simplification purposes, assume \( n = 2 \), i.e., two branches are created, one with the closure of the item \( S_j^j \rightarrow \cdot S_j^j_1 \) and one with the closure of the item \( S_j^j \rightarrow \cdot S_j^j_2 \). Denote the item-set in the first branch by \( I_1 \), and the item-set in the second branch by \( I_2 \). Both branches proceed to shift input symbols onto the stack, until they have a valid item they can reduce by. Ultimately, after reading some word \( u \) from the input, both branches reduce by one of the rules in their original item sets, \( I_1 \) and \( I_2 \), otherwise the computation is stuck, as there is no valid derivation.

Assume that after reading \( u \), both branches reduce by a rule \( X \rightarrow \alpha \cdot \beta \)-such that \( X \rightarrow \alpha \cdot \beta \in I \). Therefore, both branches have the set \{ \( X \rightarrow \alpha \cdot \beta \) \} at the top of the stack, and the symbols of \( \beta \) below (as the split occurred when the dot was before \( \beta \)). As both are trying to reduce by \( X \rightarrow \alpha \cdot \beta \), they attempt to remove the symbols of \( \alpha \beta \) from their stacks. Therefore, after emptying \( \beta \) they both have empty stacks, and are in the same state, i.e., they are synchronized. Subsequently, the branches are collapsed, and the computation continues to empty \( \alpha \) from the parent branch. Note that the padding item-sets in the stack-branches are different, but this does not matter as they are all emptied regardless of the items they contain.

Now, assume that after reading \( u \), one branch sees \( S_j^j \rightarrow \cdot S_j^j_1 \cdot \) at the top of the stack, while the other sees \( S_j^j \rightarrow \cdot S_j^j_2 \cdot \). Therefore, both branches reduce to \( S_j^j \). Note that as each conjunctive transition has a specific \( S_j^j \) associated with it, it follows that \( g(I_1, S_j^j) = g(I_2, S_j^j) = \{ Y \rightarrow \cdot S_j^j \} \). Therefore, the next step of these branches is to reduce by \( S_j^j \), and write \( Y \) and \( g(I_1, Y) \) to their stacks.

Next, we contend that \( g(I_1, Y) = g(I_2, Y) \). By Remark 6.28, we assume that no item of the form \( Z \rightarrow \cdot Y \gamma \) is in the expansion of \( S_j^j \rightarrow \cdot S_j^j_1 \), because then \( Y \rightarrow \cdot S_j^j \) would be in the expansion of \( S_j^j \rightarrow \cdot S_j^j_1 \). Therefore, all items of the form \( Z \rightarrow \delta \cdot Y \gamma \) in \( I_1 \) and \( I_2 \) are originally from \( I \). It follows that \( g(I_1, Y) = g(I_2, Y) \). Consequently, the contents of the stack-branches is identical. As the automaton is deterministic, and the stack contents and state of the branches is identical, they will behave in the same way on any
input. It follows that when they empty, they are synchronized, and can therefore be collapsed.

Note that for this synchronization to occur, all siblings must reduce to the same rule from $I$. If this rule is of the form $Y \rightarrow S^j$ then each branch must reduce by some $S^j \rightarrow S^j_i$. As each branch has only one of the items $S^j \rightarrow S^j_i$, this can only happen only if there is a rightmost derivation of $u$ from each of the $S^j_i$ variables. Therefore, these derivations combine to form the rightmost derivation of $u$ from $S^j$.

Consider the automaton based on the goto function described in Figure 6.5. The initial item-set $I_0$ is split, and, therefore, the automaton’s first move is to open two branches, one with $I_1$ and one with $I_2$. Say the first input symbol read is $a$. At this point, it is not yet clear whether the correct reduction is by the derivation $S \Rightarrow ab$ or the derivations $S \Rightarrow S^1 \Rightarrow S^1_1 \Rightarrow a\cdot aA$ and $S \Rightarrow S^1 \Rightarrow S^1_2 \Rightarrow a\cdot aA$. Appropriately, both branches have valid item $S \rightarrow a \cdot b$ and one of the items $S^1_1 \rightarrow a \cdot aA$ or $S^1_2 \rightarrow a \cdot aA$.

The next input symbol determines whether the conjunctive derivation is correct. If the symbol is $b$, then both branches write $I_8$ at the top of their stacks. If the symbol is $a$, one branch transitions to $I_{10}$, and one to $I_{11}$. If the remaining input is of the form $a^6 \cdot 2$, both branches reduce to $S \Rightarrow S^1$. Therefore, in either case, at some point both branches are in the same configuration, and continue the computation identically, until they are ultimately collapsed.

From our automaton construction, we can derive the following.

**Theorem 6.29.** If a language is generated by an LR(0) conjunctive grammar, then it is accepted via empty stack by a deterministic SAPDA.

From Theorems 6.7 and 6.29, we obtain the following corollary.

**Corollary 6.30.** Every LR(0) conjunctive language can be parsed in linear time.

This result extends the classical LR(0) algorithm, as LR(0) conjunctive languages properly contain all finite intersections of classical LR(0) languages. Note that when applied to classical LR(0) grammars, the parsing algorithm is identical to the classical LR(0) algorithm.

**Remark 6.31.** Both classical and conjunctive LR(0) languages are not closed under complement (because the prefix property is not maintained) and under
union. However, our linear parser can be modified to work for the boolean closure of conjunctive LR(0) languages as follows. The complement of a language can be determined by running the parser and checking whether the final configuration is accepting or not. As the parser is deterministic, this method will correctly identify the words not in the language. The union of any finite number of languages can be recognized by simply making several parsing runs, one for each language. Thus, we have linear parsing for the boolean closure of LR(0) conjunctive languages, and in particular, the boolean closure of classical LR(0) languages.

### 6.4 Constructing an LR(0) Grammar from a DSAPDA

We now proceed to address the converse of Theorem 6.29, i.e., the construction of an LR(0) conjunctive grammar from a deterministic SAPDA.

**Theorem 6.32.** If a language is accepted by a deterministic SAPDA, then it is generated by an LR(0) conjunctive grammar.

To prove Theorem 6.32, we make several simplifying assumptions regarding the structure of deterministic SAPDA. Namely we assume that all conjunctive transitions are $\epsilon$ transitions, and that they all have the same number of conjuncts. Both assumptions can easily be shown not to be limiting.

Let $A = (Q, \Sigma, \Gamma, \delta, q_0, \bot)$ be a deterministic SAPDA such that each conjunctive transition has exactly $n$ conjuncts. Consider the CG $G_A = (V, \Sigma, P, S)$, where

- $V = Q \times \Gamma \times Q \cup \{A_{q\sigma X} | q \in Q, \sigma \in \Sigma \cup \{\epsilon\}, X \in \Gamma\} \cup \{S\}$
  
  $\cup \{S^q X \rho | q, p \in Q, X \in \Gamma\} \cup \{S_i^q X \rho_i | i = 1, \ldots, n, q, p \in Q, X \in \Gamma\}$

- $P$ contains the following rules.
  
  1. $S \rightarrow [q_0, \bot, p]$, for all $p \in Q$,
  2. $A_{q\sigma X} \rightarrow \sigma$,
  3. for every $\delta(q, \sigma, X) = (q', Y_1 \cdots Y_k)$ and for every choice of $p$, and $q_2, \ldots, q_k$,
    
    $[q, X, p] \rightarrow A_{q\sigma X}[q', Y_1, q_2][q_2, Y_2, q_3] \cdots [q_k, Y_k, p]$,
for every $\delta(q, \sigma, X) = (p, \epsilon)$, $[q, X, p] \rightarrow A_{q\sigma X}$, and

for every $\delta(q, \epsilon, X) = (q_1, Y_{1,1} \cdots Y_{1,k_1}) \land \cdots \land (q_n, Y_{n,1} \cdots Y_{n,k_n})$, and for every choice of $p$, and $q_s, t$’s,

\begin{enumerate}
\item $[q, X, p] \rightarrow S^{qXp}$
\item $S^{qXp} \rightarrow (S^{qXp}_{1} \& \cdots \& S^{qXp}_{n})$,
\item $S^{qXp}_{i} \rightarrow A_{q_{i\epsilon}X}[q_{i,1}, Y_{i,1}, q_{i,2}] \cdots [q_{i,k_i}, Y_{i,k_i}, p]$, $i = 1, \ldots, n$.
\end{enumerate}

Note that the grammar $G_A$ is already in augmented form. The construction is very similar to the one in [7, pp. 256–260], and is essentially a modification of the standard translation of an SAPDA into a CG, see Subsection 3.2.2. The main difference is the addition of the $S^{qXp}$ and $S^{qXp}_{i}$ variables for the augmented form, and the $A_{q_{\sigma}X}$ variables, which make it easier to prove that the grammar is $LR(0)$. Note that each $A_{q_{\sigma}X}$ variable corresponds to the specific automaton transition from $q$ on $\sigma$ with $X$ at the top of the stack. To show that $G_A$ is in fact a conjunctive $LR(0)$ grammar, we must show that there exists a valid goto function for $G_A$ such that the canonical item-set construction is conflict-free.

The proof is based on extensions of [7, Lemmas 10.6–10.9]. The extended lemmas show that for every valid goto function, the canonical set of item-sets constructed for $G_A$ is in fact conflict free. For this proof, we need to assume that all useless symbols and rules (those that derive the empty language) have been removed. The emptiness of intersections of context free languages is undecidable, and therefore not computable for conjunctive grammars. However, a correct choice of variables and rules exist, and this correct choice is the grammar we assume we are working with.

First, we formalize the correlation between grammar trace derivations and automaton trace computations. Note that the trace grammar $G_{AT}$ can be obtained from the trace automaton $A_T$ by applying the classical translation of a pushdown automaton into a context-free grammar, see e.g., [7, Theorem 5.4, pp. 116–119] (with the slight modification of adding the subscripted $A$ and $S$ variables). Therefore, as in the classical case, for each leftmost trace derivation, there is a trace computation that coordinates with it step-by-step, and vice versa. This coordination step-by-step, mandates coordination in branch/conjunct selections as well.

Note that $k_i > 0$, i.e., no empty branches are opened in the automaton.
Definition 6.33. Let $S \Rightarrow^*_T w$ be a trace derivation of $G_A$ and let $(q_0, w, \bot) \vdash^*_T (p, \epsilon, \epsilon)$ be a trace computation of $A$. Let $S' \Rightarrow^*_T w'$ be the reordering of the trace derivation as a leftmost derivation. We say that $S \Rightarrow^*_T w$ and $(q_0, w, \bot) \vdash^*_T (p, \epsilon, \epsilon)$ are matching, $S' \Rightarrow^*_T w'$ and $(q_0, w, \bot) \vdash^*_T (p, \epsilon, \epsilon)$ coordinate step-by-step.

Lemma 6.34. (Cf. [7, Lemma 10.6, pp. 257–258]) Let $[q, X, p] \in V$. For all $w \in \Sigma^*$ such that $[q, X, p] \Rightarrow^*_T w$, there is a unique matching trace computation of $A$, $(q, w, X) \vdash^*_T (p, \epsilon, \epsilon)$. Moreover, the series of transitions in the trace computation is the reverse of the series of transitions denoted by the expansion of the subscripted $A$ variables.

Proof. With the exception of the $A_{q\sigma X}$, $S^qXp$ and $SqXp_i$ variables, the construction of $G_{AT}$ from $A_T$ is identical to the classical construction of a context free grammar from a pushdown automaton, see [7, Theorem 5.4, pp 116–119]. Therefore, the fact that a matching trace computation exists stems from the same proof. As $A$ is deterministic, different trace computations on the same word $w$ have different series of conjunct transitions. Therefore, the matching trace computation is in fact unique.

The fact that the trace computation’s transitions correspond to the reverse of the expansion of the subscripted $A$ variables can be easily proven by induction on the length of the derivation.

For the basis of the induction, we consider derivations of the form

$$[q, X, p] \Rightarrow^*_T A_{q\sigma X} \Rightarrow^*_T \sigma.$$ 

In this case, the claim follows directly from the construction of $G_A$.

For the inductive step, we consider derivations that start with rules of the type

$$[q, X, p] \Rightarrow^*_T A_{q\sigma X}[q', Y_1, q_2][q_2, Y_2, q_3] \cdots [q_k, Y_k, p],$$

or rules of the type

$$[q, X, p] \Rightarrow^*_T S^qXp \Rightarrow^*_T S^q_{1}Xp \Rightarrow^*_T A_{q\sigma X}[q_i, Y_{i,1}, q_{i,2}] \cdots [q_{i,k}, Y_{i,k}, p].$$

In both cases, $A_{q\sigma X}$ and $A_{q\sigma X}$ are expanded after all the subscripted $A$ variables that are derived from the suffix of the sentential form. From the
general correlation between the derivations and the computations we know
that the first step of the computation correlates to \( q_0 \sigma X \) or \( q_0 \epsilon X \) respectively,
and that the remaining computation can be broken in to sub-computations
that each correlate with the derivations from the remaining \([p, Z, p']\) variables
from left to right. As each of these derivations is shorter than the full
derivation, and as they are expanded from right to left, we have our claim.

We now establish some properties of the item automaton constructed
from \( G_A \), based on the fact that \( A \) is a DSAPDA. When discussing trace
computations, it is important to remember that they originate from full
computations of \( A \). The following lemma describes the relationship between
various traces on a given terminal word.

**Lemma 6.35.** Let \( w \in \Sigma^* \). If \((q_0, w, \bot) \vdash_T^* (q, \epsilon, \alpha)\) and \((q_0, w, \bot) \vdash_T^* (p, \epsilon, \beta)\) are two different trace computations on \( w \), then they are sibling traces.

The proof is immediate from the following observation. As \( A \) is a deter-
nomistic automaton, all traces on a given word, must be projections of the
same, unique, computation of \( A \) on \( w \), and are therefore siblings. Note
that trace computations matching sibling trace derivations are also sib-
lings, and vice versa. This is because the sibling relation is determined
by branch/conjunct selection, which is identical in matching trace compu-
tations and derivations.

**Definition 6.36.** Let \( \gamma \) be a viable prefix, and let \( X \rightarrow \alpha \cdot \beta \) be a valid
item for \( \gamma \). Let \( S \Rightarrow_{TR}^* \gamma'X\delta' \Rightarrow_{TR}^* \gamma'\alpha\beta\delta' \) be a rightmost trace derivation
in the grammar. We say that the trace *justifies* the item if \( \gamma = \gamma'\alpha \).

Note that different traces may justify different items that are all valid
for the same viable prefix \( \gamma \).

**Lemma 6.37.** Let \( g \) be a valid goto function for \( G_A \) and let \( C_g \) be the
resulting canonical set of item sets. Let \( I \) be an item set in \( C_g \). The items
in \( I \) can be justified by a set of traces that contains no siblings.

**Proof.** We prove the claim by induction on the item set construction.

As the basis of the induction we start with the set \( I_0 \). We consider two
cases. If \( I_0 \) is regular, than all items are justified by traces with no conjunct
selections, and therefore none of them can be siblings. If \( I_0 \) is split, then, as before, no justifying trace makes a conjunct selection as items of the form \([q, X, p] \rightarrow \cdot S^q X P\) are not expanded.

For the induction step we assume that the claim holds for an item-set \( I \). Again, we consider two cases. First assume that \( I \) is regular, and let \( J \) be the item-set such that \( g(I, X) = J \). It follows that all items in \( J \) are either obtained by moving the dot over \( X \), and are therefore justified by the same traces as \( I \), or they are obtained by expansion of one of the former items, and therefore, cannot necessitate sibling branches by the same reasoning as in the base case, as conjunct selections are not expanded.

Now assume that \( I \) is split, and let \( J \) be an item-set such that \( J \in g(I, \epsilon) \). All the items in \( J \) are either items from \( I \), or expansions of items from one of the conjunct selections. As before, the expanded items do not necessitate sibling traces. Moreover, the traces that justify the items in the expansion cannot be sibling traces of the items from \( I \) as these traces are continuations of the trace justifying the \([q, X, p] \rightarrow \cdot S^q X P\) item in \( I \). Therefore, we have our claim.

Recall that we assume there are no useless variables in \( G_A \). Let \( w \in \Sigma^* \) such that \([q, X, p] \Rightarrow^* w \) for some variable \([q, X, p]\). Consider a rightmost trace derivation \([q, X, p] \Rightarrow^*_R w\). We denote the series of rules applied in this trace derivation by \( d \).

We characterize the trace computation of \( A \) that matches this derivation. Each \( A_{q\sigma X} \) matches a transition of the automaton. Let \( m(d) \) denote the reverse of the series of transitions denoted by the expansions of the subscripted \( A \) variables in \( d \). Let \( N(d) \) denote the number of transitions in the series \( m(d) \). By Lemma 6.34, \( A \) has a unique trace computation on \( w \) that matches our trace \( d \). Moreover, the sequence of automaton transitions in this trace computation is \( m(d) \).

We naturally extend these notions to \( V^* \) as follows. Let \( \gamma \in V^*, 10 \) and let \( w \in \Sigma^* \) such that \( \gamma \Rightarrow^*_R w \) by the derivation \( d \). As before, let \( m(d) \) denote the reverse of the series of transitions denoted by the expansions of the subscripted \( A \) variables in \( d \), and let \( N(d) \) denote the number of transitions in the series \( m(d) \).

\[^{10}\text{Note that as } \gamma \text{ is a viable prefix of a right-sentential form, } \gamma \text{ contains only variables.}\]
Lemma 6.38. (Cf. [7, Lemma 10.7, pp. 258–259]) Let $\gamma$ be a viable prefix of $G_A$ such that $S \Rightarrow^*_T \gamma y$. Let $w \in \Sigma^*$ be a terminal word such that $\gamma \Rightarrow^*_T w$ by the trace derivation $d$. Then, there exists a unique trace computation

$$(q_0, wy, \bot) \vdash^{N(d)}_T (q, y, \delta) \vdash^*_T (p, \epsilon, \epsilon)$$

such that the first $N(d)$ moves are the sequence $m(d)$.

Proof. As $\gamma y$ is derivable from $S$, it follows that for some $r$, $[q_0, \bot, r] \Rightarrow^*_T \gamma y$. By continuing this trace derivation with the derivation $d$, we obtain the trace $[q_0, \bot, r] \Rightarrow^*_T \gamma y \Rightarrow^*_T w y$.

By Lemma 6.34, there is a unique matching trace computation that correlates with the reverse sequence of subscripted $A$ expansions. The trace is of the form

$$(q_0, wy, \bot) \vdash^*_T (r, \epsilon, \epsilon) .$$

As the derivation is rightmost, the last $N(d)$ expansions of subscripted $A$ variables in the derivation occur after the sentential form $\gamma y$ has been reached. These expansions must therefore match the first $N(d)$ transitions in the trace computation, which are the sequence $m(d)$. That is,

$$(q_0, wy, \bot) \vdash^{N(d)}_T (q, y, \delta) ,$$

which completes the proof. \hfill \Box

Lemma 6.39. Let $g$ be a valid goto function, and let $C_g$ be the resulting canonical set of item sets. Let $\gamma$ be a viable prefix, and let $w \in \Sigma^*$ be such that $\gamma \Rightarrow^*_T w$ by the derivation $d$. Let $X \rightarrow \alpha \cdot \beta$, $Y \rightarrow \alpha' \cdot \beta'$ be valid items for $\gamma$, such that both items are in the same set $I \in C_g$. Let

$$S \Rightarrow^*_T \gamma' X y \Rightarrow \gamma' \alpha \beta y = \gamma \beta y$$

be the trace justifying $X \rightarrow \alpha \cdot \beta$ in $I$, and let

$$S \Rightarrow^*_T \gamma'' Y y' \Rightarrow \gamma'' \alpha' \beta' y' = \gamma \beta' y'$$

89
be the trace justifying $Y \rightarrow \alpha' \cdot \beta'$ in $I$. Let

$$(q_0, wy, \perp) \vdash^{N(d)}_T (p, y, \delta), (q_0, wy', \perp) \vdash^{N(d)}_T (p', y', \delta').$$

be the prefixes of the two matching trace computations provided by Lemma 6.38. Then the two trace computations are identical.

Proof. By Lemma 6.37, we can assume that $S \Rightarrow^*_{TR} \gamma' X y \Rightarrow \gamma'\alpha\beta y$ and $S \Rightarrow^*_{TR} \gamma'' Y y' \Rightarrow \gamma''\alpha'\beta' y'$ are not sibling traces. By our assumption that there are no useless symbols in the grammar, we can assume that $\alpha\beta \Rightarrow^*_{RT} u$ and $\alpha'\beta' \Rightarrow^*_{RT} u'$. We extend both these traces with these derivations and the derivation $d$ and obtain

$$S \Rightarrow^*_{TR} \gamma' X y \Rightarrow \gamma'\alpha\beta y \Rightarrow^*_{TR} wuy$$

and

$$S \Rightarrow^*_{TR} \gamma'' Y y' \Rightarrow \gamma''\alpha'\beta' y' \Rightarrow^*_{TR} wu'y'.$$

Extensions of non-sibling traces are also non-sibling, and therefore, these traces are not siblings as well. These traces match the full trace computations provided by Lemma 6.38. It follows that these full trace computations are not siblings as well.

Consider the prefixes of these trace computations

$$(q_0, w, \perp) \vdash^{N(d)}_T (p, \epsilon, \delta), (q_0, w, \perp) \vdash^{N(d)}_T (p', \epsilon, \delta').$$

By Lemma 6.35, they must either be identical, or siblings. If they are siblings, then any continuations of them are also siblings, in contradiction to the full traces not being sibling traces. Therefore, they must be identical, and we have our claim.

To show that there exists a valid goto function resulting in a conflict-free set of item-sets, we need to address the issue of grouping in the canonical item-set definition. We show that for $G_A$, every choice of grouping leads to a conflict-free item-set construction, i.e., for this grammar, grouping is not an issue.

Let $g$ be a valid goto function, and let $C_g$ be the resulting canonical set of item-sets. In the following lemmas, we show that $C_g$ is conflict-free, regardless of the choice of $g$. 

90
Lemma 6.40. (Cf. [7, Lemma 10.8, p. 259]) There are no shift-reduce conflicts in $C_G$. That is, for an item-set $I \in C_g$, if $B \rightarrow \beta \cdot \in I$, then there is no $A_{q\sigma X} \rightarrow \cdot \sigma \in I$.

Proof. Let $\gamma$ be a viable prefix such that $I \subseteq g(I_0, \gamma)$. As $\gamma$ is a viable prefix, and $B \rightarrow \beta \cdot$ is valid for $\gamma$, there is a rightmost trace derivation of the form

$$S \Rightarrow_{TR} \gamma' B \ y \Rightarrow_{TR} \gamma' \beta \ y$$

where $\gamma = \gamma' \beta$. As $A_{q\sigma X} \rightarrow \cdot \sigma$ is also valid for $\gamma$, there is a rightmost trace derivation of the form

$$S \Rightarrow_{TR} \gamma \ A_{q\sigma X} \ y' \Rightarrow_{TR} \gamma \ \sigma \ y'. $$

This means that $\gamma' \beta A_{q\sigma X}$ is also a viable prefix.

We consider the different possible forms for the rule $B \rightarrow \beta$.

1. Assume $B \rightarrow \beta$ is of the form $S \rightarrow [q_0, \perp, p]$. As $S$ does not appear on the right side of any rule, this means that $\gamma' = \epsilon$ and $\gamma = \beta = [q_0, \perp, p]$. Therefore, the second trace is of the form

$$S \Rightarrow_{TR} [q_0, \perp, p] A_{q\sigma X} \ y'. $$

However, variables of the form $[q_0, \perp, p]$, can only appear at the left-most position of a sentential form at the first step of the derivation. All subsequent sentential forms begin either with a subscripted $A$ variable, or an $S_{qXP}$ or $S_{qXPi}$ variable, until the last, which begins with a terminal.

2. Assume $B \rightarrow \beta$ is of the form $[t, Y, p] \rightarrow A_{t\tau Y} [q_1, Y_1, p], \ldots [q_k, Y_k, p]$, or $S_{t\tau Y} \rightarrow A_{teY} [q_i, Y_{i,1}, q_{i,2}] \cdots [q_{i,k}, Y_{i,k}, p], \text{ for } k \geq 0$. Note that as it corresponds to a specific automaton transition, whenever $A_{t\tau Y}$ is introduced, it is followed by exactly $k$ triples. Subsequently, if $k = 0$ then the next step of the derivation replaces $A_{t\tau Y}$ with $\tau$ in contradiction to $\gamma' A_{t\tau Y} A_{q\sigma X}$ being a prefix of a right sentential form. If $k \geq 1$, then the next step of the derivation expands the $k$-th triple. As triples
are only introduced to the right of subscripted \( A \) variables, this is also a contradiction to \( \gamma'k\beta_{q\sigma X} \) being the prefix of a right sentential form.

3. Assume \( B \to \beta \) is of the form \([r, Y, t] \to S_t^{Y_t} \) or \( S_t^{Y_t} \to S_{t}^{Y_t} \). By similar reasoning to the previous case, as soon as either \( S_t^{Y_t} \) or \( S_{t}^{Y_t} \) are introduced, they are immediately expanded. As they are alone (and therefore also rightmost) in any rule that introduces them, they cannot appear to the left of any variable in a right sentential form, in contradiction to \( \gamma'k\beta_{q\sigma X} \) being a viable prefix.

4. Assume \( B \to \beta \) is of the form \( Ap_{\epsilon}Y \to \epsilon \). First, \( \tau \) must be \( \epsilon \) because otherwise \( \gamma'k\beta_{q\sigma X}y' \) is not a right sentential form. Therefore, \( \gamma A_{p}Y \) is also a viable prefix. Let \( w \in \Sigma^* \) be a word derivable from \( \gamma \), and let \( d \) be the series of rules in a rightmost trace derivation of \( w \).

Let \( d_1 = Ap_{\epsilon}Y \to \epsilon \cdot d \) and \( d_2 = Aq_{\sigma}X \to \sigma \cdot d \). Therefore, \( w \) is derivable from \( \gamma A_{p}Y \) by \( d_1 \), and \( w\sigma \) is derivable from \( \gamma A_{q\sigma}X \) by \( d_2 \).

By Lemma 6.38, there exist matching trace computations

\[
(q_0, w, y, \perp) \vdash_T N(d) (q_1, y, \alpha_1) \vdash_T (q_2, y, \alpha_2) \vdash_T (q_3, \epsilon, \epsilon)
\]

\[
(q_0, w\sigma y', \perp) \vdash_T N(d) (p_1, \sigma y', \alpha'_1) \vdash_T (p_2, y', \alpha'_2) \vdash_T (p_3, \epsilon, \epsilon),
\]

respectively.

By Lemma 6.39, the first \( N(d) \) steps of these traces are identical, including branch selections. Therefore, we have a contradiction to the determinism of \( A \), as the \( N(d) + 1 \) step of the first trace is an \( \epsilon \) transition, and the \( N(d) + 1 \) step of the second trace is a transition on \( \sigma \).

This completes our proof that there are no shift-reduce conflicts in \( C_g \). □

**Lemma 6.41.** (Cf. [7, Lemma 10.9, pp. 259–260]) There are no reduce-reduce conflicts in \( C_g \). That is, for an item-set \( I \in C_g \), if \( B \to \beta \cdot \in I \), then there is no \( C \to \alpha \cdot \in I \).
Proof. Let $\gamma$ be a viable prefix such that $I \subseteq g(I_0, \gamma)$. As $\gamma$ is a viable prefix, and $B \rightarrow \beta$ is valid for $\gamma$, there is a rightmost trace derivation of the form

$$S \Rightarrow^*_TR \gamma' B y' \Rightarrow^*_TR \gamma' \beta y'$$

where $\gamma = \gamma' \beta$. As $C \rightarrow \alpha$ is also valid for $\gamma$, there is a rightmost trace derivation of the form

$$S \Rightarrow^*_TR \gamma'' C y'' \Rightarrow^*_TR \gamma'' \alpha y''$$

where $\gamma = \gamma'' \alpha$.

We consider the different possible forms for the rules $B \rightarrow \beta$ and $C \rightarrow \alpha$.

Note that not all combinations of rules are possible as both $\alpha$ and $\beta$ are suffixes of $\gamma$, and therefore one must be a suffix of the other.

1. If $B \rightarrow \beta$ is a rule of type 1, then, as rules of type 1 can only be applied at the first step of the derivation, it follows that $C \rightarrow \alpha$ must also be of type 1, and therefore, $\beta = \alpha$ and $B = C = S$, and we have no conflict.

2. If $B \rightarrow \beta$ is a rule of type 2, then to meet the suffix requirement, $C \rightarrow \alpha$ is also of type 2. It follows that $\gamma' = \gamma''$, $\alpha = \beta = \sigma$ and $B = A_{q \sigma X}$ while $C = A_{p \sigma Y}$. Let $w$ be a word such that $\gamma' (= \gamma'') \Rightarrow^*_TR w$ by the derivation $d$. By Lemma 6.38, there exist two trace computations, one on $w\sigma y'$ and one on $w\sigma y''$. Furthermore, by Lemma 6.39, the first $N(d)$ transitions in both traces are identical, including branch selections. Therefore, from the determinism of $A$, the next transition must be the same, i.e., $q = p$, and $X = Y$, and we have no conflict.

3. If $B \rightarrow \beta$ is a rule of type 3, then if $C \rightarrow \alpha$ is also of type 3, then $\alpha = \beta$. From $\alpha = \beta$ we can deduce that $B = C$ as $q$, $X$ and $p$ all appear in the righthand side of the rule. Therefore, we have no conflict. The only other possibility for $C \rightarrow \alpha$ is a rule of type 5(c). In this case, we would still have $\alpha = \beta$, where $\alpha, \beta$ start with some $A_{q \epsilon X}$. However, if the transition $\delta(q, \epsilon, X)$ is a conjunctive one, then it cannot appear in
a rule of type 3, as that would be a contradiction to the determinism of $A$.

4. If $B \to \beta$ is of type 4, then $C \to \alpha$ is also type 4, and $\alpha = \beta = A_{q\sigma X}$. It follows that, $B = [q, X, p]$ and $C = [q, X, p']$. However, rules of type 4 match transitions of the form $\delta(q, a, X) = (p, \epsilon)$, and therefore, from the determinism of $A$, $p = p'$, and we have no conflict.

5. If $B \to \beta$ is of type 5(a), then $C \to \alpha$ is also of type 5(a). Therefore, $\alpha = \beta = S^q_{Xp}$, $B = C = [q, X, p]$, and we have no conflict.

6. If $B \to \beta$ is a conjunct of a rule of type 5(b), then $C \to \alpha$ is also a conjunct of a rule of type 5(b). Therefore, $\alpha = \beta = S^q_{Xp}$, and trivially, $B = C = S^q_{Xp}$, and we have no conflict.

7. If $B \to \beta$ is of type 5(c), then $C \to \alpha$ is of type 5(c) or type 3. In the latter case, by interchanging the rules, we proceed like item 3 above. In the former case, we can deduce that $\gamma' = \gamma''$, $\alpha = \beta$, and $B = S^q_{Xp}$, $C = S^q_{Xp}$. Let $w$ be a word such that $\gamma'(= \gamma'') \Rightarrow^*_T R w$ by the derivation $d$. Consider the two following trace derivations:

$$
S \Rightarrow^*_T R \gamma' S^q_{Xp} y' \Rightarrow^*_T R \gamma' \beta y' \Rightarrow^*_T R wy', \\
S \Rightarrow^*_T R \gamma'' S^q_{Xp} y'' \Rightarrow^*_T R \gamma'' \beta y'' \Rightarrow^*_T R wy''.
$$

By Lemma 6.38, there are two matching trace computations. These traces make the same transitions on $w$, but then the first selects the $i$-th branch and the second selects the $k$-th. Therefore, these traces are non-identical sibling traces, in contradiction to Lemma 6.39.

Therefore, there are no reduce-reduce conflicts in $C_g$. \qed
The proof of Theorem 6.32 is immediate from Lemmas 6.40 and 6.41.

We conclude this section with the following proposition, which extends Okhotin’s result from [15].

**Proposition 6.42.** *LR*(0) conjunctive languages contain a language that does not belong to the boolean closure of deterministic classical context-free languages.

The proof of Proposition 6.42 can be derived directly from the following lemma, Example 6.2, and Theorem 6.32.

**Lemma 6.43.** The language $L_{inf}$ from Example 6.2 is not a finite intersection of context-free languages.

Recall that

$$L_{inf} = \{ a^{i_1}b a^{i_2}b^2 \cdots a^{i_n}b^n \$ ba^{i_1} ba^{i_2} \cdots ba^{i_n} \$ | n \geq 1, i_1, \ldots, i_n \geq 1 \}.$$ 

To prove the lemma, we shall use the following theorem from [25].

**Theorem 6.44.** ([25, Theorem II.3]) For all $k \geq 2$, the language

$$L_k = \{ a_1^{i_1} a_2^{i_2} \cdots a_k^{i_k} \$ a_1^{i_1} a_2^{i_2} \cdots a_k^{i_k} \$ | i_1, \ldots, i_k \geq 1 \}$$

is not an intersection of $k - 1$ context-free languages.

We now prove Lemma 6.43.

**Proof.** Note that the class of $k$-intersections of context-free languages is closed under inverse homomorphism and intersection with regular sets, see [25]. Assume to the contrary that there exists a $k$ such that $L_{inf}$ is an intersection of $k - 1$ context-free languages. First, we intersect $L_{inf}$ with a regular language as follows:

$$L_k' = L_{inf} \cap (a^+ b^+)^k (ba^+)^*$$

$$= \{ a^{i_1} b \cdots a^{i_k} b^k \$ ba^{i_1} \cdots ba^{i_k} \$ | i_1, \ldots, i_k \geq 1 \}$$

Define the homomorphism $h : \{ a_1, \ldots, a_k, b_1, \ldots, b_k, \$ \} \rightarrow \{ a, b, \$ \}$ by
$h(a_i) = a$, $h(b_i) = b^i$, $i = 1, \ldots, k$, and $h(\$) = \$. Let
\[
L''_k = h^{-1}(L'_k) \cap (a_1^+b_1) \cdots (a_k^+b_k) \$ (b_1a_1^+) \cdots (b_1a_k^+) \$
\[
= \{a_1^i b_1 \cdots a_k^i b_k \$ b_1 a_1^i \cdots b_1 a_k^i \$ \mid i_1, \ldots, i_k \geq 1\}
\]

Then, by our assumption, $L''_k$ is an intersection of $k - 1$ context-free languages. Denote these languages by $L_1, \ldots, L_{k-1}$. We can assume that each of these languages only contains words of the form $a_1^+b_1 \cdots a_k^+b_k b_1 a_1^+ \cdots b_1 a_k^+$. Define the homomorphism $g : \{a_1, \ldots, a_k, b_1, \ldots, b_k, \$\} \rightarrow \{a_1, \ldots, a_k, \$\}$ by $g(a_i) = a_i$, $g(\$) = \$, and $g(b_i) = \epsilon$, $i = 1, \ldots, k$.

We observe that

$w \in L_1 \cap \cdots \cap L_{k-1}$ if and only if $g(w) \in g(L_1) \cap \cdot \cdots \cap g(L_{k-1})$.

This observation follows trivially from the observation that words in the intersection must align on the $a_1, \ldots, a_k$ symbols, regardless of the presence or absence of the $b_1, \ldots, b_k, \$ symbols.

Therefore, we obtain

\[
\{a_1^i \cdots a_k^i \$ a_1^i \cdots a_k^i \$ \mid i_1, \ldots, i_k \geq 1\} = g(L_1) \cap \cdots \cap g(L_{k-1})
\]
in contradiction to Theorem 6.44. \hfill \Box
Chapter 7

Conclusions and Future Directions

The class of conjunctive languages is an interesting extension of context-free languages as it is a rich language class, yet it is polynomially parsable. Conjunctive grammars are an appealing grammar model, not only for the class of languages they generate, but also for the fact that they highly resemble the classical context-free grammar model, thus making them intuitive and accessible to a wider audience, and more amenable to practical use.

Since their introduction in 2001, many papers have been written on conjunctive grammars describing properties such as closure properties, decision problems, parsing algorithms, normal forms etc. One major component missing from the theoretical framework for conjunctive grammars and conjunctive languages was an automaton model. In this thesis we have addressed this gap by introducing synchronized alternating pushdown automata, and proving they accept exactly the class of conjunctive languages.

SAPDA are a natural extension of classical PDA, and they interact with conjunctive grammars in much the same way that PDA interact with context-free grammars. As such they keep in the theme of being an intuitive extension of the classical framework. SAPDA lend new intuition to the class of conjunctive languages, and provide a new set of tools to explore. As in the case of the proof of closure under inverse homomorphism, SAPDA can greatly simplify certain proofs, and may help solve some of the open questions that still remain with respect to the language class.
In this work, we have also shown that, as in the classical case, the single-state and multi-state variants of the model are equivalent, and that the one-turn sub-family of SAPDA is equivalent to the linear sub-family of conjunctive grammars. In [12], Kutrib and Malcher explore a wide range of finite-turn automata with and without turn conditions, and their relationships with closures of linear context-free languages under regular operations. It would prove interesting to explore the general case of finite-turn SAPDA, perhaps finding models for closures of linear conjunctive languages under regular operations.

One of the major advantages of conjunctive grammars is their polynomial parsing. However, for practical applications, a linear algorithm is necessary. While several published works have suggested linear-time parsing algorithms for sub-classes of conjunctive languages, they are still rather limited. In context-free languages, $LR(1)$ parsing became the standard for practical parsing algorithms used in real-world compilers. Therefore, an $LR(1)$ parsing algorithm for conjunctive languages could be a crucial step towards making the language class viable for practical use. In this work we have taken a first important step in this direction by introducing an $LR(0)$ linear-time parsing algorithm for a strong sub-family of conjunctive languages. As in the classical $LR$ parsing algorithm, our parser is essentially a deterministic SAPDA, underscoring the importance of an automaton class for conjunctive languages.

It would of course be highly desirable to extend these results for general $LR(k)$ conjunctive grammars, and specifically for the widely used $LR(1)$ grammars. It would also be interesting to explore uses for such SAPDA based compilers. Three directions seem especially promising. The first is to look for examples where conjunctive grammars give a more succinct representation of an $LR(k)$ grammar, therefore leading to more efficient parsing. The second is to explore which criteria of programming languages can be expressed with conjunctive grammars, shifting certain compiler processes from the semantic phase to the syntactic one, and the impact of such a shift on the language specification and the compilation process. The third is to find examples of $LR(k)$ conjunctive languages that can be used to describe sophisticated constructs beyond the scope of context free languages. Such examples could prove useful for areas where context free languages have been known to be lacking, such as Natural Language Parsing.
Bibliography


100


האינטראקציה בין המודלים הקלאסיים של דקדוקים חסרי קשרי הקונוקטיביות, ובכך, האינטראקציה בין המודלים משלים חזר חסר השפה התחברתית וזו של השפה התחברתית, מקום אוסף יסורים משך חזק לשפה הלל. מבואר זאת, אנו מציגים מسار של אחת השפות של מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים מודל האוטומטיים, המחברת חזר לשפה הלל, כמות התחברתית לסטורט השפה הקונוקטיבית. כמות זה, אנו מציגים M
The work of Kleinert presented in this thesis extends to the analysis of context-free languages and their connection to other classes of languages.

Kleinert's work focused on the analysis of context-free languages and their connection to other classes of languages. Specifically, Kleinert's analysis of context-free languages is extended to other classes of languages, providing a deeper understanding of their properties and relationships.

The thesis presents Kleinert's analysis of context-free languages, which is extended to other classes of languages, providing a deeper understanding of their properties and relationships.

The thesis presents Kleinert's analysis of context-free languages, which is extended to other classes of languages, providing a deeper understanding of their properties and relationships.
The focus of the research is on formal languages and their applications in computer science. It was shown that certain classes of formal languages, such as context-free languages, do not form a basis for computer science.

They integrate a high expression capability suitable for practical tasks, but are amenable to analysis and identification in polynomial time, such as arithmetic expressions, syntactic structures, etc.

In practice, this makes the class of languages suitable for real-time identification in polynomial time.

The research in some fields led to the need for alternating models that extend the context-free languages. However, they are still efficient computationally.

Sandor Abonyi introduced such models that they are called alternating context-free languages. They are a generalization of context-free languages, and they are useful in practical applications.

The advantage of the model is that it becomes intuitive and accessible due to the strong resemblance to the classical model of context-free languages. This makes it suitable for a wide range of applications.

Models of finite automata were first presented by C. Stockmeyer, et al. in 1981. These automata change the way they receive a word, whereas the classical finite automata and Turing machines are equivalent in terms of computational power.

However, an alternating finite automaton is not equivalent to the classical model.

The thesis presents an alternating model of automata that is more powerful.

The main contribution of this work is the introduction of a new model of automata that is considered as a relaxation of the classical model.
The phd thesis of Michael Kamin斯基

The introduction of the thesis is written in Hebrew.

The thesis is written in Hebrew.

The thesis includes some technical terms in both Hebrew and English.

The thesis mentions several professors, including Michael Kamin斯基.

The thesis thanks several people for their support and contributions.

The thesis concludes with a thank you to the reader.
אוטומטיו מחסנית
מחלפים מסנכרניים

תומר אייזיקוביץ

לשם מילוי חלקי של הדרישות לקבלת דוקטור לפילוסופיה

הוגש לסנט הטכניון - מכון טכנולוגי לישראל

נובמבר תשע"א - חיפה

נובמבר 2010 - חיפה
אוטומטי מחסנית
מתחלפים מסונכרנים

תמר אייזיקוביץ