June 2018

Word Blending and Other Formal Models of Bio-operations

Zihao Wang
The University of Western Ontario

Supervisor
Kari, Lila
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

Abstract

As part of ongoing efforts to view biological processes as computations, several formal models of DNA-based processes have been proposed and studied in the formal language literature. In this thesis, we survey some classical formal language word and language operations, as well as several bio-operations, and we propose a new operation inspired by a DNA recombination lab protocol known as Cross-pairing Polymerase Chain Reaction, or XPCR. More precisely, we define and study a word operation called word blending which models a special case of XPCR, where two words xwy_1 and y_2wy sharing a non-empty overlap part w generate the word xwy. Properties of word blending that we study include closure properties of the Chomsky families of languages under this operation and its iterated version, existence of solution to equations involving this operation, and its state complexity.

Keywords: Formal language models, bio-operations, DNA computing, word blending, state complexity, decidability

Co-Authorship Statement

The main contribution of this thesis consists of the article "Word blending in formal languages: The Brangelina effect" which was accepted for publication in the proceedings of the seventeenth International Conference on Unconventional Computation and Natural Computation. Note that, as customary in computer science, the author order is alphabetical. The major individual contributions are listed below.

Srujan Kumar Enaganti - topic

Lila Kari - topic, research ideas and proofs, results, manuscript writing and editing

Timothy Ng - topic, research ideas and proofs, results, manuscript writing and editing

Zihao Wang - topic, research ideas and proofs, results, manuscript writing and editing, in particular, Section 5.3, 5.4 in Chapter 5

Acknowlegements

First and foremost, I want to thank my supervisor Professor Lila Kari for her support in my academic and personal life in the past four years. Without her guidance, comments and editing, I could not have finished this thesis. I am looking forward to the following four years as a Ph.D. student supervised by her.

Second, I would like to express my gratitude to my parents who supported all my life choices, especially my decision to study abroad.

Finally, I would like to thank my graduate student colleagues Gurjit Randhawa and Stephen Solis-Reyes, as well as Beth Locke (for editing help): I had a good time working and talking with them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In [1], Adleman used DNA to solve a seven-city instance of the Directed Hamiltonian Path Problem (a well-known NP-complete problem). This new approach inspired the study of DNA-based computing. A single strand of DNA can be viewed as a word over the alphabet $\{A, C, G, T\}$, so it is natural to model bio-operations on DNA as word operations.

In 1936, a formal model of computation called Turing machine was proposed [101]. The Church-Turing thesis states that Turing machines have universal computational power [16, 102], and Turing machines are as computationally powerful as electronic computers [17, 90]. Thus, it follows that if we have a set of word operations that model DNA bio-operations, and are able to simulate a Turing machine, we can theoretically use those DNA bio-operations to build a DNA-based computer. Examples of bio-operations defined and investigated in the formal language literature include splicing, overlap assembly, contextual insertion/deletion, block substitution, hairpin completion, and template-directed extension. For example, the formal system based on intra- and inter-molecular recombinations which model the process of gene unscrambling in ciliated protozoans, was proved to have the same computational power as a Turing machine, so its bio-operations could potentially be used as a basis for a DNA-based computer [65].

In this thesis, we continue the exploration of biologically-inspired word operations by defining and studying a novel bio-operation called *word blending*, which models a special case of the Cross-pairing Polymerase Chain Reaction (XPCR) technique [29].

XPCR operates as follows: Given two single-stranded DNA molecules represented by $\alpha A\beta, \beta D\gamma$, where α, β, γ are short sequences, and A, D are relatively long sequences, a new single-stranded DNA molecule, represented by $\alpha A\beta D\gamma$, can be produced by XPCR [30]. Consider now the special case of XPCR where the input consists of two single-stranded DNA molecules, $\alpha A\beta, \beta A\gamma$. While the expected outcome of XPCR would be $\alpha A\beta A\gamma$, in practical experiments the DNA molecule $\alpha A\gamma$ was the observed output [29]. We formalized this special

case of XPCR, and called it the word blending operation. Formally, if we have two single-stranded DNA molecules, represented by $\alpha A\beta_1, \beta_2 A\gamma$, the word blending of these two input strands is the strand $\alpha A\gamma$. This thesis investigates the word blending operation in the context of formal languages.

In Chapter 2, we give a hierarchical classification of languages, called the Chomsky hierarchy of languages, based on their generative grammars, and we discuss the decidability of problems related to these families of languages.

In Chapter 3, we survey some well-studied word and language operations with their definitions, and the closure properties of the Chomsky families of languages under these operations. We also study the abstract families of languages, which are defined based on its closure under a set of operations. Moreover, a type of descriptional complexity, called state complexity, of some operations is introduced.

In Chapter 4, we briefly review some bio-operations on single strands of DNA and the word operations that model them. For each biologically-inspired operation, we study its definition, its variations, and the closure properties of the Chomsky families of languages under this operation.

In Chapter 5, which is based on the article "Word blending in formal languages: The Brangelina effect", we introduce and study a special case of XPCR by the operation called word blending, and study the closure properties of the Chomsky families of languages under this operation and its iterated version, some decidability problems related to this operation, and its state complexity.

Chapter 2

Preliminaries

This chapter contains a basic introduction to the theory of formal languages. In Section 2.1, some notations that are used in this thesis are given. In Section 2.2, a hierarchical categorization basing on generative grammars of languages is introduced with related definitions. In Section 2.3, we survey some decision problems. The remaining chapters will follow the definitions and notations in this chapter, and we will assume that the information in this chapter is known.

2.1 Notations

The notation in this thesis follows the style of [93]. An alphabet Σ is a finite nonempty set of letters. A word over the alphabet Σ contains some letters from Σ, and the word with zero letters is the empty word λ. N_a^β denotes the number of occurrences of the letter a in the word β. The number of letters in a word β is its length and is denoted by $|\beta|$. The empty word λ is a special word with $|\lambda| = 0$. The infinite set $W(\Sigma)$ contains all the words over an alphabet Σ. A language L over an alphabet Σ is a subset of $W(\Sigma)$, so $L \in 2^{W(\Sigma)}$. A language L which does not contain the empty word λ is said to be λ-free. The letter at the position i of the word α is denoted by $\alpha_{(i)}$. A language can be represented by listing all of its words, by specifying word properties, by a generative device, or by a recognition device. We also need the following definitions.

Let $\alpha = a_1a_2...a_n$ be a word over an alphabet Σ. The reverse of the word α is denoted by $\alpha^r = a_na_{n-1}...a_1$, and the reverse of the empty word λ is λ.

Given two words α, β, their concatenation is denoted by $\alpha\beta$. Given a word α, the word obtained by concatenating α for i times is denoted by α^i, where $\alpha^0 = \lambda$. The concatenation closure of a word α is defined by $\alpha^* = \bigcup_{i\geq 0} \alpha^i$.

The set of positive integers is denoted by $\mathcal{N}^+$, and the set of natural numbers is denoted

by $\mathcal{N}$. Moreover, sets with different names are assumed to be disjoint; otherwise, this can be achieved by renaming elements in those sets.

2.2 The Chomsky Hierarchy of Languages

Languages can be classified according to the properties of their generative grammars, and the most used such classification is the Chomsky hierarchy [13]. First, we define grammars and the languages generated by grammars.

Definition 2.2.1 ([93]) *A grammar is a quadruple $G = (V_N, V_T, X_0, F)$, where V_N is the non-terminal alphabet, V_T is the terminal alphabet, $X_0 \in V_N$ is the initial symbol, and F is the set of production rules of the form $\alpha \to \beta$, where α, β are words over the alphabet $V_N \cup V_T$, and α contains at least one non-terminal letter.*

Next, we define the derivations of words according to a grammar $G = (V_N, V_T, X_0, F)$.

Let $\alpha, \alpha_1, \alpha_2, \beta$ be words over the alphabet $V_N \cup V_T$. A word $\alpha_1\alpha\alpha_2$ derives another word $\alpha_1\beta\alpha_2$ in one step, denoted by $\alpha_1\alpha\alpha_2 \Rightarrow \alpha_1\beta\alpha_2$, according to the production rules in F if and only if there is a rule $\alpha \to \beta$ in F.

Let k be a natural number, and α_i, $0 \leq i \leq k$, be words over the alphabet $V_N \cup V_T$. A word α_0 derives another word α_k, denoted by $\alpha_0 \Rightarrow^* \alpha_k$, according to the production rules in F if and only if $\alpha_0 \Rightarrow \alpha_1 \Rightarrow ... \Rightarrow \alpha_{k-1} \Rightarrow \alpha_k$ according to the production rules in F. Note that if $k = 0$, no derivations happen, and this indicates a derivation of length 0. Moreover, $\Rightarrow^*$ is the reflexive transitive closure of $\Rightarrow$.

With this definition of derivations, we can show how a word is generated by a grammar $G = (V_N, V_T, X_0, F)$. A word α over the terminal alphabet V_T is said to be generated by the grammar $G = (V_N, V_T, X_0, F)$ if and only if $X_0 \Rightarrow^* \alpha$ according to the production rules in F. The language generated by the grammar, denoted by $L(G)$, is the set of the words over the terminal alphabet V_T that can be derived from X_0 according to the production rules in F.

Example The grammar $G = (V_N, V_T, X_0, F)$ generates all the binary numbers, where:

$$\begin{aligned} V_N &= \{X_0\}, \\ V_T &= \{0, 1\}, \\ F &= \{X_0 \to X_0 1, X_0 \to X_0 0, X_0 \to \lambda\}. \end{aligned}$$

The word 10101 can be derived according to the derivation $X_0 \Rightarrow X_0 1 \Rightarrow X_0 01 \Rightarrow X_0 101 \Rightarrow X_0 0101 \Rightarrow X_0 10101 \Rightarrow 10101$.

By convention, the capital letters represent non-terminal letters, the lowercase letters from the beginning of the English alphabet represent terminal letters, and the lowercase letters from the end of the English alphabet and from the Greek alphabet represent words.

2.2.1 The Family of Regular Languages

In this section, we introduce the family of regular languages, whose grammars are most restrictive, and it is the smallest family of languages in the Chomsky hierarchy.

Definition 2.2.2 ([93]) *A grammar $G = (V_N, V_T, X_0, F)$ is called a type-3 grammar if and only if its production rules in F are of the form $X \to Y\alpha$ or $X \to \alpha$, where X, Y are letters from the non-terminal alphabet V_N, and α is a word over the terminal alphabet V_T.*

A language is said to be regular if and only if it can be generated by a type-3 grammar [13]. A type-3 grammar is also called a regular grammar. The family of regular languages contains exactly all the languages that can be generated by type-3 grammars.

We can assume that the production rules of all regular grammars have at most one terminal letter due to Lemma 2.2.3.

Lemma 2.2.3 *Given a regular grammar $G = (V_N, V_T, X_0, F)$, a regular grammar G' can be constructed such that all the production rules of G' have at most one terminal letter, where $L(G) = L(G')$.*

Proof The idea of the proof is as follows.

This is achieved by the construction $G' = (V'_N, V_T, X_0, F')$, where:

$$
\begin{aligned}
V'_N &= V_N \cup \{\langle X, Y, \alpha\rangle_i \mid X \to Y\alpha \in F, |\alpha| \geq 2, 1 \leq i < |\alpha|, i \in \mathcal{N}\} \\
&\quad \cup \{\langle X, \alpha\rangle_i \mid X \to \alpha \in F, |\alpha| \geq 2, 1 \leq i < |\alpha|, i \in \mathcal{N}\}, \\
F' &= \{X \to Y \in F\} \cup \{X \to Ya \in F\} \cup \{X \to \lambda \in F\} \cup \{X \to a \in F\} \\
&\quad \cup \{X \to \langle X, Y, \alpha\rangle_1 \alpha_{(1)} \mid X \to Y\alpha \in F, |\alpha| \geq 2\} \\
&\quad \cup \{\langle X, Y, \alpha\rangle_i \to \langle X, Y, \alpha\rangle_{i+1} \alpha_{(i+1)} \mid X \to Y\alpha \in F, |\alpha| \geq 3, 1 \leq i < |\alpha| - 1, i \in \mathcal{N}\} \\
&\quad \cup \{\langle X, Y, \alpha\rangle_{|\alpha|-1} \to Y\alpha_{|\alpha|} \mid X \to Y\alpha \in F, |\alpha| \geq 2\} \\
&\quad \cup \{X \to \alpha_{(1)}\langle X, \alpha\rangle_1 \mid X \to \alpha \in F, |\alpha| \geq 2\} \\
&\quad \cup \{\langle X, \alpha\rangle_i \to \langle X, \alpha\rangle_{i+1} \alpha_{(i+1)} \mid X \to \alpha \in F, |\alpha| \geq 3, 1 \leq i < |\alpha| - 1, i \in \mathcal{N}\} \\
&\quad \cup \{\langle X, \alpha\rangle_{|\alpha|-1} \to \alpha_{|\alpha|} \mid X \to \alpha \in F, |\alpha| \geq 2\}. \quad \blacksquare
\end{aligned}
$$

Next, we show some other ways to characterize regular languages. First, we show a type of recognition devices called non-deterministic finite automata that recognize regular languages. The following model, which was original proposed to model brain activity, was later formalized as a non-deterministic finite automaton [77], and it recognizes the family of regular languages.

Definition 2.2.4 ([93]) *A non-deterministic finite automaton (NFA) is a quintuple* $M = (S, V_T, s_0, A, F)$*, where* S *is the set of states,* V_T *is the alphabet,* $s_0 \in S$ *is the initial state,* $A \subseteq S$ *is the set of final states, and* $F \subseteq \{sa \to s' \mid s, s' \in S, a \in V_T\} \cup \{s \to s' \mid s, s' \in S\}$ *is the set of transitions.*

Next, we show the configurations and derivations of NFAs.

Let $M = (S, V_T, s_0, A, F)$ be an NFA. A configuration of the NFA M is a word of the form $s\alpha$, where $s \in S, \alpha \in W(V_T)$. An accepting configuration of M is a word of the form s, where $s \in A$, and the starting configuration of M with the input word $\alpha \in W(V_T)$ is $s_0\alpha$.

Consider states $s_1, s_2 \in S$, a terminal letter $a \in V_T$, and a word $\alpha \in W(V_T)$. A configuration $s_1 a\alpha$ derives another configuration $s_2\alpha$ in one step, denoted by $s_1 a\alpha \Rightarrow s_2\alpha$, according to the transitions in F if and only if there is a transition $s_1 a \to s_2 \in F$. Moreover, a configuration $s_1\alpha$ derives another configuration $s_2\alpha$ in one step, denoted by $s_1\alpha \Rightarrow s_2\alpha$, according to the transitions in F if and only if there is a transition $s_1 \to s_2 \in F$.

Consider a number $k \in \mathcal{N}$, states $s_{i_j} \in S, 0 \leq j \leq k, j \in \mathcal{N}$, and words $\alpha_j \in W(V_T), 0 \leq j \leq k, j \in \mathcal{N}$. A configuration $s_{i_0}\alpha_0$ derives another configuration $s_{i_k}\alpha_k$, denoted by $s_{i_0}\alpha_0 \Rightarrow^* s_{i_k}\alpha_k$, according to the transitions in F if and only if $s_{i_0}\alpha_0 \Rightarrow s_{i_1}\alpha_1 \Rightarrow ... \Rightarrow s_{i_{k-1}}\alpha_{k-1} \Rightarrow s_{i_k}\alpha_k$ according to the transitions in F. Note that if $k = 0$, no derivations happen, and that indicates a derivation of length 0.

With this definition of derivations, we can show how a word is recognized by an NFA $M = (S, V_T, s_0, A, F)$.

A word $\alpha \in W(V_T)$ is said to be recognized by the NFA M if and only if $s_0\alpha \Rightarrow^* s$ according to the transitions in F, where $s \in A$. The language recognized by the NFA M, denoted by $L(M)$, is the set of all the words recognized by the NFA M.

Consider an NFA $M = (S, V_T, s_0, A, F)$ and a word $\alpha \in L(M)$. The NFA M recognizes α deriving one of the possibly multiple derivations.

Example Consider the NFA $M = (\{s_0\}, \{a\}, s_0, \{s_0\}, \{s_0 \to s_0, s_0 a \to s_0\})$ and a word $a \in L(M) = a^*$. The NFA M recognizes a deriving the derivation $s_0 a \Rightarrow s_0$ or $s_0 a \Rightarrow s_0 a \Rightarrow s_0$, etc.

Next, we show that a language can be generated by type-3 grammars if and only if it can be recognized by NFAs with the help of the following definition and lemma.

Definition 2.2.5 ([93]) *Let Σ be an alphabet. The mirror image of a word $\alpha \in W(\Sigma)$ is defined by* $\mathrm{mi}(\alpha) = \alpha^r$. *The definition of mirror image can be extended to languages by* $\mathrm{mi}(L) = \{\mathrm{mi}(\alpha) \mid \alpha \in L\}$.

Note that, for all languages L, $\mathrm{mi}(\mathrm{mi}(L)) = L$, and for all words α, $\mathrm{mi}(\mathrm{mi}(\alpha)) = \alpha$.

Lemma 2.2.6 *Let L be a language recognized by an NFA. There exists an NFA that recognizes the language* $\mathrm{mi}(L)$.

Proof The idea of the proof is as follows.

Given an NFA $M = (S, V_T, s_0, A, F)$, an NFA M' that has exactly one accepting state which has no outgoing transitions can be constructed: this NFA is $M' = (S', V_T, s_0, A', F')$, where:

$$\begin{aligned} S' &= S \cup \{s_a\}, \\ A' &= \{s_a\}, \\ F' &= F \cup \{s \to s_a \mid s \in A\}. \end{aligned}$$

Given such NFA M', an NFA M_r that recognizes the language $\mathrm{mi}(L(M))$ can be constructed: this NFA is $M_r = (S', V_T, s_a, A_r, F_r)$, where:

$$\begin{aligned} A_r &= \{s_0\}, \\ F_r &= \{s_1 a \to s_2 \mid s_2 a \to s_1 \in F'\} \\ &\quad \cup \{s_1 \to s_2 \mid s_2 \to s_1 \in F'\}. \end{aligned}$$ ∎

On the one hand, the languages recognized by NFAs can be generated by type-3 grammars. Let M be an NFA, and $M_r = (S, V_T, s_0, A, F)$ be the NFA constructed from M according to Lemma 2.2.6, where $L(M_r) = \mathrm{mi}(L(M))$. A type-3 grammar $G = (V_N, V_T, X_{s_0}, F')$ can be constructed to generate the language $L(M)$ [15], where:

$$\begin{aligned} V_N &= \{X_s \mid s \in S\}, \\ F' &= \{X_{s_1} \to X_{s_2} a \mid s_1 a \to s_2 \in F\} \\ &\quad \cup \{X_{s_1} \to X_{s_2} \mid s_1 \to s_2 \in F\} \\ &\quad \cup \{X_s \to \lambda \mid s \in A\}. \end{aligned}$$

Thus, we have that the languages recognized by NFAs can be generated by type-3 grammars.

On the other hand, the languages generated by type-3 grammars can be recognized by NFAs. Let $G = (V_N, V_T, X_0, F)$ be a type-3 grammar, and we can assume that there is at

most one terminal letter in all the production rules due to Lemma 2.2.3. An NFA $M = (S, V_T, s_{X_0}, A, F')$ that recognizes all the words in $\mathrm{mi}(L(G))$ can be constructed, where:

$$\begin{aligned} S &= \{s_X \mid X \in V_N\} \cup \{s_A\}, \\ A &= \{s_A\}, \\ F' &= \{s_X a \to s_Y \mid X \to Ya \in F\} \\ &\quad \cup \{s_X a \to s_A \mid X \to a \in F\}. \end{aligned}$$

According to Lemma 2.2.6, there exists an NFA M' that recognizes the language $L(G) = \mathrm{mi}(L(M))$.

Thus, we have shown that a language can be generated by type-3 grammars if and only if it can be recognized by NFAs, and it follows that the languages that can be recognized by NFAs are regular.

Since for all languages L generated by type-3 grammars, there exists an NFA that recognizes the language $\mathrm{mi}(L)$, we have the following Corollary.

Corollary 2.2.7 *A language L is regular if and only if* $\mathrm{mi}(L)$ *is regular.*

Next, we show a restricted version of NFA that also recognizes regular languages [89].

Definition 2.2.8 ([93]) *A deterministic finite automaton (DFA) is a quintuple $M = (S, V_T, s_0, A, F)$, where S is the set of states, V_T is the alphabet, $s_0 \in S$ is the initial state, $A \subseteq S$ is the set of final states, and $F \subseteq \{sa \to s' \mid s, s' \in S, a \in V_T\}$ is the set of transitions. Moreover, F contains at most one transition $sa \to s'$ for each pair (s, a), where $s, s' \in S, a \in V_T$.*

Note that every DFA is a NFA. Thus, the configurations and derivations of DFAs can be defined similarly as NFAs. A word $\alpha \in W(V_T)$ is said to be recognized by an DFA $M = (S, V_T, s_0, A, F)$ if and only if $s_0\alpha \Rightarrow^* s$ according to the transitions in F, where $s \in A$. The language recognized by the DFA M, denoted by $L(M)$, is the set of all the words recognized by the DFA M.

Consider a DFA $M = (S, V_T, s_0, A, F)$ and a word $\alpha \in L(M)$. The DFA M recognizes α deriving exactly one possible derivation which is of length $|\alpha|$.

A DFA $M = (S, V_T, s_0, A, F)$ is said to be complete if F contains exactly one transition $sa \to s'$ for each pair (s, a), where $s, s' \in S, a \in V_T$. A state is called a sink state if and only if no final states can be reached from it.

Let M be a DFA, and $L = L(M)$ be the language recognized by the DFA M. The DFA M is called the minimal DFA for L if there does not exist any other DFA M' recognizing the language L that has less states than the DFA M. Note that for any language that can be

recognized by a DFA, its minimal DFA is unique because this can be achieved by renaming states.

Next, we show that a language can be recognized by NFAs if and only if it can be recognized by DFAs.

For the direct implication, the languages recognized by NFAs can be recognized by DFAs. Given an NFA $M = (S, V_T, s_0, A, F)$, the DFA $M' = (2^S, V_T, s_0', A', F')$ can be constructed such that $L(M') = L(M)$ using the subset construction [69]. Let $S_1, S_2 \in 2^S$ be subsets of S, and $a \in V_T$ be a letter. We have that $S_1 a \to S_2 \in F'$ if and only $s_1 a \to s_2$ for all $s_1 \in S_1, s_2 \in S_2$.

Conversely, DFAs are NFAs. Thus, a language can be recognized by NFAs if and only if it can be recognized by DFAs, and it follows that a language is regular if and only if it can be recognized by DFAs.

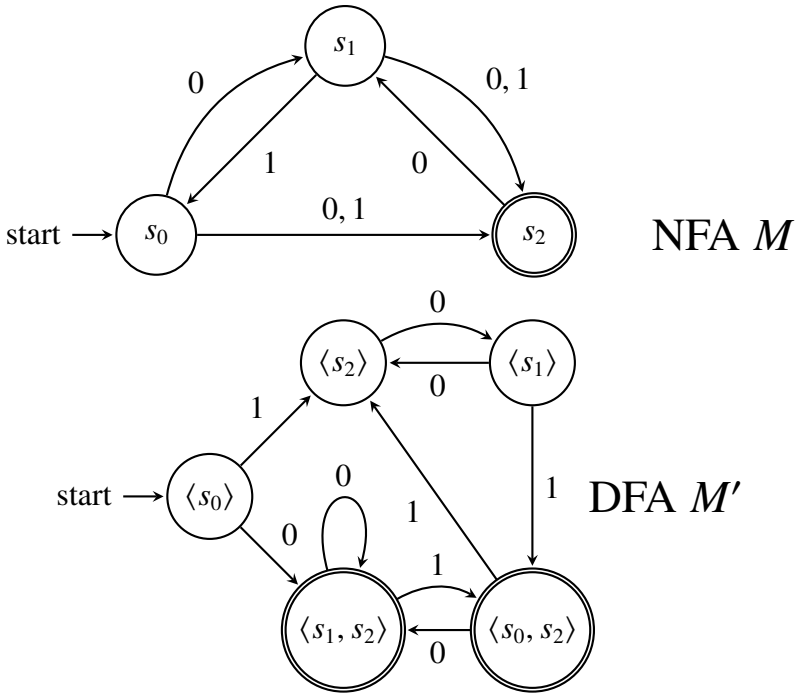

Figure 2.1: The DFA M' is constructed from the NFA M using the subset construction, where $L(M') = L(M)$, and each state of the DFA M' represents a nonempty subset of the set of states of the NFA M

Example As shown in the Figure 2.1, using the subset construction method, the NFA $M = (S, V_T, s_0, A, F)$ can be converted to the DFA $M' = (S', V_T, \langle s_0 \rangle, A', F')$, where:

$$S = \{s_0, s_1, s_2\},$$
$$V_T = \{0, 1\},$$

$$
\begin{aligned}
A &= \{s_2\},\\
F &= \{s_0 0 \to s_1, s_0 0 \to s_2, s_0 1 \to s_2, s_1 0 \to s_2, s_1 1 \to s_2, s_1 1 \to s_0, s_2 0 \to s_1\},\\
S' &= \{\langle s_0\rangle, \langle s_1\rangle, \langle s_2\rangle, \langle s_0, s_2\rangle, \langle s_1, s_2\rangle\},\\
A' &= \{\langle s_0, s_2\rangle, \langle s_1, s_2\rangle\},\\
F' &= \{\langle s_0\rangle 0 \to \langle s_1, s_2\rangle, \langle s_0\rangle 1 \to \langle s_2\rangle, \langle s_1\rangle 0 \to \langle s_2\rangle, \langle s_1\rangle 1 \to \langle s_0, s_2\rangle, \langle s_2\rangle 0 \to \langle s_1\rangle\}\\
&\quad \cup \{\langle s_0, s_2\rangle 0 \to \langle s_1, s_2\rangle, \langle s_0, s_2\rangle 1 \to \langle s_2\rangle, \langle s_1, s_2\rangle 0 \to \langle s_1, s_2\rangle, \langle s_1, s_2\rangle 1 \to \langle s_0, s_2\rangle\}.
\end{aligned}
$$

Note that NFAs are usually simpler than DFAs recognizing the same language in terms of the number of states and the number of transitions.

Next, we show another type of grammar that generates regular languages.

Definition 2.2.9 ([35]) *A grammar is said to be linear if and only if its production rules are of the form $X \to \alpha Y \beta$ or $X \to \gamma$, where X, Y are letters from the non-terminal alphabet V_N, and α, β, γ are words over the terminal alphabet V_T. It is said to be left-linear (resp. right-linear) if and only if all of the words α (resp. β) are λ.*

Note that left-linear grammars are type-3 grammars.

Next, we show that a language is regular if and only if it can be generated by a right-linear grammar.

First, we show that regular languages can be generated by right-linear grammars. Let G be a type-3 grammar, there exists a type-3 grammar $G_r = (V_N, V_T, X_0, F)$ that generates $\mathrm{mi}(L(G))$ according to Corollary 2.2.7. A right-linear grammar G' that generates the language $L(G)$ can be constructed, and this is achieved by the construction $G' = (V_N, V_T, X_0, \{X \to \alpha^r \mid X \to \alpha \in F\} \cup \{X \to \alpha^r Y \mid X \to Y\alpha \in F\})$.

Next, we show that languages that can be generated by right-linear grammars are regular. Let $G = (V_N, V_T, X_0, F)$ be a right-linear grammar, and we can assume that there is at most one terminal letter in all the production rules in F using a similar construction in Lemma 2.2.3. An NFA M that recognizes $L(G)$ can be constructed, and this is achieved by constructing the NFA $M = (S, V_T, s_{X_0}, A, F')$, where:

$$
\begin{aligned}
S &= \{s_X \mid X \in V_N\} \cup \{s_A\},\\
A &= \{s_A\},\\
F' &= \{s_X a \to s_Y \mid X \to aY \in F, X, Y \in V_N, a \in V_T\}\\
&\quad \cup \{s_X a \to s_A \mid X \to a \in F, X \in V_N, a \in V_T\}.
\end{aligned}
$$

Using these constructions, we have that a language is regular if and only if it can be generated by a right-linear grammar.

Next, we show that a language is regular if and only if it can be denoted by expressions called regular expressions. First, we define regular expressions and some related operations.

Definition 2.2.10 ([93]) *Given two languages L_1 and L_2, their union is defined by $L_1 \cup L_2 = \{\alpha \mid \alpha \in L_1 \text{ or } \alpha \in L_2\}$.*

Definition 2.2.11 ([93]) *The definition of concatenation can be extended to languages naturally by $L_1L_2 = \{\alpha\beta \mid \alpha \in L_1, \beta \in L_2\}$.*

Definition 2.2.12 ([93]) *The concatenation closure L^* of a language L can be defined recursively as $L^* = \{\lambda\} \cup \{\alpha \mid \alpha \in L\} \cup \{\alpha\beta \mid \alpha, \beta \in L^*\}$.*

Definition 2.2.13 ([93]) *Let $V, V' = \{\cup, *, \emptyset, (,)\}$ be two alphabets. A word α over the alphabet $V \cup V'$ is a regular expression over V if and only if*

- *α is a letter of V or the letter $\emptyset$, or*
- *α is of the form $(\beta \cup \gamma), (\beta\gamma)$, or β^* where β, γ are regular expressions over V.*

Next, we show what is the language denoted by a regular expression α over an alphabet V as follows.

- If $\alpha = \emptyset$, $L(\emptyset) = \emptyset$, which is the empty language.
- If $\alpha = a, a \in V$, $L(a) = \{a\}$, which is the language containing only the word a.
- If $\alpha = (\beta\gamma)$, where β, γ are regular expressions over the alphabet V, $L((\beta \cup \gamma)) = L(\beta) \cup L(\gamma)$, which is the union of two languages $L(\beta)$ and $L(\gamma)$.
- If $\alpha = (\beta\gamma)$, where β, γ are regular expressions over the alphabet V, $L((\beta\gamma)) = L(\beta)L(\gamma)$, which is the concatenation of two languages $L(\beta)$ and $L(\gamma)$.
- If $\alpha = \beta^*$, where β is a regular expression over the alphabet V, $L(\beta^*) = L(\beta)^*$, which is the concatenation closure of the language $L(\beta)$.

A language L over an alphabet V is said to be denoted by a regular expression α over the alphabet V if and only if $L = L(\alpha)$. A language is regular if and only if it can be denoted by a regular expression [61], and this can be proved using the construction in [100].

Next, we show a language operation, and we can represent the result of the operation on regular languages by regular expressions.

Definition 2.2.14 ([93]) *Consider an alphabet V, a set of alphabets $\{V_a \mid a \in V\}$, and a set of languages $\{\sigma(a) \in 2^{V_a} \mid a \in V\}$. Substitution is an operation that substitutes every letter $a \in V$ in a word with the language $\sigma(a)$. The image of the substitution σ on words is defined recursively by $\sigma(\lambda) = \lambda, \sigma(\alpha\beta) = \sigma(\alpha)\sigma(\beta)$. The image of the substitution σ on a language L is defined by $\sigma(L) = \{\alpha \mid \alpha \in \sigma(\beta) \text{ for some } \beta \in L\}$.*

We can prove the next lemma using regular expressions.

Lemma 2.2.15 *Consider a regular language L over an alphabet Σ, a set of regular languages L_a over the alphabet Σ for $a \in \Sigma$, and a substitution σ defined by $\sigma(a) = L_a$ for $a \in \Sigma$. We have that $\sigma(L)$ is a regular language.*

Proof Given a regular language L denoted by a regular expression R and a substitution σ with a set of regular expressions $\{R_a \mid a \in V_T, \sigma(a) = L(R_a)\}$, a regular expression R' can be constructed by replacing each letter a in R by R_a. It is easy to see that R' is a regular expression, and R' denotes the language $\sigma(L)$. Thus, we have that $\sigma(L)$ is regular. ∎

The following proposition summarizes the different methods mentioned in this subsection to specify a regular language.

Proposition 2.2.16 *The following statements about a language L are equivalent.*

- *L is regular.*
- *There exists a type-3 grammar that generates L.*
- *There exists an NFA that recognizes L.*
- *There exists a DFA that recognizes L.*
- *There exists a right-linear grammar that generates L.*
- *There exists a regular expression that denotes L.*

Next, we show a family of languages that is included in the family of regular languages.

A language is said to be finite if and only if it contains a finite number of words. We have the following proposition.

Proposition 2.2.17 *All finite languages are regular.*

Proof Let Σ be an alphabet, L be a finite language with $n \in \mathcal{N}$ words. We consider the following cases.

- If $n = 0$, $L = \emptyset$, which is regular.
- If $n = 1$, $L = \{\alpha\} = L(\alpha)$, which is regular.
- If $n \geq 2$, $L = \{\alpha_i \in \Sigma^* \mid 1 \leq i \leq n, i \in \mathcal{N}\} = L(\beta_n)$, where $\beta_i = (\alpha_i \cup \beta_{i-1})$ for $i \in \mathcal{N}, i \geq 3$, and $\beta_2 = (\alpha_1 \cup \alpha_2)$, which is regular. ∎

Next, we show that the family of regular languages does not contain all the possible languages over a given alphabet.

Example The language $L = \{a^n b^n \mid n \in \mathcal{N}\}$ is not regular.

The following lemma, called the pumping lemma for regular languages, gives us a method for showing that a language is not regular.

Lemma 2.2.18 ([89]) *If L is a regular language, then there exists a number $k \in \mathcal{N}^+$ such that for all words $w \in L$ whose length is at least k, there exists a decomposition of $w = xyz$, where the length of the word xy is at most k, the word y is not an empty word, and the word $xy^q z$ is in the regular language L for any number $q \in \mathcal{N}$.*

The proof is based on the idea that given a regular language L, a DFA $M = \{S, V_T, s_0, A, F\}$ that recognizes the language L, and a word $\alpha \in L$, if $|\alpha| \geq |S|$, there is a state that is visited at least twice in the derivation of α; otherwise, the length of the word α would be less than $|S|$. Then, there is a loop in the derivation of α, and the loop should happen in the first $|S|$ visited states. The basic idea behind the pumping lemma for regular languages is that removing or adding more copies of such loop part of the derivation still results in a derivation for a word in L.

Example The pumping lemma can be used, for example, to show that the language $L = \{a^n b^n \mid n \in \mathcal{N}\}$ is not regular because of the word $a^k b^k$ cannot be pumped and still belongs to the language L.

2.2.2 The Family of Context-Free Languages

In this section, we introduce the family of context-free languages, which contains all regular languages and the language $L = \{a^n b^n \mid n \in \mathcal{N}\}$.

Definition 2.2.19 ([93]) *A grammar $G = (V_N, V_T, X_0, F)$ is called a type-2 grammar if and only if its production rules in F are of the form $X \rightarrow \alpha$, where X is a letter from the non-terminal alphabet V_N, and α is a word over the alphabet $V_T \cup V_N$.*

A language is said to be context-free if and only if it can be generated by a type-2 grammar [13]. A type-2 grammar is also called a context-free grammar. Note that all regular grammars are type-2 grammars, so all regular languages are context-free. The family of context-free languages contains exactly all the languages that can be generated by type-2 grammars.

Example The language $L = \{a^n b^n \mid n \in \mathcal{N}\}$ can be generated by the type-2 grammar $G = (V_N, V_T, X_0, F)$, where:

$$\begin{aligned} V_N &= \{X_0\}, \\ V_T &= \{a, b\}, \\ F &= \{X_0 \to aX_0b, X_0 \to \lambda\}. \end{aligned}$$

Thus, if follows that L is context-free.

Note that there are no restrictions on the forms of the right hand side of production rules in F in the definition. However, there are two commonly used normal forms of context-free grammars, the Chomsky normal form and the Greibach normal form, that have restrictions on the form of the production rules in F.

Definition 2.2.20 ([13]) *A type-2 grammar $G = (V_N, V_T, X_0, F)$ is said to be in the Chomsky normal form if and only if its production rules are of the form $X \to a$ or $X \to YZ$, where $a \in V_T, X, Y, Z \in V_N$.*

Definition 2.2.21 ([40]) *A type-2 grammar $G = (V_N, V_T, X_0, F)$ is said to be in Greibach normal form if and only if its production rules are of the form $X \to a\beta$, where $a \in V_T, X \in V_N, \beta \in V_N^*$.*

Note that for all context-free languages L, there exists a context-free grammar G in the Chomsky normal form or the Greibach normal form that generates exactly the language $L\backslash\{\lambda\}$. Thus, we have the following corollary.

Corollary 2.2.22 *Given a context-free language L, there exists a context-free grammar such that all the production rules of this grammar have at most one terminal letter.*

Proof Given a context-free language L, there exists a context-free grammar $G = (V_N, V_T, X_0, F)$ in the Chomsky normal form that generate the language $L\backslash\{\lambda\}$. If L is $\lambda-$free, $L = L(G)$; otherwise, $L = L(G')$, where $G' = (V_N, V_T, X_0, F \cup \{X_0 \to \lambda\})$. In both cases, all the production rules of the grammar have at most one terminal letter. ∎

Next, we show another way to characterize context-free languages, by recognition devices called pushdown automata that can recognize exactly context-free languages [97].

Definition 2.2.23 ([93]) *A pushdown automaton (PDA) is a septuple* $M = (S, V_I, V_Z, z_0, s_0, A, F)$*, where* S *is the set of states,* V_I *is the input alphabet,* V_Z *is the stack alphabet,* $z_0 \in V_Z$ *is the initial stack letter,* $s_0 \in S$ *is the initial state,* $A \subseteq S$ *is the set of final states, and* $F \subseteq \{zs_ia \to \alpha s_j \mid z \in V_Z, a \in V_I, \alpha \in V_Z^*, s_i, s_j \in S\} \cup \{zs_i \to \alpha s_j \mid z \in V_Z, \alpha \in V_Z^*, s_i, s_j \in S\}$ *is the set of transitions.*

Note that a PDA is basically an NFA with a stack. Consider a PDA that has a production rule $zs_ia \to \alpha s_j$. If the letter at the top of its stack is z, and the input letter is a, it can change its current state from s_i to s_j and pushes α into the stack after z is removed from the top of the stack. Moreover, consider a PDA has a production rule $zs_i \to \alpha s_j$, if the letter at the top of its stack is z, it can change its current state from s_i to s_j and pushes α into the stack after z is removed from the top of the stack.

Let $M = (S, V_I, V_Z, z_0, s_0, A, F)$ be a PDA. A configuration of the PDA is a word of the form $\beta s\alpha$, where $s \in S, \alpha \in V_I^*, \beta \in V_Z^*$. Note that β is the word on the stack with the rightmost letter at the top of the stack. The initial configuration of M with input word $\alpha \in V_I^*$ is the word $z_0s_0\alpha$. An accepting configuration is a word of the form βs, where $s \in A, \beta \in V_Z^*$.

Consider states $s_1, s_2 \in S$, a stack letter $z \in V_Z$, an input letter $a \in V_I$, and stack words $\gamma, \beta \in V_Z^*$. A configuration $\beta z s_1 a \alpha$ derives another configuration $\beta\gamma s_2\alpha$ in one step, denoted by $\beta z s_1 a\alpha \Rightarrow \beta\gamma s_2\alpha$, according to the transitions in F if and only if there is a transition $zs_1a \to \gamma s_2 \in F$. Moreover, a configuration $\beta z s_1 \alpha$ derives another configuration $\beta\gamma s_2\alpha$ in one step, denoted by $\beta z s_1 \alpha \Rightarrow \beta\gamma s_2\alpha$, according to the transitions in F if and only if there is a transition $zs_1 \to \gamma s_2 \in F$.

Consider a number $k \in \mathcal{N}$, states $s_{i_j} \in S$, $0 \leq j \leq k, j \in \mathcal{N}$, input words $\alpha_j \in V_I^*$ for $0 \leq j \leq k, j \in \mathcal{N}$, and stack words $\beta_j \in V_Z^*$, $0 \leq j \leq k, j \in \mathcal{N}$. A configuration $\beta_0 s_{i_0}\alpha_0$ derives another configuration $\beta_k s_{i_k}\alpha_k$, denoted by $\beta_0 s_{i_0}\alpha_0 \Rightarrow^* \beta_k s_{i_k}\alpha_k$, according to the transitions in F if and only if $\beta_0 s_{i_0}\alpha_0 \Rightarrow \beta_1 s_{i_1}\alpha_1 \Rightarrow ... \Rightarrow \beta_{k-1} s_{i_{k-1}}\alpha_{k-1} \Rightarrow \beta_k s_{i_k}\alpha_k$ according to the transitions in F. Note that if $k = 0$, no derivations happen, and this indicates a derivation of length 0.

A word $\alpha \in V_I^*$ is said to be recognized by a PDA $M = (S, V_I, V_Z, z_0, s_0, A, F)$ if and only if $z_0s_0\alpha \Rightarrow^* \beta s$ according to the transitions in F, where $s \in A, \beta \in V_Z^*$. The language recognized by the PDA M, denoted by $L(M)$, is the set of all the words recognized by the PDA M.

Example The language $L = \{a^n c^+ b^n \mid n \in \mathcal{N}^+\}$ can be recognized by the PDA $M = (S, V_I, V_Z, e, s_0, A, F)$ illustrated in Figure 2.2, where:

$S = \{s_0, s_1, s_2, s_3\}$,

$$
\begin{aligned}
V_I &= \{a, b, c\},\\
V_Z &= \{d, e\},\\
A &= \{s_3\},\\
F &= \{es_0a \to eds_0, ds_0a \to dds_0, ds_0c \to ds_1, ds_1c \to ds_1, ds_1b \to s_2, ds_2b \to s_2, es_2 \to s_3\}.
\end{aligned}
$$

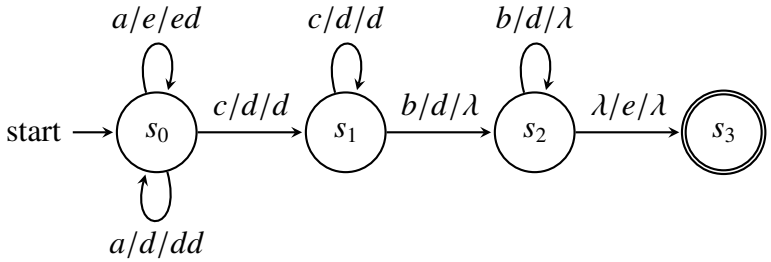

Figure 2.2: A PDA that recognizes the language $\{a^n c^+ b^n \mid n \in \mathcal{N}^+\}$

Next, we show that a language can be generated by type-2 grammars if and only if it can be recognized by PDAs.

For the direct implication, consider the derivations of a word by a type-2 grammar. If we apply the production rules to the rightmost non-terminal letter in each step, the derivation can be modeled by a stack. Thus, we can convert a type-2 grammar to a PDA using the following construction.

Given a type-2 grammar $G = (V_N, V_T, X_0, F)$, a PDA M that recognizes the language $L(G)$ can be constructed: this PDA is $M = (S, V_T, V_Z, X_0, s, A, F')$, where:

$$
\begin{aligned}
S &= \{s\},\\
V_Z &= V_T \cup V_N,\\
A &= \{s\},\\
F' &= \{asa \to s \mid a \in V_T\}\\
&\quad \cup \{Xs \to \alpha s \mid X \to \alpha \in F, \alpha \in (V_N \cup V_T)^*\}.
\end{aligned}
$$

Thus, we have that languages generated by type-2 grammars can be recognized by PDAs. Conversely, languages recognized by PDAs can be generated by type-2 grammars, which can be proven using the following lemma.

Lemma 2.2.24 ([90]) *Let $M = (S, V_I, V_Z, z_0, s_0, A, F)$ be a PDA, and $\# \notin V_Z$ be a special stack letter. A PDA $M' = (S', V_I, V_Z \cup \{\#\}, z'_0, s_s, \{s_a\}, F')$ that recognizes $L(M)$ can be constructed, where*

- *the initial state s_s is visited only once in any derivation,*
- *the initial state s_s transits to another state leaving # on the stack without reading any input letters, and*
- *all transitions into the final state s_a pop the special letter # and read no input letters.*

Proof The idea of the proof is as follows.

Let $M = (S, V_I, V_Z, z_0, s_0, A, F)$ be a PDA. A PDA $M' = (S', V_I, V'_Z, \#, s_s, A', F')$ that recognizes the language $L(M)$ and fulfills the conditions can be constructed, where:

$$\begin{aligned}
S' &= S \cup \{s_s, s_a, s_1\},\\
V'_Z &= V_Z \cup \{\#\},\\
A &= \{s_a\},\\
F' &= F \cup \{\#s \to s_a \mid s \in A\}\\
&\quad \cup \{as \to s_1 \mid a \in V_Z, s \in A \cup \{s_1\}\}\\
&\quad \cup \{\#s_1 \to s_a\}\\
&\quad \cup \{\#s_s \to \#z_0 s_0\}. \quad \blacksquare
\end{aligned}$$

For the language L recognized by an arbitrary PDA $M = (S, V_I, V_Z, s_0, A, F)$, a PDA $M' = (S \cup \{s_s, s_a, s_1\}, V_I, V_Z \cup \{\#\}, \#, s_s, \{s_a\}, F')$ that recognizes L can be constructed using the construction in the proof for Lemma 2.2.24. Based on the PDA M', a context-free grammar $G = (V_N, V_I, X_0, F)$ that generates exactly the language L can be constructed [90], where

$$\begin{aligned}
V_N &= \{\langle s_i, a, s_j\rangle \mid s_i, s_j \in S \cup \{s_s, s_a, s_1\}, a \in V_Z \cup \{\#\}\}\\
&\quad \cup \{\langle s_i, \lambda, s_j\rangle \mid s_i, s_j \in S \cup \{s_s, s_a, s_1\}\}\\
&\quad \cup \{X_0\},\\
F &= \{X_0 \to \langle s_0, \#, s_a\rangle\}\\
&\quad \cup \{\langle s_i, a, s_j\rangle \to c\langle s_k, \lambda, s_j\rangle \mid as_ic \to s_k \in F', s_j \in S \cup \{s_s, s_a, s_1\}\}\\
&\quad \cup \{\langle s_i, a, s_j\rangle \to c\langle s_k, b, s_j\rangle \mid as_ic \to bs_k \in F', s_j \in S \cup \{s_s, s_a, s_1\}\}\\
&\quad \cup \{\langle s_i, a, s_j\rangle \to c\langle s_i, b_0, s_{k_1}\rangle\langle s_{k_1}, b_1, s_{k_2}\rangle ...\langle s_{k_n}, b_n, s_j\rangle \mid as_ic \to b_0b_1...b_ns_k \in F',\\
&\quad s_{k_1}, s_{k_2}, ..., s_{k_n} \in S, s_j \in S \cup \{s_s, s_a, s_1\}, n \geq 1\}\\
&\quad \cup \{\langle s, \lambda, s\rangle \to \lambda \mid s \in S \cup \{s_s, s_a, s_1\}\}.
\end{aligned}$$

The above constructions show that a language can be generated by type-2 grammars if and only if it can be recognized by PDAs [97], and it follows that a language is context-free if and only if it can be recognized by PDAs. We have the following proposition.

Proposition 2.2.25 *The following statements about a language L are equivalent.*

- *L is context-free.*
- *There exists a type-2 grammar that generates L.*
- *There exists a PDA that recognizes L.*

Thus, $L = \{a^n c^* b^n \mid n \in \mathcal{N}^+\}$ is a context-free language since it can be recognized by the PDA in Figure 2.2.

By Definition 2.2.9, it follows that type-2 grammars are linear. Next, we introduce another restriction related to context-free grammars.

Definition 2.2.26 ([13]) *Consider a context-free grammar $G = (V_N, V_T, X_0, F)$. This grammar is said to be self-embedding if and only if there exists a non-terminal letter $X \in V_N$ such that $X \Rightarrow^* \alpha X \beta$, where $\alpha, \beta \in \{V_N \cup V_T\}^+$.*

The definition of self-embedding is extended to a context-free language based on the grammars that generate this language.

Definition 2.2.27 ([93]) *A context-free language L is said to be self-embedding if and only if all context-free grammars that generate the language L are self-embedding.*

Let $G = (V_N, V_T, X_0, F)$ be a context-free grammar. Without loss of generality, we can assume that, for any non-terminal letter $X \in V_N$, there exists a derivation $X_0 \Rightarrow^* \alpha X \beta$, where $\alpha, \beta \in (V_N \cup V_T)^*$; otherwise, everything related to X can be removed because X is not used in any derivations. Next, we show that a context-free language is not self-embedding if and only if it is regular.

Proposition 2.2.28 ([3, 47]) *Context-free languages that are not self-embedding are regular.*

Proof For the direct implication, context-free languages that are not self-embedding are regular. Let $G = (V_N, V_T, X_0, F)$ be a type-2 grammar that is not self-embedding. There are two cases to consider: whether or not every non-terminal letter $X \in V_N$ can derive X_0 in a derivation $X \Rightarrow^* \alpha X_0 \beta$, where $\alpha, \beta \in (V_N \cup V_T)^*$.

Consider the case where every non-terminal letter $X \in V_N$ can derive a word containing X_0 according to the production rules in F. For all $X \in V_N$, all the production rules that are used in the derivation $X \Rightarrow^* \alpha X_0 \beta$, where $\alpha, \beta \in (V_N \cup V_T)^*$, and have non-terminal letters on their right-hand side can be of the form $X \to \alpha Y \beta$, $X \to \alpha Y$, $X \to Y\beta$, or $X \to Y$, where $Y \in V_N, \alpha, \beta \in (V_T \cup V_N)^+$.

Note that the production rules F do not contain any rules of the form $X \to \alpha Y \beta$; otherwise, G is self-embedding because of the derivation of the form $X \Rightarrow^* \alpha_1 Y \beta_1 \Rightarrow^* \alpha_2 X_0 \beta_2 \Rightarrow^* \alpha_3 X \beta_3$, where $\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, \beta_3 \in (V_N \cup V_T)^+$.

If the production rules contain one rule of the form $X \to \alpha Y$, we have that $\alpha \in V_T^+$, $\alpha \in V_N V_T^*$, or $\alpha \in V_T^+ V_N (V_N \cup V_T)^*$. If $\alpha \in V_T^+ V_N (V_N \cup V_T)^*$, G is self-embedding as shown above; if $\alpha \in V_N V_T^*$, $\alpha = Z\gamma$, where $Z \in V_N, \gamma \in V_T^*$, G is self-embedding because of the derivation of the form $X \Rightarrow^* \alpha_1 Z \beta_1 Y \Rightarrow^* \alpha_2 X_0 \beta_2 X_0 \Rightarrow^* \alpha_3 Z \beta_3 Y \gamma_3 X_0 \Rightarrow^* \alpha_4 Z \beta_4 X_0 \gamma_4 X_0 \Rightarrow^* \alpha_5 Z \beta_5 X \gamma_5 X_0$, where $\alpha_i, \beta_i, \gamma_j \in (V_N \cup V_T)^*$ for $1 \leq i \leq 5, 3 \leq j \leq 5$. Thus, if the production rules contain one rule of the form $X \to \alpha Y$, G is right-linear, and $L(G)$ is regular.

Similarly, if the production rules contain one rule of the form $X \to Y\beta$, G is left-linear, and $L(G)$ is regular. Thus, if every non-terminal letter $X \in V_N$ can derive a word containing X_0, $L(G)$ is regular.

Now consider the case where there is a non-terminal letter $X \in V_N$ that cannot derive any word containing X_0. We want to prove by induction that if there are $n \in \mathcal{N}^+$ non-terminal letters, among which there exists a non-terminal letter $X \in V_N$ that cannot derive a word containing X_0, $L(G)$ is a regular language.

If $n = 1$, $X_0 \Rightarrow^* X_0$, so it is vacuously true that G is regular. Assume the statement holds for a grammar G with $k \geq 1$ non-terminal letters. Consider now a grammar G with $k + 1$ non-terminal letters, and among them, a non-terminal letter X_1 cannot derive a word containing X_0.

Let $G_1 = (V_{N_1}, V_T, X_1, F_1)$ be the grammar that generates exactly all the words that can be derived from X_1, where:

$$\begin{aligned} V_{N_1} &= V_N \backslash \{X_0\}, \\ F_1 &= \{X \to \alpha \in F \mid X \in V_N \backslash \{X_0\}, \alpha \in ((V_N \backslash \{X_0\}) \cup V_T)^*\}. \end{aligned}$$

Let $G_2 = (V_{N_2}, V_{T_2}, X_0, F_2)$ be the grammar that generates exactly all the words that treat X_1 as a terminal letter, where:

$$\begin{aligned} V_{N_2} &= V_N \backslash \{X_1\}, \\ V_{T_2} &= V_T \cup \{X_1\}, \\ F_2 &= F \backslash \{X_1 \to \alpha \mid \alpha \in (V_N \cup V_T)^*\}. \end{aligned}$$

We have that $L(G_1), L(G_2)$ are regular because of the induction assumption or the other case.

Let σ be a substitution defined by $\sigma(X_1) = L(G_1)$, $\sigma(a) = a$ for $a \in V_T$. We have that $L(G) = \sigma(L(G_2))$, and $L(G)$ is regular due to Lemma 2.2.15.

Thus, we have that context-free languages that are not self-embedding are regular, and the other implication is trivial. ∎

Next, we show another method to represent the derivations of type-2 grammars using parse trees.

Definition 2.2.29 ([90]) *A rooted, ordered tree is called a parse tree for a type-2 grammar* $G = (V_N, V_T, X_0, F)$ *if and only if*

- *every internal node is labeled with a non-terminal letter,*
- *its root is labeled with the initial non-terminal letter* X_0*,*
- *every leaf node is labeled with a terminal letter or* λ*, and*
- *if there is a production rule* $X \to y_1 y_2 ... y_n \in F$*, every internal node labeled with a non-terminal letter X has child nodes labeled with* $y_1, y_2, ..., y_n$*, in this order, from its left-hand side to its right-hand side .*

We define a function $f\colon\ t \to V_T^*$, where t are roots of parse trees for type-2 grammars. For a parse tree with root t, we have that $f(t)$ reads terminal letters labeling leaf nodes of the parse tree t in preorder, where the parent node is visited first, then its children are visited in a left-to-right order [63]. Note that there may exist different derivations of a word by a type-2 grammar, and it follows that there may exist different parse trees of a word generated by a type-2 grammar. Moreover, for all words generated by a type-2 grammar, there is a minimum parse tree: a parse tree such that there are no other parse trees having fewer nodes.

Example The derivation $X_0 \Rightarrow aX_0b \Rightarrow aaX_0bb \Rightarrow aabb$ of the word $aabb$ by the type-2 grammar $G = \{\{X_0\}, \{a, b\}, X_0, \{X_0 \to aX_0b, X_0 \to \lambda\}\}$ can be represented by the parse tree in Figure 2.3.

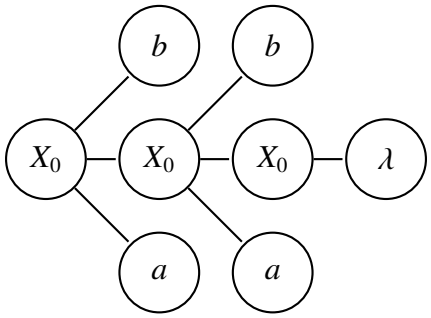

Figure 2.3: A parse tree that represents the derivation $X_0 \Rightarrow aX_0b \Rightarrow aaX_0bb \Rightarrow aabb$ of the word $aabb$ generated by a type-2 grammar for $\{a^n b^n \mid n \in \mathcal{N}\}$

Next, we have the following proposition for the special case where the size of the alphabet of a context-free language is 1.

Proposition 2.2.30 ([79]) *All context-free languages over an alphabet of size* 1 *are regular.*

Proof The branching factor b of a grammar is the largest number of letters in any right-hand side of the production rules of this grammar. For any word generated by a type-2 grammar, its derivation can be represented by a minimum parse tree. This parse tree may or may not contain a path that contains repeated non-terminal letters from the root to a leaf. Note that by paths, we mean the shortest path from the root of a parse tree to a leaf if not otherwise specified, and by parse trees, we mean minimum parse trees.

Consider a context-free grammar $G = (V_N, \{a\}, X_0, F)$ with branching factor b, which has an one-letter alphabet. For any parse tree for the grammar G with root t, if there are no paths that contain repeated non-terminal letters, the maximum length of $f(t)$ is $b^{|V_N|}$. We use T_0 to denote the set of all such parse trees, and we have that $L_0 = \bigcup_{t \in T_0} f(t)$ is a finite language.

The height of a node t' in a parse tree rooted at t is the number of edges visited in the path from t to t'. If there is at least one path that contains repeated non-terminal letters, an internal node t_1 labeled by X can be found in a parse tree such that it is the node with maximum height whose path to a leaf visits nodes labeled by other non-terminal letters at most once and by X exactly twice. Thus, another node t_2 labeled by X can be found in the subtree rooted at t_1. We call t_1 the upper repeated node of the tree rooted at t, and t_2 the lower repeated node of the tree rooted at t. Note that for any parse tree not in T_0, we can find its upper and lower repeated nodes.

Consider a parse tree not in T_0 rooted at t, which represents a derivation of the form $X_0 \Rightarrow^* \alpha_1 X \beta_1 \Rightarrow^* \alpha_1\alpha_2 X \beta_2\beta_1 \Rightarrow^* \alpha_1\alpha_2\alpha_3\beta_2\beta_1$ according to G, where $X \in V_N, \alpha_1, \alpha_2, \alpha_3, \beta_2, \beta_1 \in a^*$. Note that $\alpha_2\beta_2 \neq \lambda$; otherwise, t is not a minimum parse tree. Note also that $\alpha_1\alpha_2\alpha_3\beta_2\beta_1 \in L(G)$, and it follows that $\alpha_1\alpha_3\beta_1 \in L(G)$ because of the derivation $X_0 \Rightarrow^* \alpha_1 X \beta_1 \Rightarrow^* \alpha_1\alpha_3\beta_1$. This derivation can be represented by the parse tree rooted at t with the subtree rooted at its upper repeated node t_1 replaced by the subtree rooted at its lower repeated node t_2. Note that we can get a parse tree in T_0 from the tree not in T_0 by repeating this process.

Given a root t of a parse tree, we use $g(t)$ to denote the tree. Given the context-free grammar $G = (V_N, \{a\}, X_0, F)$, V_N can be denoted by $V_N = \{X_i \mid 1 \leq i \leq |V_N|, i \in \mathcal{N}\}$. Consider a subtree rooted at t labeled by X_i, where t is also its upper repeated node, and the node t' be its lower repeated node. This subtree represents a derivation $X_i \Rightarrow^* \alpha X_i \beta \Rightarrow^* \alpha\gamma\beta$, where $\alpha, \beta, \gamma \in a^*$. We have the language $L_i = \{\alpha\beta \mid X_i \Rightarrow^* \alpha X_i \beta \Rightarrow^* \alpha\gamma\beta$ is a derivation represented by a minimal subtree $\}$. We also have that $L_i = \{f(t)f(t')^{-1} \mid t$ is the root of a minimal subtree label by X_i, t is

the upper repeated node, t' is the lower repeated node}. Note that L_i, for $1 \leq i \leq |V_N|, i \in \mathcal{N}$, is finite.

Since the alphabet only contains a single letter, the order of the letters in the word does not matter, and it follows that

$$L(G) = \bigcup_{g(t) \in T_0} \left(f(t) \left(\bigcup_{X_i \in g(t)} L_i \right)^* \right),$$

which is regular. ∎

It follows that if we want to consider a language over an alphabet Σ that is context-free but not regular, we need to make sure that the size of the alphabet Σ is at least 2.

Definition 2.2.31 ([93]) *Two words α and β over an alphabet Σ are said to be letter-equivalent if and only if $N_a^\alpha = N_a^\beta$ for all $a \in \Sigma$.*

Definition 2.2.32 ([93]) *Two languages L_1 and L_2 over an alphabet Σ are said to be letter-equivalent if and only if for all words $\alpha \in L_1$, there exists a word $\beta \in L_2$ such that α and β are letter-equivalent, and vice versa.*

The following generalization of Proposition 2.2.30 can be proven using an approach similar to the proof of Proposition 2.2.30.

Proposition 2.2.33 ([80]) *Let Σ be an alphabet. All context-free languages over Σ have a letter-equivalent regular language over Σ.*

The family of context-free languages does not contain all the possible languages over a given alphabet because of the language $L = \{a^n b^n c^n \mid n \in \mathcal{N}\}$. The following lemma, called the pumping lemma for context-free languages, gives us a method for showing that a languages is not context-free.

Lemma 2.2.34 ([3]) *If L is a context-free language, then there exists a number $k \in \mathcal{N}^+$ such that for all words w whose length is at least k in the context-free language L, there exists a decomposition of $w = uvxyz$, where the length of the subword vxy is at most k, vy is not an empty word, and $uv^q xy^q z$ is a word in the context-free language L for any number $q \in \mathcal{N}$.*

Consider a context-free grammar $G = (V_N, V_T, X_0, F)$ and a word $\alpha \in L(G)$ long enough that the minimum parse tree of α contains a path that has at least two nodes labeled with the same non-terminal letter. The basic idea behind the pumping lemma for context-free languages is that removing or adding more copies of subwords, generated by a derivation between two repeated non-terminal letters, still results in a word in L.

Example The language $L = \{a^n b^n c^n \mid n \in \mathcal{N}\}$ is not context-free because the word $a^k b^k c^k$ cannot be pumped and still belongs to the language L.

A lemma called Ogden's lemma, which is more restrictive than Lemma 2.2.34, can also be used to prove that a language is not context-free [78].

2.2.3 The Family of Context-Sensitive Languages

In this section, we introduce the family of context-sensitive languages, which contains all context-free languages and the language $L = \{a^n b^n c^n \mid n \in \mathcal{N}\}$.

Definition 2.2.35 ([93]) *A grammar $G = (V_N, V_T, X_0, F)$ is called a type-1 grammar if and only if its production rules are of the form $\alpha_1 X \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, where $X \in V_N, \alpha_1, \alpha_2, \beta \in (V_N \cup V_T)^*, \beta \neq \lambda$. Moreover, if $X_0 \rightarrow \lambda$ is included in F, then X_0 does not occur in the right-hand side of any production rules in F.*

A language is said to be context-sensitive if and only if it can be generated by a type-1 grammar [13]. A type-1 grammar is also called a context-sensitive grammar. Note that all context-free grammars are type-1 grammars, so all context-free languages are context-sensitive. The family of context-sensitive languages contains exactly all the languages that can be generated by type-1 grammars.

Example The language $L = \{a^n b^n c^n \mid n \in \mathcal{N}\}$ can be generated by the type-1 grammar $G = (V_N, V_T, X_0, F)$, where:

$$
\begin{aligned}
V_N &= \{A, B, C, D, X, Y, X_0\}, \\
V_T &= \{a, b, c\}, \\
F &= \{X_0 \rightarrow \lambda, X_0 \rightarrow D, D \rightarrow ABC, D \rightarrow ADBC, CB \rightarrow CX\} \\
&\quad \cup \{CX \rightarrow YX, YX \rightarrow YC, YC \rightarrow BC, A \rightarrow a\} \\
&\quad \cup \{aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc\}.
\end{aligned}
$$

Thus, we have that L is context-sensitive.

Next, we show other ways to characterize context-sensitive languages. First, we show another type of grammar that generates context-sensitive languages.

Definition 2.2.36 ([14, 93]) *A grammar $G = \{V_N, V_T, X_0, F\}$ is said to be length-increasing if and only if all production rules $\alpha \rightarrow \beta \in F$ have the property that $|\alpha| \leq |\beta|$. Moreover, if $X_0 \rightarrow \lambda$ is included in F, then X_0 does not occur on the right-hand side of any production rule in F, and $X_0 \rightarrow \lambda$ is the only exception to the length restriction.*

With help of the following lemma, we can prove that a language can be generated by a length-increasing grammar if and only if it is context-sensitive.

Lemma 2.2.37 ([13]) *Given a type-1 (length-increasing) grammar $G = (V_N, V_T, X_0, F)$, a type-1 (length-increasing) grammar G' can be constructed such that all production rules containing terminal letters are of the form $X \to a$, where $X \in V_N, a \in V_T$, and G' generates exactly the language $L(G)$.*

Proof The idea of the proof is as follows.

This is achieved by constructing $G' = (V'_N, V_T, X_0, F')$ such that

$$V'_N = V_N \cup \{X_a \mid a \in V_T\},$$
$$F' = \{\sigma(\alpha) \to \sigma(\beta) \mid \alpha \to \beta \in F\} \cup \{X_a \to a \mid a \in V_T\},$$

where σ is a substitution defined by $\sigma(a) = X_a$ for $a \in V_T, \sigma(X) = X$ for $X \in V_N$. ∎

Proposition 2.2.38 ([93]) *A language can be generated by a length-increasing grammar if and only if it is context-sensitive.*

Proof The idea of the proof is as follows.

Clearly, all context-sensitive grammars are length-increasing by Definition 2.2.35. The other implication also holds by the following construction.

Given a length-increasing grammar G, where $\lambda \notin L(G)$, $G' = \{V_N, V_T, X_0, F\}$ can be constructed such that all production rules containing terminal letters are of the form $X \to a$, where $X \in V_N, a \in V_T$, according to Lemma 2.2.37. If F_0 is the set of production rules from F with all the production rules that do not follow the restrictions in Definition 2.2.35 removed, it follows that the grammar $G_0 = \{V_N, V_T, X_0, F_0\}$ is context sensitive.

We can put one production rule back using the following construction, where the resulting grammar remains context-sensitive. Let $\alpha \to \beta$ be a production rule in F, where $\alpha = X_1...X_m, \beta = Y_1...Y_n, 2 \leq m \leq n, \alpha, \beta \in V_N^+$. If we add this production rule to G_0, we have a length-increasing grammar $G'_0 = \{V_N, V_T, X_0, F_0 \cup \{\alpha \to \beta\}\}$, and a context-sensitive grammar $G_1 = \{V'_N, V_T, X_0, F_0 \cup F'_0\}$ that generates exactly the language $L(G'_0)$, where

$$V'_N = V_N \cup \{Z_i \mid 1 \leq i \leq m\},$$
$$F'_0 = \{X_1X_2...X_m \to Z_1X_2...X_m, Z_1X_2...X_m \to Z_1Z_2X_3...X_m, ..., Z_1Z_2...Z_{m-2}X_{m_1}X_m \to Z_1Z_2...Z_{m-1}X_m, Z_1Z_2...Z_{m-1}X_m \to Z_1Z_2...Z_mY_{m+1}Y_{m+2}...Y_n\}$$
$$\cup \{Z_1Z_2...Z_mY_{m+1}Y_{m+2}...Y_n \to Y_1Z_2...Z_mY_{m+1}Y_{m+2}...Y_n, Y_1Z_2...Z_mY_{m+1}Y_{m+2}...Y_n \to Y_1Y_2...Z_mY_{m+1}Y_{m+2}...Y_n, ..., Y_1Y_2...Y_{m-2}Z_{m_1}Z_mY_{m+1}Y_{m+2}...Y_n \to Y_1Y_2...Y_mY_{m+1}Y_{m+2}...Y_n\}.$$

We can repeat this process until all the removed production rules are replaced, and construct a context-sensitive grammar G_n that generates exactly the language $L(G)$. ∎

Next, we show another way to characterize context-sensitive languages by a recognition device called linear bounded automaton that recognizes context-sensitive languages [64].

Definition 2.2.39 ([93]) *A linear bounded automaton (LBA) is a sextuple $M = (S, V_I, V_T, s_0, A, F)$, where S is the set of states, $V_I \subseteq V_T$ is the input alphabet, V_T is the tape alphabet, $s_0 \in S$ is the initial state, $A \subseteq S$ is the set of final states, and $F \subseteq \{s_ia \to s_jb \mid s_i, s_j \in S, a, b \in V_T\} \cup \{s_ia \to as_j \mid s_i, s_j \in S, a \in V_T\} \cup \{cs_ia \to s_jca \mid s_i, s_j \in S, a, c \in V_T\}$ is the set of transitions. In addition, for each triple (s_i, a, s_j), where $s_i, s_j \in S, a \in V_T$, the set of transitions F either contains no rules of the form $cs_ia \to s_jca$ or contains all rules of the form $cs_ia \to s_jca$ for all $c \in V_T$.*

An LBA is basically an NFA with a tape of limited space, that is, the size of the tape is always the length of the input word. Initially, the reading position of an LBA is at the first letter of the input if the input is not λ, and its transitions are related to its movements on the tape or rewriting the tape, where

- a transition of the form $s_ia \to s_jb$ means that the LBA can rewrite the current position of tape from a to b and change its state from s_i to s_j,
- a transition of the form $s_ia \to as_j$ means that the LBA can move one step right on the tape and change its state from s_i to s_j, and
- a transition of the form $cs_ia \to s_jca$ means that the LBA can move one step left on the tape and change its state from s_i to s_j.

Note that the last sentence in Definition 2.2.39 means that an LBA may or may not move, and if it moves left, it can move left only based on the current state and the letter at the current position on the tape, independently of what the left context is.

Next, we define configurations and derivations of LBAs.

Let $M = (S, V_I, V_T, s_0, A, F)$ be an LBA. A configuration of the LBA M is a word of the form $\alpha s\beta$, where $s \in S, \alpha, \beta \in V_T^*$. The initial configuration of M is a word of the form $s_0\beta$, where $\beta \in V_I^*$ is the input word on the tape. An accepting configuration is a word of the form αs, where $\alpha \in V_T^*, s \in A$.

The cases where a configuration derives another configuration are summarized as follows:

- a configuration $\alpha s_ia\beta$ derives $\alpha s_jb\beta$ in one step, denoted by $\alpha s_ia\beta \Rightarrow \alpha s_jb\beta$, according to the transitions in F if and only if there is a transition $s_ia \to s_jb \in F$;

- a configuration $\alpha s_i a \beta$ derives $\alpha a s_j \beta$ in one step, denoted by $\alpha s_i a \beta \Rightarrow \alpha a s_j \beta$, according to the transitions in F if and only if there is a transition $s_i a \rightarrow a s_j \in F$;
- a configuration $\alpha c s_i \beta$ derives $\alpha s_j c \beta$ in one step, denoted by $\alpha c s_i \beta \Rightarrow \alpha s_i c \beta$, according to the transitions in F if and only if there is a transition $c s_i a \rightarrow s_j c a \in F$.

Consider a number $k \in \mathcal{N}$, states $s_{i_j} \in S$ for $0 \leq j \leq k, j \in \mathcal{N}$, and tape words $\alpha_j, \beta_j \in V_T^*$ for $0 \leq j \leq k, j \in \mathcal{N}$. A configuration $\alpha_0 s_{i_0} \beta_0$ derives another configuration $\alpha_k s_{i_k} \beta_k$, denoted by $\alpha_0 s_{i_0} \beta_0 \Rightarrow^* \alpha_k s_{i_k} \beta_k$, according to the transitions in F if and only if $\alpha_0 s_{i_0} \beta_0 \Rightarrow \alpha_1 s_{i_1} \beta_1 \Rightarrow ... \Rightarrow \alpha_{k-1} s_{i_{k-1}} \beta_{k-1} \Rightarrow \alpha_k s_{i_k} \beta_k$ according to the transitions in F. Note that if $k = 0$, no derivations happen, and this indicates a derivation of length 0.

A word $\beta \in V_I^*$ is said to be recognized by an LBA $M = (S, V_I, V_T, s_0, A, F)$ if and only if $s_0 \beta \Rightarrow^* \alpha s$, where $\alpha \in V_T^*, s \in A$. The language recognized by the LBA M, denoted by $L(M)$, is exactly the set of all the words that are recognized by the LBA M. Note that an LBA M recognizes λ if and only if $s_0 \in A$.

Any context-sensitive grammar G can be converted to an LBA M that recognizes $L(G)$ [64], and any LBA M can be converted to a type-1 grammar G that generates $L(M)$ [66].

It follows that a language can be generated by type-1 grammars if and only if it can be recognized by LBAs, and we also have that a language is context-sensitive if and only if it can be recognized by LBAs.

Example The language $L = \{[a^n b^n c^n] \mid n \in \mathcal{N}\}$ can be recognized by the LBA $M = (S, V_I, V_T, s_0, A, F)$, where:

$$
\begin{aligned}
S &= \{s_i \mid i \in \mathcal{N}, 0 \leq i \leq 7\}, \\
V_I &= \{a, b, c, [,]\}, \\
V_T &= \{a, b, c, d, [,]\}, \\
A &= \{s_7\}, \\
F &= \{s_0[\rightarrow [s_1\} \\
&\quad \cup \{s_1 d \rightarrow d s_1, s_1 a \rightarrow s_2 d, s_2 d \rightarrow d s_3, s_3 a \rightarrow a s_3\} \\
&\quad \cup \{s_3 d \rightarrow d s_3, s_3 b \rightarrow s_4 d, s_4 d \rightarrow d s_5, s_5 b \rightarrow b s_5\} \\
&\quad \cup \{s_5 d \rightarrow d s_5, s_5 c \rightarrow s_6 d\} \\
&\quad \cup \{e s_6 f \rightarrow s_6 e f \mid e, f \in V_T\} \\
&\quad \cup \{s_6[\rightarrow s_0[, s_1] \rightarrow] s_7\}.
\end{aligned}
$$

This LBA can reach its final state if the input is of the form $[a^* b^* c^*]$, and the numbers of occurrences of a, b, c are same. Since this language L can be recognized by this LBA, it follows that the language L is context-sensitive.

We have now introduced type-1 grammars, length-increasing grammars and LBAs, and we have the following proposition.

Proposition 2.2.40 *The following statements about a language L are equivalent.*

- *L is context-sensitive.*
- *There exists a type-1 grammar that generates L.*
- *There exists a length-increasing grammar that generates L.*
- *There exists an LBA that recognizes L.*

The family of context-sensitive languages does not contain all the possible languages over a given alphabet V_T, and this can be proven by diagonalization. Consider a method to encode context-sensitive grammars $G = (V_N, V_T, X_0, F)$ and words over the alphabet V_T. We can enumerate words $\alpha_1, \alpha_2, ...$ over the alphabet V_T in lexicographical order and the encodings of context-sensitive grammars $G_1, G_2, ...$ in lexicographical order. Consider a language $L = \{\alpha_i \mid \alpha_i \notin L(G_i), i \geq 1, i \in \mathcal{N}\}$. Since L is at least one word different from any context-sensitive language, it follows that L is not context-sensitive.

2.2.4 The Family of Recursively Enumerable Languages

In this section, we introduce the family of recursively enumerable languages, which contains all languages that can be generated by grammars.

Definition 2.2.41 ([93]) *A grammar $G = (V_N, V_T, X_0, F)$ is called a type-0 grammar if and only if there are no restrictions on the production rules in F.*

A language is said to be recursively enumerable if and only if it can be generated by a type-0 grammar [13]. A type-0 grammar is also called a recursively enumerable grammar. The family of recursively enumerable languages contains exactly all the languages that can be generated by type-0 grammars.

Next, we show another way to describe recursively enumerable languages: a recognition device called Turing machine that can recognize recursively enumerable languages [93, 101].

Definition 2.2.42 ([93]) *A Turing machine (TM) is a septuple $M = (S, V_I, V_1, V_T, s_0, A, F)$, where S is the set of states, V_T is the tape alphabet, $\# \in V_T$ is the tape boundary marker, $V_1 = V_T - \{\#\}$ is a non-empty alphabet, $V_I \subset V_1$ is the input alphabet, $s_0 \in S$ is the initial state, $A \subseteq S$ is the set of final states, $\square \in V_1$ denotes an empty cell on the tape, and $F \subseteq \{s_i a \rightarrow s_j b \mid$*

$s_i, s_j \in S, a, b \in V_1\} \cup \{s_iac \to as_jc \mid s_i, s_j \in S, a, c \in V_1\} \cup \{s_ia\# \to as_j\square\# \mid s_i, s_j \in S, a \in V_1\} \cup \{cs_ia \to s_jca \mid s_i, s_j \in S, a, c \in V_1\} \cup \{\#s_ia \to \#s_j\square a \mid s_i, s_j \in S, a \in V_1\}$ *is the set of transitions.*

In addition, for each triple (s_i, a, s_j)*, where* $s_i, s_j \in S, a \in V_1$*, the set of transitions* F *either contains no rules of the form* $s_iac \to as_jc$ *and* $s_ia\# \to as_j\square\#$ *(*$cs_ia \to s_jca$ *and* $\#s_ia \to \#s_j\square a$*) or contains all rules of the form* $s_iac \to as_jc$ *and* $s_ia\# \to as_j\square\#$ *(*$cs_ia \to s_jca$ *and* $\#s_ia \to \#s_j\square a$*) for all* $c \in V_1$*.*

Moreover, for each pair (s_i, a)*, where* $s_i \in S, a \in V_1$*, there is at most one transition of the form* $s_ia\# \to as_j\square\#$*,* $\#s_ia \to \#s_j\square a$ *and* $s_ia \to s_jb$ *in* F*, where* $s_j \in S, b \in V_1$*.*

Note that a TM is basically a *DFA* with a tape of infinite space, so we need a special marker # to mark the boundaries of the part of the tape that we use. Moreover, its transitions are related to its movements on the tape or rewriting the tape, where:

- a transition of the form $s_ia \to s_jb$ means that the TM can rewrite the current position of tape from a to b and change its state from s_i to s_j,
- a transition of the form $s_iac \to as_jc$ means that the TM can move one step right on the tape and change its state from s_i to s_j,
- a transition of the form $s_ia\# \to as_j\square\#$ means that the TM can move one step right on the tape, extend its workspace, and change its state from s_i to s_j,
- a transition of the form $cs_ia \to s_jca$ means that the TM can move one step left on the tape and change its state from s_i to s_j, and
- a transition of the form $\#s_ia \to \#s_j\square a$ means that the TM can move one step left on the tape, extend its workspace, and change its state from s_i to s_j.

Note that either a TM cannot move left (resp. right), or it can move left (resp. right), depending on the current letter read from the tape and the current state, independent of what the left (resp. right) context is. In addition, there is at most one transition that can be applied in each step.

Next, we define configurations and derivations of TMs.

Let $M = (S, V_I, V_1, V_T, s_0, A, F)$ be a TM. A configuration of the TM M is a word of the form $\#\alpha s\beta\#$, where $s \in S, \alpha, \beta \in V_1^*$. The initial configuration of M is a word of the form $\#s_0\beta\#$, where $\beta \in V_I^*$ is the input word on the tape. An accepting configuration is a word of the form $\#\alpha s\beta\#$, where $s \in A$ is an accepting state, and no more transitions can be applied to $\#\alpha s\beta\#$.

The cases where a configuration derives another configuration are summarized as follows:

- a configuration $\#\alpha s_i a\beta\#$ derives $\#\alpha s_j b\beta\#$ in one step, denoted by $\#\alpha s_i a\beta\# \Rightarrow \#\alpha s_j b\beta\#$, according to the transitions in F if and only if there is a transition $s_i a \rightarrow s_j b \in F$;
- a configuration $\#\alpha s_i a\beta\#$ derives $\#\alpha a s_j \beta\#$ in one step, denoted by $\#\alpha s a\beta\# \Rightarrow \#\alpha a s_j \beta\#$, according to the transitions in F if and only if there is a transition $s_i a \rightarrow a s_j \in F$;
- a configuration $\#\alpha s_i a\#$ derives $\#\alpha a s_j \square\#$ in one step, denoted by $\#\alpha s_i a\# \Rightarrow \#\alpha a s_j \square\#$, according to the transitions in F if and only if there is a transition $s_i a\# \rightarrow a s_j \square\# \in F$;
- a configuration $\#\alpha c s_i a\beta\#$ derives $\#\alpha s_j c a\beta\#$ in one step, denoted by $\#\alpha c s_i a\beta\# \Rightarrow \#\alpha s_j c a\beta$ #, according to the transitions in F if and only if there is a transition $c s_i a \rightarrow s_j c a \in F$;
- a configuration $\#s_i a\beta\#$ derives $\#\square s_j a\beta\#$ in one step, denoted by $\#s_i a\beta\# \Rightarrow \#\square s_j a\beta\#$, according to the transitions in F if and only if there is a transition $\#s_i a \rightarrow \#s_j \square a \in F$.

Consider a number $k \in \mathcal{N}$, states $s_{i_j} \in S, 0 \leq j \leq k, j \in \mathcal{N}$, and tape words $\alpha_j, \beta_j \in V_1^*, 0 \leq j \leq k, j \in \mathcal{N}$. A configuration $\#\alpha_0 s_{i_0} \beta_0\#$ derives another configuration $\#\alpha_k s_{i_k} \beta_k\#$, denoted by $\#\alpha_0 s_{i_0} \beta_0\# \Rightarrow^* \#\alpha_k s_{i_k} \beta_k\#$, according to the transitions in F if and only if $\#\alpha_0 s_{i_0} \beta_0\# \Rightarrow \#\alpha_1 s_{i_1} \beta_1\# \Rightarrow ... \Rightarrow \#\alpha_{k-1} s_{i_{k-1}} \beta_{k-1}\# \Rightarrow \#\alpha_k s_{i_k} \beta_k\#$ according to the transitions in F.

A word β over the input alphabet V_I is said to be recognized by a TM $M = (S, V_I, V_1, V_T, s_0, A, F)$ if and only if $s_0\beta \Rightarrow^* \alpha s\beta'$, where $\alpha s\beta'$ is an accepting configuration. The language recognized by the TM, denoted by $L(M)$, is the set of all the words that are recognized by the TM M.

Next, we show that a language can be generated by type-0 grammars if and only if it can be recognized by TMs.

Given a type-0 grammar $G = (V_N, V_T, X_0, F)$, a TM can be constructed to rewrite the initial tape of $\#X_0\#$ according to production rules in F. Thus, if a language can be generated by a type-0 grammar, it can be recognized by TMs. Next, we show the other implication.

Given a TM $M = (S, V_I, V_1, V_T, s_0, A, F)$, a type-0 grammar G that generates exactly the language $L(M)$ can be constructed. This is achieved by constructing $G = (V_N, V_I, X_0, F')$, where:

$$
\begin{aligned}
V_N &= S \cup (V_T - V_I) \cup \{X_0, X_1, X_2\}, \\
F' &= \{\alpha \rightarrow \beta \mid \beta \rightarrow \alpha \in F\} \cup \{X_0 \rightarrow \#X_2, X_2 \rightarrow X_1\#, \# \rightarrow \lambda, \#s_0 \rightarrow \lambda\} \\
&\quad \cup \{X_2 \rightarrow bX_2 \mid b \in V_1\} \cup \{X_2 \rightarrow s_1\# \mid s_1 \in A\} \cup \{X_1 \rightarrow X_1 b \mid b \in V_1\} \\
&\quad \cup \{X_1 \rightarrow s_1 a \mid a \in V_1, s_1 \in A, s_1 a \Rightarrow \emptyset \text{ according to transitions in } F\}.
\end{aligned}
$$

Thus, we have that if a language can be recognized by TMs, it can be generated by type-0 grammars, and it follows that a language can be recognized by TMs if and only if it is recursively enumerable. To summarize, we have the following proposition.

Proposition 2.2.43 *The following statements about a language L are equivalent.*

- *L is recursively enumerable.*
- *There exists a type-0 grammar that generates L.*
- *There exists a TM that recognizes L.*

Next, we show that by adding some restrictions to type-0 grammars G, the languages $L(G)$ they generate are context-sensitive.

Definition 2.2.44 ([93]) *Consider a derivation $D : X_0 = \alpha_0 \Rightarrow \alpha_1 \Rightarrow ... \Rightarrow \alpha_n = \alpha$ according to a grammar $G = (V_N, V_T, X_0, F)$. The workspace of α of the derivation D is defined by $WS_G(P, D) = \max\{|\alpha_i| \mid 0 \leq i \leq n\}$. The workspace of $\alpha \in L(G)$ is defined by $WS_G(P) = \min\{WS_G(\alpha, D) \mid D$ is a derivation of α according to the grammar $G\}$.*

Proposition 2.2.45 ([52]) *Given a type-0 grammar $G = (V_N, V_T, X_0, F)$, there exists a number $p \in \mathcal{N}$ such that $WS_G(\alpha) \leq p|\alpha|$ for all $\alpha \in L(G)$, then $L(G)$ is context-sensitive.*

The following two corollaries follow Proposition 2.2.38, where languages generated by length-increasing grammars are context-sensitive.

Corollary 2.2.46 *Let L be a recursively enumerable language over an alphabet Σ, $a, b \notin \Sigma$ be two letters, and σ be the substitution defined by $\sigma(a) = \sigma(b) = \lambda$ and $\sigma(c) = c$ for $c \in \Sigma$. A context-sensitive language $L' \subseteq a^*bL$ can be constructed, and $L = \sigma(L')$.*

Proof Given a type-0 grammar $G = (V_N, V_T, X_0, F)$ be a type-0 grammar. This is achieved by constructing $G = (V'_N, V'_T, X'_0, F')$, where:

$$\begin{aligned}
V'_N &= V_N \cup \{X'_0, Y\},\\
V'_T &= V_T \cup \{a, b\},\\
F' &= \{X'_0 \to bX_0, Yb \to ab\}\\
&\quad \cup \{\alpha \to Y^{|\alpha|-|\beta|}\beta \mid \alpha \to \beta \in F, |\alpha| > |\beta|\}\\
&\quad \cup \{\alpha \to \beta \in F \mid |\alpha| \leq |\beta|\}\\
&\quad \cup \{cY \to Yc \mid c \in V_N \cup V_T \cup \{b\}\}.
\end{aligned}$$

Since G is a length-increasing grammar, the language $L(G)$ is context-sensitive according to Proposition 2.2.38. It is easy to see that $L(G)$ becomes L if all the letters a, b are removed. ∎

Corollary 2.2.47 *Let L be a recursively enumerable language over an alphabet Σ, $a, b \notin \Sigma$ be two letters, and σ be the substitution defined by $\sigma(a) = \sigma(b) = \lambda$ and $\sigma(c) = c$ for $c \in \Sigma$. A context-sensitive language $L' \subseteq Lba^*$ can be constructed, and $L = \sigma(L')$.*

2.2.5 Summary

Now, we have presented a brief review of the families of languages in the Chomsky hierarchy. The family of regular (resp. context-free, context-sensitive, recursively enumerable) languages can be denoted by $\mathcal{L}_3$ (resp. $\mathcal{L}_2, \mathcal{L}_1, \mathcal{L}_0$). Because of the restrictions of the grammars that generate languages in these families, we have that $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$. Notice that these inclusions are strict, as shown in the previous subsections. Also, we can also use FIN, REG, CF, CS, RE to denote the family of finite, regular, context-free, context-sensitive, and recursively enumerable languages respectively. Moreover, the family of languages that can be generated by linear grammars is denoted by LIN. The families of languages in the Chomsky hierarchy are also called the Chomsky families of languages.

2.3 Decidability

In this section, we discuss the decidability of problems related to the families of languages defined above. We first introduce some common problems about languages, including the:

- Membership problem($\alpha \in L$?): Given a language L, is the word α in the language L?
- Emptiness problem($L = \emptyset$?): Given a language L, is it empty?
- Finiteness problem($|L| = \infty$?): Given a language L, does it contain infinite many words?
- Totality problem($L = \Sigma^*$?): Given a language L over the alphabet Σ, does it contain all the possible words?
- Equality problem($L_1 = L_2$?): Given two languages L_1, L_2, are they the same?

Definition 2.3.1 *A problem is decidable if and only if, for any instance of the problem, there is an algorithm that outputs "yes" or "no".*

Example The membership problem for regular languages recognized by DFAs is decidable, because for any DFA M and any word α, Algorithm 1 outputs "yes" if $\alpha \in L(M)$ and "no" otherwise.

The decidability of problems about regular and context-free languages are summarized in Table 2.1 [90].

If a problem is decidable, an algorithm exists. The algorithm can be used as a subroutine when designing new algorithms to prove the decidability of other problems. Thus, we can use proof by contradiction to prove the undecidability of problems.

Input: A *DFA* M and a word α
Output: "Yes" if $\alpha \in L(M)$ and "no" otherwise
Run M on input word α;
if *M results into an accepting configuration* **then**
    Return "Yes";
else
    Return "No";
end

Algorithm 1: An algorithm that decides the membership problem for languages recognized by *DFA*s M, and words α

	$\alpha \in L?$	$L = \emptyset?$	$\lvert L\rvert = \infty?$	$L = \Sigma^*?$	$L_1 = L_2?$
$\mathcal{L}_3$	Yes	Yes	Yes	Yes	Yes
$\mathcal{L}_2$	Yes	Yes	Yes	No	Yes

Table 2.1: Summary of decidability of problems about regular and context-free languages

Example The problem "Is a given context-free language L regular?" is undecidable. If we assume that we could decide if a context-free language is regular, then Algorithm 2 would be able to decide the totality problem for context-free languages, which is a contradiction.

Input: A context-free language L over an alphabet Σ
Output: "Yes" if $L = \Sigma^*$ and "no" otherwise
if *L is a regular language* **then**
    if $L = \Sigma^*$ **then**
        Return "Yes";
    else
        Return "No";
    end
else
    Return "No";
end

Algorithm 2: A putative algorithm that decides whether a context-free language L is $L = \Sigma^*$

Chapter 3

Word Operations

Word and language operations have been extensively studied in formal language theory, and some commonly used operations are surveyed in this section. In Section 3.1, some classical operations originating from set, linguistics and rewriting devices are surveyed, and in Section 3.2, variations of inserting words into words and deleting words from words are studied. We show their definitions and the closure properties of $\mathcal{L}_i$, $0 \leq i \leq 3$, under these operations. Such operations include union, concatenation, concatenation closure, substitution, complementation, intersection, and generalized sequential machine mapping.

Definition 3.0.1 *A languages L is said to be closed under an operation $\diamond$ if and only if application of the operation $\diamond$ on words in the language L always produces a word in the language L.*

Example The language $L = a^*$ is closed under concatenation.

Definition 3.0.2 *A family of languages $\mathcal{L}$ is said to be closed under an operation $\diamond$ if and only if application of the operation $\diamond$ on languages in the family $\mathcal{L}$ always produces a language in the family $\mathcal{L}$.*

Example For any language $L \in \mathcal{L}_3$, $\mathrm{mi}(L)$ is also regular due to Corollary 2.2.7, so $\mathcal{L}_3$ is closed under mirror image.

Let L, R be known languages, X, Y be unknown languages, and $\diamond$ be a binary word operation. Given equations of the form $X \diamond L = R$ or $L \diamond Y = R$, we want to know if there exists a solution and how to construct the solutions to these equations. These problems are studied with the help of the left and right inverse of binary word operations defined in Section 3.3.

Let $\mathcal{L}$ be a family of languages that is closed under some operations. We can prove that $\mathcal{L}$ is also closed under a new operation by showing that the new operation can be represented

as a composition of those old operations. This idea raises the study of the abstract families of languages, and it is introduced in Section 3.4.

Let $\diamond$ be an operation under which $\mathcal{L}_3$ is closed. For any regular language L_1, L_2, we can construct a DFA that recognizes the language generated by $L_1 \diamond L_2$, where $\diamond$ is a binary operation, and by $\diamond(L_1)$, where $\diamond$ is a unary operation. In Section 3.5, we study a type of descriptional complexity, called state complexity, of such operations $\diamond$ on regular languages.

Recall that by Lemma 2.2.37, we can assume that for type-1 (length-increasing) grammars $G = (V_N, V_T, X_0, F)$, all production rules that contain terminal letters $a \in V_T$ are of the form $X \to a$, where $X \in V_N$. This can be extended to context-free and recursively enumerable grammars using the same construction, and we have the following Lemma.

Lemma 3.0.3 *For a language L generated by a context-free (resp. length-increasing, context-sensitive, and recursively enumerable) grammar, there exists a context-free (resp. length-increasing, context-sensitive, and recursively enumerable) grammar $G = (V_N, V_T, X_0, F)$ that generates the language L, and all the production rules in F containing terminal letters are of the form $X \to a$, where $X \in V_N, a \in V_T$.*

3.1 Classical Operations

In this section, some classical word and language operations are defined, and closure properties of the Chomsky families of languages under these operations are summarized. The word and language operations covered in this section are:

- some operations which are defined to manipulate words, such as concatenation, concatenation closure, λ−free concatenation closure, mirror image, substitution, regular substitution, λ−free substitution, λ−free regular substitution, homomorphism, λ−free homomorphism, linear erasing, restricted homomorphism, inverse homomorphism, left quotient, left derivative, right quotient, right derivative, (proper) prefix, (proper) suffix, and (proper) infix;

- some operations which are associated with a variation of NFA, such as generalized sequential machine mapping and λ−free generalized sequential machine mapping;

- some set operations which can be applied to languages naturally because languages are sets of words, such as union, complementation, intersection, and set difference.

Union [93] Union is an operation defined on sets and can thus be applied to languages. As shown in Definition 2.2.10, given two languages L_1 and L_2, their union is defined by $L_1 \cup L_2 = \{\alpha \mid \alpha \in L_1 \text{ or } \alpha \in L_2\}$. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under union.

Indeed, given two grammars $G_1 = (V_{N_1}, V_{T_1}, X_{0_1}, F_1)$ and $G_2 = (V_{N_2}, V_{T_2}, X_{0_2}, F_2)$, where $L(G_1), L(G_2) \in \mathcal{L}_i, 0 \leq i \leq 3$, there exists a grammar G that generates exactly $L(G_1) \cup L(G_2)$, and $L(G) \in \mathcal{L}_i$. This grammar is $G = (V_N, V_T, X_0, F)$, where:

$$\begin{aligned} V_N &= V_{N_1} \cup V_{N_2} \cup \{X_0\}, \\ V_T &= V_{T_1} \cup V_{T_2}, \\ F &= F_1 \cup F_2 \cup \{X_0 \to X_{0_1}, X_0 \to X_{0_2}\}. \end{aligned}$$

Note that for the context-sensitive case, if $L(G_1)$ is not λ-free, the grammar G is not a type-1 grammar because of $X_{0_1} \to \lambda, X_0 \to X_{0_1}$ which violates the restriction in Definition 2.2.35. Thus, if $L(G_1)$ or $L(G_2)$ is not λ-free, there exists a type-1 grammar G that generates exactly $L(G_1) \cup L(G_2)$. This grammar is $G = (V_N, V_T, X_0, F)$, where:

$$\begin{aligned} V_N &= V_{N_1} \cup V_{N_2} \cup \{X_0\}, \\ V_T &= V_{T_1} \cup V_{T_2}, \\ F &= F_1 \backslash \{X_{0_1} \to \lambda\} \\ &\quad \cup F_2 \backslash \{X_{0_2} \to \lambda\} \\ &\quad \cup \{X_0 \to X_{0_1}, X_0 \to X_{0_2}, X_0 \to \lambda\}. \end{aligned}$$

Concatenation [93] Consider two words α and β over an alphabet Σ. Their concatenation is denoted by $\alpha\beta$, which is also a word over Σ, and this definition can be extended to languages, where $L_1L_2 = \{\alpha\beta \mid \alpha \in L_1, \beta \in L_2\}$ as shown in Definition 2.2.11. A semigroup is a set endowed with an associative operation, and the semigroup becomes a monoid if the set has an identity element for this operation. Thus, the set of words over an alphabet Σ with the concatenation operation is a monoid because concatenation is associative $(\alpha\beta)\gamma = \alpha(\beta\gamma)$ and because $\lambda \in \Sigma^*$ is the identity for this operation $\lambda\alpha = \alpha = \alpha\lambda$. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under concatenation, as outlined next.

For $\mathcal{L}_3$, this can be trivially proven by the definition of regular expressions. Also, it can be proven by construction. Given two DFAs $M_1 = (S_1, V_T, s_{0_1}, A_1, F_1)$ and $M_2 = (S_2, V_T, s_{0_2}, A_2, F_2)$, there exists an NFA M that M recognizes $L(M_1)L(M_2)$. This NFA is $M = (S, V_T, s_{0_1}, A_2, F)$, where:

$$\begin{aligned} S &= S_1 \cup S_2, \\ F &= F_1 \cup F_2 \cup \{s \to s_{0_2} \mid s \in A_1\}. \end{aligned}$$

Consider two grammars $G_1 = (V_{N_1}, V_{T_1}, X_{0_1}, F_1)$ and $G_2 = (V_{N_2}, V_{T_2}, X_{0_2}, F_2)$, where $L(G_1)$, $L(G_2) \in \mathcal{L}_i, 0 \leq i \leq 2$, there exists a grammar G that generates exactly $L(G_1)L(G_2)$, where $L(G) \in \mathcal{L}_i$. This grammar is $G = (V_N, V_T, Y_0, F)$, where:

$$\begin{aligned} V_N &= V_{N_1} \cup V_{N_2} \cup \{Y_0\}, \\ V_T &= V_{T_1} \cup V_{T_2}, \\ F &= F_1 \cup F_2 \cup \{Y_0 \rightarrow X_{0_1}X_{0_2}\}. \end{aligned}$$

Note that for the context-sensitive case, if $L(G_1)$ is not λ-free, the grammar G is not a type-1 grammar because of $X_{0_1} \rightarrow \lambda, X_0 \rightarrow X_{0_1}$ which violates the restriction in Definition 2.2.35. Thus, we use the following construction to avoid applying concatenation on context-sensitive languages that are not λ-free.

- If L_1 is not $\lambda-$free and L_2 is $\lambda-$free, $L_1L_2 = (L_1\backslash\{\lambda\})L_2 \cup L_2$.
- If L_1 is $\lambda-$free and L_2 is not $\lambda-$free, $L_1L_2 = L_1(L_2\backslash\{\lambda\}) \cup L_1$.
- If L_1, L_2 are not $\lambda-$free, $L_1L_2 = (L_1\backslash\{\lambda\})(L_2\backslash\{\lambda\}) \cup (L_1\backslash\{\lambda\}) \cup (L_2\backslash\{\lambda\}) \cup \{\lambda\}$.

Note that given a context-sensitive language L, which is generated by the type-1 grammar $G = (V_N, V_T, X_0, F)$ and is not λ-free, the context-sensitive grammar $G' = (V_N, V_T, X_0, F\backslash\{X_0 \rightarrow \lambda\})$ generates the $\lambda-$free language $L(G)\backslash\{\lambda\}$. Since $\mathcal{L}_1$ is closed under union, it follows that it is closed under concatenation.

Concatenation Closure [93] As shown in Section 2.1, the concatenation closure of a word α is defined by $\alpha^* = \bigcup_{i\geq 0} \alpha^i$, where $\alpha^0 = \lambda$. As shown in Definition 2.2.12, the concatenation closure L^* of a language L can be defined recursively as $L^* = \{\lambda\} \cup \{\alpha \mid \alpha \in L\} \cup \{\alpha\beta \mid \alpha, \beta \in L^*\}$.

If we denote the set $\{\alpha_1\alpha_2...\alpha_i \mid \alpha_1, \alpha_2, ..., \alpha_i \in L, i \in \mathcal{N}^+\}$ by L^i, and $L^0 = \{\lambda\}$, then the concatenation closure L^* of a language L can also be defined by $L^* = \bigcup_{i=0}^{\infty} L^i$. The concatenation closure of a language is also called the Kleene star or the iteration of a language. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under concatenation closure.

For $\mathcal{L}_3$, this can be trivially proven by the definition of regular expressions. Also, it can be proven by construction. Given a regular grammar $G = (V_N, V_T, X_0, F)$, where all the production rules in F have at most one terminal letter due to Lemma 2.2.3, a grammar $G' = (V_N, V_T, X_0, F \cup \{X \rightarrow X_0a \mid X \rightarrow a \in F\} \cup \{X \rightarrow X_0 \mid X \rightarrow \lambda \in F\} \cup \{X_0 \rightarrow \lambda\})$ can be constructed such that G' generates exactly $L(G)^*$.

Given a context-free grammar $G = (V_N, V_T, X_0, F)$, a context-free grammar $G' = (V_N \cup \{Y_0\}, V_T, Y_0, F \cup \{Y_0 \rightarrow \lambda, Y_0 \rightarrow Y_0X_0\})$ can be constructed such that G' generates exactly $L(G)^*$.

To prove that $\mathcal{L}_0, \mathcal{L}_1$ are closed under concatenation closure using a similar approach to that of $\mathcal{L}_2$, the boundaries between occurrences of the language L in L^i need to be marked; otherwise, a production rule may be applied with its left-hand side coming from different occurrences of the language L due to the same terminal alphabet V_T and non-terminal alphabet V_N.

Example Consider the type-0 grammar $G = (\{a, b\}, \{X_0, X_1\}, X_0, \{X_0 \to X_1X_1X_1, X_1 \to a, X_1X_1 \to b\})$ that generates the language $L(G) = \{aaa, ab, ba\}$. Its concatenation closure is $L(G)^* = \{aaa, ab, ba\}^*$. If we use a similar construction to that of $\mathcal{L}_2$, the grammar $G' = (\{X_0, X_1, Y_0\}, \{a, b\}, Y_0, \{X_0 \to X_1X_1X_1, X_1 \to a, X_1X_1 \to b, Y_0 \to \lambda, Y_0 \to Y_0X_0\})$ can be constructed. However, G' generates the word $bbb \notin L(G)^*$ by the derivation $Y_0 \Rightarrow Y_0X_0 \Rightarrow Y_0X_0X_0 \Rightarrow X_0X_0 \Rightarrow X_0X_1X_1X_1 \Rightarrow X_1X_1X_1X_1X_1X_1 \Rightarrow bX_1X_1X_1X_1 \Rightarrow bbX_1X_1 \Rightarrow bbb$. Thus, this construction cannot be used without modification.

We will employ instead the following construction. For a context-sensitive (resp. recursively enumerable) grammar G, regardless of whether $L(G)$ is $\lambda-$free or not, the concatenation closure $L(G)^*$ always contains the empty word λ, so $L(G)^* = L(G')^*$, where $L(G') = L(G)\backslash\{\lambda\}$.

Given an arbitrary context-sensitive grammar $G = (V_N, V_T, X_0, F)$, the context-sensitive grammar $G' = (V_N, V_T, X_0, F\backslash\{X_0 \to \lambda\})$ generates the language $L(G)\backslash\{\lambda\}$. Given an arbitrary recursively enumerable grammar $G = (V_N, V_T, X_0, F)$, where all the production rules in F containing terminal letters are of the form $X \to a$ due to Lemma 3.0.3, there exists a recursively enumerable grammar $G' = (V_N, V_T, X_0, F')$ generates the language $L(G)\backslash\{\lambda\}$, and this is achieved by the construction: $F' = (F\backslash\{X \to \lambda \in F\}) \cup \{XY \to Y, YX \to Y \mid X \to \lambda \in F, Y \in V_N\}$.

Consider a context-sensitive (resp. recursively enumerable) language L, and a context-sensitive (resp. recursively enumerable) grammar $G = (V_N, V_T, X_0, F)$ that generates the language $L\backslash\{\lambda\}$, where all the production rules in F containing terminal letters are of the form $X \to a$ due to Lemma 3.0.3. We can now construct the context-sensitive (resp. recursively enumerable) grammar G' that generates exactly $L(G)^*$. This is achieved by constructing $G' = (V'_N, V_T, Y_0, F')$, where:

$$\begin{aligned} V'_N &= V_N \cup \{Y_0, Y_1\}, \\ F' &= F \cup \{Y_0 \to \lambda, Y_0 \to X_0, Y_0 \to Y_1X_0\} \\ &\quad \cup \{Y_1a \to Y_1X_0a \mid a \in V_T\} \\ &\quad \cup \{Y_1a \to X_0a \mid a \in V_T\}. \end{aligned}$$

In this construction, the non-terminal letter Y_1 makes sure that the letter immediately beside

X_0, on the right, is a terminal letter, so non-terminal letters from different derivations of L do not interfere with each other.

λ-Free Concatenation Closure [93] The λ-free concatenation closure of a word α is denoted by $\alpha^+ = \{\alpha^i \mid i \in \mathcal{N}^+\}$. The λ-free concatenation closure of a language L^+ can be defined recursively as $L^+ = \{\alpha \mid \alpha \in L\} \cup \{\alpha\beta \mid \alpha, \beta \in L^+\}$.

Alternatively, $L^+ = \bigcup_{i=1}^{\infty} L^i$. If a language L is not $\lambda-$free, $L^+ = L^*$; otherwise, $L^+ = L^* \backslash \{\lambda\}$. The λ-free concatenation closure of a language is also called the Kleene plus of L. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under λ-free concatenation closure.

Given a regular expression R, the regular expression $(R)^*(R)$ denotes the regular language $L(R)^+$. If the language L generated by a context-free (resp. context-sensitive, recursively enumerable) grammar G is $\lambda-$ free, a similar grammar G' to the grammar constructed for concatenation closure, but without the production rule $Y_0 \rightarrow \lambda$, generates exactly the $\lambda-$ free context-free (resp. context-sensitive, recursively enumerable) language L^+; otherwise, the same construction to that of concatenation closure can be used.

Mirror Image [93] Let Σ be an alphabet. As shown in Definition 2.2.5, the mirror image of a word $\alpha = a_1 a_2 ... a_k$, where $a_i \in \Sigma, 1 \leq i \leq k$, is defined by $\text{mi}(\alpha) = a_k a_{k-1} ... a_2 a_1, \text{mi}(\lambda) = \lambda$, and the mirror image of a language L is defined by $\text{mi}(L) = \{\text{mi}(\alpha) \mid \alpha \in L\}$. Note that $\alpha = \text{mi}(\text{mi}(\alpha)), L = \text{mi}(\text{mi}(L))$, where $\alpha \in \Sigma^*, L \subseteq \Sigma^*$. A word α or a language L is said to be palindromic if and only if its mirror image is itself, that is $\text{mi}(\alpha) = \alpha$ or $\text{mi}(L) = L$. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under mirror image.

Indeed, $\mathcal{L}_3$ is closed under mirror image due to Corollary 2.2.7.

Given a context-free (resp. context-sensitive, recursively enumerable) grammar $G = (V_N, V_T, X_0, F)$, where all the production rules in F containing terminal letters are of the form $X \rightarrow a$ due to Lemma 3.0.3, the context-free (resp. context-sensitive, recursively enumerable) grammar G' generates the language $\text{mi}(L(G))$. This is achieved by constructing $G' = (V_N, V_T, X_0, \{\alpha \rightarrow \beta \mid \text{mi}(\alpha) \rightarrow \text{mi}(\beta) \in F, \alpha, \beta \in V_N^*\} \cup \{X \rightarrow a \in F \mid X \in V_N, a \in V_T\})$.

Substitution [93] As shown in Definition 2.2.14, substitution, $\sigma : V^* \rightarrow 2^{V'^*}$, where $V' = \bigcup_{a \in V} V_a$, is an operation that substitutes every letter a from the alphabet V with a language $\sigma(a)$ over the alphabet V_a. Substitution is defined recursively by $\sigma(\lambda) = \lambda, \sigma(\alpha\beta) = \sigma(\alpha)\sigma(\beta)$. The substitution of a language L is defined by $\sigma(L) = \{\alpha \mid \alpha \in \sigma(\beta) \text{ for some } \beta \in L\}$. Note that substitution is associative. A substitution σ is said to be regular (resp. context-free, context-sensitive, recursively enumerable) if and only if the language $\sigma(a)$ is regular (resp. context-free, context-sensitive, recursively enumerable), for all $a \in V$. A family of languages is said to be

closed under substitution if and only if whenever L is in the family and σ is a substitution such that each $\sigma(a)$ is in the family, then $\sigma(L)$ is also in the family. All families $\mathcal{L}_i$, $i = 0, 2, 3$, are closed under substitution, but $\mathcal{L}_1$ is not closed under substitution.

Indeed, $\mathcal{L}_3$ is closed under substitution due to Lemma 2.2.15.

Given a context-free language L generated by the context-free grammar $G = (V_N, V_T, X_0, F)$, where all the production rules in F containing terminal letters are of the form $X \to a$ due to Lemma 3.0.3, and a context-free substitution σ with the languages generated by the context-free grammars $\{G_a \mid G_a = (V_N^a, V_T^a, X_0^a, F^a), a \in V_T\}$, the context-free grammar G' that generates exactly the language $\sigma(L)$ can be constructed. This is achieved by the construction $G' = (V_N', V_T', X_0, F')$, where:

$$\begin{aligned}
V_N' &= V_N \cup \bigcup_{a \in V_T} V_N^a, \\
V_T' &= \bigcup_{a \in V_T} V_T^a, \\
F' &= F \backslash \{X \to a \mid X \in V_N, a \in V_T\} \\
&\quad \cup \{X \to X_0^a \mid X \to a \in F, X \in V_N, a \in V_T\} \\
&\quad \cup \bigcup_{a \in V_T} F^a.
\end{aligned}$$

According to Corollary 2.2.46, a context-sensitive language L' can be constructed from any recursively enumerable language L over an alphabet Σ, which contains words of the form $a^* b\alpha$, where $\alpha \in L$. Note now that $\sigma(L') = L$, where the context-sensitive substitution σ is defined by $\sigma(a) = \sigma(b) = \lambda, \sigma(c) = c$ for $c \in \Sigma$, so $\mathcal{L}_1$ is not closed under substitution.

Given a recursively enumerable language L and a recursively enumerable substitution σ, the boundary between different substitutions needs to be marked; otherwise, a production rule in the substitution may be applied with its left-hand side coming from different substitutions due to the same non-terminal alphabet V_N^a for the substitution of the same terminal letter a.

Example Consider the type-0 grammar $G = (\{X_0, X_1\}, \{a\}, X_0, \{X_0 \to X_1X_1, X_1 \to a\})$ that generates the language $L(G) = \{aa\}$. Let σ be the substitution of a with the language generated by $G_a = (\{X_0^a\}, \{a\}, X_0^a, \{X_0^a \to aa, X_0^aX_0^a \to a\})$. The result of applying this substitution on $L(G)$ is $\sigma(L(G)) = \{aaaa\}$. If we use the similar construction as for $\mathcal{L}_2$, the grammar $G' = (\{X_0, X_1, X_0^a\}, \{a\}, \{X_0 \to X_1X_1, X_1 \to X_0^a, X_0^a \to aa, X_0^aX_0^a \to a\})$ would be constructed. However, G' generates the word $a \notin \sigma(L(G))$ by the derivation $X_0 \Rightarrow X_1X_1 \Rightarrow X_0^aX_1 \Rightarrow X_0^aX_0^a \Rightarrow a$.

We will employ instead the following construction. Consider a recursively enumerable language L generated by the recursively enumerable grammar $G = (V_N, V_T, X_0, F)$, and a recursively enumerable substitution σ with languages generated by the recursively enumerable

grammars $\{G_a \mid G_a = (V_N^a, V_T^a, X_0^a, F^a), a \in V_T\}$. We assume that all the production rules in these grammar containing terminal letters are of the form $X \to a$ due to Lemma 3.0.3. The recursively enumerable grammar G' that generates exactly the language $\sigma(L)$ can be constructed. This is achieved by the construction $G' = (V'_N, V'_T, Y_0, F')$, where:

$$
\begin{aligned}
V'_N &= V_N \cup \{Y_0, Y\} \\
&\quad \cup \{A_a \mid a \in V_T\} \\
&\quad \cup \bigcup_{a \in V_T} V_N^a, \\
V'_T &= \bigcup_{a \in V_T} V_T^a, \\
F' &= \bigcup_{a \in V_T} F^i \\
&\quad \cup F \backslash \{X \to a \mid X \in V_N, a \in V_T\} \\
&\quad \cup \{X \to A_a \mid X \to a \in F, a \in V_T\} \\
&\quad \cup \{YA_a \to YX_0^a \mid a \in V_T\} \\
&\quad \cup \{Ya \to aY \mid a \in \bigcup_{a \in V_T} V_T^a\} \\
&\quad \cup \{Y_0 \to YX_0, Y \to \lambda\}.
\end{aligned}
$$

In this construction, the non-terminal Y makes sure that no non-terminal letters from different substitutions exist at the same time.

Regular Substitution [93] Given a language L over an alphabet Σ, if all the substitution languages $\sigma(a_i)$, where $a_i \in \Sigma, 0 \leq i \leq |\Sigma|$, are regular, the substitution $\sigma(L)$ is called a regular substitution. Using the same constructions and counterexample respectively as those in the proofs of the closure of the Chomsky families of languages under substitution, all families $\mathcal{L}_i$, $i = 0, 2, 3$, are closed under regular substitution, but $\mathcal{L}_1$ is not closed under regular substitution.

λ-Free Substitution [93] Given a language L over an alphabet Σ, if all the substitution languages $\sigma(a)$, where $a \in \Sigma$, are $\lambda-$free, the substitution is called a λ-free substitution. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under λ-free substitution.

Indeed, for $\mathcal{L}_i$, $i = 0, 2, 3$, their closure under λ-free substitution can be proven using the same constructions as those in the proofs of their closure under substitution.

Given a context-sensitive language L generated by a context-sensitive grammar G, a recursively enumerable grammar $G' = (V_N, V_T, X_0, F)$ can be constructed in the same way as in

the substitution proof for $\mathcal{L}_0$. Because all substitution languages, as well as L, are context-sensitive, their grammars are length-increasing. Thus, the only production rule of G' that is not length-increasing is $Y \to \lambda$. According to Proposition 2.2.45, since no words with two occurrences of Y can be derived according to G', a constant $p = 2$ exists such that $WS_{G'}(\alpha) \leq p|\alpha|$ for all nonempty words $\alpha \in L(G')$, so we have that $L(G')$ is a context-sensitive language, and G' generates exactly the language $\sigma(L)$.

λ-Free Regular Substitution [93] Given a language L over an alphabet Σ, if all the substitution languages $\sigma(a)$, where $a \in \Sigma$, are $\lambda-$free and are regular, the substitution is called a λ-free regular substitution. Using the same construction, respectively, as in the proofs of the closure of the Chomsky families of languages under λ-free substitution, all families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under λ-free regular substitution.

Homomorphism [93] Given a language L over an alphabet Σ, if all the substitution languages $\sigma(a)$, where $a \in \Sigma$, only contain a single word, the substitution is called a homomorphism. Using the same construction and counterexample respectively as in the proofs of the closure of the Chomsky families of languages under substitution, all families $\mathcal{L}_i$, $i = 0, 2, 3$, are closed under homomorphism, but $\mathcal{L}_1$ is not closed under homomorphism.

λ-free Homomorphism [93] Given a language L over an alphabet Σ, if all the substitution languages from $\sigma(a)$, where $a \in \Sigma$, only contain a non-empty single word, the substitution is called a λ-free homomorphism. Using the same construction respectively as in the proofs of the closure of the Chomsky families of languages under λ-free substitution, all families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under λ-free homomorphism.

Linear Erasing [93] Given a language L over an alphabet Σ and a homomorphism h, if there exists $k \in \mathcal{N}$ such that $|\alpha| \leq k|h(\alpha)|$, for all words $\alpha \in L$, then the homomorphism h is called a k-linear erasing with respect to L. A family of languages $\mathcal{L}$ is said to be closed under linear erasing if and only if $h(L) \in \mathcal{L}$ for all $L \in \mathcal{L}$ and for all $k-$linear erasings h, where $k \in \mathcal{N}$. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under linear erasing.

Indeed, all families $\mathcal{L}_i$, $i = 0, 2, 3$, are closed under linear erasing because they are closed under arbitrary homomorphism.

Let L be a context-sensitive language generated by the context-sensitive grammar $G = (V_N, V_T, X_0, F)$, where all the production rules in F containing terminal letters are of the form $X \to a$ due to Lemma 3.0.3, and h be a k-linear erasing with respect to L for some fixed $k \geq 0$. If $k = 0$, we have that $L = \{\lambda\}$ or $L = \emptyset$ which are both context-sensitive. If $k \geq 1$, the recursively

enumerable grammar $G' = (V_N, V'_T, X_0, F')$ can be constructed such that $L(G') = h(L)$, where:

$$V'_T = \bigcup_{a \in V_T} V_T^a,$$
$$F' = F \backslash \{X \to a \mid a \in V_T\}$$
$$\cup \{X \to h(a) \mid X \to a \in F, X \in V_N, a \in V_T\}.$$

Note that, for any non-empty word $\beta \in L(G')$, there is a word $\alpha \in L$ such that $h(\alpha) = \beta$. Since $WS_{G'}(\beta) = max\{|\alpha|, |\beta|\} \leq k * |\beta|$, it follows that $L(G')$ is context-sensitive according to Proposition 2.2.45. Thus, $\mathcal{L}_1$ is closed under linear erasing.

Restricted Homomorphism [93] Let L be a language over the alphabet Σ, $c \in \Sigma$ be a letter, $k \in \mathcal{N}^+$ be a positive integer, and h be a homomorphism defined by $h(c) = \lambda, h(a) = a$, for $a \in \Sigma$. If $c^k \notin \inf(L)$, then the homomorphism h is said to be k-restricted on L. A family of languages $\mathcal{L}$ is said to be closed under restricted homomorphism if and only if $h(L) \in \mathcal{L}$ for all $L \in \mathcal{L}$ and for all $k-$restricted homomorphism h, where $k \geq 1$. For all non-empty words $\alpha \in L$, we have that $|\alpha| \leq k|h(\alpha)|$, and it follows that h is a special case of $k-$linear erasing with respect to L. Thus, all families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under restricted homomorphism.

Inverse Homomorphism [93] Given two alphabets Σ_1, Σ_2, a mapping $g : \Sigma_2^* \to \Sigma_1^*$ is called an inverse homomorphism if there exists a homomorphism $h : \Sigma_1^* \to \Sigma_2^*$ such that $g(\alpha) = \{\beta \in \Sigma_1^* \mid h(\beta) = \alpha, \alpha \in \Sigma_2^*\}$. For a language L over Σ_2, we define $h^{-1}(L) = g(L) = \{\beta \in \Sigma_1^* \mid h(\beta) = \alpha, \alpha \in L\}$. Given a homomorphism h over an alphabet Σ, if h is a λ-free homomorphism, then we define that $h^{-1}(\lambda) = \lambda$; otherwise, $h^{-1}(\lambda) = \Sigma'^+$, where $\Sigma' = \{a \mid a \in \Sigma, h(a) = \lambda\}$. Thus, $h^{-1}(\lambda)$ is regular. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under inverse homomorphism.

Proposition 3.1.1 ([93]) *All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under inverse homomorphism.*

Proof The idea of the proof is as follows.

Consider a language $L \in \mathcal{L}_i$, $0 \leq i \leq 3$, over an alphabet Σ and an alphabet $\Sigma_0 = \{a_1, a_2, ..., a_r\}$. Let h be a homomorphism mapping Σ_0^* to Σ^*. We want to show that $h^{-1}(L) \in \mathcal{L}_i$.

Let $k = \max\{|h(a)| \mid a \in \Sigma_0\} + 1$ be a positive integer, and $\Sigma_1 = \{a'_1, a'_2, ..., a'_r\}$ be an alphabet, σ be a $\lambda-$free regular substitution defined by $\sigma(a) = \Sigma_1^* a \Sigma_1^*$ for all $a \in \Sigma$, h_1 be a $\lambda-$free homomorphism defined by $h_1(a'_i) = a_i$ for $1 \leq i \leq r$ and $h_1(a) = c$ for $a \in \Sigma$, and h_2 be a homomorphism defined by $h_2(c) = \lambda$ and $h_2(a_i) = a_i$ for $1 \leq i \leq r$.

If L is $\lambda-$free, we have that $h^{-1}(L) = h_2\left(h_1\left(\sigma(L) \cap \{a'_i h(a_i) \mid 1 \leq i \leq r\}^*\right)\right)$; otherwise, we have that $h^{-1}(L) = h^{-1}(L - \{\lambda\}) \cup h^{-1}(\{\lambda\}) = h_2\left(h_1\left(\sigma(L) \cap \{a'_i h(a_i) \mid 1 \leq i \leq r\}^*\right)\right) \cup h^{-1}(\{\lambda\})$. Note that h_2 is $k-$restricted on $L_\sigma = h_1\left(\sigma(L) \cap \{a'_i h(a_i) \mid 1 \leq i \leq r\}^*\right)$. Moreover, note that

$h^{-1}(\{\lambda\})$ is regular. Since the families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under λ−free regular substitution, restricted homomorphism, union with regular languages, and intersection with regular languages, it follows that they are closed under inverse homomorphism. ∎

Complementation [48, 64, 93, 98] Because a language can be viewed as a set of strings, some set operations can also be defined on languages. Given an alphabet Σ and a language L, the complement of L is defined by $\sim L = \{\alpha \in \Sigma^* \mid \alpha \notin L\} = \Sigma^* \backslash L$. Both families $\mathcal{L}_i$, $i = 1, 3$, are closed under complementation, but neither family $\mathcal{L}_i$, $i = 0, 2$, is closed under complementation.

Indeed, given a regular language L recognized by a DFA $M = (S, \Sigma, s_0, A, F)$, the DFA $M' = (S, \Sigma, s_0, S - A, F)$ that recognizes the language $\sim L(M)$ can be constructed.

By De Morgan's law, $L_1 \cap L_2 = \sim ((\sim L_1) \cup (\sim L_2))$. Since $\mathcal{L}_2$ is closed under union, if it were closed under complementation, it would be closed under intersection, which contradicts the following example. Thus, $\mathcal{L}_2$ is not closed under complementation.

Example Consider two context-free languages $L_1 = \{a^n b^n c^m \mid n, m \in \mathcal{N}\}$ and $L_2 = \{a^n b^m c^m \mid n, m \in \mathcal{N}\}$, their intersection $\{a^n b^n c^n \mid n \in \mathcal{N}\}$ is not context-free, as shown in Subsection 2.2.2.

$\mathcal{L}_1$ is closed under complementation, and this can be proven using a space complexity argument, which is not the focus of this thesis, so it is omitted here.

$\mathcal{L}_0$ is not closed under complementation, and this can be proven using a computablility argument, which is not the focus of this thesis, so it is omitted here.

Intersection [66, 90, 93, 96] Given two languages L_1 and L_2, their intersection is denoted by $L_1 \cap L_2 = \{\alpha \mid \alpha \in L_1, \alpha \in L_2\}$. All families $\mathcal{L}_i$, $i = 0, 1, 3$, are closed under intersection, but $\mathcal{L}_2$ is not closed under intersection. In addition, $\mathcal{L}_2$ is closed under intersection with regular languages.

Indeed, both families $\mathcal{L}_3$ and $\mathcal{L}_1$ are closed under union and complementation, and by De Morgan's law, $L_1 \cap L_2 = \sim ((\sim L_1) \cup (\sim L_2))$, so they are also closed under intersection.

From the previous example, it follows that $\mathcal{L}_2$ is not closed under intersection.

The situation is different if one of the languages is regular. Consider a context-free language L_1 recognized by a PDA $M_1 = (S_1, V_I, V_Z, z_0, s_{0_1}, A_1, F_1)$ and a regular language L_2 recognized by a DFA $M_2 = (S_2, V_T, s_{0_2}, A_2, F_2)$. The PDA that recognizes the language $L_1 \cap L_2$ can be constructed. This is achieved by the construction $M = (S, V_I', V_Z, z_0, s_0, A, F)$, where:

$$S = \{\langle s_1, s_2 \rangle \mid s_1 \in S_1, s_2 \in S_2\},$$
$$V_I' = V_I \cap V_T,$$

$$
\begin{aligned}
s_0 &= \langle s_{0_1}, s_{0_2} \rangle, \\
A &= \{\langle s_1, s_2 \rangle \mid s_1 \in A_1, s_2 \in A_2\}, \\
F &= \{z\langle s_1, s_2 \rangle a \to \alpha\langle s_3, s_4 \rangle \mid z s_1 a \to \alpha s_3 \in F_1, s_2 a \to s_4 \in F_2\} \\
&\quad \cup \{z\langle s_1, s_2 \rangle \to \alpha\langle s_3, s_2 \rangle \mid z s_1 \to \alpha s_3 \in F_1, s_2 \in S_2\}.
\end{aligned}
$$

In the construction, the PDA M simulates the PDA M_1 and the DFA M_2 on an input word $\alpha \in (V_I \cup V_T)^*$ in parallel.

$\mathcal{L}_0$ is closed under intersection, and this can be proven using a universal Turing machine, which is not the focus of this thesis, so it is omitted here.

Set Difference [93, 96] Given two languages L_1 and L_2 over an alphabet Σ, their difference is denoted by $L_1 \backslash L_2 = \{\alpha \mid \alpha \in L_1, \alpha \notin L_2\}$, that is $L_1 \backslash L_2 = L_1 \cap (\sim L_2)$. Both families $\mathcal{L}_i$, $i = 1, 3$, are closed under set difference, but neither family $\mathcal{L}_i$, $i = 0, 2$, is closed under set difference.

Since $\mathcal{L}_i$, $i = 1, 3$, are closed under intersection and complementation, and $L_1 \backslash L_2 = L_1 \cap (\sim L_2)$, it follows that they are closed under set difference.

Regarding $\mathcal{L}_i$, $i = 0, 2$, if they were closed under set difference, they would be closed under complementation since $\sim L = \Sigma^* \backslash L$. Thus, they are not closed under set difference.

Generalized Sequential Machine Mapping [93] First, we need to define a variation of an NFA called a generalized sequential machine.

Definition 3.1.2 ([93]) *A generalized sequential machine (GSM) is a sextuple $g = (V_I, V_O, S, s_0, A, F)$, where V_I is the input alphabet, V_O is the output alphabet, S is the set of states, $s_0 \in S$ is the initial state, $A \subseteq S$ is the set of final states, and $F \subseteq \{s_i a \to P s_j \mid s_i, s_j \in S, a \in V_I, P \in V_O^*\}$ is the set of transitions.*

Note that a GSM is essentially an NFA with an output and with no transitions of the form $s_i \to s_j$.

A configuration of a GSM $g = (V_I, V_O, S, s_0, A, F)$ is a word in $V_O^* s V_I^*$, where $s \in S$. An accepting configuration of g is a word in $V_O^* s$, where $s \in A$, and the initial configuration of g with the input word $\alpha \in V_I^*$ is $s_0 \alpha$.

Consider two states $s_1, s_2 \in S$, two words $\alpha, \alpha' \in V_I^*$ over the input alphabet, and two words $\beta, \beta' \in V_O^*$ over the output alphabet. A configuration $\beta s_1 \alpha$ derives another configuration $\beta' s_2 \alpha'$ in one step, denoted by $\beta s_1 \alpha \Rightarrow \beta' s_2 \alpha'$, according to the transitions in F if and only if there is a transition $s_1 a \to P s_2 \in F$, where $\alpha = a\alpha', \beta' = \beta P$.

Consider a number $k \in \mathcal{N}$, states $s_{i_j} \in S$, $0 \leq j \leq k, j \in \mathcal{N}$, input words $\alpha_j \in V_I^*$, $0 \leq j \leq k, j \in \mathcal{N}$, and output words $\beta_j \in V_O^*$, $0 \leq j \leq k, j \in \mathcal{N}$. A configuration $\beta_0 s_{i_0} \alpha_0$

derives another configuration $\beta_k s_{i_k} \alpha_k$, denoted by $\beta_0 s_{i_0} \alpha_0 \Rightarrow^* \beta_k s_{i_k} \alpha_k$, in k steps according to the transitions in F if and only if $\beta_0 s_{i_0} \alpha_0 \Rightarrow \beta_1 s_{i_1} \alpha_1 \Rightarrow ... \Rightarrow \beta_{k-1} s_{i_{k-1}} \alpha_{k-1} \Rightarrow \beta_k s_{i_k} \alpha_k$ according to the transitions in F. Note that if $k = 0$, this indicates a derivation of length 0, where no transitions is applied.

With the definition of derivations, we can define the action of a GSM.

A word $\beta \in V_O^*$ is said to be output by the GSM $g = (V_I, V_O, S, s_0, A, F)$ on the input word $\alpha \in V_I^*$, denoted by $\beta \in g(\alpha)$, if and only if $s_0 \alpha \Rightarrow^* \beta s$ according to the transitions in F, where $s \in A$. With the input languages L over the alphabet V_I, the language output by the GSM g is denoted by $g(L) = \{\beta \mid \beta \in g(\alpha), \alpha \in L\}$.

Note that a GSM can be viewed as a unary operation. Given a GSM $g = (V_I, V_O, S, s_0, A, F)$ and a language L, the image of L under the GSM mapping g is $g(L)$. All families $\mathcal{L}_i$, $i = 0, 2, 3$, are closed under GSM mapping. However, $\mathcal{L}_1$ is not closed under GSM mapping. To prove these, we need the following lemma.

Lemma 3.1.3 ([93]) *Let T be a function defined on an alphabet Σ such that $T(a, b) = 0$ or $T(a, b) = 1$ for all $a, b \in \Sigma$. The language $L = \{b_1 b_2 ... b_n \mid n \geq 1, b_i \in \Sigma, T(b_i, b_{i+1}) = 1$ for $1 \leq i \leq n - 1\}$ is regular.*

Proof The idea of the proof is as follows.

According to Proposition 2.2.16, L is regular if there exists a right-linear grammar G that generates exactly L. This right-linear grammar is $G = (V_N, V_T, X_0, F)$, where:

$$
\begin{aligned}
V_N &= \{X_i \mid 0 \leq i \leq r\}, \\
V_T &= \{a_i \mid 0 \leq i \leq r\}, \\
F &= \{X_0 \rightarrow a_i X_i \mid 1 \leq i \leq r\} \\
&\quad \cup \{X_i \rightarrow a_j X_j \mid 1 \leq i, j \leq r, T(a_i, a_j) = 1\} \\
&\quad \cup \{X_i \rightarrow \lambda \mid 1 \leq i \leq r\}. \qquad \blacksquare
\end{aligned}
$$

Next, we can prove the following proposition regarding the closure properties of $\mathcal{L}_i$, $i = 0, 2, 3$, under GSM mapping.

Proposition 3.1.4 ([93]) *All families $\mathcal{L}_i$, $i = 0, 2, 3$, are closed under GSM mapping.*

Proof The idea of the proof is as follows.

Let $g = (V_I, V_O, S, s_0, A, F)$ be a GSM, $\Sigma_1 = \{[s_i, a, \alpha, s_j] \mid s_i a \rightarrow \alpha s_j \in F\}$ be an alphabet, and T be a function on Σ_1 defined by $T([s_i, a, \alpha, s_j], [s_i', a', \alpha', s_j']) = 1$ if $s_j = s_i'$, and $T([s_i, a, \alpha, s_j], [s_i', a', \alpha', s_j']) = 0$ otherwise.

According to Lemma 3.1.3, the language $R = \{b_1b_2...b_n \mid n \geq 1, b_i \in \Sigma_1, T(b_i, b_{i+1}) = 1$ for $1 \leq i \leq n-1\}$ is regular. Note that we want the language R to represent the derivation of the generalized sequential machine g. Thus, the first letter of words from R should contain the initial state s_0, and the last letter should contain one of the accepting states $s \in A$. Let $R_2 = \{[s_0, a, \alpha, s_j] \in \Sigma_1\}, R_3 = \{[s_i, a, \alpha, s_j] \in \Sigma_1 \mid s_j \in A\}$ be languages over the alphabet Σ_1. If $s_0 \notin A$, we define $R_1 = (R_2\Sigma_1^*R_3 \cup (R_2 \cap R_3)) \cap R$; otherwise, we define $R_1 = ((R_2\Sigma_1^*R_3 \cup (R_2 \cap R_3)) \cap R) \cup \{\lambda\}$. Note that in both cases, R_1 is regular, and it represent a derivation of the GSM g.

Next, we extract the output word from the words representing derivations. Let h_1 be the homomorphism defined by $h_1([s_i, a, \alpha, s_j]) = a$ for $[s_i, a, \alpha, s_j] \in \Sigma_1$, and h_2 be the homomorphism defined by $h_2([s_i, a, \alpha, s_j]) = \alpha$ for $[s_i, a, \alpha, s_j] \in \Sigma_1$. We have that $g(L) = h_2(h_1^{-1}(L) \cap R_1)$.

Since the families $\mathcal{L}_i$, $i = 0, 2, 3$, are closed under intersection with regular languages, inverse homomorphism and arbitrary homomorphism, it follows that they are closed under GSM mapping. ∎

According to Corollary 2.2.46, a context-sensitive language L' can be constructed from any recursively enumerable language L over an alphabet Σ, and $g(L') = L$. Consider a GSM $g = (V_I, \Sigma, S, s_0, A, F)$, where:

$$\begin{aligned} V_I &= \Sigma \cup \{a, b\}, \\ S &= \{s_0, s_1\}, \\ A &= \{s_1\}, \\ F &= \{s_0a \to s_0, s_0b \to s_1\} \\ &\quad \cup \{s_1c \to cs_1 \mid c \in \Sigma\}. \end{aligned}$$

We have that $g(L') = L$, and it follows that $\mathcal{L}_1$ is not closed under GSM mapping.

λ-free Generalized Sequential Machine Mapping [93] A GSM is said to be λ-free if and only if its transitions are $F \subseteq \{s_ia \to Ps_j \mid s_i, s_j \in S, a \in V_I, P \in V_O^+\}$. Given a λ-free GSM $g = (V_I, V_O, S, s_0, A, F)$ and a language L, the image of L under the λ-free GSM g is $g(L)$. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under λ-free GSM mapping.

A similar proof to Proposition 3.1.4, with h_2 as a $\lambda-$free homomorphism, can be used here. Since the families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under intersection with regular languages, inverse homomorphism and $\lambda-$free homomorphism, they are closed under λ-free GSM mapping.

Left Quotient [36, 54, 93] Given two languages L_1 and L_2, the left quotient of L_1 by L_2 is denoted by $L_2^{-1}/L_1 = \{\beta \mid \alpha\beta \in L_1, \alpha \in L_2\}$. Both families $\mathcal{L}_0$ and $\mathcal{L}_3$ are closed under left

quotient, $\mathcal{L}_2$ is closed under left quotient by regular languages, and $\mathcal{L}_1$ is closed under left quotient by finite languages.

For the regular case, consider a DFA $M = (S, \Sigma, s_0, A, F)$ that recognizes the language L, two states $s_1, s_2 \in S$, and an arbitrary language L_2 over the alphabet Σ. Let the language $L_M^{(s_1,s_2)} = \{\alpha \mid s_1\alpha \Rightarrow^* s_2, \alpha \in \Sigma^*\}$ be the set of all words that can be read deriving any derivations from s_1 to s_2 L, and h be the homomorphism defined by $h(\#) = \lambda$ and $h(a) = a$ for $a \in \Sigma$. An NFA M' can be constructed such that $h(L(M') \cap \#\Sigma^*) = L_2^{-1_l}L_1$. This is achieved by the construction $M' = (S, \Sigma \cup \{\#\}, s_0, A, F \cup \{s_0\# \rightarrow s' \mid L_M^{(s_0,s')} \cap L_2 \neq \emptyset\})$. Since $\mathcal{L}_3$ is closed under arbitrary homomorphism and intersection, the left quotient of regular languages by an arbitrary language is regular.

Consider now the case where $L_1 = L$ is a regular (context-free, recursively enumerable) language over the alphabet Σ, and $L_2 = R$ over the alphabet Σ is regular. Let $c \notin \Sigma$ be a special letter, h be a homomorphism defined by $h(c) = \lambda, h(a) = a$ for $a \in \Sigma$, and g be a GSM defined by $g = (V_I, \Sigma, S, s_0, A, F)$, where:

$$
\begin{aligned}
V_I &= \Sigma \cup \{c\},\\
S &= \{s_0, s_1\},\\
A &= \{s_0\},\\
F &= \{s_0c \rightarrow s_1\}\\
&\quad \cup \{s_0a \rightarrow s_0 \mid a \in \Sigma\}\\
&\quad \cup \{s_1a \rightarrow as_1 \mid a \in \Sigma\}.
\end{aligned}
$$

We have that $R^{-1_l}L = g(h^{-1}(L) \cap Rc\Sigma^*)$. Since $\mathcal{L}_i$, $i = 0, 2, 3$, are closed under inverse homomorphism, GSM mapping and intersection with regular languages, it follows that they are also closed under left quotient by regular languages.

Consider two context-free languages L_1 and L_2, where $L_1 = a\{b^ja^j \mid j \in \mathcal{N}^+\}^*$, and $L_2 = \{a^ib^{2i} \mid i \in \mathcal{N}^+\}^*$. We have that $L_2^{-1_l}L_1 \cap a^+ = \{a^{2^n} \mid n \in \mathcal{N}^+\}$ which is not context-free by Lemma 2.2.34. Since $\mathcal{L}_2$ is closed under intersection with regular languages, it follows that it is not closed under left quotient.

Regarding the context-sensitive case, according to Corollary 2.2.46, a context-sensitive language L' can be constructed from any recursively enumerable language L, and $\{a^*b\}^{-1_l}L' = L$. Thus, $\mathcal{L}_1$ is not closed under left quotient by regular languages.

The situation is different for left quotient of context-sensitive languages by finite languages. Consider a context-sensitive language L generated by a context-sensitive grammar $G = (V_N, V_T, X_0, F)$ and a word $\alpha = a_1a_2...a_n \in V_T^*$. A recursively enumerable grammar G' that generates $\{\alpha\}^{-1_l}L$ can be constructed. This is achieved by the construction $G' = (V'_N, V_T, Y_0, F')$,

where:

$$\begin{aligned}
V'_N &= V_N \cup \{Y_0, Y_1, Y_2\} \\
&\quad \cup \{A_a \mid a \in V_T\}, \\
F' &= F \cup \{Y_0 \to Y_1 a_1 a_2 ... a_n Y_2 X_0, Y_1 Y_2 \to \lambda\} \\
&\quad \cup \{Y_2 a \to A_a Y_2 \mid a \in V_T\} \\
&\quad \cup \{bA_a \to A_a b \mid a, b \in V_T\} \\
&\quad \cup \{Y_1 A_a a \to Y_1 \lambda \mid a \in V_T\}.
\end{aligned}$$

According to Proposition 2.2.45, a constant $p = 2 * |\alpha| + 3$ exists such that $WS_{G'}(\beta) \leq p * |\beta|$ for all nonempty words $\beta \in L(G')$. Thus, if α is the longest word in L, $\mathcal{L}_1$ is closed under left quotient by finite languages L because the work space is bounded by $2 * |\alpha| + 3$.

Left Derivative [93] Given two languages L_1 and L_2, if L_2 only contains a single word α, the left quotient of L_1 by L_2 is called the left derivative of L_1 with respect to α, and it is denoted by $\partial_\alpha L_1$. From the proofs of the closure properties under left quotient, it follows that all families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under left derivative.

Right Quotient [36, 54, 93] Given two languages L_1 and L_2, the right quotient of L_1 by L_2 is defined by $L_1 L_2^{-1} = \{\alpha \mid \alpha\beta \in L_1, \beta \in L_2\}$. We have similar results and proofs as for left quotient. Both families $\mathcal{L}_0$ and $\mathcal{L}_3$ are closed under right quotient, $\mathcal{L}_2$ is closed under right quotient by regular languages, and $\mathcal{L}_1$ is closed under right quotient by finite languages.

Right Derivative [93] Given two languages L_1 and L_2, if L_2 only contains a single word α, the right quotient of L_1 by L_2 is called the right derivative of L_1 with respect to α, and it is denoted by $\partial^r_\alpha L_2$. From the proofs of the closure properties under right quotient, all families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under right derivative.

Prefix/Suffix/Infix [93] All words α over the alphabet Σ have a decomposition $\alpha = xyz$, where $x, y, z \in \Sigma^*$, and the subwords x, y, z of the word α are called the prefix, infix and suffix of the word α respectively. The set of prefixes (resp. infixes, suffixes) of a word α is denoted by $\text{pref}(\alpha)$ (resp. $\text{inf}(\alpha), \text{suff}(\alpha)$). If the empty word λ and the word α itself are removed from the set of prefixes (resp. infixes, suffixes) of the word α, then it is called the set of proper prefix (resp. infix, suffix) of the word α, denoted by $\text{Pref}(\alpha)$ (resp. $\text{Inf}(\alpha), \text{Suff}(\alpha)$). The prefix (resp. suffix, infix) operation on words can be extended to languages by $\text{pref}(L) = \{\alpha \mid \alpha\beta \in L, \alpha, \beta \in \Sigma^*\}$ (resp. $\text{suff}(L) = \{\beta \mid \alpha\beta \in L, \alpha, \beta \in \Sigma^*\}$, $\text{inf}(L) = \{\beta \mid \alpha_1\beta\alpha_2 \in L, \alpha_1, \alpha_2, \beta \in \Sigma^*\}$). The

proper prefix (infix, suffix) operation on languages can be defined in a similar manner. All families $\mathcal{L}_i$, $i = 0, 2, 3$, are closed under (proper) prefix, (proper) suffix and (proper) infix, but $\mathcal{L}_1$ is not.

Indeed, let L be a language over the alphabet Σ. We have that $\text{pref}(L) = L(\Sigma^*)^{-1}$, $\text{Pref}(L) = (L(\Sigma^+)^{-1})\backslash\{\lambda\}$, $\text{suff}(L) = (\Sigma^*)^{-1_l}L = \text{mi}\,(\text{pref}\,(\text{mi}\,(L)))$, $\text{Suff}(L) = ((\Sigma^+)^{-1_l}L)\backslash\{\lambda\}$, $\text{inf}(L) = \text{pref}(\text{suff}(L))$, $\text{Inf}(L) = \text{Pref}(L) \cup \text{Suff}(L)$. Since all families $\mathcal{L}_i$, $i = 0, 2, 3$, are closed under left quotient by regular languages, mirror image, right quotient by regular languages, union and difference with regular languages, it follows that they are closed under (proper) prefix, (proper) suffix and (proper) infix.

Next, we show the closure properties of $\mathcal{L}_1$ under prefix, suffix and infix. According to Corollary 2.2.46, a specific context-sensitive language L_1 can be constructed from any recursively enumerable language L over the alphabet Σ, and $\partial_b(\text{suff}(L_1) \cap b\Sigma^*) = L$. According to Corollary 2.2.47, a specific context-sensitive language L_2 can be constructed from any recursively enumerable language L over the alphabet Σ, and $\partial_b^r(\text{pref}(L_2) \cap \Sigma^* b) = L$. Note that $L_3 = L_1 b$ is context-sensitive because $\mathcal{L}_1$ is closed under concatenation. Moreover, $\partial_b(\partial_b^r(\text{inf}(L_3) \cap b\Sigma^* b)) = L$. Thus, $\mathcal{L}_1$ is not closed under prefix, suffix and infix.

Next, we show the closure properties of $\mathcal{L}_1$ under proper prefix, proper suffix and proper infix. Let $c \notin \Sigma$ be a letter, and h be a $\lambda-$free homomorphism defined by $h(b) = bc$ and $h(d) = d$ for $d \in \Sigma \cup \{a\}$. The three languages $L_1' = h(L_1), L_2' = h(L_2), L_3' = L_1' c$ that are context-sensitive because $\mathcal{L}_1$ is closed under concatenation and $\lambda-$free homomorphism. Since $\partial_c(\text{Suff}(L_1') \cap c\Sigma^*) = \partial_c^r(\text{Pref}(L_2') \cap \Sigma^* c) = \partial_c(\partial_c^r(\text{Inf}(L_3') \cap c\Sigma^* c)) = L$, it follows that $\mathcal{L}_1$ is not closed under proper prefix, proper suffix and proper infix.

3.2 Insertion and Deletion Operations

In this section, word and language operations related to inserting words into other words and deleting words from other words are defined, and the closure properties of the Chomsky families of languages under these operations are summarized.

Sequential Insertion [54] Given two words α, β over an alphabet Σ, the sequential insertion of β into α is defined by $\alpha \leftarrow \beta = \{\alpha_1\beta\alpha_2 \mid \alpha = \alpha_1\alpha_2\}$, and the sequential insertion of a language L_2 into another language L_1 is defined by $L_1 \leftarrow L_2 = \bigcup_{\alpha\in L_1, \beta\in L_2}(\alpha \leftarrow \beta)$. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under sequential insertion.

Indeed, let L_1, L_2 be two languages over the alphabet Σ, $\# \notin \Sigma$ be a letter, a homomorphism h defined by $h(\#) = \lambda$, $h(a) = a$ for $a \in \Sigma$, and a λ-free substitution σ defined by $\sigma(a) = a$ for $a \in \Sigma$, $\sigma(\#) = L_2\backslash\{\lambda\}$. We have the following cases:

- if $\lambda \notin L_1 \cup L_2$, $L_1 \leftarrow L_2 = \sigma(h^{-1}(L_1) \cap \Sigma^*\#\Sigma^*)$;
- if $\lambda \in L_2 - L_1$, $L_1 \leftarrow L_2 = \sigma(h^{-1}(L_1) \cap \Sigma^*\#\Sigma^*) \cup L_1$;
- if $\lambda \in L_1 - L_2$, $L_1 \leftarrow L_2 = \sigma(h^{-1}(L_1) \cap \Sigma^*\#\Sigma^*) \cup L_2$;
- if $\lambda \in L_1 \cap L_2$, $L_1 \leftarrow L_2 = \sigma(h^{-1}(L_1) \cap \Sigma^*\#\Sigma^*) \cup L_1 \cup L_2$.

Since all families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under inverse homomorphism, λ-free substitution, intersection with regular languages, and union, it follows that they are also closed under sequential insertion.

Parallel Insertion [54] Let Σ be an alphabet, $\alpha \in \Sigma^*$ be a word, and $L \subseteq \Sigma^*$ be a language. The parallel insertion of L into α is defined by $\alpha \Leftarrow L = \{\beta_0 a_1 \beta_1 a_2 ... a_k \beta_k \mid \alpha = a_1 a_2 ... a_k, \beta_0, \beta_1, ..., \beta_k \in L\}$, and the parallel insertion of a language L_2 into another language L_1 is defined by $L_1 \Leftarrow L_2 = \bigcup_{\alpha \in L_1} (\alpha \Leftarrow L_2)$. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under parallel insertion.

Indeed, let L_1, L_2 be two languages over the alphabet Σ, and σ be a $\lambda-$free substitution defined by $\sigma(a) = aL_2$ for $a \in \Sigma$. We have that $L_1 \Leftarrow L_2 = L_2\sigma(L_1)$. Since all families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under concatenation and λ-free substitution, it follows that they are also closed under parallel insertion.

Sequential Deletion [55] Given two words α, β over the alphabet Σ, the sequential deletion of β from α is defined by $\alpha \rightarrow \beta = \{\alpha_1\alpha_2 \mid \alpha = \alpha_1\beta\alpha_2, \alpha_1, \alpha_2 \in \Sigma^*\}$, and the sequential deletion of a language L_2 from another language L_1 is defined by $L_1 \rightarrow L_2 = \bigcup_{\alpha \in L_1, \beta \in L_2} (\alpha \rightarrow \beta)$. Both families $\mathcal{L}_i$, $i = 0, 3$, are closed under sequential deletion, but neither family $\mathcal{L}_i$, $i = 1, 2$, is closed under sequential deletion.

Consider first the case $L_1 \rightarrow L_2$, where L_1 is a regular language over the alphabet Σ, and L_2 is an arbitrary language over the alphabet Σ. Consider a DFA $M_1 = (S, \Sigma, s_0, A, F)$ that generates the language L_1 and two states $s_1, s_2 \in S$. Let the language $L_M^{(s_1,s_2)} = \{\alpha \mid s_1\alpha \Rightarrow^* s_2, \alpha \in \Sigma^*\}$ be the set of all words that can be read deriving any derivations from s_1 to s_2, and h be a homomorphism defined by $h(\#) = \lambda, h(a) = a$ for $a \in \Sigma$. An NFA M can be constructed such that $L_1 \rightarrow L_2 = h(L(M) \cap \Sigma^*\#\Sigma^*)$. This is achieved by the construction $M = (S, \Sigma \cup \{\#\}, s_0, A, F \cup \{s\# \rightarrow s' \mid s, s' \in S, L_{M_1}^{(s,s')} \cap L_2 \neq \emptyset\})$. We note that M recognizes the language $L_1 \rightarrow L_2$ with deleted subwords replaced by #. Since $\mathcal{L}_3$ is closed under intersection and arbitrary homomorphism, the sequential deletion of arbitrary languages from regular languages is regular.

Consider now the case where $L_1 = L$ is a context-free language over the alphabet Σ, and $L_2 = R$ is recognized by the DFA $M = (S, \Sigma, s_0, A, F)$. A GSM g such that $L \rightarrow R = g(L) \cup \{\lambda \mid L \cap R \neq \emptyset\}$ can be constructed. This is achieved by the construction $g = (\Sigma, \Sigma, S', s_0', A', F')$, where:

$$
\begin{aligned}
S' &= S \cup \{s_0', s_f\},\\
A' &= \{s_f\},\\
F' &= F \cup \{s_0'a \rightarrow as_0' \mid a \in \Sigma\}\\
&\quad \cup \{s_0'a \rightarrow s \mid s_0a \rightarrow s \in F\}\\
&\quad \cup \{sa \rightarrow s_f \mid sa \rightarrow s' \in F, s' \in A\}\\
&\quad \cup \{s_fa \rightarrow as_f \mid a \in \Sigma\}\\
&\quad \cup \{s_0'a \rightarrow s_f \mid s_0a \rightarrow s \in F, s \in A\}\\
&\quad \cup \{s_0'a \rightarrow as_f \mid a \in \Sigma, \lambda \in R\}.
\end{aligned}
$$

Thus, any family of languages closed under GSM mapping is also closed under sequential deletion with regular languages. Since $\mathcal{L}_2$ is closed under GSM mapping, it follows that it is also closed under sequential deletion with regular languages.

Consider two context-free languages $L_1 = \#a\{b^ia^i \mid i \in \mathcal{N}^+\}^*, L_2 = \#\{a^ib^{2^i} \mid i \in \mathcal{N}^+\}^*$, we have that $(L_1 \rightarrow L_2) \cap a^+ = \{a^{2^n} \mid n \in \mathcal{N}^+\}$ which is not context-free by Lemma 2.2.34. Since $\mathcal{L}_2$ is closed under intersection with regular languages, it follows that it is not closed under sequential deletion.

According to Proposition 2.2.46, a specific context-sensitive language L' can be constructed from any recursively enumerable language L, and $L' \rightarrow a^*b = L$. Thus, $\mathcal{L}_1$ is not closed under sequential deletion.

Dipolar Deletion [55] Given two words α, β over an alphabet Σ, the dipolar deletion of β from α is defined by $\alpha \leftrightarrow \beta = \{\alpha_2 \mid \alpha = \alpha_1\alpha_2\alpha_3, \beta = \alpha_1\alpha_3, \alpha_1, \alpha_2, \alpha_3 \in \Sigma^*\}$, and the dipolar deletion of a language L_2 from another language L_1 is defined by $L_1 \leftrightarrow L_2 = \bigcup_{\alpha \in L_1, \beta \in L_2}(\alpha \leftrightarrow \beta)$. Using proofs similar to the proofs for sequential deletion, we have that both families $\mathcal{L}_i$, $i = 0, 3$, are closed under dipolar deletion, but neither family $\mathcal{L}_i$, $i = 1, 2$, is closed under dipolar deletion.

Shuffle [49, 54] Given two words α, β over an alphabet Σ, the shuffle of β into α is defined by $\alpha \coprod \beta = \{\alpha_1\beta_1\alpha_2\beta_2...\alpha_k\beta_k \mid \alpha = \alpha_1\alpha_2...\alpha_k, \beta = \beta_1\beta_2...\beta_k, k \in \mathcal{N}^+; \alpha_i, \beta_i \in \Sigma^*, 1 \leq i \leq k\}$, and the shuffle of a language L_2 into another language L_1 can be defined by $L_1 \coprod L_2 =$

$\bigcup_{\alpha\in L_1,\beta\in L_2}(\alpha \coprod \beta)$. All families $\mathcal{L}_i$, $i = 0, 1, 3$, are closed under shuffle, but $\mathcal{L}_2$ is not closed under shuffle.

For the regular case, let L_1 be a regular language recognized by the DFA $M_1 = (S_1, \Sigma, s_{0_1}, A_1, F_1)$ and L_2 be a regular language recognized by the DFA $M_2 = (S_2, \Sigma, s_{0_2}, A_2, F_2)$. An NFA that recognizes the language $L(M_1) \coprod L(M_2)$ can be constructed. This is achieved by the construction $M = (S, \Sigma, \langle s_{0_1}, s_{0_2}\rangle, A, F)$, where:

$$\begin{aligned}
S &= \{\langle s_1, s_2\rangle \mid s_1 \in S_1, s_2 \in S_2\},\\
A &= \{\langle s_1, s_2\rangle \mid s_1 \in A_1, s_2 \in A_2\},\\
F &= \{\langle s_1, s_2\rangle a \to \langle s_3, s_2\rangle \mid s_1 a \to s_3 \in F_1, s_2 \in S_2\}\\
&\quad \cup \{\langle s_1, s_2\rangle a \to \langle s_1, s_4\rangle \mid s_2 a \to s_4 \in F_2, s_1 \in S_1\}.
\end{aligned}$$

Note that a PDA can be constructed in a similar way for the shuffle of regular languages into context-free languages and for the shuffle of context-free languages into regular languages. Thus, $\mathcal{L}_3$ and $\mathcal{L}_2$ are closed under shuffle into regular languages, and the shuffle of regular languages into context-free languages is context-free.

Consider two context-free languages $L_1 = \{a^n b^n \mid n \in \mathcal{N}^+\}$, $L_2 = \{c^m d^m \mid m \in \mathcal{N}^+\}$, we have that $(L_1 \coprod L_2) \cap a^+c^+b^+d^+ = \{a^n c^m b^n d^m \mid m, n \in \mathcal{N}^+\}$ which is not context-free by Lemma 2.2.34. Since $\mathcal{L}_2$ is closed under intersection with regular languages, it follows that it is not closed under shuffle.

Let L_1 be a λ-free context-sensitive language generated by the context-sensitive grammar $G_1 = (V_{N_1}, V_{T_1}, X_{0_1}, F_1)$, and L_2 be a λ-free context-sensitive language generated by the context-sensitive grammar $G_2 = (V_{N_2}, V_{T_2}, X_{0_2}, F_2)$. We assume that all the production rules in F_1, F_2 containing terminal letters are of the form $X \to a$ due to Lemma 3.0.3. A length-increasing grammar G that generates exactly the language $L_1 \coprod L_2$ can be constructed. This is achieved by the construction $G = (V_N, V_T, X_0, F)$, where:

$$\begin{aligned}
V_N &= V_{N_1} \cup V_{N_2} \cup \{X_0\} \cup \{X_a \mid a \in V_{T_2}\},\\
V_T &= V_{T_1} \cup V_{T_2},\\
F &= F_1 \cup \{X \to X_a \mid X \to a \in F_2, a \in V_{T_2}\}\\
&\quad \cup \{X_0 \to X_{0_1} X_{0_2}\}\\
&\quad \cup \{bX_a \to X_a b \mid a \in V_{T_2}, b \in V_{T_1}\}\\
&\quad \cup \{X_a \to a \mid a \in V_{T_2}\}.
\end{aligned}$$

The other cases regarding the empty word λ are as following:

- if $\lambda \in L_2 \backslash L_1$, $L_1 \coprod L_2 = (L_1 \coprod (L_2 \backslash \{\lambda\})) \cup L_1$;

- if $\lambda \in L_1 \backslash L_2$, $L_1 \coprod L_2 = ((L_1 \backslash \{\lambda\}) \coprod L_2) \cup L_2$;
- if $\lambda \in L_2 \cap L_1$, $L_1 \coprod L_2 = ((L_1 \backslash \{\lambda\}) \coprod (L_2 \backslash \{\lambda\})) \cup L_1 \cup L_2$.

Since $\mathcal{L}_1$ is closed under union and difference with regular languages, it is also closed under shuffle.

3.3 Invertible Binary Operations

Decision problems regarding equations involving formal languages and operations are an important theoretical aspect of formal language study. For example, given a language L over an alphabet Σ and two homomorphisms h, g, mapping words over Σ, the decision problems related to whether or not $g(L) = h(L)$ were studied in [2, 19, 20]. In this section, equations of the form $L \diamond Y = R$ or $X \diamond L = R$ are considered, where $\diamond$ is a binary word operation extended to languages by $L \diamond Y = \bigcup_{\alpha \in L, \beta \in Y} (\alpha \diamond \beta)$, L, R are known languages, and X, Y are unknown languages. The solutions to these equations and decision problems about the existence of solutions and singleton solutions were studied in [56], and we briefly describe them here.

Consider an alphabet Σ and a binary word operation $\square$. We can define the right-inverse of the word operation $\square$.

Definition 3.3.1 ([56]) *A binary word operation $\diamond$ is called the right-inverse of $\square$ if and only if for all $\alpha, \beta, \gamma \in \Sigma^*$, we have that $\gamma \in (\alpha \square \beta)$ if and only if $\beta \in (\alpha \diamond \gamma)$.*

Example Sequential insertion and reversed dipolar deletion[1] are right-inverses of each other because $\gamma \in (\alpha \leftarrow \beta)$ if and only if $\beta \in (\gamma \leftrightarrow \alpha)$ for all $\alpha, \beta, \gamma \in \Sigma^*$.

We can define the left-inverse of $\square$ in a similar way.

Definition 3.3.2 ([56]) *A binary word operation $\diamond$ is called the left-inverse of $\square$ if and only if for all $\alpha, \beta, \gamma \in \Sigma^*$, we have that $\gamma \in (\alpha \square \beta)$ if and only if $\alpha \in (\gamma \diamond \beta)$.*

Example Concatenation and right quotient are the left-inverses of each other because for all $\alpha, \beta, \gamma \in \Sigma^*$, we have that $\gamma \in (\alpha\beta)$ if and only if $\alpha \in \gamma\beta^{-1}$.

Next, we use left-inverse and right-inverse to solve equations of the form $L \diamond Y = R$ or $X \diamond L = R$. Consider an alphabet Σ, two known languages L, R over the alphabet Σ, and a binary operation $\diamond$.

Consider first the case where right operand in the equation is unknown.

[1] A binary word operation $\square^r$ is called the reversed operation of $\square$ if and only if for all $\alpha, \beta \in \Sigma^*$, we have that $\alpha \square^r \beta = \beta \square \alpha$.

Proposition 3.3.3 ([56]) *Consider an equation $L \diamond Y = R$. If there is a solution to this equation, the language $Y' = (L \square R^c)^c$ is also a solution to the equation, where $\square$ is the right-inverse of $\diamond$, and all the solutions to this equation are subsets of Y'.*

Proof We want to prove first that the Y' is a solution, by double inclusion.

On the one hand, we can prove by contradiction that $L \diamond Y' \subseteq R$. Assume that $L \diamond Y' \nsubseteq R$ means that there exists a word $\gamma \in \Sigma^*$ in $L \diamond Y'$ but not in R. Then, there exist words $\alpha \in L, \beta \in Y'$ such that $\gamma \in (\alpha \diamond \beta)$, we have that $\beta \in (\alpha \square \gamma) \subseteq (L \square R^c)$, which contradicts the fact that $\beta \in Y'$. Thus, we have that $L \diamond Y' \subseteq R$.

On the other hand, we can prove by contradiction that for all languages Y over the alphabet Σ such that $L \diamond Y = R$, we have that $Y \subseteq Y'$. Assume that there exists a language Y such that $L \diamond Y = R$ but $Y \nsubseteq Y'$. This implies that there exists a word $\beta \in \Sigma^*$ in Y but not in Y', which means β is in $L \square R^c$. Then, there exist words $\alpha \in L, \gamma \in R^c$ such that $\beta \in \alpha \square \gamma$, and it follows that $\gamma \in \alpha \diamond \beta \subseteq L \diamond Y = R$, which contradicts the fact that $\gamma \in R^c$. Thus, we have that $L \diamond Y' \subseteq R$, and it follows that Y' is the maximal solution. ∎

In a similar approach, we arrive at the following proposition for cases where the left operand in the equation is unknown.

Proposition 3.3.4 ([56]) *Consider an equation $X \diamond L = R$. If there is a solution to the equation, the language $X' = (R^c \square L)^c$ is a solution to the equation, where $\square$ is the left-inverse of $\diamond$, and all the solutions to the equation are a subset of X'.*

Note that X' is the maximal solution to the equation.

We can ask the following two decision questions regarding equations of the form $L \diamond Y = R$ or $X \diamond L = R$.

1. Is there a solution to the equation?
2. Is there a singleton solution to the equation?

We can usually determine whether problem 1 is decidable, regarding equations of the form $L \diamond Y = R$ or $X \diamond L = R$, by a general method with the help of Proposition 3.3.3 and Proposition3.3.4, as follows.

Example Consider an alphabet Σ, two regular languages L, R over the alphabet Σ, and a binary operation $\diamond$ whose right-inverse is $\square$. Also, assume that $\mathcal{L}_3$ is closed under $\diamond$ and $\square$. The problem "Is there a solution to the equation $L \diamond Y = R$?" is decidable because of Algorithm 3, and the maximal solution can be effectively constructed.

Input: Two regular languages L, R, two binary operations $\diamond, \square$ under which $\mathcal{L}_3$ is closed, and which are right-inverses of each other
Output: "Yes" if there is a solution to the equation $L \diamond Y = R$ and "no" otherwise Let Y' be the regular language $(L \square R^c)^c$;
Let R' be the regular language $L \diamond Y'$;
if $R' = R$ **then**
  Return "Yes";
else
  Return "No";
end

Algorithm 3: An algorithm that decides the problem "Is there a solution to the equation $L \diamond Y = R$?"

When we try to decide the existence of a singleton solution to equations of the form of $L \diamond \{w\} = R$ or $\{w\} \diamond L = R$, we need to analyze the properties of the languages $L, R, \{w\}$.

Example Consider an alphabet Σ, two regular languages L, R over the alphabet Σ, and an equation $L\{w\} = R$. If there is a singleton solution $\{w\}$, the word w should not be longer than the shortest word in R due to the nature of concatenation, so we have a finite set of words to check. Thus, the existence of a singleton solution to the equation is decidable.

When we study a new binary word operation $\diamond$, this chapter provides us a direction of research. We can ask the previous two decision questions regarding equations of the form $L \diamond Y = R$ or $X \diamond L = R$.

3.4 Abstract Families of Languages

In Section 2.2, the families of languages were defined based on restrictions to the production rules of their grammars. We can also define a family of languages based on their closure properties under some operations.

Definition 3.4.1 ([37]) *A family of languages $\mathcal{L}$ is called an abstract family of languages (AFL) if and only if*

- *there is a language containing a non-empty word in the family $\mathcal{L}$, and*
- *the family $\mathcal{L}$ is closed under union, concatenation, λ−free concatenation closure, inverse homomorphism, λ−free homomorphism, and intersection with regular languages.*

Note that all of the families $\mathcal{L}_i$, where $0 \leq i \leq 3$, are AFLs. If $\lambda-$free homomorphism is replaced by arbitrary homomorphism, we have a more restricted definition.

Definition 3.4.2 ([37]) *A family of languages $\mathcal{L}$ is called a full abstract family of languages if and only if*

- *there is a language containing a non-empty word in the family $\mathcal{L}$, and*
- *the family $\mathcal{L}$ is closed under union, concatenation, $\lambda-$free concatenation closure, intersection with regular languages, arbitrary homomorphism, and inverse homomorphism.*

Note that the family of context-sensitive languages is not a full AFL.

Note that we can prove that a family of languages, which is known to be closed under a set O of operations, is closed under an operation $\diamond$ by representing $\diamond$ in terms of operations from O, as shown in Corollary 3.4.3 of Proposition 3.1.1.

Corollary 3.4.3 ([93]) *If a family $\mathcal{L}$ of languages is closed under $\lambda-$free regular substitution, restricted homomorphism, union with regular languages, and intersection with regular languages, then $\mathcal{L}$ is closed under inverse homomorphism.*

We are also interested in finding the smallest and most restricted set of operations to represent a new operation.

Example If we replace "union with regular languages" with "union", Corollary 3.4.3 also holds . However, the resulting set of operations is sufficient but not necessary.

Note that we can prove that a family of languages, which includes a language containing a non-empty word and is closed under some other set O of operations, is an AFL by representing the required operations in the definition of AFLs in terms of the operations from O, as shown in the proof the following proposition.

Proposition 3.4.4 ([93]) *If a family $\mathcal{L}$ of languages includes a language containing a non-empty word and is closed under union, $\lambda-$free concatenation closure, $\lambda-$free regular substitution, intersection with regular languages, and restricted homomorphism, then $\mathcal{L}$ is an AFL.*

Proof If $\mathcal{L}$ is $\lambda-$free, we have that $\mathcal{L}$ is closed under inverse homomorphism by Proposition 3.1.1. Thus, $\mathcal{L}$ is an AFL in this case.

If $\mathcal{L}$ is not $\lambda-$free, for all languages $L \in \mathcal{L}$ over an alphabet Σ and any homomorphism, we have that $h^{-1}(L') \in \mathcal{L}$ where $L' = L \cap \Sigma^+$. Since $L = L' \cup \{\lambda\}$, it follows that $h^{-1}(L) = h^{-1}(L') \cup h^{-1}(\lambda)$, so we only need to prove that $h^{-1}(\lambda) \in \mathcal{L}$ because $\mathcal{L}$ is closed under intersection

with regular languages. Note that $\{\lambda\} \in \mathcal{L}$ because there exists a language $L_1 \in \mathcal{L}$ such that $\lambda \in L_1$, $\{\lambda\} = L_1 \cap \{\lambda\}$, and $\mathcal{L}$ is closed under intersection with regular languages. Since there is a language $L_2 \in \mathcal{L}$ containing a non-empty word α, we have that $\{\alpha\} = L_2 \cap \{\alpha\} \in \mathcal{L}$. Since $\{\alpha\}$ is $\lambda-$free, there exists a inverse homomorphism that maps $\{\alpha\}$ to $\{a \in \Sigma\} \in \mathcal{L}$. Thus, we have that there exists a $\lambda-$free regular substitution that maps $\{a\}$ to $L_3 \subseteq \Sigma^*$, and $L_3 \in \mathcal{L}_3$. Thus, with any inverse homomorphism, there exists a language $L_3 \subseteq \Sigma^*$ such that $h^{-1}(\{\lambda\}) = L_3$. Thus, $\mathcal{L}$ is closed under inverse homomorphism.

Since $\mathcal{L}$ is closed under union, $\lambda-$free homomorphism, inverse homomorphism, and intersection with regular languages, it is closed under concatenation [93]. It follows that $\mathcal{L}$ is an AFL in this case. ∎

Note that since L_i, $0 \leq i \leq 3$, are AFLs, and L_i, for $i = 0, 2, 3$, are full AFLs, we can prove that L_i, for $0 \leq i \leq 3$, are closed under an operation if the corresponding (full) AFL is closed under that operation.

Example According to Corollary 3.4.5 of Proposition 3.1.4, we have that $\mathcal{L}_3$ is closed under GSM mapping.

Corollary 3.4.5 ([93]) *Every full AFL is closed under GSM mapping.*

When we study a new operation $\diamond$, this chapter provides us a method to prove that some Chomsky families of languages are closed under $\diamond$ by proving a (full) AFL is closed under $\diamond$.

3.5 State Complexity

In this section, we give a brief introduction to state complexity and some state complexity results about operations under which $\mathcal{L}_3$ is closed. State complexity is a complexity measure for $\mathcal{L}_3$, and it is based on the DFA model [106].

The size of a DFA $M = (S, V_T, s_0, A, F)$ is determined by either the number of states $|S|$, or the number of transitions $|F|$. An upper bound of the number of transitions of any DFA $M = (S, V_T, s_0, A, F)$ is given by $|V_T| * |S'|$. It follows that both ways used to determine the size of a DFA are related, and in this section we are interested in the number of states.

Definition 3.5.1 ([34]) *The state complexity of a regular language L, denoted by sc(L), is defined as the number of states in the minimal complete DFA M that recognizes L.*

Example The state complexity of the regular language $L = a\{a, b\}^*b$ over the alphabet $\Sigma = \{a, b\}$ is 4, because the DFA in Figure 3.1 is the minimal complete DFA that recognizes L.

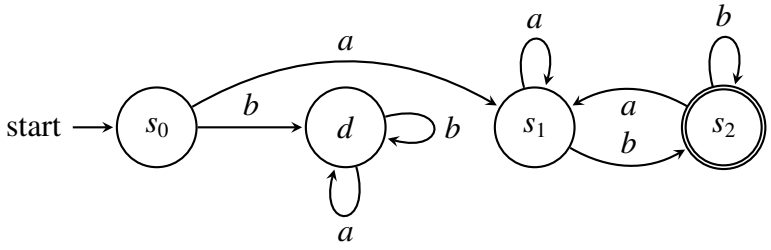

Figure 3.1: The minimal complete DFA with 4 states that recognizes $a(\{a\} \cup \{b\})^*b$

Definition 3.5.2 ([106]) *A regular language L is called an m−state DFA language if and only if L can be recognized by a DFA with m states.*

The definition of state complexity can be extended to operations under which $\mathcal{L}_3$ is closed.

Definition 3.5.3 ([107]) *Let $\diamond$ be an operation under which $\mathcal{L}_3$ is closed. If $\diamond$ takes k operands, let L_i be an m_i−state DFA language for $1 \leq i \leq k$. The state complexity of an operation $\diamond$ is defined as the number of states, in terms of m_i, $1 \leq i \leq k$, that is sufficient and necessary (in the worst case) for a minimal complete DFA to recognize the regular language generated by applying $\diamond$ on regular languages L_i, $1 \leq i \leq k$.*

Example The state complexity of concatenation between an m−state DFA language L_1 with an n−state DFA language L_2 over an arbitrary alphabet Σ is $m \cdot 2^n - 2^{n-1}$, where $m, n > 1$ [107].

Next, we show that a DFA with $m \cdot 2^n - 2^{n-1}$ states is sufficient and necessary to recognize the language L_1L_2.

Let L_1 be an m−state DFA language recognized by a DFA $M_1 = (S_1, \Sigma, s_{0_1}, A_1, F_1)$ with m states, and L_2 be an n−state DFA language recognized by a DFA $M_2 = (S_2, \Sigma, s_{0_2}, A_2, F_2)$ with n states. A DFA $M = (S, \Sigma, s_0, A, F)$ that recognizes the language L_1L_2 can be constructed, where:

$$S = \{\langle s_i, T\rangle \mid s_i \in S_1, T \in 2^{S_2}\} - \{\langle s_i, T\rangle \mid s_i \in A_1, T \in 2^{S_2 - \{s_{0_2}\}}\},$$

$$s_0 = \begin{cases} \langle s_{0_1}, \emptyset\rangle & \text{if } s_{0_1} \in A_1, \\ \langle s_{0_1}, \{s_{0_1}\}\rangle & \text{otherwise,} \end{cases}$$

$$A = \{\langle s_i, T\rangle \mid \langle s_i, T\rangle \in S, T \cap A_1 \neq \emptyset\},$$

$$F = \{\langle s_i, T\rangle a \to \langle s_j, T'\rangle \mid s_i a \to s_j \in F_1, s_j \in A_1, T' = \{s_{0_2}\} \cup \bigcup_{s_k \in T} \{s_l \mid s_k a \to s_l \in F_2\}\}$$

$$\cup \{\langle s_i, T\rangle a \to \langle s_j, T' >\mid s_i a \to s_j \in F_1, s_j \notin A_1, T' = \bigcup_{s_k \in T} \{s_l \mid s_k a \to s_l \in F_2\}\}.$$

The set T tracks all the possible current states for M_2. Every time that M_1 reaches an final state, the initial state of M_2 is added to T. Note that this DFA contains $(m - |A_1|) \cdot 2^n - 2^{n-1}$ states which is maximized when $|A_1| = 1$.

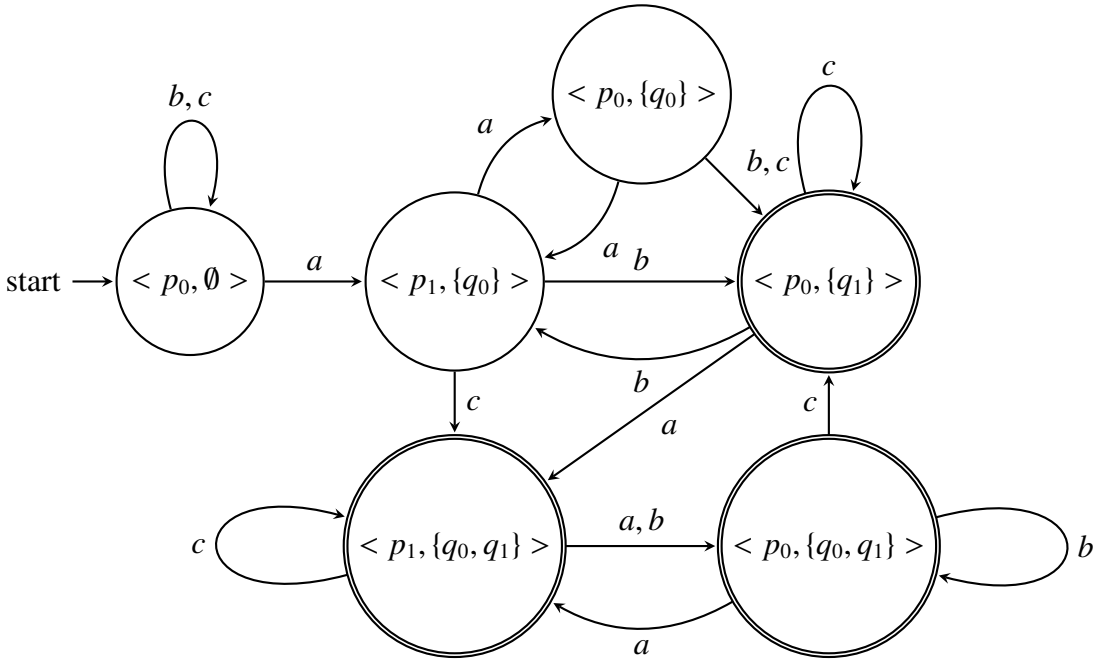

Figure 3.2: The minimal DFA that recognizes the concatenation of two 2-state DFA languages

The bound $m \cdot 2^n - 2^{n-1}$ is also necessary because it is reached when L_1, L_2 are recognized by minimal complete DFAs $M_1 = (S_1, \Sigma, s_{0_1}, A_1, F_1)$ and $M_2 = (S_2, \Sigma, s_{0_2}, A_2, F_2)$, where:

$$
\begin{aligned}
S_1 =& \{p_i \mid 0 \leq i \leq m-1\},\\
A_1 =& \{p_{m-1}\},\\
F_1 =& \{p_i a \to p_j \mid j = (i+1) \bmod m, p_i, p_j \in S_1\}\\
&\cup \{p_i b \to p_0 \mid p_i \in S_1\}\\
&\cup \{p_i c \to p_i \mid p_i \in S_1\},\\
S_2 =& \{q_i \mid 0 \leq i \leq n-1\},\\
A_2 =& \{q_{n-1}\},\\
F_2 =& \{q_i b \to q_j \mid j = (i+1) \bmod n, q_i, q_j \in S_2\}\\
&\cup \{q_i c \to q_1 \mid q_i \in S_2\}\\
&\cup \{q_i a \to q_i \mid q_i \in S_2\}.
\end{aligned}
$$

If $m = n = 2$, the DFA constructed by this procedure is minimal, and it is shown in

Figure 3.2.

Note that we specified that $m, n > 1$. If $n = 1$, a different bound would be calculated; the state complexity of concatenation of an $m-$state DFA language with a 1-state DFA language is m [107]. Also, we specified that the bound above was reached when an arbitrary alphabet was used. If we have a single-letter alphabet Σ, the state complexity of concatenation of an $m-$state DFA language with an n-state DFA language is $m \cdot n$ if m and n are relatively prime [107]. Thus, when we consider the state complexity of operations, we need to consider these special cases.

The state complexities of union, intersection, mirror image, concatenation, concatenation closure, left quotient with an arbitrary language L and right quotient with an arbitrary language L, are studied in [8, 34, 89, 106, 107], and are summarized in Table 3.1.

Operation	State Complexity	Worst Case
$L_m \cup L_n$	mn	$L_m = \{\alpha \in \{a,b\}^* \mid n_a(\alpha) \neq 0 \bmod m\}$ $L_n = \{\alpha \in \{a,b\}^* \mid n_b(\alpha) \neq 0 \bmod n\}$
$L_m \cap L_n$	mn	$L_m = \{\alpha \in \{a,b\}^* \mid n_a(\alpha) = 0 \bmod m\}$ $L_n = \{\alpha \in \{a,b\}^* \mid n_b(\alpha) = 0 \bmod n\}$
$\sim L_m$	m	All cases
$L_m L_n$	$(2m-1)2^{n-1}$	L_m, L_n shown in Figure 3.2
L_m^*	$2^{m-1} + 2^{m-2}$	$L_m = \{\alpha \in \{a,b\}^* \mid N_a(\alpha) = 2n+1, n \in \mathcal{N}\}$
$L^{-1_l} L_m$	$2^m - 1$	$L = \{a,b\}^*, L_m = (b^*a\{a,b\}^{m-1})^* b^* a \{a,b\}^{m-2}$
$L_m L^{-1}$	m	$L = \{\lambda\}$

Table 3.1: State complexities of some operations under which $\mathcal{L}_3$ is closed, where the operands are an $m-$state DFA language L_m and an $n-$state DFA language L_n over an arbitrary alphabet Σ, where $m, n > 1$.

As shown in Proposition 3.1.1, operations can be represented as a composition of other basic operations.

Example A new binary operation $\diamond$ is defined by $L_m \diamond L_n = (L_m \cup L_n)^*$, and $\mathcal{L}_3$ is closed under $\diamond$ because it is closed under union and concatenation closure. An upper bound for the state complexity can be calculated using function composition, where L_m, L_n are over an arbitrary alphabet and $m, n > 1$, such as $sc(L_m \diamond L_n) = 2^{m \cdot n - 1} + 2^{m\dot{n}-2}$. However, this is different from the exact state complexity $2^{m+n-1} - 2^{m-1} - 2^{n-1} + 1$ [34].

It has also been proven that there does not exist an algorithm to calculate the state complexity of an operation which is a composition of some basic regularity-preserving operations [94].

Chapter 4

Biologically-Inspired Operations

The emergence of the field of DNA computing [1] has motivated the definition of many language operations that are biologically inspired. Biological processes on DNA can be viewed as functions on the information stored on DNA, so studying such operations can ultimately lead to biological computers which are as computationally powerful as electronic computers and can naturally achieve high degrees of parallelism.

In Section 4.1, we will briefly introduce DNA and some DNA related processes. In Section 4.2, biologically inspired word operations and their properties are surveyed.

4.1 Biological Background

Nucleotides are the basic building blocks of deoxyribonucleic acid (DNA). Each nucleotide contains a five-carbon sugar with its carbon atoms labeled by $1'$ to $5'$, a nucleobase (adenine (A), cytosine (C), guanine (G) or thymine (T)) connected to the carbon labeled by $1'$, a hydroxyl group connected to the carbon labeled by $3'$, and a phosphate group connected to the carbon labeled by $5'$. For example, a nucleotide with adenine as its nucleobase is illustrated in Figure 4.1, where the five-carbon sugar is at the center, the adenine nucleobase is at the upper right, the phosphate group is at the left, and the hydroxyl group is at the bottom. The hydroxyl group of a nucleotide can form a covalent bond with the phosphate group of another nucleotide with the help of enzymes. If we have a single-stranded DNA molecule of length $m \in \mathcal{N}^+$ and a single-stranded DNA molecule of length $n \in \mathcal{N}^+$, they can form a single-stranded DNA molecule of length $m+n$ with one free phosphate group and one free hydroxyl group by forming a covalent bond. This chain of alternating sugar and phosphate groups forms the backbone of a single-stranded DNA molecule. A short DNA molecule is called an oligonucleotide.

Single-stranded DNA molecules are considered to be directional, from the free phosphate

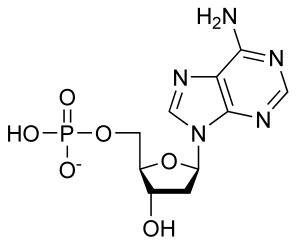

Figure 4.1: A nucleotide consists of a five-carbon sugar (center) with its carbon labeled from 1′ to 5′ clockwise, an adenine nucleobase (upper right) connected to the 1′ carbon, a phosphate group (left) connected to the 5′ carbon, and a hydroxyl group (bottom) connected to the 3′ carbon [105]

group on its 5′ end to the nucleotide with the free hydroxyl group on its 3′ end. If we read the nucleobases of a single-stranded DNA molecule in this direction, a single-stranded DNA molecule can be translated into a word over the alphabet $\Sigma = \{A, C, G, T\}$. In addition, by convention, we write the strand in the 5′ to 3′ orientation. The nucleobase C is the Watson-Crick complement the nucleobase G, and vice versa; also, the nucleobase A is the Waston-Crick complement of the nucleobase T, and vice versa [103]. For a nucleobase $a \in \Sigma$, its Watson-Crick complement is denoted by $\bar{a}$; for a word $\alpha \in \Sigma^*$, its Waston-Crick complement is denoted by $\bar{\lambda} = \lambda$, $\bar{\alpha} = a_1 a_2 ... a_k$, where $\alpha = a'_1 a'_2 ... a'_k, a_i = \bar{a'_i}$ for $1 \leq i \leq k$. We can consider the following definition.

Definition 4.1.1 ([57]) *Let $\Sigma = \{A, C, G, T\}$ be an alphabet. The mapping $\theta : \Sigma^* \to \Sigma^*$ defined by $\theta(\lambda) = \lambda, \theta(A) = T, \theta(T) = A, \theta(C) = G, \theta(G) = C, \theta(\alpha\beta) = \theta(\beta)\theta(\alpha)$, where $\alpha, \beta \in \Sigma^+$, is an involution.*

Note that θ^2 is the identity, where $\theta(\theta(\alpha)) = \alpha$, $\alpha \in \Sigma^*$. In addition, $\theta(\alpha) = \text{mi}(\bar{\alpha}), \alpha \in \Sigma^*$. With this definition, we consider double-stranded DNA molecules.

Let $\alpha, \beta \in \Sigma^*$ be two words representing single-stranded DNA molecules. The two single-stranded DNA molecules can form a partially double-stranded DNA molecule as illustrated in Figure 4.2 if and only if $\alpha = xy, \beta = \theta(yz)$, where $x, y, z \in \Sigma^+$. If $x = z = \lambda$, two single-stranded DNA molecules y and $\theta(y)$ can form a double-stranded DNA molecule. These processes are called hybridization, where the nucleobase of each nucleotide in y forms hydrogen bonds with the nucleobase of the corresponding nucleotide in $\bar{y}$.

If a single-stranded DNA molecule hybridizes with itself, a hairpin structure can be formed with intrastrand base-pairings [5]. Some often more complex structures can be engineered: 3-armed junctions result from hybridizations of each pair among three single-stranded DNA

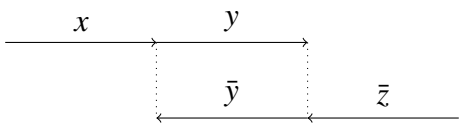

Figure 4.2: The hybridization between two non-empty single-stranded DNA molecules represented by α, β over the alphabet $\Sigma = \{A, C, G, T\}$, where $\alpha = xy, \beta = \theta(yz), x, y, z \in \Sigma^+$, results in a partially double-stranded DNA molecule

molecules [24]; double-crossover units result from hybridizations of four single-stranded DNA molecules [33], etc.

The end of a DNA molecule is called a blunt end if and only it terminates in a base pair. The end of a DNA molecule is called a sticky end if and only if it has a protruding single-stranded overhang with unpaired nucleotides. Note that in Figure 4.2, the DNA molecule has a sticky end; if $x = \lambda$, the DNA molecule has a blunt end. Denaturation is the process where all the hydrogen bonds that bind two complementary nucleobases are broken.

Example The DNA duplex in Figure 4.2 can result in two single-stranded DNA xy and $\theta(yz)$ after denaturation.

Next, we show some enzymes that can act on DNA strands.

Restriction enzymes recognize and cleave double-stranded DNA molecules at specific nucleotide sequences, called restriction sites, by breaking one covalent bond on each strand [91]. Given a solution containing double-stranded DNA molecules, restriction enzymes can cut DNA molecules at specific sites, and produce shorter double-stranded DNA molecules with blunt or sticky ends (the cut is offset, leaving single-stranded overhangs on each strand).

DNA ligases join DNA backbones to each other by creating a new covalent bond [68]. DNA ligases can fill in the backbone gap between two neighbour nucleotides in a double-stranded DNA molecule with breaks in its backbone. Note that restriction enzymes and DNA ligase are like the scissors and glue in DNA editing.

Consider a single-stranded oligonucleotide, called a primer, and a longer single-stranded DNA molecule, called a template. The primer base-pairs with the template, and DNA polymerases extend the primer at the $3'$ end according to the template by adding complementary nucleotides to the primer [67].

Example If we have a primer $\alpha \in \Sigma^+$ and a template $\beta\theta(\alpha)\gamma \in \Sigma^+$, where $\beta, \gamma \in \Sigma^*$, DNA polymerases can extend the primer to $\alpha\theta(\beta)$.

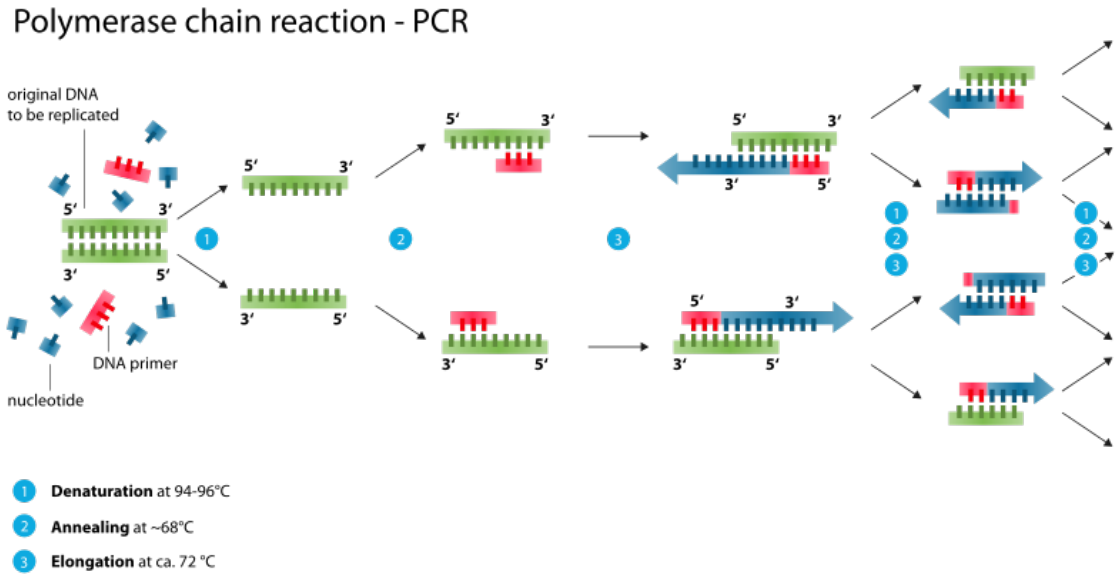

Figure 4.3: PCR: Given a solution of free nucleotides, designed primers that match the prefix of the desired sequence and the prefix of the reversed complement of the desired sequence, and the original DNA molecules, PCR replicates the desired sequence exponentially [104]

Polymerase Chain Reaction (PCR) is a technique developed by Kary Mullis to replicate a segment of DNA using DNA polymerases [4]. We can extract a desired sequence of a double-stranded DNA molecule, and replicate it exponentially by PCR as illustrated in Figure 4.3. Initially, we have a solution of the original double-stranded DNA molecule, free nucleotides and designed DNA primers that match the prefix of the desired sequence and the prefix of the reversed complement of the desired sequence. In each cycle, denaturation separates double-stranded DNA molecules into single-stranded DNA molecules; primers hybridize with the longer single-stranded DNA molecules; finally, DNA polymerases extend the primers using the free nucleotides, using the longer strands as templates. After several iterations, most of the double-stranded DNA molecules in the solution consist of the strands of the desired sequence and its reversed complement.

Next, we consider a special case of PCR, called Cross-pairing Polymerase Chain Reaction (XPCR), as illustrated in Figure 4.4 [30]. The input of XPCR are two double-stranded DNA molecules and two primers, where one primer base-pairs with one single strand of one of the double-stranded DNA molecules, the other primer hybridize with one single strand of the other double-stranded DNA molecule, and the remaining two single strands of these two double-stranded DNA molecules hybridize with each other. The output of XPCR contains the recombination of these two double-stranded DNA molecules. In other word, given two double-

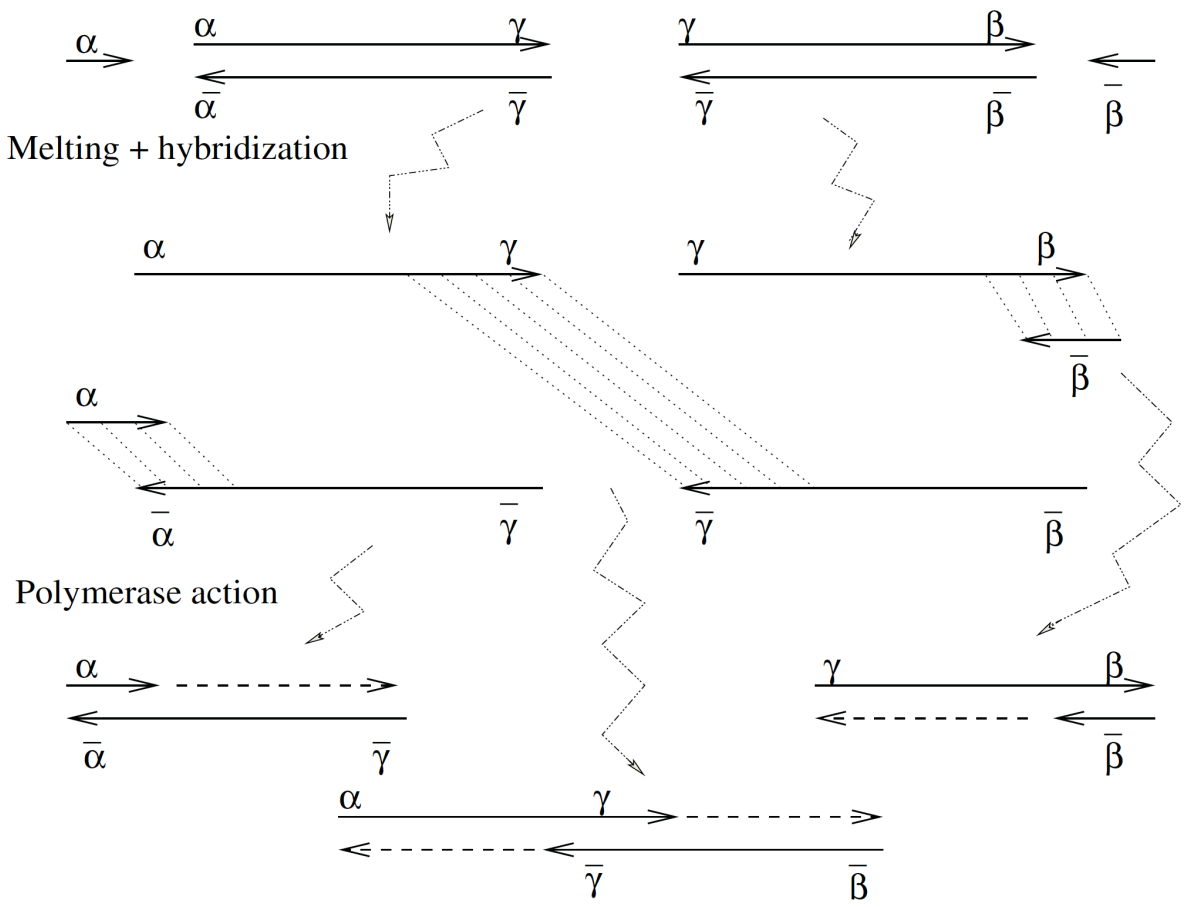

Figure 4.4: XPCR: Given two double-stranded DNA molecules $\alpha w\gamma, \gamma w'\beta$ and primers $\gamma, \theta(\beta)$ (denoted by $\bar{\beta}$ in this figure), XPCR generates the double-stranded DNA molecule $\alpha w\gamma w'\beta$ [30]

stranded DNA molecules $\alpha w\gamma, \gamma w'\beta$ and primers $\gamma, \theta(\beta)$, XPCR generates the double-stranded DNA molecule $\alpha w\gamma w'\beta$.

4.2 Biologically-Inspired Word Operations

4.2.1 Splicing

Given a solution containing double-stranded DNA molecules, restriction enzymes can cut DNA molecules at specific sites, and produce shorter strands with blunt or sticky ends (the cut is offset, leaving single-stranded overhangs on each strand). If ligase enzymes are added to the solution, some potentially new double-stranded DNA molecules can be produced. DNA recombination through the action of restriction and ligase enzymes was formalized by a word operation called splicing introduced by Head in [42].

Definition 4.2.1 *Let Σ be an alphabet. A splicing rule is a word $r = u_1\#u_2\$u_3\#u_4$, where $u_1, u_2, u_3, u_4 \in \Sigma^*, \#, \$ \notin \Sigma$.*

Note that splicing rules $r = u_1\#u_2\$u_3\#u_4$ formalize the fact that u_1u_2 and u_3u_4 represent the restriction sites of restriction enzymes on double-stranded DNA molecules, while u_1u_4 represents the double strand formed by ligating the compatible single-stranded overhangs of double-stranded DNA strands that result from cutting the input DNA strands by the two restriction enzymes with such sites u_1u_2 and u_3u_4. Given two words x and y and a splicing rule string r, a word z can be generated by applying r on x and y as follows:

$$z \in \{x_1u_1u_4y_2 \mid x = x_1u_1u_2x_2, y = y_1u_3u_4y_2, r = u_1\#u_2\$u_3\#u_4, x_1, x_2, y_1, y_2, u_1, u_2, u_3, u_4 \in \Sigma^*\}.$$

Formally, we write $(x, y) \vdash_r z$, and this process is illustrated in Figure 4.5.

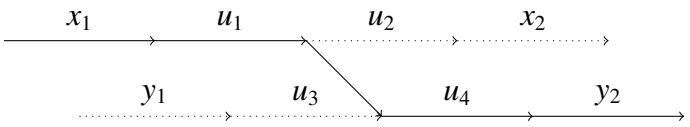

Figure 4.5: Splicing of words $x_1u_1u_2x_2$ and $y_1u_3u_4y_2$ with a splicing rule $r = u_1\#u_2\$u_3\#u_4$ results in the word $x_1u_1u_4y_2$

Definition 4.2.2 ([43]) *A splicing scheme is a pair $\sigma = (V, R)$, where V is an alphabet, and $R \subseteq V^*\#V^*\$V^*\#V^*$ is the set of splicing rules.*

Given a splicing scheme $\sigma = (V, R)$ and a language L over the alphabet V, the language obtained by applying splicing rules R to L is $\sigma(L) = \{z \in V^* \mid x, y, \in L, r \in R, (x, y) \vdash_r z\}$. In practice, only finitely many different restriction enzymes and ligase enzymes exist, so a finite splicing scheme whose set of splicing rules is finite is necessary to model any practical situation. However, it is theoretically interesting to study infinite splicing schemes which can generate languages from known families of languages. For example, given a splicing scheme $\sigma = (\{a, b\}, \{a\#a\$b\#b\})$ and a regular language $L = a^* \cup b^*$, the language $\sigma(L)$ is a^+b^+.

Next, we introduce some variations of splicing schemes.

Definition 4.2.3 ([81]) *Consider a finite splicing scheme $\sigma = (V, R)$ and a constant number $k \in \mathcal{N}^+$. The splicing scheme σ is called a $k-$limited splicing scheme if and only if for all splicing rules $r = u_1\#u_2\$u_3\#u_4 \in R$, we have that* $\max\{|u_1|, |u_2|, |u_3|, |u_4|\} \leq k$.

For example, $\sigma = (\{a\}, \{a\#a\$b\#b, a\#b\$\lambda\#a\})$ is a $1-$limited splicing scheme, which is also called a unary splicing scheme. The family of $k-$limited splicing schemes is denoted by k. For example, we have that 2 is the family of $2-$limited splicing schemes.

Definition 4.2.4 ([43]) *Consider a splicing scheme $\sigma = (V, R)$. The splicing scheme σ is called a finite (resp. regular, context-free, context-sensitive, recursively enumerable) splicing scheme if R is a finite (resp. regular, context-free, context-sensitive, recursively enumerable) language.*

The family of finite (resp. regular, context-free, context-sensitive, recursively enumerable) splicing schemes is denoted by fin (resp. reg, cf, cs, re), and we have that $1 \subset 2 \subset 3... \subset fin \subset reg \subset cf \subset cs \subset re$, since $FIN \subset REG \subset CF \subset CS \subset RE$.

Consider two families of languages $\mathcal{L}_L, \mathcal{L}_R$. The family of languages generated by applying splicing schemes $\sigma = (V, R)$ with splicing rules $R \in \mathcal{L}_R$ to languages $L \in \mathcal{L}_L$ is denoted by $S(\mathcal{L}_L, \mathcal{L}_R) = \{\sigma(L) \mid L \in \mathcal{L}_L, \sigma = (V, R), R \in \mathcal{L}_R\}$. With this notation, we have the following lemma.

Lemma 4.2.5 ([43]) *If $\mathcal{L}_L \subseteq \mathcal{L}'_L$ and $\mathcal{L}_R \subseteq \mathcal{L}'_R$, we have that $S(\mathcal{L}_L, \mathcal{L}_R) \subseteq S(\mathcal{L}'_L, \mathcal{L}'_R)$ for all $\mathcal{L}_L, \mathcal{L}'_L, \mathcal{L}_R, \mathcal{L}'_R$.*

Thus, for a family of languages $\mathcal{L}$, we have that $S(\mathcal{L}, FIN) \subseteq S(\mathcal{L}, REG) \subseteq S(\mathcal{L}, CF) \subseteq S(\mathcal{L}, CS) \subseteq S(\mathcal{L}, RE)$. With this notation, we can also define the closure property of a family of languages under splicing.

Definition 4.2.6 ([43]) *A family $\mathcal{L}_L$ of languages is said to be closed under splicing of $\mathcal{L}_R$ type if $S(\mathcal{L}_L, \mathcal{L}_R) \subseteq \mathcal{L}_L$.*

Consider a language $L \in FIN$ and a splicing scheme $\sigma \in re$, $\sigma(L) \in FIN$. For a finite set of words, there are finitely many ways to recombine them, so $S(FIN, RE) \subseteq FIN$. It follows that the family of finite languages is closed under splicing of $\mathcal{L}_R \subseteq RE$. The closure properties of various families of languages under splicing are summarized in Table 4.1 [43].

$S(\mathcal{L}_L, \mathcal{L}_R)$	$\mathcal{L}_R = FIN$	$\mathcal{L}_R = REG$	$\mathcal{L}_R = LIN$	$\mathcal{L}_R = CF$	$\mathcal{L}_R = CS$	$\mathcal{L}_R = RE$
$\mathcal{L}_L = FIN$	FIN	FIN	FIN	FIN	FIN	FIN
$\mathcal{L}_L = REG$	REG	REG	REG, LIN	REG, CF	REG, RE	REG, RE
$\mathcal{L}_L = LIN$	LIN, CF	LIN, CF	RE	RE	RE	RE
$\mathcal{L}_L = CF$	CF	CF	RE	RE	RE	RE
$\mathcal{L}_L = CS$	RE	RE	RE	RE	RE	RE
$\mathcal{L}_L = RE$	RE	RE	RE	RE	RE	RE

Table 4.1: Closure properties of various families of languages under splicing [43], where if a family of languages $\mathcal{L}$ is in the cell marked by $\mathcal{L}_L, \mathcal{L}_R$, then $S(\mathcal{L}_L, \mathcal{L}_R) \subseteq \mathcal{L}$, and if two families of language $\mathcal{L}_1, \mathcal{L}_2$ is in the cell marked by $\mathcal{L}_L, \mathcal{L}_R$, then $\mathcal{L}_1 \subseteq S(\mathcal{L}_L, \mathcal{L}_R) \subseteq \mathcal{L}_2$

Now, we consider the iterated splicing.

Definition 4.2.7 ([43]) *Let σ be a splicing scheme $\sigma = (\Sigma, R)$. The iterated splicing σ applied to a language over the alphabet Σ is $\sigma^*(L) = \bigcup_{i\geq 0} \sigma^i(L)$, where $\sigma^0(L) = L$ and $\sigma^{i+1}(L) = \sigma^i(L) \cup \sigma(\sigma^i(L))$ for $i \geq 1$.*

Consider two families of languages $\mathcal{L}_L, \mathcal{L}_R$. The family of languages generated by iteratively applying splicing scheme $\sigma = (\Sigma, R)$ with splicing rules $R \in \mathcal{L}_R$ on languages $L \in \mathcal{L}_L$ is denoted by $H(\mathcal{L}_L, \mathcal{L}_R) = \{\sigma^*(L) \mid L \in \mathcal{L}_L, \sigma = (\Sigma, R), R \in \mathcal{L}_R\}$. With this notation, we can define the closure of a family of languages under iterated splicing.

Definition 4.2.8 ([43]) *Let be a family of languages, and if $H(\mathcal{L}_L, \mathcal{L}_R) \subseteq \mathcal{L}_L$, $\mathcal{L}_L$ is said to be closed under iterated splicing of $\mathcal{L}_R$ type.*

Consider a language $L \in REG$ and a splicing scheme $\sigma \in fin$. It is proven $\sigma^*(L) \in REG$ [43], and it follows that $H(REG, FIN) \subseteq REG$. Thus, we have that the family of regular languages is closed under iterated splicing with finite splicing schemes. The closure properties of various families of languages under iterated splicing are summarized in Table 4.2 [43].

$H(\mathcal{L}_L, \mathcal{L}_R)$	$\mathcal{L}_R = FIN$	$\mathcal{L}_R = REG$	$\mathcal{L}_R = LIN$	$\mathcal{L}_R = CF$	$\mathcal{L}_R = CS$	$\mathcal{L}_R = RE$
$\mathcal{L}_L = FIN$	FIN, REG	FIN, RE	FIN, RE	FIN, RE	FIN, RE	FIN, RE
$\mathcal{L}_L = REG$	REG	REG, RE	REG, RE	REG, RE	REG, RE	REG, RE
$\mathcal{L}_L = LIN$	LIN, CF	LIN, RE	LIN, RE	LIN, RE	LIN, RE	LIN, RE
$\mathcal{L}_L = CF$	CF	CF, RE	CF, RE	CF, RE	CF, RE	CF, RE
$\mathcal{L}_L = CS$	CS, RE	CS, RE	CS, RE	CS, RE	CS, RE	CS, RE
$\mathcal{L}_L = RE$	RE	RE	RE	RE	RE	RE

Table 4.2: Closure properties of various families of languages under iterated splicing [43], where if a family of languages $\mathcal{L}$ is in the cell marked by $\mathcal{L}_L, \mathcal{L}_R$, then $H(\mathcal{L}_L, \mathcal{L}_R) \subseteq \mathcal{L}$, and if two families of language $\mathcal{L}_1, \mathcal{L}_2$ is in the cell marked by $\mathcal{L}_L, \mathcal{L}_R$, $\mathcal{L}_1 \subseteq H(\mathcal{L}_L, \mathcal{L}_R) \subseteq \mathcal{L}_2$

Note that $FIN \subseteq H(FIN, RE) \subseteq RE$ because for any recursively enumerable language L over an alphabet Σ, there exists a language $L' \subseteq H(FIN, RE)$ over an arbitrary alphabet such that $L = L' \cap \Sigma^*$ [82].

Next, we introduce systems that use iterated splicing to generate words.

Definition 4.2.9 ([75]) *A simple H system is a triple $G = (V, M, A)$, where V is an alphabet, $M \subseteq V$ is the set of markers that indicate the possible locations of splicing, and A is a finite language. Let σ_M be the splicing scheme of G such that $\sigma_M = (V, \{a\#\$a\# \mid a \in M\})$. The language generated by the simple H system G is defined by $L(G) = \sigma_M^*(A)$.*

Since we known that $H(FIN, FIN) \subseteq REG$ as shown in Table 4.2, and σ_M and A are finite for simple H systems $G = (V, M, A)$, it follows that $L(G)$ is regular. For example, let G be a simple H system, where $G = (\{a, b, c\}, \{a, b\}, \{abcabc\})$, and $L(G) = \{abc\}^*$ which is regular.

Note that the generative capacity of a simple H system is limited. Next, we will introduce a more general H system that may use an arbitrarily large set of rules. There are several definitions of extended H systems as shown in [42, 84, 88], and we show Păun's definition here.

Definition 4.2.10 ([88]) *An extended H system is a quadruple $H = (V, T, A, R)$, where V is an alphabet, $T \subseteq V$ is the terminal alphabet, A is a language over the alphabet V, and $R \subseteq V^*\#V^*\$V^*\#V^*$ is the set of splicing rules. Let $\sigma = (V, R)$ be the splicing scheme of H. The language generated by the extended H system H is defined by $L(H) = \sigma^*(A) \cap T^*$.*

Note that extended H systems $H = (V, T, A, R)$ with regular splicing rules R and a finite language A can generate recursively enumerable languages [82, 87]. Thus, extended H systems are computationally universal.

4.2.2 Overlap Assembly

Cross-paring Polymerase Chain Reaction (XPCR) amplifies the desired DNA sequence $\alpha A\gamma B\beta$ from two double-stranded DNA molecules whose single strands are $\alpha A\gamma, \gamma B\beta$ and their reversed complements [30]. The main product of XPCR can be modeled as the following process.

Let $x = \alpha\gamma, y = \theta(\gamma\beta)$ be two single-stranded DNA molecules. In a suitable environment, x and y can form partially double-stranded DNA molecules. With DNA polymerase enzymes, a completely double-stranded DNA molecule is generated, and two new single-stranded DNA molecules $\alpha\gamma\beta$ and $\theta(\alpha\gamma\beta)$ are produced after denaturation.

This process was formalized as a binary word operation called linear self-assembly in [18], the chop operation in [46] and overlap assembly in [26]. It can also be considered as a special case of a semantic shuffle on trajectories, studied in [22]. The iteration of this process is used to generate double-stranded DNA molecules which are combinations of shorter, partially double-stranded DNA molecules [53].

Definition 4.2.11 ([26]) *Given two words x, y over an alphabet Σ, the overlap assembly of x with y is defined by $x \overline{\odot} y = \{uvw \mid x = uv, y = vw, u, w \in \Sigma^*, v \in \Sigma^+\}$ (illustrated in Figure 4.6).*

The definition of overlap assembly can be extended to languages L_x, L_y as $L_x \overline{\odot} L_y = \bigcup_{x \in L_x, y \in L_y} x \overline{\odot} y$.

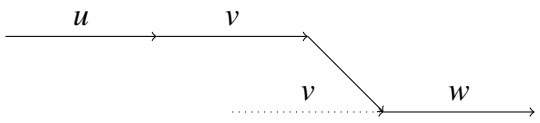

Figure 4.6: The overlap assembly of two words uv and vw over an alphabet Σ, where $u, w \in \Sigma^*, v \in \Sigma^+$, results in the word uvw

The closure properties of the Chomsky families of languages under overlap assembly are studied in [26], and are summarized in Table 4.3. The state complexity of overlap assembly is studied in [9].

$L_L \,\overline{\odot}\, L_R$	$L_R \in FIN$	$L_R \in REG$	$L_R \in CF$	$L_R \in CS$	$L_R \in RE$
$L_L \in FIN$	FIN	REG	CF	CS	RE
$L_L \in REG$	REG	REG	CF	CS	RE
$L_L \in CF$	CF	CF	CS	CS	RE
$L_L \in CS$	CS	CS	CS	CS	RE
$L_L \in RE$	RE	RE	RE	RE	RE

Table 4.3: Closure properties of various families of languages under overlap assembly

Next, we consider the iterated version of overlap assembly in the context of the assembly operation defined in [18].

Definition 4.2.12 ([18]) *The iterated version of assembly on a language L over an alphabet Σ is denoted by $\mathcal{S}^+(L) = \bigcup_{n\geq 1} \mathcal{S}^n(L)$, where $\mathcal{S}^n(L) = L \,\overline{\odot}\, \mathcal{S}^{n-1}(L)$, $\mathcal{S}^0(L) = L$.*

The closure properties of the Chomsky families of languages under iterated assembly are studied in [18], and are summarized in Table 4.4.

$L \in \mathcal{L}$	$\mathcal{S}^+(L)$
$L \in REG$	REG
$L \in CF$	CF
$L \in CS$	RE
$L \in RE$	RE

Table 4.4: Closure properties of the Chomsky families of languages under iterated assembly

Some variations of the overlap assembly, assembly and iterated assembly are defined as follows.

If we require that the non-overlapping parts of each word cannot be empty, we have a binary word operation called restricted assembly operation.

Definition 4.2.13 ([18]) *Let x, y be non-empty words over an alphabet Σ. The binary word operation called restricted assembly operation $\mathcal{R}(x, y)$ is defined by $\mathcal{R}(x, y) = \{uvw \mid x = uv, y = vw, u, v, w \neq \lambda\}$. This definition can be extended to languages $\mathcal{R}(L_1, L_2) = \bigcup_{x_1 \in L_1, x_2 \in L_2} \mathcal{R}(x_1, x_2)$ for languages L_1, L_2 over an alphabet Σ. The iterated version of restricted assembly to a language L over an alphabet Σ is denoted by $\mathcal{R}^+(L) = \bigcup_{n \geq 1} \mathcal{R}^n(L)$, where $\mathcal{R}^n(L) = \mathcal{R}(L, \mathcal{R}^{n-1}(L))$, $\mathcal{R}^0(L) = L$.*

If we require that the non-overlapping parts of each word cannot be empty with no restrictions on the overlapping parts, we have a binary word operation denoted by $\bigotimes$.

Definition 4.2.14 ([51]) *Let x, y be arbitrary words over an alphabet Σ. We can define that $x \bigotimes y = \{uvw \mid uw \neq \lambda, x = uv, y = vw\}$. This definition can be extended to languages $L_1 \bigotimes L_2 = \bigcup_{x_1 \in L_1, x_2 \in L_2} x_1 \bigotimes x_2$ for languages L_1, L_2 over an alphabet Σ. The iterated version of $\bigotimes$ to words y over an alphabet Σ is denoted by $p^{\bigotimes k+1} = \bigcup_{x \in p^{\bigotimes k}} x \bigotimes y$ for $k \in \mathcal{N}$, where $p^{\bigotimes 0} = \{\lambda\}$.*

Note that $a^* \bigotimes b^* = \{a^+b^*\} \cup \{a^*b^+\}$, where $a^* \overline{\odot} b^* = \emptyset$.

If we choose the longest or the shortest non-empty overlapped part when we perform overlap assembly on two non-empty words x, y over an alphabet Σ, we have the following definitions.

Definition 4.2.15 ([46]) *Let x, y be non-empty words over an alphabet Σ. The binary word operation called max chop $\bigodot_{max}$ is defined by $x \bigodot_{max} y = \{uvw \mid x = uv, y = vw, v \in \Sigma^+, \forall u', v', w' : x = u'v', y = v'w', |v'| \leq |v|\}$.*

Definition 4.2.16 ([46]) *Let x, y be non-empty words over an alphabet Σ. The binary word operation called min chop $\bigodot_{min}$ is defined by $x \bigodot_{min} y = \{uvw \mid x = uv, y = vw, v \in \Sigma^+, \forall u', v', w' : x = u'v', y = v'w', |v'| \geq |v|\}$.*

The max and min chop operations can be extended to languages naturally. Consider a regular language $L = a^+b^+a^+$, we have that $L \bigodot_{max} L = a^+b^+a^+$ and $L \bigodot_{min} L = a^+b^+a^+b^+a^+$. Note that if we allow the overlapped part be empty in the max chop operation, we define an operation called short concatenation [10].

If we choose the length of the overlapped part be at least $N \in \mathcal{N}^+$, we have the following definition.

Definition 4.2.17 ([23]) *Let x, y be non-empty words over an alphabet Σ, and N be a positive integer. We can define that $x \bigodot_N y = \{uvw \mid x = uv, y = vw, |v| \geq N\}$.*

If we consider the case that the overlapped part is a single character, we have the following definition.

Definition 4.2.18 ([45]) *Let x, y be non-empty words over an alphabet Σ. The binary word operation called fusion $\bigodot$ is defined by $x \bigodot y = \{uvw \mid x = uv, y = vw, v \in \Sigma\}$.*

This operation has been called the Latin product in [38].

4.2.3 Contextual Insertion/Deletion

In this subsection, we introduce two language operations called contextual insertion and contextual deletion defined and studied in [60]. Both bio-operations can potentially be implemented in the lab using PCR site-specific oligonucleotide mutagenesis [21].

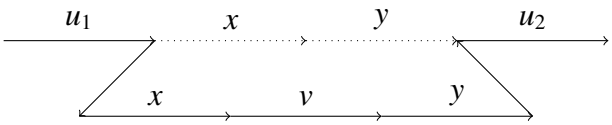

Figure 4.7: The $(x, y)-$contextual insertion of the word v into the word $u = u_1xyu_2$ result in the word u_1xvyu_2

Definition 4.2.19 ([60]) *Let Σ be an alphabet. A pair $(x, y) \in \{(\Sigma^*, \Sigma^*)\}$ is called a context. The $(x, y)-$contextual insertion of v into u is defined by $u \underset{(x,y)}{\longleftarrow} v = \{u_1xvyu_2 \mid u = u_1xyu_2, u_1, u_2, v \in \Sigma^*\}$ (illustrated in Figure 4.7).*

Note that if the context is (λ, λ), we have the regular insertion operation. Moreover, if xy is not an infix of u, the result of contextual insertion is empty.

If we have a set of given contexts, we have the following definition.

Definition 4.2.20 ([60]) *Let Σ be an alphabet, and $C \subseteq \{(x, y) \mid x, y \in \Sigma^*\}$ be a set of given contexts. The $C-$contextual insertion of v into u is defined by $u \underset{C}{\leftarrow} v = \{u_1xvyu_2 \mid u = u_1xyu_2, (x, y) \in C, u_1, u_2, v \in \Sigma^*\}$.*

Note that the definition of $C-$contextual insertion can be extended to languages L_1, L_2 as $L_1 \underset{C}{\leftarrow} L_2 = \bigcup_{u \in L_1, v \in L_2} u \underset{C}{\leftarrow} v$. If the context set C is finite, the $C-$contextual insertion is called

finite $C-$contextual insertion. All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under finite $C-$contextual insertion [60].

A related operation, contextual deletion, can be defined as follows.

Definition 4.2.21 ([60]) *Let Σ be an alphabet, and $(x, y) \in \{(\Sigma^*, \Sigma^*)\}$ be a context. The (x, y)-contextual deletion of v from u is defined by $u \underset{(x,y)}{\longrightarrow} v = \{u_1xyu_2 \mid u = u_1xvyu_2, u_1, u_2, v \in \Sigma^*\}$ (illustrated in Figure 4.8).*

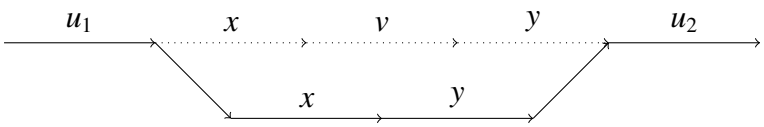

Figure 4.8: The $(x, y)-$contextual deletion of the word v from the word $u = u_1xvyu_2$ results in the word u_1xyu_2

Definition 4.2.22 ([60]) *Let Σ be an alphabet, and $C \subseteq \{(x, y) \mid x, y \in \Sigma^*\}$ be a set of given contexts. The $C-$contextual deletion of v from u is defined by $u \underset{C}{\rightarrow} v = \{u_1xyu_2 \mid u = u_1xvyu_2, (x, y) \in C, u_1, u_2, v \in \Sigma^*\}$, and it can be extended to languages as $L_1 \underset{C}{\rightarrow} L_2 = \bigcup_{u \in L_1, v \in L_2} u \underset{C}{\rightarrow} v$.*

Note that the $(\lambda, \lambda)-$contextual deletion is sequential deletion. $\mathcal{L}_0$ and $\mathcal{L}_3$ are closed under finite $C-$contextual deletion, and $\mathcal{L}_2$ is closed under finite $C-$contextual deletion with regular languages, but neither $\mathcal{L}_2$ nor $\mathcal{L}_1$ is closed under finite $C-$contextual deletion [60].

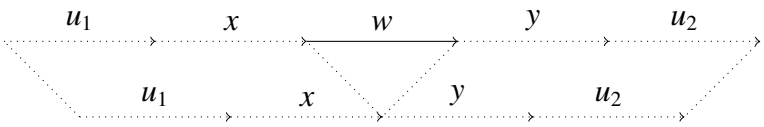

Figure 4.9: The $(x, y)-$contextual dipolar deletion of the word $v = u_1xyu_2$ from the word $u = u_1xwyu_2$ results in the word w

A similar operation, contextual dipolar deletion, can be defined as follows.

Definition 4.2.23 ([60]) *Let Σ be an alphabet, and $(x, y) \in \{(\Sigma^*, \Sigma^*)\}$ be a context. The (x, y)-contextual dipolar deletion of v from u can be defined by $u \underset{(x,y)}{\rightleftharpoons} v = \{w \mid u = u_1xwyu_2, v = u_1xyu_2, u_1, u_2, w \in \Sigma^*\}$ (illustrated in Figure 4.9).*

Let $C \subseteq \{(x, y) \mid x, y \in \Sigma^*\}$ be a set of given contexts. The $C-$contextual dipolar deletion $\underset{C}{\rightleftharpoons}$ of v from u is defined by $u \underset{C}{\rightleftharpoons} v = \{w \mid u = u_1xwyu_2, v = u_1xyu_2, u_1, u_2, w \in \Sigma^*, (x, y) \in C\}$, and it can be extended to languages as $L_1 \underset{C}{\rightleftharpoons} L_2 = \bigcup_{u \in L_1, v \in L_2} u \underset{C}{\rightleftharpoons} v$.

It is shown in [60] that $\mathcal{L}_0$ and $\mathcal{L}_3$ are closed under finite $C-$contextual dipolar deletion, but neither $\mathcal{L}_1$ nor $\mathcal{L}_2$ is closed under finite $C-$contextual dipolar deletion.

A system using contextual insertion and deletion is defined and proven to be computationally universal [59, 60, 83, 99].

The languages that are closed under contextual insertion and deletion, and the decidability of the existence of solutions to equations of the form $L \diamond Y = R$ and $X \diamond L = R$, where $\diamond$ is either contextual insertion or contextual deletion, were also studied in [60].

4.2.4 Block Substitution

Errors may occurr in DNA encoded information, where part of a DNA strand is replaced with another strand of the same length, as illustrated in Figure 4.10, and this process can be modeled by the bio-operation called block substitution introduced in [58].

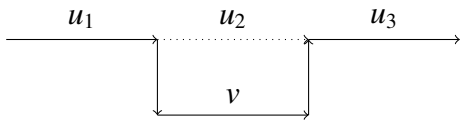

Figure 4.10: The block substitution in the word $u = u_1u_2u_3$ by the word v , where $|v| = |u_2|$, results in the word u_1vu_3

Note that in this process we only consider one such occurrence of error, and the resulting strand can be the same as the original strand.

Definition 4.2.24 ([58]) *Let Σ be an alphabet, and u, v be words over the alphabet. The block substitution in u by v is defined by $u \bowtie_b v = \{u_1vu_3 \mid u = u_1u_2u_3, |u_2| = |v|, u_1, u_2, u_3 \in \Sigma^*\}$.*

Note that v represents the error introduced into u.

Definition 4.2.25 ([58]) *Let Σ be an alphabet, and u, v be words over the alphabet. The block substitution of v in u is defined by $u \triangle_b v = \{u_1u_2u_3 \mid u = u_1vu_3, |u_2| = |v|, u_1, u_2, u_3 \in \Sigma^*\}$.*

Note that v represents the subword of u that is the introduced error, and that the length of words after block substitution is the same as the original. We have another operation $\triangleright_b$ to extract the error introduced by block substitution.

Definition 4.2.26 ([58]) *Let u, v be two words over an alphabet Σ. The operation $\triangleright_b$ is defined by $u \triangleright_b v = \{\beta \mid u = u_1\alpha u_2, v = u_1\beta u_2, |\alpha| = |\beta|, u_1, u_2, \alpha, \beta \in \Sigma^*\}$.*

The closure properties under $\bowtie_b, \triangle_b, \triangleright_b$ are studied in [58], and are summarized as follows:

- All families $\mathcal{L}_i$, $0 \leq i \leq 3$, are closed under $\bowtie_b, \triangle_b, \triangleright_b$ with regular languages;
- All families $\mathcal{L}_i$, $i = 0, 1, 3$, are closed under $\bowtie_b, \triangle_b$, but not $\mathcal{L}_2$;
- Both families $\mathcal{L}_0, \mathcal{L}_3$ are closed under $\triangleright_b$, but neither families $\mathcal{L}_1, \mathcal{L}_2$.

Note that the operation $\bowtie_b$ is the left inverse of $\triangle_b$, and that the operation $\triangleright_b$ is the right inverse of $\bowtie_b$ [58].

4.2.5 Hairpin Completion

A single-stranded DNA molecule w may form a hairpin structure by Waston-Crick base pairing [5] with itself. There are three possible structures we consider, given by $w = \alpha\beta\theta(\alpha), w = \alpha\beta\theta(\alpha)\theta(\gamma)$, or $w = \gamma\alpha\beta\theta(\alpha)$, as illustrated in Figure 4.11. A method using DNA hairpin formation to test the satisfiability of a given Boolean formula using laboratory techniques was proposed in [92], and hairpin related languages were studied in [11].

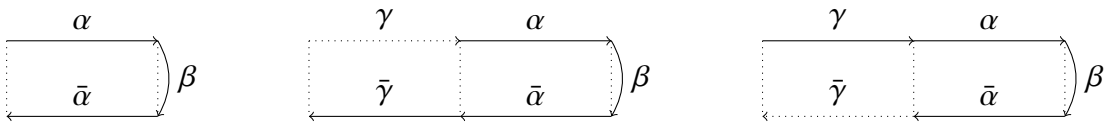

Figure 4.11: Possible hairpin structures that can be formed from single-stranded DNA molecules $\alpha\beta\theta(\alpha), \alpha\beta\theta(\alpha)\theta(\gamma), \gamma\alpha\beta\theta(\alpha)$

The sticky ends can be used as a template by the DNA polymerase enzyme to completely extend the strand to the other end, which results in hairpin structures with blunt ends. Thus, we have that $\alpha\beta\theta(\alpha)\theta(\gamma)$ and $\gamma\alpha\beta\theta(\alpha)$ can be extended to $\gamma\alpha\beta\theta(\alpha)\theta(\gamma)$. This process is formalized as a unary word operation called hairpin completion $\rightarrow$ defined in [12].

Definition 4.2.27 ([12]) *Let Σ be an alphabet, $k \in \mathcal{N}^+$ be a positive integer, and $w \in \Sigma^*$ be a word. The $k-$hairpin completion $\rightarrow_k$ of w is defined by*

$$w \rightarrow_k = w \rightharpoonup_k \cup w \rightharpoondown_k,$$

where

$$w \rightharpoonup_k = \{\gamma w \mid w = \alpha\beta\theta(\alpha)\theta(\gamma), |\gamma| = k, \alpha, \beta \in \Sigma^+, \gamma \in \Sigma^*\},$$

$$w \rightharpoondown_k = \{w\theta(\gamma) \mid w = \gamma\beta\theta(\alpha), |\gamma| = k, \alpha, \beta \in \Sigma^+, \gamma \in \Sigma^*\}.$$

Note that $\rightarrow_k$ considers hairpin structures with sticky ends of length k, where $\rightharpoonup_k$ considers hairpin structures with sticky ends of length k at the $3'$ end, and $\leftharpoondown_k$ considers hairpin structures with sticky ends of length k at the $5'$ end. Because the hairpin completion operation considers all lengths, we have the following definition.

Definition 4.2.28 ([12]) *Let Σ be an alphabet, and $w \in \Sigma^*$ be a word. The hairpin completion of w is defined by $w \rightarrow = \bigcup_{k\geq 1} w \rightarrow_k$.*

The hairpin completion operation can be extended to languages L as $L \rightarrow_k = \bigcup_{w\in L} w \rightarrow_k$, $L \rightarrow = \bigcup_{w\in L} w \rightarrow$. It is shown in [12] that $\mathcal{L}_0$ and $\mathcal{L}_1$ are closed under hairpin completion, but $\mathcal{L}_2$ and $\mathcal{L}_3$ are not.

Next, we consider the iterated version of hairpin completion.

Definition 4.2.29 ([12]) *Let Σ be an alphabet, k be a positive integer, and $w \in \Sigma^*$ be a word. The iterated version of the $k-$hairpin completion of w is defined by*

$$w(\rightarrow_k)^* = \bigcup_{n\in\mathcal{N}} w(\rightarrow_k)^n,$$

where $w(\rightarrow_k)^{n+1} = (w(\rightarrow_k)^n) \rightarrow_k$ for $n \geq 0$, $w(\rightarrow_k)^0 = \{w\}$. It is extended to languages L as $L(\rightarrow_k)^* = \bigcup_{w\in L} w(\rightarrow_k)^*$.

Definition 4.2.30 ([12]) *Let Σ be an alphabet, and $w \in \Sigma^*$ be a word. The iterated version of the hairpin completion of w is defined by*

$$w(\rightarrow)^* = \bigcup_{n\in\mathcal{N}} w(\rightarrow)^n,$$

where $w(\rightarrow)^{n+1} = (w(\rightarrow)^n) \rightarrow$ for $n \geq 0$, $w(\rightarrow)^0 = \{w\}$. It is extended to languages L as $L(\rightarrow)^* = \bigcup_{w\in L} w(\rightarrow)^*$.

It was shown in [12] that $\mathcal{L}_0$ and $\mathcal{L}_1$ are closed under iterated hairpin completion, but $\mathcal{L}_2$ and $\mathcal{L}_3$ are not. Bounded hairpin completion and bounded iterated hairpin completion were defined and studied in [50, 62], where the maximum length of sticky ends are bounded by a constant.

Removing, rather than completing, the sticky ends from strands with hairpin structures was formalized as a unary word operation called hairpin reduction, and was studied in [73, 74]. The (iterated) bounded hairpin reduction was studied in [50]. Hairpin lengthening was introduced to formalize the case where hairpin extension occurs but may not be complete (possibly resulting in hairpin structures with sticky ends), and was studied in [71]. Its iterated version was studied in [72]. The case where hairpin reduction is not complete was formalized as a word operation, called hairpin shortening, and studied in [72], and the (iterated) bounded hairpin shortening was studied in [72].

4.2.6 Template-Directed Extension

The DNA polymerase enzyme extends a short single-stranded DNA molecule called a primer according to a longer single-stranded DNA molecule called the template. This process can be modeled by template-directed extension introduced in [27].

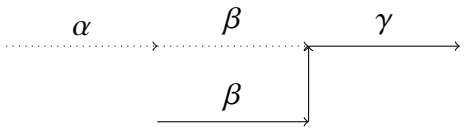

Figure 4.12: The $x-$directed extension that extends the primer word β according to the template word $x = \alpha\beta\gamma$ results in the word $\beta\gamma$

Definition 4.2.31 ([27]) *Let Σ be an alphabet, $x \in \Sigma^*$ be the template, and $y \in \Sigma^*$ be the primer. The $x-$directed extension of y is defined by $x \bigoplus y = \{w \in \Sigma^* \mid x = \alpha y\beta, w = y\beta, \alpha, \beta \in \Sigma^*\}$ (illustrated in Figure 4.12).*

Note that we consider the case that the template and primer are λ, which is not realistic in the real world. The x-directed extension of y is extended to languages L_1, L_2 by $L_1 \bigoplus L_2 = \bigcup_{x \in L_1, y \in L_2} x \bigoplus y$. The closure properties of various families of languages under template-directed extension is summarized in Table 4.5 [27].

$L_1 \bigoplus L_2$	FIN or REG	CF	CS	RE
REG	REG	CF	CS	RE
CF	CF	CS	CS	RE
CS	RE	RE	RE	RE
RE	RE	RE	RE	RE

Table 4.5: Closure properties of various families of languages under template-directed extension

Chapter 5

Word blending in formal languages: The Brangelina effect

In this chapter[1], we define and investigate a binary word operation that formalizes an experimentally observed outcome of DNA computations, performed to generate a small gene library and implemented using a DNA recombination technique called Cross-pairing Polymerase Chain Reaction (XPCR). The word blending between two words xwy_1 and y_2wz that share a non-empty overlap w, results in xwz. We study closure properties of families in the Chomsky hierarchy under word blending, language equations involving this operation, and its descriptional state complexity when applied to regular languages. Interestingly, this phenomenon has been observed independently in linguistics, under the name "blend word", "mot-valise" or "portmanteau", and is responsible for the creation of words in the English language such as smog (*smo*ke + f*og*), labradoodle (*labrado*r + p*oodle*), and Brangelina (*Bra*d + *Angelina*).

5.1 Introduction

Cross-pairing Polymerase Chain Reaction (XPCR) is an experimental DNA protocol introduced in [30] for extracting all the strands containing a given substrand from a heterogeneous pool of DNA strands. XPCR was then employed to implement several DNA recombination algorithms [32], for the creation of the solution space for a SAT problem [28], and for mutagenesis [31]. The combinatorial power of such a technique has been explained by logical-symbolic schemes in [70], while algorithms to create combinatorial libraries were improved

[1]The chapter is adapted from the paper "Word blending in formal languages: The Brangelina effect" accepted for publication in the proceedings of the seventeenth International Conference on Unconventional Computation and Natural Computation, and is co-authored with Dr. Srujan Kumar Enaganti, Dr. Lila Kari and Dr. Timothy Ng

and experimented in [29, 31].

The formal language operation called overlap assembly, introduced in [18] under the name of self-assembly, and further investigated in [9, 25, 26], also models a special case of XPCR: The overlap assembly of two strings αx and $x\beta$ that share a non-empty overlap x, results in the string $\alpha x\beta$. A particular case of overlap assembly, called "chop operation", where the overlap consists of a single letter, was studied in [44, 45], and generalized to an arbitrary length overlap in [46]. Other similar operations have been studied in the literature, such as the "short concatenation" [10], which uses only the maximum-length (possibly empty) overlap y between operands, the "Latin product" of words [38] where the overlap consists of only one letter, and the operation $\bigotimes$ which imposes the restriction that the non-overlapping part x is not empty [51]. Overlap assembly can also be considered as a particular case of "semantic shuffle on trajectories" with trajectory $0^*\sigma^+1^*$ or as a generalization of the operation $\bigodot_N$ from [23] which imposes the length of the overlap to be at least N. Many similar biological phenomena and operations can also be modeled using splicing systems [85, 86]. However, modeling these operations often does not require the full power of splicing. Properties of splicing languages under restrictions such as symmetry and reflexivity have been studied in [6, 39].

Returning to the biological process that motivated the study of overlap assembly, the XPCR procedure has been successfully used to join two different genes if they are attached to compatible primers [29]. Formally, $\alpha A\gamma$ and $\gamma D\beta$ were combined to produce $\alpha A\gamma D\beta$ (here A and D are gene sequences and α, γ and β are primers used). However, when $A = D$, that is, when two sequences containing the same gene were combined by XPCR, the result was not as expected. More specifically, when using XPCR with two strings $\alpha A\gamma$ and $\gamma A\beta$, instead of obtaining the expected $\alpha A\gamma A\beta$, the experiments repeatedly produced the result $\alpha A\beta$.

In this chapter, we define and investigate a formal language operation called word blending, that formalizes this experimentally observed outcome of XPCR: The word blending of two words xAy_1 and y_2Az that share a non-empty overlap A results in xAz. Interestingly, this phenomenon has been observed independently in linguistics [41], under the name "blend word", "mot-valise" or "portmanteau", and is responsible for the creation of words in the English language such as smog (*smo*ke + f*og*), labradoodle (*labrado*r + po*odle*), emoticon (*emoti*on + *icon*), and Brangelina (*Bra*d + *Angelina*).

The chapter is organized as follows. Section 5.2 details the biological motivation behind the study of word blending, and introduces the main definitions and notations. Section 5.3 studies closure properties of the Chomsky families of languages under word blending, its right and left inverses, as well as iterated word blending. Section 5.4 investigates the decidability of existence of solutions to some language equations involving word blending, and Section 5.5 studies the descriptional state complexity of this operation when applied to regular languages.

5.2 Preliminaries

The biological phenomenon we model in this chapter was observed during the XPCR-based experiments, initially intended to achieve the concatenation of two or more genes (genomic DNA strands). It was observed in [29] that, in the particular case where the two genes to be concatenated were one and the same, that is, when the two input DNA strands were $\alpha A\gamma$ and $\gamma A\beta$ (here A represents a gene sequence), the output of a PCR-based amplification with primers α and β was $\alpha A\beta$. This output was different from the expected $\alpha A\gamma A\beta$, which had been the anticipated result. (Indeed, experiments using XPCR for the purpose of concatenating two different genes A and D flanked by primers, that is, when the two input strands were $\alpha A\gamma$ and $\gamma D\beta$, had resulted in the output $\alpha A\gamma D\beta$. This "expected" output of XPCR was modeled by the previously mentioned operation of overlap assembly, given by $\alpha A\gamma + \gamma D\beta = \alpha A\gamma D\beta$.)

Generalizing this experimentally newly-observed phenomenon to the case where the end words of the input strings are different, we model this string recombination as follows. Given two non-empty words x, y over an alphabet Σ, we define the word blending, or simply blending, of x with y as

$$x \bowtie y = \{z \in \Sigma^+ \mid \exists \alpha, \beta, \gamma_1, \gamma_2 \in \Sigma^*, \exists w \in \Sigma^+ : x = \alpha w \gamma_1, y = \gamma_2 w \beta, z = \alpha w \beta\}.$$

The definition of blending can be extended to languages L_1 and L_2 by

$$L_1 \bowtie L_2 = \bigcup_{x \in L_1, y \in L_2} x \bowtie y.$$

Note that, for a realistic model, we would need additional restrictions such as the fact that w, γ_1 and γ_2 should be of a sufficient length and should not appear as a substring in the other strings involved.

We can also extend the blending operation to an iterated version on a language. Let $L \subseteq \Sigma^*$ be a language. We define the iterated (word) blending of L by $L^{\bowtie_0} = L$ and $L^{\bowtie_i} = L \bowtie L^{\bowtie_{i-1}}$. We define the iterated blending closure of L by

$$L^{\bowtie_*} = \bigcup_{i \geq 0} L^{\bowtie_i}.$$

We observe that the result of the iterated blending operation can be generated by a splicing system with null context splicing rules [42]. Splicing rules in [42] are of the form $(u_1, z, u_2; u_3, z, u_4)$. For such a rule, if we have strings $x = x_1 u_1 z u_2 x_2$ and $y = y_1 u_3 z u_4 y_2$, we obtain the word $x_1 u_1 z u_4 y_2$. A splicing rule is a null context rule when $u_1, u_2, u_3, u_4 = \lambda$. It is easy to see that the language $L^{\bowtie_*}$ can be generated from a splicing scheme with rules of the form $(\lambda, w, \lambda; \lambda, w, \lambda)$ for every word $w \in \Sigma^+$. The relationship between iterated blending and splicing will be discussed in greater detail in Section 5.3.

5.3 Closure Properties

In this section, we prove that the families of regular, context-free and recursively enumerable languages are closed under blending, and that the family of context-sensitive languages is not. The section also contains closure properties of Chomsky families of languages under the right and left inverse of word blending, as well as under iterated word blending.

The following lemma shows that word blending is equivalent to a special version where only one-letter overlaps are utilized.

Lemma 5.3.1 *If x, y are non-empty words in Σ^+, then*

$$x \bowtie y = \{z \in \Sigma^+ \mid \exists \alpha, \beta, \gamma_1, \gamma_2 \in \Sigma^*, \exists a \in \Sigma : x = \alpha a \gamma_1, y = \gamma_2 a \beta, z = \alpha a \beta\}.$$

This result can be extended to languages in the natural way. Then from this lemma, we can show that the word blending of two languages can be obtained by combining right quotient, concatenation, left quotient and union operations, as follows.

Proposition 5.3.2 *Given languages $L_1, L_2 \subseteq \Sigma^+$,*

$$L_1 \bowtie L_2 = \bigcup_{a \in \Sigma} \left(L_1 (a\Sigma^*)^{-1}\right) a \left((\Sigma^* a)^{-1_l} L_2\right).$$

Proof ($\subseteq$) Let $z \in L_1 \bowtie L_2$. Then, by Lemma 5.3.1, $z = \alpha a \beta$, for some $x \in L_1$ and $y \in L_2$ such that $x = \alpha a \gamma_1$, $y = \gamma_2 a \beta$ where $a \in \Sigma, \alpha, \beta, \gamma_1, \gamma_2 \in \Sigma^*$. It is clear that $\alpha \in L_1 (a\Sigma^*)^{-1}$ and $\beta \in (\Sigma^* a)^{-1_l} L_2$, so $z = \alpha a \beta \in \left(L_1 (a\Sigma^*)^{-1}\right) a \left((\Sigma^* a)^{-1_l} L_2\right)$.
($\supseteq$) Let $z \in \bigcup_{a \in \Sigma} \left(L_1 (a\Sigma^*)^{-1}\right) a \left((\Sigma^* a)^{-1_l} L_2\right)$. Then there exists $a \in \Sigma$ and words $\alpha, \gamma_1, \gamma_2, \beta \in \Sigma^*$, such that $z = \alpha a \beta$, where $x = \alpha a \gamma_1 \in L_1$, $y = \gamma_2 a \beta \in L_2$, which implies that $z \in L_1 \bowtie L_2$. ∎

Corollary 5.3.3 *Every full AFL is closed under word blending.*

We note that the families of regular languages, context-free languages and recursively enumerable languages are all full AFLs [93].

Proposition 5.3.4 *The family of context-sensitive languages is not closed under word blending.*

Proof Let L_0 be a recursively enumerable language over Σ, that is not context-sensitive. It is known that a context-sensitive language L_1 over $\Sigma \cup \{a, b\}$ with $a, b \notin \Sigma$, can be constructed such that L_1 consists of words of the form Pba^i where $i \geq 0$ and $P \in L_0$ and, in addition, for every $P \in L_0$ there is an $i \geq 0$ such that $Pba^i \in L_1$ (see, e.g., [93]).

Since it is obvious that $L_1 \bowtie \{b\} = \{Pb \mid P \in L_0\}$, which is not context sensitive, it follows that the family of context sensitive languages is not closed under word blending with singleton words. ∎

From Lemma 5.3.1, we can also show that the word blending of two languages can be obtained by combining inverse homomorphism, union, intersection with regular languages, concatenation and finite splicing, as follows.

Proposition 5.3.5 *Consider languages $L_1, L_2 \subseteq \Sigma^+$, two special letters $!, ? \notin \Sigma$, a homomorphism h_1 defined by $h_1(!) = \lambda$, $h_1(a) = a$ for $a \in \Sigma$, another homomorphism h_2 defined by $h_2(?) = \lambda$, $h_2(a) = a$ for $a \in \Sigma$,, and a splicing scheme $\sigma = (\Sigma, \{a\#!\$a\#? \mid a \in \Sigma\})$. The word blending can be represented as $L_1 \bowtie L_2 = h_2(\sigma(L))$, where*

$$L = (h_1^{-1}(L_1) \cap (\Sigma^*!\Sigma^*)) \cup (h_2^{-1}(L_2) \cap (\Sigma^*?\Sigma^*)).$$

Proof ($\subseteq$) Let $z \in L_1 \bowtie L_2$. Then by Lemma 5.3.1, $z = \alpha a \beta$, for some $x \in L_1$ and $y \in L_2$ such that $x = \alpha a \gamma_1$, $y = \gamma_2 a \beta$, where $a \in \Sigma$, $\alpha, \beta, \gamma_1, \gamma_2 \in \Sigma^*$. It is clear that $\alpha a!\gamma_1, \gamma_2 a?\beta \in L$, so $\alpha a?\beta \in \sigma(L)$. It follows that $z \in h_2(\sigma(L))$.

($\supseteq$) Let $z \in h_2(\sigma(L))$. Then there exists $a \in \Sigma$ and words $\alpha, \gamma_1, \gamma_2, \beta \in \Sigma^*$, such that $z = \alpha a \beta$ and $\alpha a?\beta \in \sigma(L)$. It follows that there exists $\alpha a!\gamma_1, \gamma_2 a?\beta \in L$, so $\alpha a \gamma_1 \in L_1$ and $\gamma_2 a \beta \in L_2$, which implies that $z \in L_1 \bowtie L_2$. ∎

Recall that, given a binary word operation $\diamond$, the binary word operation $\square$ is called the *right-inverse of* $\diamond$ [56] if and only if for every triplet of words $u, y, w \in \Sigma^*$ the following relation holds: $w \in (u \diamond y)$ if and only if $y \in (u \square w)$. In other words, the operation $\square$ is called the right-inverse of $\diamond$ if it can be used to recover the right operand y in $u \diamond y$, from the other operand u and a word $w \in (u \diamond y)$ in the result. Define now the binary word operation $\bowtie^r$ as $u \bowtie^r w = \bigcup_{a \in \Sigma} \Sigma^* a \left(\left(u (a\Sigma^*)^{-1} a \right)^{-1_l} w \right)$. Informally, given a word $w = \alpha a \beta \in (\alpha a \gamma_1 \bowtie \gamma_2 a \beta)$, the operation $\bowtie^r$ outputs the right operand $y = \gamma_2 a \beta$ of word blending, if it is given as inputs the result $w = \alpha a \beta \in (u \bowtie y)$ and the left operand $u = \alpha a \gamma_1$. The definition of $\bowtie^r$ can be extended to languages naturally.

Proposition 5.3.6 *The operation $\bowtie^r$ is the right-inverse of $\bowtie$.*

Proof If $w \in u \bowtie y$, there exist $\alpha, \beta, \gamma_1, \gamma_2 \in \Sigma^*, b \in \Sigma$ such that $w = \alpha b \beta, u = \alpha b \gamma_1, y = \gamma_2 b \beta$ by Lemma 5.3.1. Then, we have that

$$\begin{aligned} y = \gamma_2 b \beta &\in \Sigma^* b \beta \\ &= \Sigma^* b \left((\alpha b)^{-1_l} (\alpha b \beta) \right) \\ &\subseteq \Sigma^* b \left(\left(\left((\alpha b \gamma_1) (b\Sigma^*)^{-1} \right) b \right)^{-1_l} (\alpha b \beta) \right) \\ &\subseteq \bigcup_{a \in \Sigma} \Sigma^* a \left(\left(\left((\alpha b \gamma_1) (a\Sigma^*)^{-1} \right) a \right)^{-1_l} (\alpha b \beta) \right) \end{aligned}$$

If $y \in u \bowtie^r w = \bigcup_{a\in\Sigma} \Sigma^* a \left(\left(\left(u\left(a\Sigma^*\right)^{-1}\right)a\right)^{-1_l} w\right)$, then there exist $b \in \Sigma$, and $\gamma_2 \in \Sigma^*, \gamma_3 \in \left(u\left(b\Sigma^*\right)^{-1}\right)b$ such that $y = \gamma_2 b\left(\gamma_3^{-1_l} w\right)$. This implies that

$$\begin{aligned}
w &\in \left(u\left(b\Sigma^*\right)^{-1}\right) b\left(\gamma_3^{-1_l} w\right) \\
&= \left(u\left(b\Sigma^*\right)^{-1}\right) b\left((\gamma_2 b)^{-1_l}\left(\gamma_2 b\left(\gamma_3^{-1_l} w\right)\right)\right) \\
&\subseteq \left(u\left(b\Sigma^*\right)^{-1}\right) b\left((\Sigma^* b)^{-1_l} y\right) \\
&\subseteq \bigcup_{a\in\Sigma} \left(u\left(a\Sigma^*\right)^{-1}\right) a\left((\Sigma^* a)^{-1_l} y\right) \\
&= u \bowtie y \qquad \blacksquare
\end{aligned}$$

Corollary 5.3.7 *The families of regular languages and recursively enumerable languages are closed under the right inverse of the blending. Moreover, if L_1 is an arbitrary language and L_2 is a regular language, then $L_1 \bowtie^r L_2$ is regular; if L_1 is a regular language and L_2 is a context-free language, then $L_1 \bowtie^r L_2$ is context-free.*

Proposition 5.3.8 *The family of context-free languages is not closed under the right inverse of blending.*

Proof Consider the context-free languages $L_1 = \{a\$(b^{i_1}a^{i_1}\$)\cdots(b^{i_n}a^{i_n}\$) \mid n \geq 1, i_m \geq 1 \text{ for } 1 \leq m \leq n\}$, $L_2 = \{(a^{j_1}\$b^{2j_1})\cdots(a^{j_k}\$b^{2j_k})(a^j\$c^{2j}) \mid j \geq 1, k \geq 1, j_m \geq 1 \text{ for } 1 \leq m \leq k\}$ and the regular language $R = \{\$c^*\}$.

We now show that $(L_1 \bowtie^r L_2) \cap R = \{\$c^{2^n} \mid n \geq 2\}$. Since words in R start with \$ and contain only one symbol \$, the only cases in which the words in $L_1 \bowtie^r L_2$ have the pattern of the words in R are the cases of word pairs where the overlap letter is \$, and a prefix ending in \$ in the word from L_1 matches the prefix ending in the last occurrence of \$ in the word from L_2. More precisely, let $u = a\$b^{i_1}a^{i_1}\$b^{i_2}a^{i_2}\$\cdots b^{i_m}a^{i_m}\$\cdots b^{i_n}a^{i_n}\$ \in L_1$ and $v = a^{j_1}\$b^{2j_1}a^{j_2}\$b^{2j_2}\cdots a^{j_m}\$b^{2j_m}a^j\$c^{2j} \in L_2$. For a word $w \in (L_1 \bowtie^r L_2)$ to belong to R, we must have

$$a\$b^{i_1}a^{i_1}\$b^{i_2}a^{i_2}\$\cdots b^{i_m}a^{i_m}\$ = a^{j_1}\$b^{2j_1}a^{j_2}\$b^{2j_2}\cdots a^{j_m}\$b^{2j_m}a^j\$,$$

which implies $j_1 = 1, j_2 = i_1 = 2j_1 = 2, \ldots, j = i_m = 2j_m = 2^m$. Thus, $w = \$c^{2j} = \$c^{2^{m+1}}$, which implies $(L_1 \bowtie^r L_2) \cap R = \{\$c^{2^n} \mid n \geq 2\}$.

Since the family of context-free languages is closed under intersection with regular languages, it follows that it is not closed under the right inverse of blending. $\blacksquare$

Proposition 5.3.9 *The family of context-sensitive languages is not closed under the right inverse of blending.*

Proof Let L_0 be a recursively enumerable language over Σ, that is not context-sensitive, and the context-sensitive language L_1 over $\Sigma \cup \{a, b\}$ with $a, b \notin \Sigma$, that can be associated to L_0 such that L_1 consists of words of the form $a^i bP$ where $i \geq 0$ and $P \in L_0$ and, in addition, for every $P \in L_0$ there is an $i \geq 0$ such that $a^i bP \in L_1$.

The result now follows as $(L_1 \bowtie^r \{a^* b\}) \cap \Sigma^* b \Sigma^* = \{\Sigma^* bP \mid b \notin \Sigma, P \in L_0\}$, which is not context-sensitive, and the family of context-sensitive languages is closed under intersection with regular languages. ∎

Recall that given a binary word operation $\diamond$, the binary word operation $\square$ is called the *left-inverse of* $\diamond$ iff for every triplet of words $x, v, w \in \Sigma^*$ the following relation holds: $w \in (x \diamond v)$ if and only if $x \in (w \square v)$ [56].

Proposition 5.3.10 *The left inverse of blending can be expressed using the right inverse of blending, and mirror image as* $w \bowtie^l v = \mathrm{mi}\,(\mathrm{mi}\,(v) \bowtie^r \mathrm{mi}\,(w))$.

Proof If $w \in x \bowtie v$, there exist $\alpha, \beta, \gamma_1, \gamma_2 \in \Sigma^*, b \in \Sigma$ such that $w = \alpha b \beta, x = \alpha b \gamma_1, v = \gamma_2 b \beta$ by Lemma 5.3.1. Then, since $x = \alpha b \gamma_1 = \mathrm{mi}\,(\mathrm{mi}\,(\gamma_1)\, b\, \mathrm{mi}\,(\alpha))$, we have

$$
\begin{aligned}
& x \in \mathrm{mi}\,(\Sigma^* b\, \mathrm{mi}\,(\alpha)) \\
& \quad = \mathrm{mi}\left(\Sigma^* b \left((\mathrm{mi}\,(\beta)\, b)^{-1_l} (\mathrm{mi}\,(\beta)\, b\, \mathrm{mi}\,(\alpha))\right)\right) \\
& \quad = \mathrm{mi}\left(\Sigma^* b \left(\left((\mathrm{mi}\,(\beta)\, b\, \mathrm{mi}\,(\gamma_2))\,(b\, \mathrm{mi}\,(\gamma_2))^{-1}\, b\right)^{-1_l} \mathrm{mi}\,(w)\right)\right) \\
& \quad = \mathrm{mi}\left(\Sigma^* b \left(\left(\left(\mathrm{mi}\,(v)\,(b\, \mathrm{mi}\,(\gamma_2))^{-1}\right) b\right)^{-1_l} \mathrm{mi}\,(w)\right)\right) \\
& \quad \subseteq \mathrm{mi}\left(\bigcup_{a \in \Sigma} \left(\Sigma^* a \left(\left(\left(\mathrm{mi}\,(v)\,(a\Sigma^*)^{-1}\right) a\right)^{-1_l} \mathrm{mi}\,(w)\right)\right)\right) \\
& \quad = \mathrm{mi}\,(\mathrm{mi}\,(v) \bowtie^r \mathrm{mi}\,(w)) \\
& \quad = w \bowtie^l v
\end{aligned}
$$

Now consider $x \in w \bowtie^l v$. Then,

$$
\begin{aligned}
x \in w \bowtie^l v &= \mathrm{mi}\,(\mathrm{mi}\,(v) \bowtie^r \mathrm{mi}\,(w)) \\
&= \mathrm{mi}\left(\bigcup_{a \in \Sigma} \Sigma^* a \left(\left(\left(\mathrm{mi}\,(v)\,(a\Sigma^*)^{-1}\right) a\right)^{-1_l} \mathrm{mi}\,(w)\right)\right).
\end{aligned}
$$

Thus, there exist $b \in \Sigma, \gamma_1 \in \Sigma^*, \gamma_3 \in (\mathrm{mi}(v)(b\Sigma^*)^{-1})b$ such that

$$
x = \mathrm{mi}\left(\mathrm{mi}\,(\gamma_1)\, b \left(\gamma_3^{-1_l} \mathrm{mi}\,(w)\right)\right) = \left(w\, \mathrm{mi}\,(\gamma_3)^{-1}\right) b \gamma_1
$$

Then we have

$$\begin{aligned} w &\in \left(w \operatorname{mi} (\gamma_3)^{-1}\right) b \left((\Sigma^* b)^{-1_l} v\right) \\ &\subseteq \left(w \operatorname{mi} (\gamma_3)^{-1}\right) b\gamma_1 (b\Sigma^*)^{-1} b \left((\Sigma^* b)^{-1_l} v\right) \\ &\subseteq \bigcup_{a\in\Sigma} \left(\left(\left(w \operatorname{mi} (\gamma_3)^{-1}\right) b\gamma_1\right) (a\Sigma^*)^{-1}\right) a \left((\Sigma^* a)^{-1_l} v\right) \\ &= \bigcup_{a\in\Sigma} \left(x (a\Sigma^*)^{-1}\right) a \left((\Sigma^* a)^{-1_l} v\right) \\ &= x \bowtie v \quad \blacksquare \end{aligned}$$

Because all families of languages in the Chomsky hierarchy are closed under mirror image, their closure properties under the left-inverse of word blending are the same as their closure properties under the right-inverse of word blending.

We now consider the iterated blending operation $\bowtie_*$. Recall that, as mentioned in Section 5.2, for any language $L \subseteq \Sigma^*$, the language $L^{\bowtie_*}$ can be generated by a splicing system with null-context splicing rules defined as 6-tuples, as in [42]. As shown in [7], every splicing system where the rules are defined by 6-tuples, can also be implemented by a splicing system as defined in [86], which uses 4-tuple rules (see Definition 5.3.11). This connection, together with Proposition 5.3.2, allows us to express iterated word blending using so-called simple splicing systems [75], themselves a particular case of splicing systems based on 4-tuple splicing rules.

Definition 5.3.11 ([86]) *Let $\sigma = (\Sigma, R)$ be a splicing scheme, where Σ is the alphabet and R is a set of rules $R \subseteq \Sigma^*\#\Sigma^*\$\Sigma^*\#\Sigma^*$. A rule $(u_1, u_2; u_3, u_4)$ is a word $u_1\#u_2\$u_3\#u_4 \in R$. For two strings $x, y \in \Sigma^*$, we have*

$$\begin{aligned} \sigma(x, y) = \{x_1 u_1 u_4 y_2 \mid & x = x_1 u_1 u_2 x_2, y = y_1 u_3 u_4 y_2; \\ & x_1, x_2, y_1, y_2 \in \Sigma^*, u_1\#u_2\$u_3\#u_4 \in R\}. \end{aligned}$$

For a language L, we define $\sigma(L) = L \cup \bigcup_{x,y\in L} \sigma(x, y)$ and we define the iterated splicing of L by $\sigma^*(L) = \bigcup_{i\geq 0} \sigma^i(L)$ with $\sigma^0(L) = L$ and $\sigma^{i+1}(L) = \sigma(\sigma^i(L))$.

Simple splicing schemes are splicing schemes as above, but restricted to rules of the form $(a, \lambda; a, \lambda)$ for $a \in \Sigma$. Note that for two languages L_1 and L_2 over Σ, we now have that

$$L_1 \bowtie L_2 = \bigcup_{x\in L_1, y\in L_2} \sigma_{\bowtie}(x, y),$$

where $\sigma_{\bowtie}$ is the simple splicing scheme $\sigma_{\bowtie} = (\Sigma, R)$ with $R = \{a\#\lambda\$a\#\lambda \mid a \in \Sigma\}$.

This observation together with Proposition 5.3.2 which showed that the word blending of two languages can be written $L_1 \bowtie L_2 = \bigcup_{a\in\Sigma}(L_1(a\Sigma^*)^{-1})a((\Sigma^* a)^{-1_l} L_2)$, gives us the following result.

Proposition 5.3.12 *For any language $L \subseteq \Sigma^*$, we have $\sigma_{\bowtie}(L) = L \bowtie L$ and $\sigma^*_{\bowtie}(L) = L^{\bowtie_*}$.*

We note that the splicing scheme $\sigma_{\bowtie}$ is finite, since the number of rules depends only on the number of symbols in Σ, and it is unary, since the rules use words of length at most 1. We also note that, even though in [75] consideration is restricted to the case when L is a finite language, the properties of the splicing systems obtained therein imply the following closure properties.

Proposition 5.3.13 *Every full AFL is closed under iterated word blending.*

Proof Recall that $L^{\bowtie_*} = \sigma^*_{\bowtie}(L)$ and that $\sigma^*_{\bowtie}$ is finite and unary. For a splicing rule $u_1\#u_2\$u_3\#u_4$, the words u_1 and u_4 are called visible sites and u_2 and u_3 are invisible sites. In [85], it is shown that full AFLs are closed under regular splicing systems with finitely many visible sites. Since $\sigma^*_{\bowtie}$ is finite, the rules of $\sigma^*_{\bowtie}$ contain only finitely many visible sites. ∎

Now, we will give an explicit construction for $L^{\bowtie_*}$ when L is a regular language. We will require the following lemma concerning the structure of words generated by the iterated blending operation.

Lemma 5.3.14 *Let $L \subseteq \Sigma^+$ be a language. Then for each word $w \in L^{\bowtie_*}$, there exists $n \in \mathbb{N}$ such that there are words $u_i \in \inf(L), 1 \leq i \leq n$ and $\alpha_j \in \Sigma^*, 1 \leq j \leq n$ and symbols $a_k \in \Sigma, 1 \leq k \leq n-1$ where*

1. *for $n > 1$,*
   - *(a) $w = \alpha_1 a_1 \alpha_2 a_2 \cdots a_{n-1}\alpha_n$,*
   - *(b) $u_i = a_{i-1}\alpha_i a_i \in \inf(L)$ for all $2 \leq i \leq n-1$,*
   - *(c) $u_1 = \alpha_1 a_1 \in \text{pref}(L)$ and $u_n = a_{n-1}\alpha_n \in \text{suff}(L)$,*
2. *$u_1 = w \in L$ for $n = 1$.*

Proof Let $w \in L^{\bowtie_*}$. Then $w \in L^{\bowtie_j}$ for some $j \geq 0$. We will prove the statement by induction on j. If $j = 0$ then $w \in L$ and the statement holds taking $n = 1$. Assume that the statement holds for words in $L^{\bowtie_j}$ for any $j \leq k$, and consider a word $w \in L^{\bowtie_{k+1}} = L \bowtie L^{\bowtie_k}$. This implies that $w \in x \bowtie y$ for some $x \in L$ and $y \in L^{\bowtie_k}$. By the induction hypothesis, either $y \in L$ or $y = \beta_1 b_1 \beta_2 b_2 \cdots b_{m-1}\beta_m$ for some $m \geq 2$ with $b_i \in \Sigma$, $1 \leq i \leq m-1$ and $\beta_j \in \Sigma^*$, $1 \leq j \leq m$ satisfying the conditions of the Lemma.

If $y \in L$, then $x \bowtie y$ consists of all words of the form $\alpha_1 a_1 \alpha_2$ where $x = \alpha_1 a_1 \gamma_1$ for $\alpha_1, \gamma_1 \in \Sigma^*$ and $a_1 \in \Sigma$ and $y = \gamma_2 a_1 \alpha_2$ for some $\gamma_2, \alpha_2 \in \Sigma^*$. It is easy to see that $\alpha_1 a_1 \alpha_2$ satisfies the conditions of the Lemma.

Otherwise, if $y = \beta_1 b_1 \beta_2 b_2 \cdots b_{m-1}\beta_m$ for some $m \geq 2$, then the set $x \bowtie y$ consists of words of the form $\alpha'_1 a'_1 \beta'_\ell b_\ell \cdots b_{m-1}\beta_m$ where $\alpha'_1 a'_1 \in \text{pref}(x)$ and $1 \leq \ell \leq m$. Here, we observe that in order for the blend to occur, we have $a'_1\beta'_\ell \in \text{suff}(b_{\ell-1}\beta_\ell)$. Then by definition, we have $\alpha'_1 a'_1 \in \text{pref}(x) \subseteq \text{pref}(L)$ and $a'_1\beta'_\ell b_\ell \in \text{inf}(L)$ and the rest follows. ∎

Proposition 5.3.15 *Given an NFA A, there exists an NFA A' which can be effectively constructed, and which recognizes the language $L(A)^{\bowtie *}$.*

Proof Given an NFA $A = (Q, \Sigma, s, \delta, F)$, we can construct an NFA $A' = (Q', \Sigma, s', \delta', F')$ that recognizes the language $L(A)^{\bowtie *}$. Informally, the machine operates by guessing when a blend occurs. Recall from Lemma 5.3.14, words of $L(A)^{\bowtie *}$ are of the form $\alpha_1 a_1 \alpha_2 a_2 \cdots a_{n-1}\alpha_n$. When a blend occurs on a symbol a_i, the machine then simulates the operation of A on the blended suffix α_{i+1} and continues to guess when the next blend may occur. This process repeats until the machine reaches a final state of A and accepts or it does not and the machine rejects.

Formally, we define A' by

- $Q' = \{\langle p\rangle, \langle q, r\rangle \mid p, q, r \in Q\}$,
- $s' = \langle s\rangle$,
- $F' = \{\langle q\rangle \mid q \in F\}$,

and the transition function is defined by

- $\delta'(\langle q\rangle, a) = \{\langle q'\rangle, \langle q', r'\rangle \mid q' \in \delta(q, a), r' \in \bigcup_{p\in Q} \delta(p, a)\}$,
- $\delta'(\langle q, r\rangle, a) = \{\langle r'\rangle, \langle r', p'\rangle \mid r' \in \delta(r, a), p' \in \bigcup_{p\in Q} \delta(p, a)\}$.

First, we show that $L(A') \subseteq L(A)^{\bowtie *}$. Consider a word $w \in L(A')$. There exists a sequence of states, or path, in A' on w from $\langle s\rangle$ to $\langle q'_n\rangle$ where $\langle q'_n\rangle \in F'$. Recall that states of A' are of the form $\langle q\rangle$ or $\langle q, r\rangle$. Of the states on the path defined by the computation of w, we consider those states of the form $\langle q, r\rangle$ and label them $\langle q'_i, q_{i+1}\rangle$ for $0 \leq i \leq n$. Each state $\langle q'_i, q_{i+1}\rangle$ is entered upon reading a symbol which we will call $a_i \in \Sigma$.

Now for each $1 \leq i \leq n-1$, consider the path in A' between $\langle q'_{i-1}, q_i\rangle$ and $\langle q'_i, q_{i+1}\rangle$. Between these two states, each state on the path is of the form $\langle q\rangle$, as we have already labeled states that are of the form $\langle q, r\rangle$. This implies that there is a word $\alpha_{i+1}a_{i+1}$ which takes A from q_i to q'_i. But this means that $a_i\alpha_{i+1}a_{i+1}$ is a subword of some word recognized by A. Then we can write $w = \alpha_1 a_1 \alpha_2 a_2 \cdots a_{n-1}\alpha_n$ where $\alpha_1 a_1 \in \text{pref}(L(A))$ since it takes A from s to s' and $\alpha_n \in \text{suff}(L(A))$ since it takes A from a state q_n to $q'_n \in F$. Thus, we have $w \in L(A)^{\bowtie *}$.

Now, we show that $L(A)^{\bowtie_*} \subseteq L(A')$. Consider a word $w \in L(A)^{\bowtie_*}$. We can write $w = \alpha_1 a_1 \alpha_2 a_2 \cdots a_{n-1}\alpha_n$. Since each $a_i\alpha_{i+1}a_{i+1}$ is a subword of a word recognized by A, there must exist a path between two states of A, say q_i and q'_i, on each word $\alpha_{i+1}a_{i+1}$. Then a path can be traced in A' from the initial state $\langle s\rangle$ by

$$\langle s\rangle \xrightarrow{\alpha_1 a_1} \langle q'_0, q_1\rangle \xrightarrow{\alpha_2 a_2} \langle q'_1, q_2\rangle \xrightarrow{\alpha_3 a_3} \cdots \xrightarrow{\alpha_{n-1}a_{n-1}} \langle q'_{n-1}, q_n\rangle \xrightarrow{\alpha_n} \langle q'_n\rangle.$$

Recall that by Lemma 5.3.14, $\alpha_1 a_1 \in \mathrm{pref}(L(A))$ and therefore there is a path from s to a state q'_0 in A. Note also that since $a_{n-1}\alpha_n \in \mathrm{suff}(L(A))$, the state q'_n which is reached on a path on α_n must be an accepting state of A and therefore $\langle q'_n\rangle \in F'$ and $w \in L(A')$. ∎

This construction gives us a way to test whether a regular language L is closed under iterated blending.

Proposition 5.3.16 *Let L be a regular language. It is decidable whether or not L is closed under $\bowtie_*$.*

Proof Let A be an NFA that recognizes L. By the construction given in Proposition 5.3.15, we can construct an NFA A' that recognizes $L^{\bowtie_*}$. Testing equivalence of two NFAs is known to be decidable [93] and therefore, testing whether $L = L^{\bowtie_*}$ is decidable. ∎

Let $L, B \subseteq \Sigma^*$ be two languages. We say that B is a base of L (with respect to $\bowtie$) if $L = B^{\bowtie_*}$. In [75], it is shown that it is decidable whether or not a regular language is generated by a simple splicing scheme and a finite language base. Here, we extend the result to consider the case when the base need not be finite.

Proposition 5.3.17 *It is decidable whether or not a regular language has a base over $\bowtie_*$.*

Proof Let $L \subseteq \Sigma^*$ be a regular language given as a finite automaton. Let $R = \{w \in \Sigma^* \mid |w|_a \leq 2 \text{ for all } a \in \Sigma\}$ be the set of words in which each symbol of Σ appears at most twice. We claim that if L is closed under $\bowtie_*$, it must be generated by a base $B \subseteq L \cap R$.

Suppose otherwise and that L is generated by a base B' which is not a subset of $L \cap R$. Then B' contains a word of the form $w = x_1 a x_2 a x_3 a x_4$, where $x_1, x_2, x_3, x_4 \in \Sigma^*$ and $a \in \Sigma$. Let $w_1 = x_1 a x_2 a x_4$ and $w_2 = x_1 a x_3 a x_4$ and note that $w_1, w_2 \in w \bowtie w$ and therefore $w_1, w_2 \in L$. Furthermore, we have $w \in w_1 \bowtie w_2$ and we can define an equivalent base $B'' = (B' \setminus \{w\}) \cup \{w_1, w_2\}$.

Now, we show that we only need to repeat this procedure a finite number of times. One only needs to consider words of length at most n, where n is the pumping length of L. Indeed, consider a word u of length greater than n. Since L is regular, we can write $u = xy^2z$ where

$x, y, z \in \Sigma^*$ and $xyz \in L$ is of length at most n. But then we have $xyz \in u \bowtie u$. Thus, it suffices to consider only those words in L of length up to n.

We can test whether the base B obtained via this process generates L. Using the construction of Proposition 5.3.15, we can construct an NFA C that recognizes $B^{\bowtie_*}$ and test whether $L(C) = L(A)$. This is decidable since NFA equivalence is known to be decidable [93]. ∎

As a consequence, we are able to not only decide whether a regular language is closed under $\bowtie_*$, but if it is, we know there always exists a finite base that generates it.

Corollary 5.3.18 *Let L be a regular language closed under $\bowtie_*$. Then L can be generated by a finite base.*

Note that in [75] languages generated by simple splicing schemes are assumed to have finite bases by definition. There it was also shown that the class of languages generated by these simple splicing schemes is a subclass of the family of regular languages. Here we do not have the finite base restriction, and Corollary 5.3.18 shows that allowing regular bases does not give simple splicing schemes and iterated word blending any more power than restricting bases to be finite.

5.4 Decision Problems

This section investigates the existence of solutions to language equations of the type $X \bowtie L = R$ and $L \bowtie Y = R$, where L, R are given known languages, X, Y are unknown languages, and $\bowtie$ is the word blending operation.

Proposition 5.4.1 *The existence of a solution Y to the equation $L \bowtie Y = R$ is decidable for given regular languages L and R.*

Proof According to [56], since $\bowtie^r$ is the right-inverse of word blending, if there exists a solution Y to the given equation, then $Y' = (L \bowtie^r R^c)^c$ is also a solution. Moreover, in this case Y' is the maximal solution, in the sense that it includes all the other solutions to the equation. Since the family of regular languages is closed under $\bowtie^r$ and complement, the algorithm for deciding the existence of a solution starts with constructing $L \bowtie Y'$, which is also regular, and checking whether $L \bowtie Y'$ equals R. As equality of regular languages is decidable [76], if the answer to the question "Is $L \bowtie Y'$ equal to R?" is "yes", then a solution to the equation exists, and Y' is such a solution. If the answer is "no", then the equation has no solution. ∎

Proposition 5.4.2 *The existence of a solution X to the equation $X \bowtie L = R$ is decidable for regular languages L and R.*

Proof Similar to the preceding proof, and using the closure of the family of regular languages under the left-inverse of word blending. ∎

Proposition 5.4.3 *The existence of a singleton solution* $\{w\}$ *to the equation* $L \bowtie \{w\} = R$ *is decidable for regular languages* L *and* R.

Proof If R is empty, a singleton solution $\{w\}$ to the equation $L \bowtie \{w\} = R$ exists if and only if L does not use all the letters from the alphabet Σ. The decision algorithm will check the emptiness of all regular languages $L \cap \Sigma^* a \Sigma^*$, where $a \in \Sigma$: If any of them is empty, then $\{w\} = \{a\}$ is a singleton solution, otherwise no singleton solution exists.

We now consider the case when R is not empty. If there is a singleton solution $\{w\}$ to the equation $L \bowtie \{w\} = R$, where $L, R \subseteq \Sigma^+, w \in \Sigma^+$ then there is a shortest singleton solution of length $k \geq 1$, denoted by $w_s = a_1 a_2 \cdots a_k$, with $a_1, a_2, \ldots, a_k \in \Sigma$. We now want to show that the number of states in any finite state automaton that accepts R is at least k.

If $lg(w_s) = 1$, then $\lambda \notin R$, so the number of states of any finite state machine that recognizes R is at least 2, which is greater than the length of w_s.

Suppose $k \geq 2$. Define $L_i = (L \bowtie a_i)a_{i+1} \cdots a_k$ for $1 \leq i < k$, and define $L_k = L \bowtie a_k$. Then, we have $R = \bigcup_{i=1}^{k} L_i$. Note that $L_1 \nsubseteq \bigcup_{i=2}^{k} L_i$, as otherwise $a_2 a_3 \cdots a_k$ would be a shorter singleton solution than w_s—a contradiction.

Let $\alpha \in L_1 \subseteq R$; α can be represented as $\alpha = \alpha_1 a_1 a_2 \cdots a_k$, where $\alpha_1 \in \Sigma^*$. Assume now that R is recognized by a DFA $M = (Q, \Sigma, q_0, \delta, F)$ with $n < k$ states. Then there is a derivation

$$q_0 \alpha_1 a_1 a_2 \cdots a_k \Longrightarrow^* q_{i_1} a_1 a_2 \cdots a_k \Longrightarrow q_{i_2} a_2 \cdots a_k \Longrightarrow \cdots \Longrightarrow q_{i_k} a_k \Longrightarrow q_{i_{k+1}}.$$

Because M has $n < k$ states, there is a state that occurs twice in the set $\{q_{i_2}, q_{i_3}, \ldots, q_{i_{k+1}}\}$.

If $q_{i_j} = q_{i_{k+1}}$ where $2 \leq j \leq k$, then $\alpha_1 a_1 \cdots a_{j-1}(a_j \cdots a_k)^+ \subseteq R$, and so there exists a word $\alpha_2 \in \Sigma^*$ such that $\alpha_1 a_1 \cdots a_{j-1}(a_j \cdots a_k)^+ \alpha_2 \subseteq L$. Thus, we have that

$$\alpha \in \alpha_1 a_1 \cdots a_{j-1}(a_j \cdots a_k)^+ \alpha_2 \bowtie a_k \subseteq L_k \subseteq \bigcup_{i=2}^{k} L_i$$

.

If $q_{i_j} = q_{i_h}$ where $2 \leq j < h \leq k$, then $\alpha_1 a_1 \cdots a_{j-1}(a_j \cdots a_{h-1})^+ a_h \cdots a_k \subseteq R$, and so there exists a word $\alpha_2 \in \Sigma^*$ such that $\alpha_1 a_1 \cdots a_{j-1}(a_j \cdots a_{h-1})^+ \alpha_2 \subseteq L$. Thus, we have that

$$\alpha \in (\alpha_1 a_1 \cdots a_{j-1}(a_j \cdots a_{h-1})^+ \alpha_2 \bowtie a_{h-1})a_h \cdots a_k \subseteq L_{h-1} \subseteq \bigcup_{i=2}^{k} L_i$$

.

In either case, for all words $\alpha \in L_1$, $\alpha \in \bigcup_{i=2}^{k} L_i$. Thus, we have that $L_1 \subseteq \bigcup_{i=2}^{k} L_i$, which is a contradiction.

For the equation $L \bowtie \{w\} = R$, if there is a singleton solution, there is a singleton solution w_s of minimal length k, and the number of states in any DFA for R is at least k. If the minimal DFA that generates R has k states, the algorithm for deciding the existence of a singleton solution will check all the words β, where $lg(\beta) \leq k$. The answer is"yes" if this algorithm finds a string β such that $L \bowtie \{\beta\} = R$, and "no" otherwise. ∎

Proposition 5.4.4 *The existence of a singleton solution $\{w\}$ to the equation $\{w\} \bowtie L = R$ is decidable for regular languages L and R.*

Proof Similar to the preceding proof, and using the fact that the minimal length of a singleton solution to the equation is equal to or smaller than the number of states of any DFA that recognizes R. ∎

Proposition 5.4.5 *The existence of a singleton solution $\{w\}$ to the equation $L \bowtie \{w\} = R$ is undecidable for regular languages R and context-free languages L.*

Proof Assume, for the sake of contradiction, that the existence of a singleton solution $\{w\}$ to the equation $L \bowtie \{w\} = R$ is decidable for regular languages R and context-free languages L.

Given an arbitrary context-free language L' over an alphabet Σ, the context-free language $L_1 = \#\Sigma^+\# \cup L'\$$ can be constructed where $\#, \$ \notin \Sigma$. Note now that the equation $L_1 \bowtie \{w\} = \Sigma^*\$$ has a singleton solution $\{w\}$ if and only if $L' = \Sigma^*$ and the solution is $\{w\} = \{\$\}$.

Thus, if we could decide the problem in the proposition, we would be able to decide whether or not $L' = \Sigma^*$ for arbitrary context-free languages L', which is impossible. ∎

Corollary 5.4.6 *The existence of a solution Y to the equation $L \bowtie Y = R$ is undecidable for regular languages R and context-free languages L.*

Proposition 5.4.7

1. *The existence of a singleton solution $\{w\}$ to the equation $\{w\} \bowtie L = R$ is undecidable for a regular language R and a context-free language L.*
2. *The existence of a solution X to the equation $X \bowtie L = R$ is undecidable for a regular language R and a context-free language L.*

5.5 State Complexity

By Proposition 5.3.2, the family of regular languages is closed under word blending. Thus, we can consider the state complexity of the blending operation on two regular languages. Recall from Proposition 5.3.2 that the blending of two languages can be expressed as a series of union,

concatenation, and quotient operations. While the state complexity of each of these operations is known, the state complexity of a combination of operations is not necessarily the same as the composition of the state complexities of the operations [95].

First, for illustrative purposes, we will construct an NFA that recognizes the blending of two languages given by DFAs. Let $A_m = (Q_m, \Sigma, \delta_m, s_m, F_m)$ be a DFA with $m \geq 1$ states that recognizes the language L_m and let $A_n = (Q_n, \Sigma, \delta_n, s_n, F_n)$ be a DFA with $n \geq 1$ states that recognizes the language L_n. We construct an NFA $B' = (Q', \Sigma, \delta', s', F')$, where $Q' = Q_m \cup Q_n$, $s' = s_m$, $F' = F_n$, and the transition function $\delta' : Q' \times \Sigma \to 2^{Q'}$ is defined for all $q \in Q'$ and $a \in \Sigma$ by

$$\delta'(q,a) = \begin{cases} \bigcup_{p \in Q_n} \delta_n(p,a) & \text{if } q \in Q_m \text{ and } \delta_m(q,a) \text{ is not the sink state,} \\ \delta_m(q,a) & \text{if } q \in Q_m \text{ and } \delta_m(q,a) \text{ is the sink state,} \\ \delta_n(q,a) & \text{if } q \in Q_n. \end{cases}$$

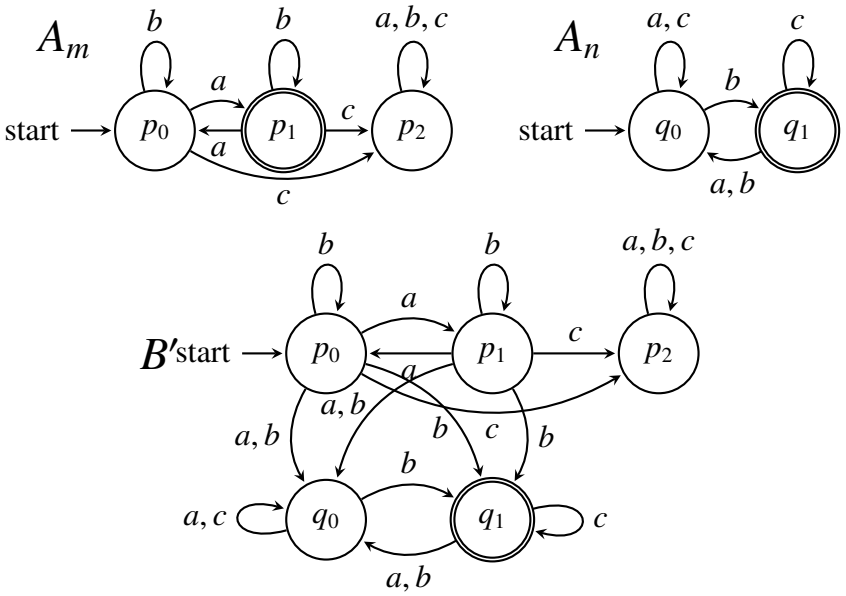

Figure 5.1: The NFA B' recognizes the blend of the languages recognized by the DFAs A_m and A_n

In Figure 5.1, we define two DFAs A_m and A_n and show the NFA B' resulting from the construction described above. Intuitively, the machine B' operates by first reading the input word assuming that it is the prefix of some word recognized by A_m. Since the blending occurs on only one symbol, the machine guesses at which symbol the blend occurs. Once the blend occurs the machine continues and assumes the rest of the word is the suffix of some word recognized by A_n.

Proposition 5.5.1 *The NFA B' recognizes the language $L_m \bowtie L_n$.*

Proof First, we show that $L_m \bowtie L_n \subseteq L(B')$. Let $w \in L_m \bowtie L_n$ and write $w = \alpha a \beta$ for a symbol $a \in \Sigma$ and words $\alpha, \beta \in \Sigma^*$ such that for some words $\gamma_1, \gamma_2 \in \Sigma^*$, we have $\alpha a \gamma_1 \in L_m$ and $\gamma_2 a \beta \in L_n$. Since αa is a prefix of a word in L_m, let $p = \delta_m(s_m, \alpha a) \in \delta'(s', \alpha a)$. However, since $a\beta$ is a suffix of a word of L_n, there exists a state $q \in Q_n$ such that $\delta_n(q, a) = r$ and by the definition of δ', we have $r \in \delta'(s', \alpha a)$. From here, we observe that we must have $\delta_n(r, \beta) \in F_n$ and therefore $\delta'(r, \beta) \in F'$ and w is accepted by B'.

Next, we show that $L(B') \subseteq L_m \bowtie L_n$ and consider a word $w \in L(B')$. By definition, must exist a path on w from $s' = s_m$ to a state in F_n and we can divide the path into two parts. The first part consists of transitions among states of A_m and the latter part consists of transitions among states of Q_n. Observe that the only way for a transition from a state $p \in Q_m$ to a state $q \in Q_n$ to be defined is if for some symbol $a \in \Sigma$, $\delta_m(p, a)$ is not a transition to a sink state and that $\delta_n(r, a) = q$ for some $r \in Q_n$. Thus, we can write $w = xay$ for words $x, y \in \Sigma^*$, where $a \in \Sigma$ is the symbol on which the path transitions from states of A_m to states of A_n. Then this implies that xa is a prefix of some word in L_m and ay is a suffix of some word in L_n. Therefore, $w \in L_m \bowtie L_n$ by definition and thus, we have shown that B' recognizes $L_m \bowtie L_n$. ∎

Now, using the same basic idea, we will construct a DFA that recognizes the language of the blending of the two languages recognized by two given DFAs A_m and A_n. We construct a DFA $A' = (Q', \Sigma, \delta', s', F')$ where

- $Q' = Q_m \times 2^{Q_n}$,
- $s' = (s_m, \emptyset)$,
- $F' = \{(q, P) \in Q_m \times 2^{Q_n} \mid P \cap F_n \neq \emptyset\}$,
- $\delta'((q, P), a) = (\delta_m(q, a), P')$ for $a \in \Sigma$, where

$$P' = \begin{cases} \bigcup_{p \in P} \delta_n(p, a) & \text{if } \delta_m(q, a) \text{ is the sink state,} \\ \bigcup_{p \in Q_n} \delta_n(p, a) & \text{otherwise.} \end{cases}$$

Figure 5.2 shows the DFA A' that results from following the construction described above, where A_m and A_n are the DFAs shown in Figure 5.1, and Figure 5.3 shows the minimal DFA. Each state of A' is a pair consisting of a state of A_m and a subset of states of A_n. Informally, we can divide the computation of a word into two phases. In the first phase, states of the form (q, P) are reached where q is not the sink state of A_m. Here, the set P is determined solely by the input symbol as the machine tries to guess the symbol on which the blending occurs. In the

second phase, the machine reaches states $(q_\emptyset, P)$, where $q_\emptyset$ is the sink state of A_m. The second phase only occurs when the blend occurs and the input that has been read is no longer a prefix of a word recognized by A_m. In this phase, the set P is determined by the transition function of A_n. We will show this formally in the following.

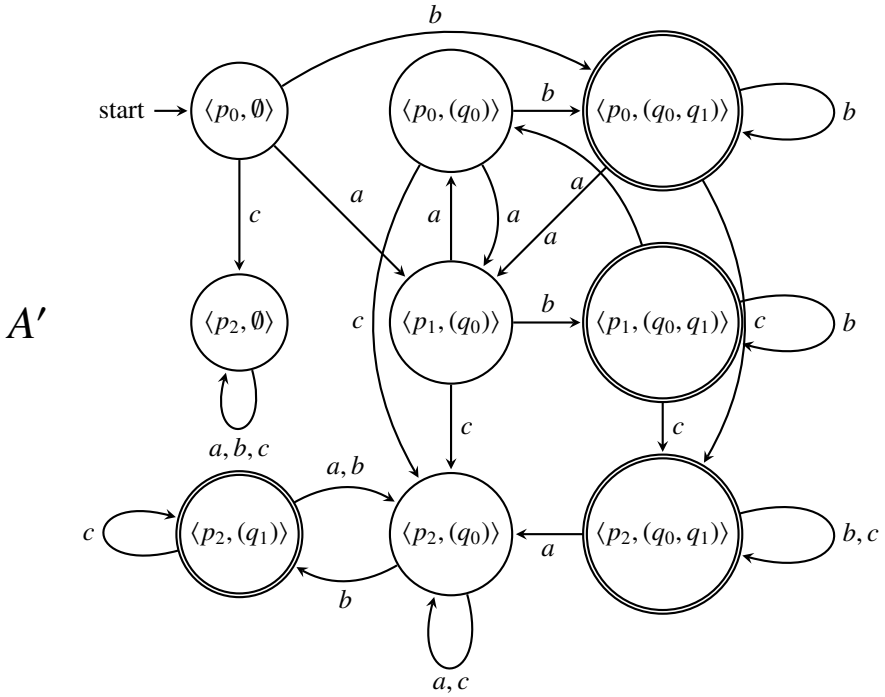

Figure 5.2: The DFA A' recognizes the blend of the languages recognized by A_m and A_n from Figure 5.1

Proposition 5.5.2 *The DFA A' recognizes the language $L_m \bowtie L_n$.*

Proof First, to show that $L_m \bowtie L_n \subseteq L(A')$, consider a word $w \in L_m \bowtie L_n$. Then $w = \alpha a \beta$ for some symbol $a \in \Sigma$ and words $\alpha, \beta \in \Sigma^*$ where for some $\gamma_1, \gamma_2 \in \Sigma^*$, we have $\alpha a \gamma_1 \in L_m$ and $\gamma_2 a \beta \in L_n$. Observe that since αa is a prefix of a word in L_m, the state $\delta_m(s_m, \alpha a)$ is not the sink state. Similarly, since $a\beta$ is the suffix of a word in L_n, there exists at least one state in Q_n that has an incoming transition on the symbol a.

If $\beta = \lambda$, then $a \in \text{suff}(L_n)$ and $\delta'(s', \alpha a) \in F'$ and therefore, $w \in L(A')$. So suppose $\beta = \sigma\beta'$ for some symbol $\sigma \in \Sigma$ and $\beta' \in \Sigma^*$. We assume that $\delta_m(s_m, \alpha a \sigma)$ is the sink state, since otherwise, we can write $w = \alpha' \sigma \beta'$ where $\alpha' = \alpha a$ and repeat the same process. Then reading σ takes us from $(\delta_m(s_m, \alpha a), P)$ to the state $(q_\emptyset, P')$, where $q_\emptyset$ denotes the sink state of A_m and $P' = \bigcup_{p \in P} \delta_n(p, a)$. Since β is the suffix of a word in L_n, there exists a state $p \in P$ such

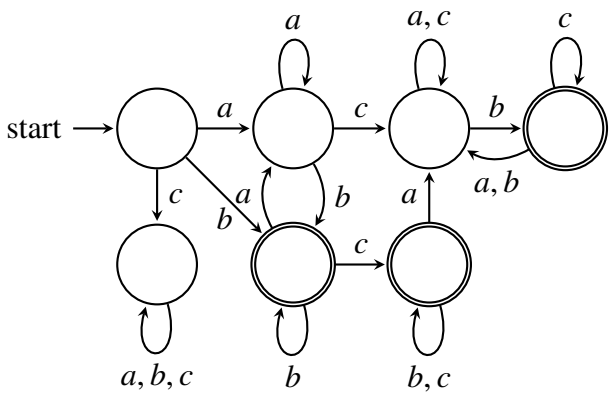

Figure 5.3: The minimal DFA for the blend of the languages recognized by A_m and A_n from Figure 5.1

that $\delta_n(p, \beta) \in F_n$. Thus, reading the rest of β takes us to a state $(q_\emptyset, P'')$ with $P'' \cap F_n \neq \emptyset$ and therefore, w is recognized by A'.

To show that $L(A') \subseteq L_m \bowtie L_n$, we consider a word w recognized by A'. That is, upon reading w, the machine A' reaches a final state (q', P') with $P' \cap F_n \neq \emptyset$. First, suppose that q' is not the sink state of A_m. We can write $w = w'a$ for a symbol $a \in \Sigma$ and a word $w' \in \Sigma^*$. Since q' is not the sink state of A_m, the word w is a prefix of some word in L_m and we have $w'a\gamma_1 \in L_m$ for some $\gamma_1 \in \Sigma^*$. By the definition of the transition function, a is a suffix of a word in L_n and we have $\gamma_2 a \in L_n$ for some $\gamma_2 \in \Sigma^*$. Thus, $w \in w'a\gamma_1 \bowtie \gamma_2 a$ and therefore $w \in L_m \bowtie L_n$.

Now, suppose that q' is the sink state of A_m. Let $w = \alpha ab\beta$ such that αab is the shortest prefix of w that enters the sink state q' of A_m. Then αa is a prefix of a word in L_m so we have $\alpha a\gamma_1 \in L_m$ for some $\gamma_1 \in \Sigma^*$. Reading αa takes us to the state $\delta'(s', \alpha a) = (q, P)$ such that q is some state of A_m which is not the sink state and $P = \bigcup_{p \in Q_n} \delta_n(p, a)$.

We claim that $ab\beta$ is a suffix of a word in L_n. To see this, we observe that since reading b from (q, P) takes A' to the state (q', P''), where q' is the sink state of A_m, any transitions from P'' no longer depend solely on the input symbol. Therefore, there must exist a path in A_n from a state $r \in P$ to a final state of A_n on the word $ab\beta$. We can write $\beta' = b\beta$ and thus there exists a word γ_2 such that $\gamma_2 a\beta' \in L_n$. Therefore, $w \in \alpha a\gamma_1 \bowtie \gamma_2 a\beta'$ and $w \in L_m \bowtie L_n$ as desired.

Thus, we have shown that $L(A') = L_m \bowtie L_n$. ∎

A simple count of the number of states in the state set of A' gives us as many as $m2^n$ states. We will show that, depending on the size of the alphabet, not all of these states are necessarily reachable. First, we consider the case where the alphabet is unary.

Proposition 5.5.3 *Let L_m and L_n be regular languages defined over a unary alphabet such that L_m is recognized by an m-state DFA and L_n is recognized by an n-state DFA. Then the state complexity of $L_m \bowtie L_n$ is $m + n - 1$ if both L_m and L_n are finite or 1 otherwise. Furthermore, this bound is reachable.*

Proof Recall that by Proposition 5.3.2, $L_m \bowtie L_n = (L_m(a^+)^{-1})a((a^+)^{-1}{}_l L_n)$. If either L_m or L_n are infinitely large, then we have $L_m \bowtie L_n = a^*$, in which case the state complexity of $L_m \bowtie L_n$ is 1. If both L_m and L_n are finite, then it is easy to see that the state complexity of $L_m \bowtie L_n$ is $m + n - 1$. ∎

Now, we will consider the state complexity when the languages are defined over alphabets of size greater than 1.

Lemma 5.5.4 *The DFA A' requires at most $(m-1)\cdot(k-1) + 2^n + 1$ states if $k = |\Sigma| \leq 2^n$; otherwise, it requires at most $(m-1)\cdot(2^n-1) + 2^n + 1$ states.*

Proof First, observe that in order to maximize the number of reachable states of A', the DFA A_m must contain a state that cannot reach an accepting state. Otherwise, if every state of A_m can reach an accepting state, then by definition of A', we have $L(A_m) \bowtie L(A_n) \subseteq \text{pref}(L(A_m))$. One can construct a DFA for $\text{pref}(L(A_m))$ by modifying A_m so that every state of A_m is a final state. In this case, A' would then require at most m states. Thus, we assume that A_m contains a sink state $q_\emptyset$ which cannot reach an accepting state.

Consider the case where $k \leq 2^n$ and the transition function δ' on a state (q, P), where $q \neq q_\emptyset$. Then for each symbol $a \in \Sigma$, there is only one possible reachable set of states P in A_n. This gives us up to $(m-1)\cdot k$ reachable states. However, we claim that in order for two states (q, P) and (q, P') with $P \neq P'$ to be distinguishable, q must contain a transition to $q_\emptyset$. Otherwise, for every symbol $a \in \Sigma$, we have $\delta'((q, P), a) = \delta'((q, P'), a)$ by definition. Thus, since every state must contain at least one transition to $q_\emptyset$ and A_m is deterministic, A' has only at most $(m-1)\cdot(k-1)$ reachable states of this form.

Similarly, for the case where $k > 2^n$, A' has only at most $(m-1)\cdot(2^n-1)$ reachable states of this form.

Next, consider that there are up to 2^n reachable states $(q_\emptyset, P)$ as derived from the subset construction.

Notice that the initial state $s' = (s_m, \emptyset)$ does not belong to any of the above sets. Adding all of these states together, we have at most $(m-1)\cdot(k-1) + 2^n + 1$ reachable states if $k \leq 2^n$; otherwise, we have at most $(m-1)\cdot(2^n-1) + 2^n + 1$ reachable states. ∎

Lemma 5.5.5 *Let $k \geq 3$ and $m, n \geq 2$. There exist families of DFAs A_m with m states and B_n with n states defined over an alphabet with k letters such that a DFA recognizing $A_m \bowtie B_n$ requires at least $(m-1)\cdot(k-1)+2^n+1$ states.*

Proof Let $\Sigma = \{a_1, \ldots, a_{k-2}, b, c\}$. We will define the DFAs A_m and B_n over Σ.

Let $A_m = (P_m, \Sigma, \delta_m, s_m, F_m)$ where $Q_m = \{0, \ldots, m-1\}$, $s_m = 0$, and $F_m = \{m-2\}$. We define the transition function δ_m by

- $\delta_m(p, a_i) = p$ for all $0 \leq p \leq m-2$ and $1 \leq i \leq k-2$,
- $\delta_m(p, b) = p+1$ for $0 \leq p \leq m-2$,
- $\delta_m(m-1, \sigma) = m-1$ for all $\sigma \in \Sigma$.

The DFA A_m is shown in Figure 5.4.

Let $B_n = (Q_n, \Sigma, \eta_n, s_n, F_n)$ where $Q_n = \{0, \ldots, n-1\}$, $s_n = 0$, and $F_n = \{n-1\}$. We will define the transition function η_n by

- $\eta_n(q, b) = q+1 \bmod n$ for $0 \leq q \leq n-1$,
- $\eta_n(q, c) = q$ for $0 \leq q \leq n-1$.

For transitions on symbols a_i with $1 \leq i \leq k-2$, we define an enumeration of the subsets of Q_n and let $Q_n[i]$ be the ith subset of Q_n. Any arbitrary enumeration of subsets of Q_n suffices for this proof subject to the condition that

1. for $0 \leq i \leq k-2$, each i corresponds to a particular subset of Q_n and
2. we reserve $Q_n[0] = Q_n$ and $Q_n[1] = \{0, 1, \ldots, n-2\}$.

That is, $Q_n[i] \neq Q_n[j]$ iff $i \neq j$ for $0 \leq i, j \leq k-2$. Also note that while we have defined $Q_n[0]$, there is no symbol a_0. We will show later that, by our definitions, the role of a_0 will be played by b. If $k > 2^n$, then this property cannot hold but it is clear that we can enumerate all 2^n subsets of Q_n.

Then we define transitions on $a_i \in \Sigma$ by

$$\eta(q, a_i) = \begin{cases} q & \text{if } q \in Q_n[i], \\ q + \min_{(q+j \bmod n) \in Q_n[i]} j \bmod n & \text{if } q \notin Q_n[i]. \end{cases}$$

In other words, for each state $q \in Q_n$, the transition on the symbol a_i goes to the "next" state that is in $Q_n[i]$. If $q \in Q_n[i]$, then that q itself is the "next" state.

We will show that A' contains $(m-1)\cdot(k-1)+2^n+1$ reachable and distinguishable states.

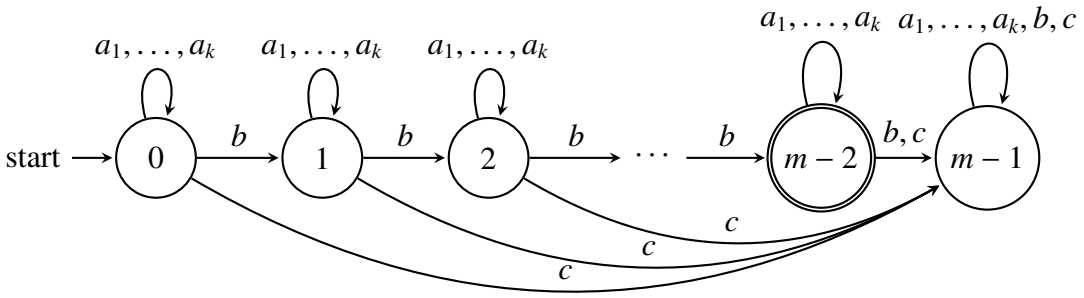

Figure 5.4: The DFA A_m.

First, to show that the states are reachable, we note that $s' = (s_m, \emptyset)$ is clearly reachable as the initial state. Then, we observe that for $1 \leq i \leq k-2$, the state $(q, Q_n[i])$ with $q \in Q_m \setminus \{m-1\}$ is reachable on the word $b^q a_i$ and $(q, Q_n[0])$ is reachable on the word b^q. Since the only symbol not used here is c, this gives us $(m-1) \cdot (k-1)$ states.

Now we consider states of the form $(m-1, P)$, where $P \subseteq Q_n$. Observe that $(m-1, Q_n)$ can be reached on the word b^{m-1}. Also, note that $(m-1, \emptyset)$ can be reached on the letter c from $(0, \emptyset)$.

Next, we will show that all states of the form $(m-1, P)$, where $P = Q_n \setminus T$ for some $T \subseteq Q_n$ are reachable from $(m-1, Q_n)$ by induction on $|T|$. First, consider $|T| = 1$ and $T = \{t\}$. Then, we have $Q_n \xrightarrow{a_1 b^{t+1}} Q_n \setminus \{t\}$.

Assume that all states $(m-1, Q_n \setminus T')$ are reachable from $(m-1, Q_n)$, where $u = |T'| \geq 1$. We will show that all states $(m-1, Q_n \setminus T)$ are reachable from $(m-1, Q_n)$, where $|T| = u+1 < |Q_n|$. Let $P = Q_n \setminus T = \{t_1, t_2, \ldots, t_{k-1}\}$, where elements in P are ordered in the ascending order. If $t_1 = 0$, $t_{k-1} \neq n-1$, then we have

$$(m-1, \{0, t_2, \cdots, t_{k-1}, n-1\}) \xrightarrow{a_1} (m-1, \{0, t_2, \ldots, t_{k-1}\}) = (m-1, P). \tag{5.1}$$

Thus, $(m-1, P)$ is reachable from the state $(m-1, P \cup \{n-1\})$, which is reachable from $(m-1, Q_n)$ by assumption.

If $t_1 = 0, t_{k-1} = n-1$, there exists a largest integer $v \notin P$, where $1 \leq v < n-1$, then we have

$$(m-1, \{w + n - v - 1 \bmod n \mid w \in P\}) \xrightarrow{b^{v+1}} (m-1, P).$$

Thus, $(m-1, P)$ is reachable from $(m-1, Q_n)$ by (5.1).

If $t_1 > 0$, then we have

$$(m-1, \{0, t_2 - t_1, \cdots, t_{k-1} - t_1\}) \xrightarrow{b^{t_1}} (m-1, \{t_1, t_2, \ldots, t_{k-1}\}).$$

That is, $(m-1, P)$ is reachable from $(m-1, Q_n)$ by (5.1). Thus, all states $(m-1, Q_n \setminus T)$ with $|T| = u+1$ are reachable.

Thus, we have an additional 2^n reachable states of the form $(m-1, P)$, giving us a total of $(m-1)\cdot(k-1) + 2^n + 1$ reachable states.

Next, we will show that these states are pairwise distinguishable. Consider two states (q, P) and (q', P'). First, we fix $P = P'$ and assume without loss of generality that $q < q'$. Then the two states are distinguished by the word $b^{m-1-q}a_1^n$.

Now, we consider when $P \neq P'$. In this case, reading c takes the state (q, P) to $(m-1, P)$ and (q', P') to $(m-1, P')$. Then without loss of generality, there exists an element $t \in P$ and $t \notin P'$. Then these states are distinguished by the word b^{n-t}.

Thus, we have shown that all $(m-1)\cdot(k-1) + 2^n + 1$ states are reachable and pairwise distinguishable. ∎

These results together give us the following proposition.

Proposition 5.5.6 *Let A_m be a DFA with m states recognizing the language L_m and let A_n be a DFA with n states recognizing the language L_n, where L_m and L_n are defined over an alphabet Σ of size k. Then*

$$sc(L_m \bowtie L_n) \leq (m-1)\cdot(k-1) + 2^n + 1,$$

and this bound can be reached in the worst case.

Chapter 6

Conclusions

In this thesis, we defined a novel bio-operation called word blending, which is inspired by Cross-pairing Polymerase Chain Reaction (XPCR), and we studied closure properties of the Chomsky families of languages under this operation and its iterated version, the decidability of existence of solution to equations involving this operation, and the state complexity of this operation.

In Chapter 2, we overviewed the families of languages in the Chomsky hierarchy. We defined the family of regular languages, context-free languages, context-sensitive languages and recursively enumerable languages using the form of their generative grammars, and also surveyed other ways of characterizing them. Moreover, some decidability problems related to these language families were given, together with a general approach to prove whether a problem is decidable or not.

In Chapter 3, we surveyed some commonly used word operations, including their definitions and outlines of proofs of the closure properties of the families of languages in the Chomsky hierarchy under these operations. Given a binary word operation $\diamond$, two known languages L, R over an alphabet Σ, and unknown languages X, Y over the alphabet Σ, we also describe a method to solve equations of the form $X \diamond L = R$ and $L \diamond Y = R$ using the right inverse and the left inverse of $\diamond$. Next, we discussed abstract families of languages, that is, families of languages that are closed under some operations, and discussed the idea that an operation can be represented by a composition of some other operations. Finally, we introduced state complexity, and showed some examples of how to find the state complexity of an operation.

In Chapter 4, the biological background related to bio-operations was given, and some biologically-inspired operations were surveyed including their definitions, their iterated versions, their restricted versions, and closure properties of various language families under these operations.

In Chapter 5, we introduced and studied the word operation called word blending, to model

a special case of XPCR, where two DNA sequences $\alpha A\beta, \beta A\gamma$ generate $\alpha A\gamma$. Let Σ be an alphabet. Following the approach used in Chapter 4, we first defined the operation, as $x \bowtie y = \{\alpha w\beta \mid x = \alpha w\gamma_1, y = \gamma_2 w\beta, \alpha, \beta, \gamma_1, \gamma_2, w \in \Sigma^+\}$. Then, we studied the closure properties of the Chomsky families of languages under this operation and its iterated version. Moreover, we studied its state complexity and some equations involving this operation, using methods discussed in Chapter 3.

Note that our model is a generalization of this special case of XPCR, which however required $\gamma_1 = \gamma_2$ in the XPCR experiment that were reported in [29]. In the future, we could consider a restricted version of word blending with the following definition, which is more closely aligned to the experimental XPCR whereby $\gamma_1 = \gamma_2 = \gamma$.

Definition 6.0.1 *Let x, y be two non-empty words over an alphabet Σ. The binary word operation called restricted word blending $\bowtie'$ is defined by $x \bowtie' y = \{\alpha w\beta \mid x = \alpha w\gamma, y = \gamma w\beta, \gamma \in \Sigma^+\}$.*

Moreover, in [46], the authors have investigated two variations of overlap assembly, using the longest or the shortest overlap part. Thus, we can also extend our study in this direction. For the case where the word blending operation uses the longest common non-empty infix, we have the following definition.

Definition 6.0.2 *Let x, y be two non-empty words over an alphabet Σ. The binary word operation called max word blending $\bowtie_{max}$ is defined by $x \bowtie_{max} y = \{\alpha w\beta \mid x = \alpha w\gamma_1, y = \gamma_2 w\beta, \gamma \in \Sigma^+, \forall \alpha', \beta', \gamma_1', \gamma_2' \in \Sigma^*, w' \in \Sigma^+ : x = \alpha' w' \gamma_1', y = \gamma_2' w' \beta', |w| \geq |w'|\}$.*

The min word blending operation could be defined similarly, but if the word blending can be applied to two words, these words must have a common part of minimum length 1, so the word blending operation and the min word blending operation are same due to Lemma 5.3.1.

Lastly, similar to the variation of the overlap assembly operation studied in [23], we can have a word blending operation where the length of the shared non-empty infix is larger than a positive integer $n \in \mathcal{N}^+$. Moreover, the restricted word blending with these length restrictions could also be considered if found to be mathematically interesting.

Bibliography

[1] Leonard Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994.

[2] Jürgen Albert, Karel Culik, and Juhani Karhumäki. Test sets for context free languages and algebraic systems of equations over a free monoid. *Information and Control*, 52(2):172–186, 1982.

[3] Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. On formal properties of simple phrase structure grammars. *STUF-Language Typology and Universals*, 14(1-4):143–172, 1961.

[4] John M. S. Bartlett and David Stirling. A short history of the polymerase chain reaction. In John M. S. Bartlett and David Stirling, editors, *PCR Protocols*, volume 226 of *Methods in Molecular Biology*, pages 3–6. Humana Press, Totowa, NJ, 2003.

[5] David Bikard, Céline Loot, Zeynep Baharoglu, and Didier Mazel. Folded DNA in action: Hairpin formation and biological functions in prokaryotes. *Microbiology and Molecular Biology Reviews*, 74(4):570–588, 2010.

[6] Paola Bonizzoni, Clelia De Felice, and Rosalba Zizza. The structure of reflexive regular splicing languages via Schützenberger constants. *Theoretical Computer Science*, 334(1):71–98, 2005.

[7] Paola Bonizzoni, Claudio Ferretti, Giancarlo Mauri, and Rosalba Zizza. Separating some splicing models. *Information Processing Letters*, 79(6):255–259, 2001.

[8] Janusz Brzozowski. Quotient complexity of regular languages. In Jürgen Dassow, Giovanni Pighizzini, and Bianca Truthe, editors, Proc. *Eleventh International Workshop on Descriptional Complexity of Formal Systems,* (DCFS 2009), volume 3 of *Electronic Proceedings in Theoretical Computer Science*, pages 17–28, 2009.

[9] Janusz Brzozowski, Lila Kari, Bai Li, and Marek Szykula. State complexity of overlap assembly. *Computing Research Repository*, abs/1710.06000, 2017.

[10] Alexandru Carausu and Gheorghe Păun. String intersection and short concatenation. *Revue Roumaine de Mathématiques Pures et Appliquées*, 26(5):713–726, 1981.

[11] Juan Castellanos and Victor Mitrana. Some remarks on hairpin and loop languages. In Masami Ito, Sheng Yu, and Gheorghe Păun, editors, *Words, Semigroups, and Transductions*, pages 47–58. World Scientific, 2001.

[12] Daniela Cheptea, Carlos Martın-Vide, and Victor Mitrana. A new operation on words suggested by DNA biochemistry: Hairpin completion. In Jean-Guillaume Dumas, editor, Proc. *Transgressive Computing*, pages 105–114, 2006.

[13] Noam Chomsky. On certain formal properties of grammars. *Information and control*, 2(2):137–167, 1959.

[14] Noam Chomsky. Formal properties of grammars. In Robert D. Luce, Robert R. Bush, and Eugene Galanter, editors, *Handbook of Mathematical Psychology*, volume 2, pages 323–418. New York: Wiley, 1963.

[15] Noam Chomsky and George Miller. Finite state languages. *Information and Control*, 1(2):91–112, 1958.

[16] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[17] Robert W. Coffin, Harry E. Goheen, and Walter R. Stahl. Simulation of a turing machine on a digital computer. In James D. Tupac, editor, Proc. *Fall Joint Computer Conference,* (AFIPS 1963), volume 24 of *American Federation of Information Processing Societies*, pages 35–43, New York, NY, USA, 1963. ACM.

[18] Erzsébet Csuhaj-Varjú, Ion Petre, and György Vaszil. Self-assembly of strings and languages. *Theoretical Computer Science*, 374(1):74–81, 2007.

[19] Karel Culik and Arto Salomaa. On the decidability of homomorphism equivalence for languages. *Journal of Computer and System Sciences*, 17(2):163–175, 1978.

[20] Karel Culik and Arto Salomaa. Test sets and checking words for homomorphism equivalence. *Journal of Computer and System Sciences*, 20(3):379–395, 1980.

[21] Carl W. Dieffenbach and Gabriela S. Dveksler, editors. *PCR Primer: A Laboratory Manual*. Cold Spring Harbor Laboratory Press, Cold Spring Harbor, USA, second edition, 2003.

[22] Michael Domaratzki. Semantic shuffle on and deletion along trajectories. In Cristian S. Calude, Elena Calude, and Michael J. Dinneen, editors, Proc. *Developments in Language Theory,* (DLT 2004), volume 3340 of *LNCS*, pages 163–174, Berlin, Heidelberg, 2005. Springer.

[23] Michael Domaratzki. Minimality in template-guided recombination. *Information and Computation*, 207(11):1209–1220, 2009.

[24] Derek R. Duckett and David M. J. Lilley. The three-way DNA junction is a Y-shaped molecule in which there is no helix-helix stacking. *The European Molecular Biology Organization journal*, 9(5):1659–1664, 1990.

[25] Srujan Kumar Enaganti, Oscar H. Ibarra, Lila Kari, and Steffen Kopecki. Further remarks on DNA overlap assembly. *Information and Computation*, 253:143–154, 2017.

[26] Srujan Kumar Enaganti, Oscar H. Ibarra, Lila Kari, and Steffen Kopecki. On the overlap assembly of strings and languages. *Natural Computing*, 16(1):175–185, 2017.

[27] Srujan Kumar Enaganti, Lila Kari, and Steffen Kopecki. A formal language model of DNA polymerase enzymatic activity. *Fundamenta Informaticae*, 138(1-2):179–192, 2015.

[28] Giuditta Franco. A polymerase based algorithm for SAT. In Mario Coppo, Elena Lodi, and G. Michele Pinna, editors, Proc. *Theoretical Computer Science,* (ICTCS 2005), volume 3701 of *LNCS*, pages 237–250. Springer, 2005.

[29] Giuditta Franco, Francesco Bellamoli, and Silvia Lampis. Experimental analysis of XPCR-based protocols. *Computing Research Repository*, abs/1712.05182, 2017.

[30] Giuditta Franco, Cinzia Giagulli, Carlo Laudanna, and Vincenzo Manca. DNA extraction by XPCR. In Claudio Ferretti, Giancarlo Mauri, and Claudio Zandron, editors, Proc. *DNA Computing,* (DNA 10), volume 3384 of *LNCS*, pages 104–112, 2005.

[31] Giuditta Franco and Vincenzo Manca. Algorithmic applications of XPCR. *Natural Computing*, 10(2):805–819, 2011.

[32] Giuditta Franco, Vincenzo Manca, Cinzia Giagulli, and Carlo Laudanna. DNA recombination by XPCR. In Alessandra Carbone and Niles A. Pierce, editors, Proc. *DNA Computing,* (DNA 11), volume 3892 of *LNCS*, pages 55–66, 2006.

[33] Tsu J. Fu and Nadrian C Seeman. DNA double-crossover molecules. *Biochemistry*, 32(13):3211–3220, 1993.

[34] Yuan Gao, Nelma Moreira, Rogério Reis, and Sheng Yu. A survey on operational state complexity. *Journal of Automata, Languages and Combinatorics*, 21(4):251–310, 2016.

[35] Seymour Ginsburg. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, Inc., New York, NY, USA, 1966.

[36] Seymour Ginsburg. *Algebraic and Automata-Theoretic Properties of Formal Languages*. Elsevier Science Inc., New York, NY, USA, 1975.

[37] Seymour Ginsburg and Sheila Greibach. Abstract families of languages. In Proc. *Eighth Annual Symposium on Switching and Automata Theory,* (SWAT 1967), pages 128–139, Washington, DC, USA, 1967. IEEE Computer Society.

[38] Jonathan S. Golan. *The Theory of Semirings with Applications in Mathematics and Theoretical Computer Science*. Addison-Wesley Longman Ltd., 1992.

[39] Elizabeth Goode and Dennis Pixton. Recognizing splicing languages: Syntactic monoids and simultaneous pumping. *Discrete Applied Mathematics*, 155(8):989–1006, 2007.

[40] Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *Journal of the ACM*, 12(1):42–52, 1965.

[41] Stefan Th. Gries. Shouldn't it be breakfunch? A quantitative analysis of blend structure in English. *Linguistics*, 42(3):639–667, 2004.

[42] Tom Head. Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49(6):737–759, 1987.

[43] Tom Head, Gheorghe Păun, and Dennis Pixton. Language theory and molecular genetics: Generative mechanisms suggested by DNA recombination. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 2, pages 295–360. Springer, Berlin, Heidelberg, 1997.

[44] Markus Holzer and Sebastian Jakobi. Chop operations and expressions: Descriptional complexity considerations. In Giancarlo Mauri and Alberto Leporati, editors, Proc. *Developments in Language Theory,* (DLT 2011), volume 6795 of *LNCS*, pages 264–275, Berlin, Heidelberg, 2011. Springer.

[45] Markus Holzer and Sebastian Jakobi. State complexity of chop operations on unary and finite languages. In Martin Kutrib, Nelma Moreira, and Rogério Reis, editors, *Descriptional Complexity of Formal Systems*, volume 7386 of *LNCS*, pages 169–182, Berlin, Heidelberg, 2012. Springer.

[46] Markus Holzer, Sebastian Jakobi, and Martin Kutrib. The chop of languages. *Theoretical Computer Science*, 682:122–137, 2017.

[47] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969.

[48] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on computing*, 17(5):935–938, 1988.

[49] Masami Ito. *Algebraic Theory of Automata and Languages*. World Scientific Press, 2004.

[50] Masami Ito, Peter Leupold, and Victor Mitrana. Bounded hairpin completion. In Adrian Horia Dediu, Armand Mihai Ionescu, and Carlos Martín-Vide, editors, Proc. *Language and Automata Theory and Applications,* (LATA 2009), volume 5457 of *LNCS*, pages 434–445, Berlin, Heidelberg, 2009. Springer.

[51] Masami Ito and Gerhard Lischke. Generalized periodicity and primitivity for words. *Mathematical Logic Quarterly*, 53(1):91–106, 2007.

[52] Neil D. Jones. A survey of formal language theory. Technical Report 3, University of Western Ontario, Computer Science Department, 1966.

[53] Peter D. Kaplan, Qi Ouyang, David S. Thaler, and Albert Libchaber. Parallel overlap assembly for the construction of computational DNA libraries. *Journal of Theoretical Biology*, 188(3):333–341, 1997.

[54] Lila Kari. *On insertion and deletion in formal languages*. PhD thesis, University of Turku, 1991.

[55] Lila Kari. Deletion operations: Closure properties. *International Journal of Computer Mathematics*, 52(1-2):23–42, 1994.

[56] Lila Kari. On language equations with invertible operations. *Theoretical Computer Science*, 132(1-2):129–150, 1994.

[57] Lila Kari, Rob Kitto, and Gabriel Thierrin. Codes, involutions, and DNA encodings. In Wilfried Brauer, Hartmut Ehrig, Juhani Karhumäki, and Arto Salomaa, editors, *Formal and Natural Computing: Essays Dedicated to Grzegorz Rozenberg*, volume 2300 of *LNCS*, pages 376–393. Springer, Berlin, Heidelberg, 2002.

[58] Lila Kari and Elena Losseva. Block substitutions and their properties. *Fundamenta Informaticae*, 73(1, 2):165–178, 2006.

[59] Lila Kari, Gheorghe Păun, Gabriel Thierrin, and Sheng Yu. At the crossroads of DNA computing and formal languages: Characterizing recursively enumerable languages using insertion-deletion systems. In Harvey Rubin and David Harlan Wood, editors, Proc. *DNA Based Computers III,* (DNA 3), volume 48 of *DIMACS*, pages 329–346, 1999.

[60] Lila Kari and Gabriel Thierrin. Contextual insertions/deletions and computability. *Information and Computation*, 131(1):47–61, 1996.

[61] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, U.S. Air Force Project Rand, 1951.

[62] Steffen Kopecki. On iterated hairpin completion. *Theoretical Computer Science*, 412(29):3629–3638, 2011.

[63] Murat Kural. Tree traversal and word order. *Linguistic Inquiry*, 36(3):367–387, 2005.

[64] Shigeyuki Kuroda. Classes of languages and linear-bounded automata. *Information and Control*, 7(2):207–223, 1964.

[65] Laura F. Landweber and Lila Kari. Universal molecular computation in ciliates. In Laura F. Landweber and Erik Winfree, editors, *Evolution as Computation*, pages 257–274, Berlin, Heidelberg, 2002. Springer.

[66] Peter S. Landweber. Three theorems on phrase structure grammars of type 1. *Information and Control*, 6(2):131–136, 1963.

[67] I. Robert Lehman, Maurice J. Bessman, Ernest S. Simms, and Arthur Kornberg. Enzymatic synthesis of deoxyribonucleic acid I. Preparation of substrates and partial purification of an enzyme from Escherichia coli. *Journal of Biological Chemistry*, 233(1):163–170, 1958.

[68] I. Robert Lehnman. DNA ligase: Structure, mechanism, and function. *Science*, 186(4166):790–797, 1974.

[69] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, 2nd edition, 1997.

[70] Vincenzo Manca and Giuditta Franco. Computing by polymerase chain reaction. *Mathematical Biosciences*, 211(2):282–298, 2008.

[71] Florin Manea, Carlos Martín-Vide, and Victor Mitrana. Hairpin lengthening: Language theoretic and algorithmic results. *Journal of Logic and Computation*, 25(4):987–1009, 2015.

[72] Florin Manea, Robert Mercas, and Victor Mitrana. Hairpin lengthening and shortening of regular languages. In Henning Bordihn, Martin Kutrib, and Bianca Truthe, editors, *Languages Alive: Essays Dedicated to Jürgen Dassow on the Occasion of His 65th Birthday*, volume 7300 of *LNCS*, pages 145–159. Springer, Berlin, Heidelberg, 2012.

[73] Florin Manea and Victor Mitrana. Hairpin completion versus hairpin reduction. In S. Barry Cooper, Benedikt Löwe, and Andrea Sorbi, editors, Proc. *Computation and Logic in the Real World,* (CiE 2007), volume 4497 of *LNCS*, pages 532–541, Berlin, Heidelberg, 2007. Springer.

[74] Florin Manea, Victor Mitrana, and Takashi Yokomori. Two complementary operations inspired by the DNA hairpin formation: Completion and reduction. *Theoretical Computer Science*, 410(4):417–425, 2009.

[75] Alexandru Mateescu, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. Simple splicing systems. *Discrete Applied Mathematics*, 84(1):145–163, 1998.

[76] Alexandru Mateescu and Arto Salomaa. Formal languages: An introduction and a synopsis. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of formal languages*, volume 1, pages 1–39. Springer, Berlin, Heidelberg, 1997.

[77] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[78] William Ogden. A helpful result for proving inherent ambiguity. *Mathematical Systems Theory*, 2(3):191–194, 1968.

[79] Rohit J Parikh. Language generating devices. In *Quarterly Progress Report*, volume 60, pages 199–212. Research Laboratory of Electronics, M.I.T., 1961.

[80] Rohit J. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.

[81] Gheorghe Păun. On the splicing operation. *Discrete Applied Mathematics*, 70(1):57–79, 1996.

[82] Gheorghe Păun. Regular extended H systems are computationally universal. *Journal of Automata, Languages and Combinatorics*, 1(1):27–36, 1996.

[83] Gheorghe Păun, Mario J Pérez-Jiménez, and Takashi Yokomori. Representations and characterizations of languages in chomsky hierarchy by means of insertion-deletion systems. *International Journal of Foundations of Computer Science*, 19(4):859–871, 2008.

[84] Dennis Pixton. Regularity of splicing languages. *Discrete Applied Mathematics*, 69(1):101–124, 1996.

[85] Dennis Pixton. Splicing in abstract families of languages. *Theoretical Computer Science*, 234:135–166, 2000.

[86] Gheorghe Păun. On the splicing operation. *Discrete Applied Mathematics*, 70(1):57–79, 1996.

[87] Gheorghe Păun. DNA computing based on splicing: Universality results. *Theoretical Computer Science*, 231(2):275–296, 2000.

[88] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. Computing by splicing. *Theoretical Computer Science*, 168(2):321–336, 1996.

[89] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[90] Elaine Rich. *Automata, Computability and Complexity: Theory and Applications*. Pearson Prentice Hall Upper Saddle River, 2008.

[91] Richard J Roberts and Kenneth Murray. Restriction endonuclease. *CRC Critical Reviews in Biochemistry*, 4(2):123–164, 1976.

[92] Kensaku Sakamoto, Hidetaka Gouzu, Ken Komiya, Daisuke Kiga, Shigeyuki Yokoyama, Takashi Yokomori, and Masami Hagiya. Molecular computation by DNA hairpin formation. *Science*, 288(5469):1223–1226, 2000.

[93] Arto Salomaa. *Formal Languages*. Academic Press, Inc., New York, NY, 1977.

[94] Arto Salomaa, Kai Salomaa, and Sheng Yu. Undecidability of the state complexity of composed regular operations. In Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications,* (LATA 2011), volume 6638 of *LNCS*, pages 489–498, Berlin, Heidelberg, 2011. Springer.

[95] Kai Salomaa and Sheng Yu. On the state complexity of combined operations and their estimation. *International Journal of Foundations of Computer Science*, 18(4):683–698, 2007.

[96] Stephen Scheinberg. Note on the boolean properties of context free languages. *Information and Control*, 3(4):372–375, 1960.

[97] Marcel Paul Schützenberger. On context-free languages and push-down automata. *Information and Control*, 6(3):246–264, 1963.

[98] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[99] Akihiro Takahara and Takashi Yokomori. On the computational power of insertion-deletion systems. In Masami Hagiya and Azuma Ohuchi, editors, Proc. *DNA Computing,* (DNA 8), volume 2568 of *LNCS*, pages 269–280, Berlin, Heidelberg, 2002. Springer.

[100] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[101] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265, 1937.

[102] Alan M. Turing. Intelligent machinery. In Donald Michie and Bernard Meltzer, editors, Proc. *Fifth Annual Machine Intelligence Workshop*, volume 5 of *Machine Intelligence*, pages 3–23. Edinburgh University Press, 1969.

[103] James D. Watson and Francis H. C. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

[104] Wikipedia, the free encyclopedia. Polymerase chain reaction - PCR, 2014. [Online; accessed April 23, 2018].

[105] Wikipedia, the free encyclopedia. Damp chemical structure, 2015. [Online; accessed April 27, 2018].

[106] Sheng Yu. State complexity of regular languages. *Journal of Automata, Languages and Combinatorics*, 6(2):221–234, 2001.

[107] Sheng Yu, Qingyu Zhuang, and Kai Salomaa. The state complexities of some basic operations on regular languages. *Theoretical Computer Science*, 125(2):315–328, 1994.

Curriculum Vitae

Name:	Zihao Wang
Post-Secondary Education and Degrees:	University of Western Ontario London, ON 2016 - 2018 M.Sc.(Computer Science) University of Western Ontario London, ON 2013 - 2016 B.Sc.(Computer Science)
Honours and Awards:	Robert and Ruth Lumsden Scholarships In Science 2015
Related Work Experience:	Research Assistant The University of Western Ontario 2016 - 2017

Publications:

Srujan K. Enaganti, Lila Kari, Timothy Ng, and Zihao Wang. Word blending in formal languages: The Brangelina effect. In Susan Stepney and Sergey Verlan, editors, Proc. *Seventeenth International Conference on Unconventional Computation and Natural Computation*, (UCNC 2018, in press).