# Contents

1 Introduction 1-1
   1.1 Purpose of Course ........................................... 1-1
   1.2 Mathematical Background .................................... 1-2

2 Languages 2-1
   2.1 Introduction .................................................. 2-1
   2.2 Alphabets, Strings, and Languages .......................... 2-1
   2.3 Set Relations and Operations ................................. 2-6
   2.4 Functions and Operations .................................... 2-10
   2.5 Closures ........................................................ 2-13

3 Recursive Definitions 3-1
   3.1 Definition ...................................................... 3-1
   3.2 Examples ....................................................... 3-1

4 Regular Expressions 4-1
   4.1 Some Definitions ............................................... 4-1
   4.2 Defining Languages Using Regular Expressions ............. 4-2
   4.3 The Language EVEN-EVEN ..................................... 4-8
   4.4 More Examples and Definitions ............................... 4-11
CONTENTS

5 Finite Automata 5-1
  5.1 Introduction ............................................. 5-1
  5.2 Finite Automata ......................................... 5-4
  5.3 Examples of FA ........................................... 5-8

6 Transition Graphs 6-1
  6.1 Introduction ............................................. 6-1
  6.2 Definition of Transition Graph ......................... 6-3
  6.3 Examples of Transition Graphs ......................... 6-5

7 Kleene’s Theorem 7-1
  7.1 Kleene’s Theorem .......................................... 7-1
  7.2 Proof of Part 1: FA ⇒ TG ................................. 7-2
  7.3 Proof of Part 2: TG ⇒ RegExp ............................ 7-2
  7.4 Proof of Part 3: RegExp ⇒ FA ............................. 7-12
  7.5 Nondeterministic Finite Automata ....................... 7-27
  7.6 Properties of NFA ....................................... 7-29

8 Finite Automata with Output 8-1
  8.1 Moore Machines .......................................... 8-1
  8.2 Mealy Machines .......................................... 8-2
  8.3 Properties of Moore and Mealy Machines ............... 8-5

9 Regular Languages 9-1
  9.1 Properties of Regular Languages ....................... 9-1
  9.2 Complementation of Regular Languages ................. 9-8
  9.3 Intersections of Regular Languages ..................... 9-10
Preface

These lecture notes are a revision of what I used to teach my CIS 341 class (Introduction to Logic and Automata) at NJIT in the spring semester of 1996. The course textbook is currently *Introduction to Computer Theory, Second Edition* (Wiley, 1997) by Daniel I. A. Cohen, and the development of the material in these notes corresponds to the layout there. My notes are meant to be a supplement (not a replacement) for the textbook in the course.

My lectures for CIS 341 in spring, 1996, were videotaped, and I tried to follow these notes as closely as possible. However, there are a number of places where the material in the notes does not exactly match that which is in the video tapes. There were several reasons for this. First, students often asked questions on material not covered in my notes. Second, I frequently made up examples in the middle of my lectures, and so those are not in the notes. Third, during some lectures I decided not to cover particular material in my notes for various reasons (e.g., lack of time).

The following is a rough guideline for how the tapes correspond to the pages in this set of lecture notes:

**Tape 1:** Syllabus, pages 1-1 to 2-18

**Tape 2:** Pages 2-17 to 3-4

**Tape 3:** Pages 4-1 to 4-6

**Tape 4:** Pages 4-6 to 4-11

**Tape 5:** Pages 4-11 to 5-12

**Tape 6:** Pages 5-12 to 6-5

**Tape 7:** Pages 6-5 to 7-12
CONTENTS

Tape 8: Pages 7-14 to 7-22
Tape 9: Pages 7-24 to 8-1
Tape 10: Pages 8-1 to 9-1
Tape 11: Pages 9-1 to 10-4
Tape 12: Pages 9-4 and 10-4 to 10-11
Tape 13: Pages 10-11 to 11-6
Tape 14: Pages 11-6 to 11-13, and handout on “Regular Expressions in the Real World: egrep” from Floyd and Beigel, *The Language of Machines*.
Tape 15: Pages 11-8 to 12-14
Tape 16: Pages 12-14 to 12-18
Tape 17: Pages 12-18 to 13-5
Tape 18: Pages 13-5 to 13-15
Tape 19: Pages 13-15 to 13-20
Tape 20: Pages 13-20 to 14-8
Tape 21: Pages 14-8 to 14-14
Tape 22: Pages 14-14 to 15-2
Tape 23: Pages 15-2 to 15-22
Tape 24: Pages 15-21 to 15-32
Tape 25: Pages 15-32 to 17-5
Tape 26: Pages 17-5 to 17-9
Supplement 1: Pages 17-9 to 19-14, skipping Chapter 18
Supplement 2: Pages 23-1 to 23-9
Finally, as anyone who has written a large document knows, it is virtually impossible to eliminate all of the errors. I have proofread these notes many times, but I am sure there are still a number of mistakes in it.

Marvin Nakayama
August, 2003
Chapter 1

Introduction

1.1 Purpose of Course

Course covers the theory of computers:

- Not concerned with actual hardware and software.
- More interested in abstract questions of the frontiers of capability of computers.
- More specifically, what can and what cannot be done by any existing computer or any computer ever built in the future.
- We will study different types of theoretical machines that are mathematical models for actual physical processes.
- By considering the possible inputs on which these machines can work, we can analyze their various strengths and weaknesses.
- We can then develop what we may believe to be the most powerful machine possible.
- Surprisingly, it will not be able to perform every task, even some easily described tasks.
1.2 Mathematical Background

In this class, we will be seeing a number of theorems and proofs.

To be able to understand how to prove a theorem, we first have to understand how theorems are stated.

Many (but not all) theorems are stated as “if $p$, then $q$”, where $p$ and $q$ are statements.

**Example:** If a word $w$ has more $e$’s than $o$’s, then $w$ has at least one $e$.

**Example:** If a word $w$ has $m$ a’s and $n$ e’s in it, then the word $w$ has at least $m + n$ letters in all.

**Example:** If $x^2 = 0$, then $x = 0$.

So what does “if $p$, then $q$” mean?

- If a theorem stated in this form is to be true, then it means that if $p$ is true, then $q$ must also be true.

- Note that this does not say that if $q$ is true, then $p$ must also be true. This may or may not be the case.

**Example:** The statement, “If a word $w$ has at least one $e$, then $w$ has more e’s than o’s” is not true.

For example, consider the word “exploration” or “Exxon.”

**Example:** If a word $w$ has at least $m + n$ letters in all, then the word $w$ has $m$ a’s and $n$ e’s in it.

For example, suppose $m = n = 1$, and consider the word “goof.”

**Example:** If $x = 0$, then $x^2 = 0$.

So now how do we prove a result?

We do it by arguing very carefully, where each step in our argument follows logically from the previous step.

There are several ways of proving that a statement “if $p$, then $q$” holds:
• One way is to use a direct argument:

**Example:** Prove: If a word \( w \) has more \( e \)'s than \( o \)'s, then \( w \) has at least one \( e \).

**Proof.** Let \( n_e \) be the number of \( e \)'s in \( w \), and let \( n_o \) be the number of \( o \)'s in \( w \). Since \( w \) has more \( e \)'s than \( o \)'s, we must have that \( n_e > n_o \), or in other words \( n_e \geq n_o + 1 \). But since \( w \) cannot have fewer than zero \( o \)'s, we must have that \( n_o \geq 0 \). Therefore, \( n_e \geq n_o + 1 \geq 0 + 1 = 1 \). Thus, \( w \) has at least one \( e \). □

• Another way of proving results is by contradiction. We do this by assuming that \( p \) is true and that \( q \) is not true, and then showing that an inconsistency results.

**Example:** Prove: If \( x^2 = 0 \), then \( x = 0 \).

**Proof.** Suppose that \( x^2 = 0 \) but \( x \neq 0 \). Then either \( x > 0 \) or \( x < 0 \). But if \( x > 0 \), then \( x^2 > 0 \), and if \( x < 0 \), then \( x^2 > 0 \). In either case, \( x^2 > 0 \). This contradicts the assumption that \( x^2 = 0 \). □

**Example:** Prove: If \( x > 0 \) with \( x \in \mathbb{R} \), then \( x^2 > 0 \).

**Proof.** Suppose that \( x^2 = 0 \). Then \( x = 0 \) so \( x \neq 0 \). □

There are several equivalent ways of stating “if \( p \), then \( q \)”

• “if not \( q \), then not \( p \)”
• “\( p \) only if \( q \)”
• “\( q \) if \( p \)”
• “\( p \) implies \( q \)”
• “\( p \) is sufficient for \( q \)”
• “\( q \) is necessary for \( p \)”

**Example:** Let \( x \) be a real number. If \( x > 0 \), then \( x^2 > 0 \).

This is equivalent to stating
• “If $x^2 > 0$ is not true (i.e., $x^2 \leq 0$), then $x > 0$ is not true (i.e., $x \leq 0$).”

• This is also equivalent to stating “$x > 0$ only if $x^2 > 0$."

• This is also equivalent to stating “$x^2 > 0$ if $x > 0$.”

• This is also equivalent to stating “$x > 0$ implies $x^2 > 0$.”

Often, the two statements

1. “$p$ only if $q$” (i.e., “if $p$, then $q$”) and
2. “$p$ if $q$” (i.e., “if $q$, then $p$”)

are combined into “$p$ if and only if $q$” (or “$p$ is a necessary and sufficient condition for $q$”).

In order for this statement to be true, we need to show that both statements
1 and 2 above are true.

Definition: An integer $n$ is an even number if $n = 2k$ for some $k = 0, 1, 2, 3, \ldots$.

Definition: An integer $n$ is an odd number if $n = 2k + 1$ for some $k = 0, 1, 2, 3, \ldots$.

Definition: An integer $n$ is a positive even number if $n = 2k$ for some $k = 1, 2, 3, \ldots$. 
Chapter 2

Languages

2.1 Introduction

- In English, there are at least three different types of entities: letters, words, sentences.
- Letters are from a finite alphabet \( \{ a, b, c, \ldots, z \} \)
- Words are made up of certain combinations of letters from the alphabet. Not all combinations of letters lead to a valid English word.
- Sentences are made up of certain combinations of words. Not all combinations of words lead to a valid English sentence.
- So we see that some basic units are combined to make bigger units.
- We want to abstract this to a different level.
- In particular, we will be studying so-called *formal languages*.

2.2 Alphabets, Strings, and Languages

**Definition**: A *set* is an unordered collection of objects or elements. Sets are written with curly braces \( \{ \} \), and the elements in the set are written within the curly braces.
Examples:

- The set \{a, b, c\} has elements a, b, and c.
- The sets \{a, b, c\} and \{b, c, b, a, a\} are the same since order does not matter in a set and since redundancy does not count.
- The set \{a\} has element a. Note that \{a\} and a are different things; \{a\} is a set with one element a.
- The set \{x^n : n = 1, 2, 3, \ldots\} consists of x, xx, xxx, \ldots.
- The set of even numbers is \{0, 2, 4, 6, 8, 10, 12, \ldots\} = \{2n : n = 0, 1, 2, \ldots\}. In particular, note that 0 is an even number.
- The set of positive even numbers is \{2, 4, 6, 8, 10, 12, \ldots\} = \{2n : n = 1, 2, 3, \ldots\}.
- The set of odd numbers is \{1, 3, 5, 7, 9, 11, 13, \ldots\} = \{2n + 1 : n = 0, 1, 2, \ldots\}.

**Definition:** An alphabet, denoted by \(\Sigma\), is a finite set of fundamental units (called letters) out of which we build structure.

Examples:

- The alphabet of lower-case Roman letters is \(\Sigma = \{a, b, c, \ldots, z\}\). (There are 26 lower-case Roman letters.)
- The alphabet of upper-case Roman letters is \(\Sigma = \{A, B, C, \ldots, Z\}\). (There are 26 upper-case Roman letters.)
- The alphabet of Arabic numerals is \(\Sigma = \{0, 1, 2, \ldots, 9\}\). (There are 10 Arabic numerals.)

**Definition:** A string over an alphabet is a finite sequence of letters from the alphabet.

Examples:

- *cat*, *food*, *c*, and *bbedwxq* are strings over the alphabet \(\Sigma = \{a, b, c, \ldots, z\}\).
• 0173 is a string over the alphabet \( \Sigma = \{0, 1, 2, \ldots, 9\} \).

**Definition:** The *empty string* or *null string*, which we shall denote by \( \Lambda \), is the string consisting of no letters, no matter what language we’re considering.

**Definition:** Given two strings \( w_1 \) and \( w_2 \), we define the *concatenation* of \( w_1 \) and \( w_2 \) to be the string \( w_1w_2 \).

**Examples:**

• If \( w_1 = xx \) and \( w_2 = x \), then \( w_1w_2 = xxx \).
• If \( w_1 = abb \) and \( w_2 = ab \), then \( w_1w_2 = ababb \) and \( w_2w_1 = ababb \).
• If \( w_1 = \Lambda \) and \( w_2 = ab \), then \( w_1w_2 = ab \).
• If \( w_1 = bb \) and \( w_2 = \Lambda \), then \( w_1w_2 = bb \).
• If \( w_1 = \Lambda \) and \( w_2 = \Lambda \), then \( w_1w_2 = \Lambda \); i.e., \( \Lambda\Lambda = \Lambda \).

**Definition:** For any string \( w \), we define \( w^n \) for \( n \geq 0 \) inductively as follows:

• \( w^0 = \Lambda \);
• \( w^{n+1} = w^nw \) for any \( n \geq 0 \).

**Example:** If \( w = \text{cat} \), then \( w^0 = \Lambda \), \( w^1 = \text{cat} \), \( w^2 = \text{catcat} \), \( w^3 = \text{catcatcat} \), and so on.

**Definition:** Given a string \( s \), a *substring* of \( s \) is any part of the string \( s \); i.e., \( w \) is a substring of \( s \) if there exist strings \( x \) and \( y \) (either or both possibly null) such that \( s = xwy \).

**Examples:**

• Take the string 472828. Then \( \Lambda \), 282, 4, and 472828 are all substrings of 472828.
• 48 is not a substring of 472828.
Definition: A *formal language* $L$ is a set of strings over an alphabet for which there are explicit rules for the strings in the set. Throughout these notes, we will only consider formal languages, and so we will simplify the discussion by saying *language* instead of *formal language*.

Examples:

- Computer languages, e.g., C or C++ or Java, are formal languages with alphabet $\Sigma = \{a, b, \ldots, z, A, B, \ldots, Z, , 0, 1, 2, \ldots, 9, >, <, =, +, -, *, /, (, ), , , &, !, %, ^, \{, \}, |, ’, :, ; \}$. The rules of syntax define the rules for the language.

- The set of valid variable names in C++ is a formal language. What are the alphabet and rules defining valid variable names in C++?

Definition: Those strings that are permissible in the language $L$ are called *words of the language* $L$.

Remarks:

- A language is just a specific collection of strings.

- We will use the words *string* and *word* interchangeably.

- Thus, for a given string $w$ and a particular language $L$, we might call $w$ a *word* even if it is not in the language $L$.

Let us consider some simple examples of languages:

Example: Alphabet $\Sigma = \{x\}$.
Language

$$L_0 = \{\Lambda, x, xx, xxx, xxxx, \ldots\}$$
$$= \{x^n \text{ for } n = 0, 1, 2, 3, \ldots\}$$
$$= \{x^n : n = 0, 1, 2, 3 \ldots\}$$

where we interpret $x^n$ to be the string of $n$ $x$’s strung together. In particular, $x^0 = \Lambda$. Note that

- $L_0$ includes $\Lambda$ as a word.
• there are different ways we can specify a language.

**Example:** Alphabet $\Sigma = \{x\}$.

Language

$$L_1 = \{x, xx, xxx, xxxx, \ldots\}$$

$$= \{x^n \text{ for } n = 1, 2, 3, \ldots\}$$

$$= \{x^n : n = 1, 2, 3, \ldots\}$$

Note that

• $L_1$ doesn’t include $\Lambda$ as a word.

• there are different ways we can specify a language.

**Example:** Alphabet $\Sigma = \{x\}$.

Language

$$L_2 = \{x, xxx, xxxxx, xxxxxxx, \ldots\}$$

$$= \{x^\text{odd}\}$$

$$= \{x^{2n+1} : n = 0, 1, 2, 3, \ldots\}$$

**Example:** Alphabet $\Sigma = \{0, 1, 2, \ldots, 9\}$.

Language

$$L_3 = \{\text{any string of alphabet letters that does not start with the letter "0"}\}$$

$$= \{1, 2, 3, \ldots, 9, 10, 11, \ldots\}$$

**Definition:** For any set $S$, we use the notation “$w \in S$” to denote that $w$ is an element of the set $S$. Also, we use the notation “$y \notin S$” to denote that $y$ is not an element of the set $S$.

**Example:** If $L_1 = \{x^n : n = 1, 2, 3, \ldots\}$, then $x \in L_1$ and $xxx \in L_1$, but $\Lambda \notin L_1$.

**Definition:** The set $\emptyset$, which is called the *empty set*, is the set consisting of no elements.
CHAPTER 2. LANGUAGES

Fact: Note that \( \Lambda \notin \emptyset \) since \( \emptyset \) has no elements.

Example: Let \( \Sigma = \{a, b\} \), and we can define a language \( L \) consisting of all strings that begin with \( a \) followed by zero or more \( b \)'s; i.e.,

\[
L = \{a, ab, abb, abb, \ldots\} = \{ab^n : n = 0, 1, 2, \ldots\}.
\]

2.3 Set Relations and Operations

Definition: If \( A \) and \( B \) are sets, then \( A \subseteq B \) (\( A \) is a subset of \( B \)) if \( w \in A \) implies that \( w \in B \); i.e., each element of \( A \) is also an element of \( B \).

Examples:

- Suppose \( A = \{ab, ba\} \) and \( B = \{ab, ba, aaa\} \). Then \( A \subseteq B \), but \( B \not\subset A \).
- Suppose \( A = \{x, xx, xxx, \ldots\} \) and \( B = \{\Lambda, x, xx, xxx, \ldots\} \). Then \( A \subseteq B \), but \( B \not\subset A \).
- Suppose \( A = \{ba, ab\} \) and \( B = \{aa, bb\} \). Then \( A \not\subset B \) and \( B \not\subset A \).

Definition: Let \( A \) and \( B \) be 2 sets. \( A = B \) if \( A \subseteq B \) and \( B \subseteq A \).

Examples:

- Suppose \( A = \{ab, ba\} \) and \( B = \{ab, ba\} \). Then \( A \subseteq B \) and \( B \subseteq A \), so \( A = B \).
- Suppose \( A = \{ab, ba\} \) and \( B = \{ab, ba, aaa\} \). Then \( A \subseteq B \), but \( B \not\subset A \), so \( A \neq B \).
- Suppose \( A = \{x, xx, xxx, \ldots\} \) and \( B = \{x^n : n \geq 1\} \). Then \( A \subseteq B \) and \( B \subset A \), so \( A = B \).

Definition: Given two sets of strings \( S \) and \( T \), we define

\[
S + T = \{w : w \in S \text{ or } w \in T\}
\]
to be the union of $S$ and $T$; i.e., $S + T$ consists of all words either in $S$ or in $T$ (or in both).

Examples:

- Suppose $S = \{ab, bb\}$ and $T = \{aa, bb, a\}$. Then $S + T = \{ab, bb, aa, a\}$.

**Definition:** Given two sets $S$ and $T$ of strings, we define

$$S \cap T = \{w : w \in S \text{ and } w \in T\},$$

which is the intersection of $S$ and $T$; i.e., $S \cap T$ consists of strings that are in both $S$ and $T$.

**Definition:** Sets $S$ and $T$ are disjoint if $S \cap T = \emptyset$.

Examples:

- Suppose $S = \{ab, bb\}$ and $T = \{aa, bb, a\}$. Then $S \cap T = \{bb\}$.
- Suppose $S = \{ab, bb\}$ and $T = \{ab, bb\}$. Then $S \cap T = \{ab, bb\}$.
- Suppose $S = \{ab, bb\}$ and $T = \{aa, ba, a\}$. Then $S \cap T = \emptyset$, so $S$ and $T$ are disjoint.

**Definition:** For any 2 sets $S$ and $T$ of strings, we define $S - T = \{w : w \in S, w \notin T\}$.

Examples:

- Suppose $S = \{a, b, bb, bbb\}$ and $T = \{a, bb, bab\}$. Then $S - T = \{b, bbb\}$.
- Suppose $S = \{ab, ba\}$ and $T = \{ab, ba\}$. Then $S - T = \emptyset$.

**Definition:** For any set $S$, we define $|S|$, which is called the cardinality of $S$, to be the number of elements in $S$.

Examples:

- Suppose $S = \{ab, bb\}$ and $T = \{a^n : n \geq 1\}$. Then $|S| = 2$ and $|T| = \infty$. 

• If $S = \emptyset$, then $|S| = 0$.

**Definition:** If $S$ is any set, we say that $S$ is *finite* if $|S| < \infty$. If $S$ is not finite, then we say that $S$ is *infinite*.

**Examples:**

• Suppose $S = \{ab, bb\}$. Then $S$ is finite.
• Suppose $T = \{a^n : n \geq 1\}$. Then $T$ is infinite.

**Fact:** If $S$ and $T$ are 2 disjoint sets (i.e., $S \cap T = \emptyset$), then $|S + T| = |S| + |T|$.

**Fact:** If $S$ and $T$ are any 2 sets such that $|S \cap T| < \infty$, then

$$|S + T| = |S| + |T| - |S \cap T|.$$  

In particular, if $S \cap T = \emptyset$, then $|S + T| = |S| + |T|$.

**Examples:**

• Suppose $S = \{ab, bb\}$ and $T = \{aa, bb, a\}$. Then
  
  - $S + T = \{ab, bb, aa, a\}$
  - $S \cap T = \{bb\}$
  - $|S| = 2$
  - $|T| = 3$
  - $|S \cap T| = 1$
  - $|S + T| = 4$.

• Suppose $S = \{ab, bb\}$ and $T = \{aa, ba, a\}$. Then
  
  - $S + T = \{ab, bb, aa, ba, a\}$
  - $S \cap T = \emptyset$
  - $|S| = 2$
  - $|T| = 3$
  - $|S \cap T| = 0$
  - $|S + T| = 5$.  


CHAPTER 2. LANGUAGES

Definition: The Cartesian product (or direct or cross product) of two sets $A$ and $B$ is the set $A \times B = \{(x, y) : x \in A, y \in B\}$ of ordered pairs.

Examples:

- If $A = \{ab, ba, bbb\}$ and $B = \{bb, ba\}$, then

  \[ A \times B = \{(ab, bb), (ab, ba), (ba, bb), (ba, ba), (bbb, bb), (bbb, ba)\}. \]

  Note that $(ab, ba) \in A \times B$.

  Also, note that

  \[ B \times A = \{(bb, ab), (bb, ba), (bb, bbb), (ba, ab), (ba, ba), (ba, bbb)\}. \]

  Note that $(bb, ba) \in B \times A$, but $(bb, ba) \notin A \times B$, so $B \times A \neq A \times B$.

We can also define the Cartesian product of more than 2 sets.

Definition: The Cartesian product (or direct or cross product) of $n$ sets $A_1, A_2, \ldots, A_n$ is the set

\[ A_1 \times A_2 \times \cdots \times A_n = \{(x_1, x_2, \ldots, x_n) : x_i \in A_i \text{ for } i = 1, 2, \ldots, n\} \]

of ordered $n$-tuples.

Examples:

- Suppose

  \[ A_1 = \{ab, ba, bbb\}, \quad A_2 = \{a, bb\}, \quad A_3 = \{ab, b\}. \]

  Then

  \[ A_1 \times A_2 \times A_3 = \{(ab, a, ab), (ab, a, b), (ab, bb, ab), (ab, bb, b), (ba, a, ab), (ba, a, b), (ba, bb, ab), (ba, bb, b), (bbb, a, ab), (bbb, a, b), (bbb, bb, ab), (bbb, bb, b)\}. \]

  Note that $(ab, a, ab) \in A_1 \times A_2 \times A_3$. 

**Definition:** If $S$ and $T$ are sets of strings, we define the *product set* (or *concatenation*) $ST$ to be

$$ST = \{ w = w_1w_2 : w_1 \in S, w_2 \in T \}$$

**Examples:**

- If $S = \{a, aa, aaa\}$ and $T = \{b, bb\}$, then
  $$ST = \{ab, abb, aab, aabb, aaabb\}$$

- If $S = \{a, ab, aba\}$ and $T = \{\Lambda, b, ba\}$, then
  $$ST = \{a, ab, aba, abb, abba, abab, ababa\}$$

- If $S = \{\Lambda, a, aa\}$ and $T = \{\Lambda, bb, bbbb, bbbbb, \ldots\}$, then
  $$ST = \{\Lambda, a, aa, bb, abb, aabb, bbbb, abbb, \ldots\}$$

**Definition:** For any set $S$, define $2^S$, which is called the *power set*, to be the set of all possible subsets of $S$; i.e., $2^S = \{A : A \subset S\}$.

**Example:** If $S = \{a, bb, ab\}$, then

$$2^S = \{\emptyset, \{a\}, \{bb\}, \{ab\}, \{a, bb\}, \{a, ab\}, \{bb, ab\}, \{a, bb, ab\}\}.$$  

**Fact:** If $|S| < \infty$, then $|2^S| = 2^{|S|}$; i.e., there are $2^{|S|}$ different subsets of $S$.

### 2.4 Functions and Operations

**Definition:** For any string $s$, the *length* of $s$ is the number of letters in $s$. We will sometimes denote the length of a string $s$ by length$(s)$ or by $|s|$.

**Examples:**

- length$(cat) = 3$. Also, $|cat| = 3$. If we define a string $s$ such that $s = cat$, then $|s| = 3$.  

Definition: A function (or operator, operation, map, or mapping) $f$ maps each element in a domain $D$ into a single element in a range $R$. We denote this by $f : D \rightarrow R$. Also, we say that the mapping $f$ is defined on the domain $D$ and that $f$ is an $R$-valued mapping. In particular, if the range $R \subset \mathbb{R}$, i.e., if the range is a subset of the real numbers, then we say that $f$ is a real-valued mapping.

Examples:

- Let $\mathbb{R}$ denote the real numbers, and let $\mathbb{R}_+$ denote the non-negative real numbers. We can define a function $f : \mathbb{R} \rightarrow \mathbb{R}_+$ as $f(x) = x^2$.
- If we define $f$ such that $f(3) = 4$ and $f(3) = 8$, then $f$ is not a function since it maps 3 to more than one value.
- Let $D$ be any collection of strings, and let $R$ be the non-negative integers. Then we can define $f : D \rightarrow R$ to be such that for any string $s \in D$, $f(s) = |s|$, which is the length of $s$.
- We can define a function $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ to be $f(x, y) = x + y$.
- Let $L_1$ and $L_2$ be two sets of strings. Then we can define the concatenation operator as the function $f : L_1 \times L_2 \rightarrow L_1L_2$ such that
  
  $f(w_1, w_2) = w_1w_2$

- Language $L_1 = \{x^n : n \geq 1\}$ from before. Can concatenate $a = xxx$ and $b = x$ to get $ab = xxxx$. Note that $a, b \in L_1$ and that $ab \in L_1$.
- Language $L_2 = \{x^{2n+1} : n \geq 0\}$ from before. Can concatenate $a = xxx$ and $b = x$ to get $ab = xxxx$. Note that $a, b \in L_2$ but that $ab \notin L_2$.

Definition: For a mapping $f$ defined on a domain $D$, we define $f(D) = \{f(x) : x \in D\}$.
i.e., \( f(D) \) is the set of all possible values that the mapping \( f \) can take on when applied to values in \( D \).

**Example:**

- If \( f(x) = x^2 \) and \( D = \mathbb{R} \), then \( f(D) = \mathbb{R}_+ \), the set of non-negative real numbers.

**Definition:** Suppose \( f \) is a mapping defined on a domain \( D \). We say that \( D \) is closed under mapping \( f \) if \( f(D) \subseteq D \); i.e., if \( x \in D \) implies that \( f(x) \in D \). In other words, \( D \) is closed under \( f \) if applying \( f \) to any element in \( D \) results in an element in \( D \).

**Definition:** Suppose \( f \) is a mapping defined on a domain \( D \times D \). We say that \( D \) is closed under mapping \( f \) if \( f(D, D) \subseteq D \); i.e., if \( (x, y) \in D \times D \) implies that \( f(x, y) \in D \).

**Examples:**

- \( L_1 = \{ x^n : n = 1, 2, 3, \ldots \} \) is closed under concatenation.
- \( L_2 = \{ x^{2n+1} : n = 0, 1, 2, \ldots \} \) is not closed under concatenation since \( x \) concatenated with \( x \) yields \( xx \notin L_2 \).

**Definition:** For any string \( w \), the reverse of \( w \), written as \( \text{reverse}(w) \) or \( w^R \), is the same string of letters written in reverse order. Thus, if \( w = w_1w_2\cdots w_n \), where each \( w_i \) is a letter, then \( \text{reverse}(w) = w_nw_{n-1}\cdots w_1 \).

**Examples:**

- For \( xxxx \in L_1 = \{ x^n : n = 1, 2, 3, \ldots \} \), \( \text{reverse}(xxxx) = xxxx \in L_1 \). We can show that \( L_1 \) is closed under reversal.
- Recall \( L_3 \) is the set of strings over the alphabet \( \Sigma = \{ 0, 1, 2, \ldots, 9 \} \) such that the first letter is not 0. For \( 48 \in L_3 \), \( \text{reverse}(48) = 84 \in L_3 \).
- **Example:** For \( 90210 \in L_3 \), \( \text{reverse}(90210) = 01209 \notin L_3 \). Thus, \( L_3 \) is not closed under reversal.
**Definition**: Over the alphabet $\Sigma = \{a, b\}$, the language PALINDROME is defined as

$$\text{PALINDROME} = \{\Lambda \text{ and all strings } x \text{ such that } \text{reverse}(x) = x\} = \{\Lambda, a, b, aa, bb, aaa, aba, \ldots \}$$

Note that for the language PALINDROME, the words $abba, a \in \text{PALINDROME}$, but their concatenation $abbaa$ is not in $\text{PALINDROME}$.

**Definition**: Suppose $f : D \to \mathbb{R}$ and $g : D \to \mathbb{R}$ are real-valued mappings such that $f(x) \leq g(x)$ for all $x \in D$. Then $f$ is a bounded above by $g$, or $g$ is an upper bound for $f$. In addition, if there exists some $x \in D$ such that $f(x) = g(x)$, then we say that $g$ is a tight upper bound for $f$.

**Examples**:

- If $f(x) = \sin(x)$ and $g(x) = 2$ for all $x \in \mathbb{R}$, then $g$ is an upper bound of $f$, but $g$ is not a tight upper bound of $f$.
- If $f(x) = \sin(x)$ and $g(x) = 1$ for all $x \in \mathbb{R}$, then $g$ is a tight upper bound of $f$.
- Suppose $f(x) = x$ and $g(x) = x^2$. Then $g$ is an upper bound for $f$ for all $x \geq 1$, and $g$ is tight since $g(x) = f(x)$ for $x = 1$.
- Suppose $f(x) = x^2$ and $g(x) = 2^x$. Then $g$ is an upper bound for $f$ for all $x \geq 4$. Also, $g$ is a tight upper bound over $x \geq 4$ since $g(x) = f(x)$ for $x = 4$.

### 2.5 Closures

**Definition**: Given an alphabet $\Sigma$, let $\Sigma^*$ be the closure of the alphabet, which is defined to be the language in which any string of letters from $\Sigma$ (with possible repetition of letters) is a word in $\Sigma^*$, even the null string $\Lambda$. This notation is also known as the Kleene star. Thus,

$$\Sigma^* = \{w = w_1 w_2 \cdots w_n : n \geq 0, \ w_i \in \Sigma \text{ for } i = 1, 2, \ldots, n\},$$

where we define $w_1 w_2 \cdots w_n = \Lambda$ when $n = 0$. 


Example: Alphabet $\Sigma = \{x\}$. Then, the closure of $\Sigma$ is
$$\Sigma^* = \{\Lambda, x, xx, xxx, \ldots\}$$

Example: Alphabet $\Sigma = \{0, 1, 2, \ldots, 9\}$. Then, the closure of $\Sigma$ is
$$\Sigma^* = \{\Lambda, 0, 1, 2, \ldots, 9, 00, 01, 02, 03, \ldots\}$$

We can think of the Kleene star as an operation that makes an infinite language (i.e., a language with infinitely many words) out of an alphabet.

Definition: Given a set $S$ of strings, we define $S^n$, $n \geq 1$, to be
$$S^n = SS \cdots S$$
$n$ times
$$= \{w = w_1w_2\cdots w_n : w_i \in S, \ i = 1, 2, \ldots, n\}.$$ 

Note that $S^1 = S$. We also define $S^0 = \{\Lambda\}$.

Example: If $S = \{ab, bbb\}$, then $S^1 = S$, and
$$S^2 = \{abab, abbbb, bbbab, bbbbbb\}$$
$$S^3 = \{ababab, ababbbb, abbbbab, abbbbbbb, bbbabab, bbbabbbb, bbbbbbab, bbbbbbbbb\}$$

We can also apply the star-operator to sets of words:

Definition: If $S$ is a set of words, then $S^*$ is the set of all finite strings formed by concatenating words from $S$, where any word may be used as often as we like, and where the null string is also included; i.e.,
$$S^* = S^0 + S^1 + S^2 + S^3 + \cdots.$$ 

In set notation,
$$S^* = \{w = w_1w_2w_3\cdots w_n : n \geq 0 \text{ and } w_i \in S\ \text{for all } i = 1, 2, 3, \ldots, n\},$$
where we interpret $w_1w_2w_3\cdots w_n$ for $n = 0$ to be the null string $\Lambda$. Thus, $S^0 = \{\Lambda\}$ for any set $S$. In particular, if $S = \emptyset$, we still have $S^0 = \{\Lambda\}$.

Example: If $S = \{ba, a\}$, then
$$S^* = \{\Lambda \text{ plus any word composed of factors of } ba\ \text{and } a\}
= \{\Lambda, a, aa, ba, aaa, aba, baa, aaaa, aaba, \ldots\}.$$
If \( w \in S^* \), can \( bb \) ever be a substring of \( w \)? No.

**Proof.**

- Suppose \( xy \) is a substring of length 2 of \( w \), where \( x \) and \( y \) are single letters.
- Since \( w \in S^* \), we can write \( w = w_1w_2 \cdots w_n \), for some \( n \geq 0 \), where each \( w_i \in S, i = 1, 2, \ldots, n \).
- Since \( S = \{ba, a\} \), there are five possibilities for how the 2-letter substring \( xy \) could have arisen:
  1. \( xy \) is the concatenation of two 1-letter words from \( S \); i.e., for some \( i = 1, 2, \ldots, n-1 \), we have that \( xy = w_iw_{i+1} \), where \( w_i \) and \( w_{i+1} \) are words from \( S \) having only one letter each. Since the only 1-letter word from \( S \) is \( a \), we must have that \( w_i = w_{i+1} = a \). In this case, \( xy = aa \), which is not \( bb \).
  2. \( xy \) is a 2-letter word from \( S \); i.e., for some \( i = 1, 2, \ldots, n \), we have that \( xy = w_i \), where \( w_i \) is a 2-letter word from \( S \). Since the only 2-letter word from \( S \) is \( ba \), we must have that \( w_i = ba \). In this case, \( xy = ba \), which is not \( bb \).
  3. \( xy \) is the concatenation of a 1-letter word from \( S \) and the first letter of a 2-letter word from \( S \); i.e., for some \( i = 1, 2, \ldots, n-1 \), we have that \( xy = w_iw_{i+1,1} \), where
    - \( w_i \) is a 1-letter word from \( S \).
    - \( w_{i+1} \) is a 2-letter word of \( S \) with \( w_{i+1} = w_{i+1,1}w_{i+1,2} \) and \( w_{i+1,1} \) and \( w_{i+1,2} \) are the two letters of \( w_{i+1} \).
    Since the only 1-letter word from \( S \) is \( a \), we must have that \( w_i = a \). Since the only 2-letter word from \( S \) is \( ba \), we must have that \( w_{i+1} = ba \), whose first letter is \( b \). In this case, \( xy = ab \), which is not \( bb \).
  4. \( xy \) is the concatenation of the second letter of a 2-letter word from \( S \) and a 1-letter word from \( S \); i.e., for some \( i = 1, 2, \ldots, n-1 \), we have that \( xy = w_{i,2}w_{i+1} \), where
    - \( w_i \) is a 2-letter word of \( S \) with \( w_i = w_{i,1}w_{i,2} \) and \( w_{i,1} \) and \( w_{i,2} \) are the two letters of \( w_i \).
    - \( w_{i+1} \) is a 1-letter word from \( S \).
    Since the only 2-letter word from \( S \) is \( ba \), we must have that \( w_i = ba \), whose second letter is \( a \). Since the only 1-letter word from \( S \) is \( a \), we must have that \( w_{i+1} = a \). In this case, \( xy = aa \), which is not \( bb \).
5. \(xy\) is the concatenation of the second letter of a 2-letter word from 
\(S\) and the first letter of a 2-letter word from \(S\); i.e., for some \(i = 1, 2, \ldots, n - 1\), we have that 
\(xy = w_{i,2}w_{i+1,1}\), where
- \(w_i\) is a 2-letter word of \(S\) with \(w_i = w_{i,1}w_{i,2}\) and \(w_{i,1}\) and \(w_{i,2}\) 
  are the two letters of \(w_i\).
- \(w_{i+1}\) is a 2-letter word of \(S\) with \(w_{i+1} = w_{i+1,1}w_{i+1,2}\) and \(w_{i+1,1}\) 
  and \(w_{i+1,2}\) are the two letters of \(w_{i+1}\).
Since the only 2-letter word from \(S\) is \(ba\), we must have that 
\(w_i = w_{i+1} = ba\), whose first letter is \(b\) and whose second letter is \(a\). In 
this case, \(xy = ab\), which is not \(bb\).

- This exhausts all of the possibilities for how a 2-letter substring \(xy\) can 
arise in this example. Since all of them result in \(xy \neq bb\), we have 
completed the proof.

\[\text{Example: If } S = \{xx, xxx\}, \text{ then} \]
\[S^* = \{\Lambda \text{ and all strings of more than one } x\} \]
\[= \{\Lambda, xx, xxx, xxxx, \ldots\}\]

To prove that a certain word is in the closure language \(S^*\), we must show how 
it can be written as a concatenation of words in \(S\).

\[\text{Example: If } S = \{ba, a\}, \text{ then } aaba \in S^* \text{ since we can break } aaba \text{ into the} \]
\[\text{factors } a \in S, a \in S, \text{ and } ba \in S; \text{ i.e., } aaba = (a)(a)(ba). \]

Note that there is only one way to do the above factorization into words from 
\(S\); we then say the factorization is \textit{unique}.

\[\text{Example: If } S = \{xx, xxx\}, \text{ then } xxxxxx \in S^* \text{ since } xxxxxx = (xx)(xx)(xx) = \]
\[(xxx)(xxx). \]

Here, the factorization is not unique.

\[\text{Example: If } S = \emptyset, \text{ then } S^* = \{\Lambda\}. \]

\[\text{Example: If } S = \{\Lambda\}, \text{ then } S^* = \{\Lambda\}. \]

\textbf{Remarks:}
• Two words are considered the same if all their letters are the same and in the same order, so there is only one possible word of no letters, Λ.

• There is an important difference between the word that has no letters Λ and the language that has no words, which we denote by ∅.

• It is not true that Λ is a word in the language ∅ since ∅ doesn’t have any words at all.

• If a language $L$ does not contain the word Λ and we wish to add it to $L$, we use the “union of sets” operation denoted by “+” to form $L + \{Λ\}$.

• Note that $L \neq L + \{Λ\}$ if Λ ∉ $L$.

• Note that $L = L + ∅$.

Definition: If $S$ is some set of words, then $S^+ = S^1 + S^2 + S^3 + \cdots$, which is the set of all finite strings formed by concatenating some positive number of strings from $S$.

Example: If $Σ = \{x\}$, then $Σ^+ = \{x, xx, xxx, \ldots\}$.

Definition: If $A$ and $B$ are sets, then $A \subset B$ ($A$ is a subset of $B$) if $w \in A$ implies that $w \in B$; i.e., each element of $A$ is also an element of $B$.

Suppose that we have two sets $A$ and $B$, and we want to prove that $A = B$. One way of proving this is to show that

1. $A \subset B$, and
2. $B \subset A$.

Example: Suppose $A = \{x, xx\}$ and $B = \{x, xx, xxx\}$. Note that $A \subset B$, but $B \not\subset A$, and so $A \neq B$.

Theorem 1 For any set $S$ of strings, we have that $S^* = S^{**}$.

Proof. The way we will prove this is by showing two things:

1. $S^{**} \subset S^*$
CHAPTER 2. LANGUAGES

2. \( S^* \subset S^{**} \)

To show part 1, we have to prove that any word \( w_0 \) in \( S^{**} \) is also in \( S^* \).

- Note that since \( w_0 \in S^{**} \), \( w_0 \) is made up of factors, say \( w_1, w_2, \ldots, w_k \), \( k \geq 0 \), from \( S^* \); i.e., \( w_0 = w_1w_2\cdots w_k \), with \( k \geq 0 \) and \( w_i \in S^* \) for \( i = 1, 2, \ldots, k \).

- Also, each factor \( w_i, i = 1, 2, \ldots, k \), is from \( S^* \), and so it is made up of a nonnegative number of factors from \( S \); i.e., \( w_i = w_{i,1}w_{i,2}\cdots w_{i,n_i} \), with \( n_i \geq 0 \) and \( w_{i,j} \in S \) for \( j = 1, 2, \ldots, n_i \).

- Therefore, we can write

\[
\begin{align*}
  w_0 &= w_1w_2\cdots w_k \\
  &= w_{1,1}w_{1,2}\cdots w_{1,n_1}w_{2,1}w_{2,2}\cdots w_{2,n_2}\cdots w_{k,1}w_{k,2}\cdots w_{k,n_k},
\end{align*}
\]

where each \( w_{i,j} \in S, i = 1, 2, \ldots, k, j = 1, 2, \ldots, n_i \). So the original word \( w_0 \in S^{**} \) is made up of factors from \( S \).

- But \( S^* \) is just the language made up of the different factors in \( S \).

- Therefore, \( w_0 \in S^* \).

- Since \( w_0 \) was arbitrary, we have just shown that every word in \( S^{**} \) is also a word in \( S^* \); i.e., \( S^{**} \subset S^* \).

To show part 2, note that in general, for any set \( A \), we know that \( A \subset A^* \). Hence, letting \( A = S^* \), we see that \( S^* \subset S^{**} \).
Chapter 3

Recursive Definitions

3.1 Definition

A recursive definition is characteristically a three-step process:

1. First, we specify some basic objects in the set. The number of basic objects specified must be finite.

2. Second, we give a finite number of rules for constructing more objects in the set from the ones we already know.

3. Third, we declare that no objects except those constructed in this way are allowed in the set.

3.2 Examples

Example: Consider the set P-EVEN, which is the set of positive even numbers.

We can define the set P-EVEN in several different ways:

- We can define P-EVEN to be the set of all positive integers that are evenly divisible by 2.

- P-EVEN is the set of all $2n$, where $n = 1, 2, \ldots$. 
• P-EVEN is defined by these three rules:

  **Rule 1** 2 is in P-EVEN.
  **Rule 2** If \( x \) is in P-EVEN, then so is \( x + 2 \).
  **Rule 3** The only elements in the set P-EVEN are those that can be produced from the two rules above.

Note that the first two definitions of P-EVEN are much easier to apply than the last.

In particular, to show that 12 is in P-EVEN using the last definition, we would have to do the following:

1. 2 is in P-EVEN by Rule 1.
2. \( 2 + 2 = 4 \) is in P-EVEN by Rule 2.
3. \( 4 + 2 = 6 \) is in P-EVEN by Rule 2.
4. \( 6 + 2 = 8 \) is in P-EVEN by Rule 2.
5. \( 8 + 2 = 10 \) is in P-EVEN by Rule 2.
6. \( 10 + 2 = 12 \) is in P-EVEN by Rule 2.

We can make another definition for P-EVEN as follows:

**Rule 1** 2 is in P-EVEN.
**Rule 2** If \( x \) and \( y \) are both in P-EVEN, then \( x + y \) is in P-EVEN.
**Rule 3** No number is in P-EVEN unless it can be produced by rules 1 and 2.

Can use the new definition of P-EVEN to show that 12 is in P-EVEN:

1. 2 is in P-EVEN by Rule 1.
2. \( 2 + 2 = 4 \) is in P-EVEN by Rule 2.
3. \( 4 + 4 = 8 \) is in P-EVEN by Rule 2.
4. \( 4 + 8 = 12 \) is in P-EVEN by Rule 2.
Example: Let PALINDROME be the set of all strings over the alphabet \( \Sigma = \{a, b\} \) that are the same spelled forward as backwards; i.e., \( \text{PALINDROME} = \{w : w = \text{reverse}(w)\} = \{\Lambda, a, b, aa, bb, aab, bab, bba, aaaa, abba, \ldots\} \).

A recursive definition for PALINDROME is as follows:

**Rule 1** \( \Lambda, a, \) and \( b \) are in PALINDROME.

**Rule 2** If \( w \in \text{PALINDROME} \), then so are \( awa \) and \( bwb \).

**Rule 3** No other string is in PALINDROME unless it can be produced by rules 1 and 2.

Example: Let us now define a set AE of certain valid arithmetic expressions. The set AE will not include all possible arithmetic expressions.

The alphabet of AE is

\[ \Sigma = \{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ +\ -\ \ast\ /\ (\ )\} \]

We recursively define AE using the following rules:

**Rule 1** Any number (positive, negative, or zero) is in AE.

**Rule 2** If \( x \) is in AE, then so are \( (x) \) and \( -(x) \).

**Rule 3** If \( x \) and \( y \) are in AE, then so are

(i) \( x + y \) (if the first symbol in \( y \) is not \( - \))
(ii) \( x - y \) (if the first symbol in \( y \) is not \( - \))
(iii) \( x \ast y \)
(iv) \( x/y \)
(v) \( x \ast y \) (our notation for exponentiation)

**Rule 4** AE consists of only those things can be created by the above three rules.

For example,

\[ (5 \ast (8 + 2)) \]
and
\[ 5 - (8 + 1)/3 \]
are in AE since they can be generated using the above definition.
However,
\[ ((6 + 7)/9 \]
and
\[ 4(/9 * 4) \]
are not since they cannot be generated using the above definition.
Now we can use our recursive definition of AE to show that
\[ 8 * 6 - ((4/2) + (3 - 1) * 7)/4 \]
is in AE.

1. Each of the numbers are in AE by Rule 1.
2. \( 8 * 6 \) is in AE by Rule 3(iii).
3. \( 4/2 \) is in AE by Rule 3(iv).
4. \( (4/2) \) is in AE by Rule 2.
5. \( 3 - 1 \) is in AE by Rule 3(ii).
6. \( (3 - 1) \) is in AE by Rule 2.
7. \( (3 - 1) * 7 \) is in AE by Rule 3(iii).
8. \( (4/2) + (3 - 1) * 7 \) is in AE by Rule 3(i).
9. \( ((4/2) + (3 - 1) * 7) \) is in AE by Rule 2.
10. \( ((4/2) + (3 - 1) * 7)/4 \) is in AE by Rule 3(iv).
11. \( 8 * 6 + ((4/2) + (3 - 1) * 7)/4 \) is in AE by Rule 3(i).
Chapter 4

Regular Expressions

4.1 Some Definitions

Definition: If $S$ and $T$ are sets of strings of letters (whether they are finite or infinite sets), we define the product set of strings of letters to be

$$ST = \{ w = w_1w_2 : w_1 \in S, w_2 \in T \}$$

Example: If $S = \{a, aa, aaa\}$ and $T = \{b, bb\}$, then

$$ST = \{ab, abb, aab, aabb, aaab, aaabb\}$$

Example: If $S = \{a, ab, aba\}$ and $T = \{\Lambda, b, ba\}$, then

$$ST = \{a, ab, aba, abb, abba, abab, ababa\}$$

Example: If $S = \{\Lambda, a, aa\}$ and $T = \{\Lambda, bb, bbbb, bbbbb\, \ldots\}$, then

$$ST = \{\Lambda, a, aa, bb, abb, aabb, bbbb, abbb, \ldots\}$$

Definition: Let $s$ and $t$ be strings. Then $s$ is a substring of $t$ if there exist strings $u$ and $v$ such that $t = usv$. 
Example: Suppose $s = aba$ and $t = aababb$. Then $s$ is a substring of $t$ since we can define $u = a$ and $v = bb$, and then $t = usv$.

Example: Suppose $s = abb$ and $t = aaabb$. Then $s$ is a substring of $t$ since we can define $u = aa$ and $v = \Lambda$, and then $t = usv$.

Example: Suppose $s = bb$ and $t = aababa$. Then $s$ is not a substring of $t$.

Definition: Over the alphabet $\Sigma = \{a, b\}$, a string contains a double letter if it has either $aa$ or $bb$ as a substring.

Example: Over the alphabet $\Sigma = \{a, b\}$,

1. The string $abaabab$ contains a double letter.
2. The string $bb$ contains a double letter.
3. The string $aba$ does not contain a double letter.
4. The string $abbba$ contains two double letters.

### 4.2 Defining Languages Using Regular Expressions

Previously, we defined the languages:

- $L_1 = \{x^n \text{ for } n = 1, 2, 3, \ldots\}$
- $L_2 = \{x, xxx, xxxx, \ldots\}$

But these are not very precise ways of defining languages.

- So we now want to be very precise about how we define languages, and we will do this using regular expressions.
• Languages that are associated with these regular expressions are called regular languages and are also said to be defined by a finite representation.

• Regular expressions are written in bold face letters and are a way of specifying the language.

• Recall that we previously saw that for sets $S, T$, we defined the operations
  - $S + T = \{ w : w \in S \text{ or } w \in T \}$
  - $ST = \{ w = w_1w_2 : w_1 \in S, w_2 \in T \}$
  - $S^* = S^0 + S^1 + S^2 + \cdots$
  - $S^+ = S^1 + S^2 + \cdots$

• We will precisely define what a regular expression is later. But for now, let’s work with the following sketchy description of a regular expression.

• Loosely speaking, a regular expression is a way of specifying a language in which the only operations allowed are
  - union (+),
  - concatenation (or product),
  - Kleene-* closure,
  - superscript-+

The allowable symbols are parentheses, $\Lambda$, and $\emptyset$, as well as each letter in $\Sigma$ written in boldface. No other symbols are allowed in a regular expression. Also, a regular expression must only consist of a finite number of symbols.

• To introduce regular expressions, think of

  $$x = \{ x \};$$

  i.e., $x$ represents the language (i.e., set) consisting of exactly one string, $x$. Also, think of

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

  so $a$ is the language consisting of exactly one string $a$, and $b$ is the language consisting of exactly one string $b$. 
• Using this interpretation, we can interpret $ab$ to mean

$$ab = \{a\}\{b\} = \{ab\}$$

since the concatenation (or product) of the two languages $\{a\}$ and $\{b\}$ is the language $\{ab\}$.

• We can also interpret $a + b$ to mean

$$a + b = \{a\} + \{b\} = \{a,b\}$$

• We can also interpret $a^*$ to mean

$$a^* = \{a\}^* = \{\Lambda, a, aa, aaa, \ldots\}$$

• We can also interpret $a^+$ to mean

$$a^+ = \{a\}^+ = \{a, aa, aaa, \ldots\}$$

• Also, we have

$$(ab + a)^*b = (\{a\}\{b\} + \{a\})^*\{b\} = \{ab, a\}^*\{b\}$$

Example: Previously, we saw language

$$L_4 = \{\Lambda, x, xx, xxx, \ldots\} = \{x\}^* = \text{language}(x^*)$$

Example: Language

$$L_1 = \{x, xx, xxx, xxxx, \ldots\} = \text{language}(xx^*) = \text{language}(x^*x) = \text{language}(x^+) = \text{language}(x^*xx^*) = \text{language}(x^*x^+)$$

Note that there are several different regular expressions associated with $L_1$. 
Example: alphabet $\Sigma = \{a, b\}$
language $L$ of all words of the form one $a$ followed by some number (possibly zero) of $b$'s.

$$L = \text{language}(ab^*)$$

Example: alphabet $\Sigma = \{a, b\}$
language $L$ of all words of the form some positive number of $a$’s followed by exactly one $b$.

$$L = \text{language}(aa^*b)$$

Example: alphabet $\Sigma = \{a, b\}$
language $L = \text{language}(ab^*a)$, which is the set of all strings of $a$’s and $b$’s that have at least two letters, that begin and end with one $a$, and that have nothing but $b$’s inside (if anything at all).

$$L = \{aa, aba, abba, abbb, \ldots\}$$

Example: alphabet $\Sigma = \{a, b\}$
The language $L$ consisting of all possible words over the alphabet $\Sigma$ has the following regular expression:

$$(a + b)^*$$

Other regular expressions for $L$ include $(a^*b^*)^*$ and $(\Lambda + a + b)^*$.

Example: alphabet $\Sigma = \{x\}$
language $L$ with an even number (possibly zero) of $x$’s

$$L = \{\Lambda, xx, xxxx, xxxxxx, \ldots\} = \text{language}((xx)^*)$$

Example: alphabet $\Sigma = \{x\}$
language $L$ with a positive even number of $x$’s

$$L = \{xx, xxxx, xxxxxx, \ldots\} = \text{language}(xx(xx)^*) = \text{language}((xx)^+)$$
Example: alphabet $\Sigma = \{x\}$
language $L$ with an odd number of $x$’s

$$L = \{x, xxx,xxxxx,\ldots\}$$

$= \text{language}(x(xx)^*)$

$= \text{language}((xx)^*x)$

Is $L = \text{language}(x^*xx^*)$ ?
No, since it includes the word $(xx)x(x)$.

Example: alphabet $\Sigma = \{a, b\}$
language $L$ of all three-letter words starting with $b$

$$L = \{baa, bab, bba, bbb\}$$

$= \text{language}(b(a+b)(a+b))$

$= \text{language}(baa + bab + bba + bbb)$

Example: alphabet $\Sigma = \{a, b\}$
language $L$ of all words starting with $a$ and ending with $b$

$$L = \{ab, aab, abb, aabb, abab, abbb, \ldots\}$$

$= \text{language}(a(a + b)^*b)$

Example: alphabet $\Sigma = \{a, b\}$
language $L$ of all words starting and ending with $b$

$$L = \{b, bb, bab, bbb, baab, babb, bbab, bbbb, \ldots\}$$

$= \text{language}(b + b(a + b)^*b)$

Example: alphabet $\Sigma = \{a, b\}$
language $L$ of all words with exactly two $b$’s

$L = \text{language}(a^*ba^*ba^*)$

Example: alphabet $\Sigma = \{a, b\}$
language $L$ of all words with at least two $b$’s

$L = \text{language}((a + b)^*b(a + b)^*b(a + b)^*)$
Note that $bbaaba \in L$ since

$$bbaaba = (\Lambda)b(\Lambda)b(aaba) = (b)b(aa)b(a)$$

**Example:** alphabet $\Sigma = \{a, b\}$
language $L$ of all words with at least two $b$’s

$$L = \text{language}(a^*ba^*b(a + b)^*)$$

Note that $bbaaba \in L$ since $bbaaba = \Lambda b \Lambda b aaba$

**Example:** alphabet $\Sigma = \{a, b\}$
language $L$ of all words with at least one $a$ and at least one $b$

$$L = \text{language}((a + b)^*a(a + b)^*b(a + b)^* + (a + b)^*b(a + b)^*a(a + b)^*)$$

$$= \text{language}((a + b)^*a(a + b)^*b(a + b)^* + bb^*aa^*)$$

where

- the first regular expression comes from separately considering the two cases:
  1. requiring an $a$ before a $b$, 
  2. requiring a $b$ before an $a$.
- the second expression comes from the observation that the first term in the first expression only omits words that are of the form some $b$’s followed by some $a$’s.

**Example:** alphabet $\Sigma = \{a, b\}$
language $L$ consists of $\Lambda$ and all strings that are either all $a$’s or $b$ followed by a nonnegative number of $a$’s

$$L = \text{language}(a^* + ba^*)$$

$$= \text{language}((\Lambda + b)a^*)$$

**Theorem 5** If $L$ is a finite language, then $L$ can be defined by a regular expression.
Proof. To make a regular expression that defines the language \( L \), turn all the words in \( L \) into boldface type and put pluses between them.

Example: language

\[ L = \{aba, \ abba, \ bbaab\} \]

Then a regular expression to define \( L \) is

\[ aba + abba + bbaab \]

4.3 The Language EVEN-EVEN

Example: Consider the regular expression

\[ E = [aa + bb + (ab + ba)(aa + bb)^*(ab + ba)]^* \]

We now prove that the regular expression \( E \) generates the language EVEN-EVEN, which consists exactly of all strings that have an even number of \( a \)'s and an even number of \( b \)'s; i.e.,

\[ \text{EVEN-EVEN} = \{\Lambda, \ aa, \ bb, \ aabb, \ abab, \ abba, \ baba, \ bbaa, \ aaaabb, \ldots\} \]

Proof.

- Let \( L_1 \) be the language generated by the regular expression \( E \).
- Let \( L_2 \) be the language EVEN-EVEN.
- So we need to prove that \( L_1 = L_2 \), which we will do by showing that \( L_1 \subseteq L_2 \) and \( L_2 \subseteq L_1 \).
- First note that any word generated by \( E \) is made up of “syllables” of three types:

  \[
  \begin{align*}
  \text{type}_1 &= \text{aa} \\
  \text{type}_2 &= \text{bb} \\
  \text{type}_3 &= (ab + ba)(aa + bb)^*(ab + ba) \\
  E &= [\text{type}_1 + \text{type}_2 + \text{type}_3]^*
  \end{align*}
  \]
• We first show that \( L_1 \subset L_2 \):

- Consider any string \( w \in L_1 \); i.e., \( w \) can be generated by the regular expression \( E \).
- We need to show that \( w \in L_2 \).
- Note that since \( w \) can be generated by the regular expression \( E \), the string \( w \) must be made up of syllables of type 1, 2, or 3.
- Each of these types of syllables generate an even number of a’s and an even number of b’s.
  * type 1 syllable generates 2 a’s and 0 b’s.
  * type 2 syllable generates 0 a’s and 2 b’s.
  * type 3 syllable \((ab + ba)(aa + bb)^*(ab + ba)\) generates
    - exactly 1 a and 1 b at the beginning,
    - exactly 1 a and 1 b at the end,
    - and generates either 2 a’s or 2 b’s at a time in the middle.
    - Thus, the type 3 syllable generates an even number of a’s and an even number of b’s.
- Thus, the total string must have an even number of a’s and an even number of b’s.
- Therefore, \( w \in \text{EVEN-EVEN} \), so we can conclude that \( L_1 \subset L_2 \).

• Now we want to show that \( L_2 \subset L_1 \); i.e., we want to show that any word with an even number of a’s and an even number of b’s can be generated by \( E \).

- Consider any string \( w = w_1w_2w_3 \cdots w_n \) with an even number of a’s and an even number of b’s.
- If \( w = \Lambda \), then iterate the outer star of the regular expression \( E \) zero times to generate \( \Lambda \).
- Now assume that \( w \neq \Lambda \).
- Let \( n = \text{length}(w) \).
- Note that \( n \) is even since \( w \) consists solely of a’s and b’s and since the number of a’s is even and the number of b’s is even.
- Thus, we can read in the string \( w \) two letters at a time from left to right.
Use the following algorithm to generate \( w = w_1w_2w_3 \cdots w_n \) using the regular expression \( E \):

1. Let \( i = 1 \).

2. Do the following while \( i \leq n \):
   
   (a) If \( w_i = a \) and \( w_{i+1} = a \), then iterate the outer star of \( E \) and use the type 1 syllable \( aa \).
   
   (b) If \( w_i = b \) and \( w_{i+1} = b \), then iterate the outer star of \( E \) and use the type 2 syllable \( bb \).
   
   (c) If \( (w_i = a \text{ and } w_{i+1} = b) \) or if \( (w_i = b \text{ and } w_{i+1} = a) \), then choose the type 3 syllable \( (ab + ba)(aa + bb)^*(ab + ba) \), and do the following:
      
      * If \( (w_i = a \text{ and } w_{i+1} = b) \), then choose \( ab \) in the first part of the type 3 syllable.
      
      * If \( (w_i = b \text{ and } w_{i+1} = a) \), then choose \( ba \) in the first part of the type 3 syllable.
      
      * Do the following while either \( (w_{i+2} = a \text{ and } w_{i+3} = a) \) or \( (w_{i+2} = b \text{ and } w_{i+3} = b) \):
         
         * Let \( i = i + 2 \).
         
         * If \( w_i = a \text{ and } w_{i+1} = a \), then iterate the inner star of the type 3 syllable, and use \( aa \).
         
         * If \( w_i = b \text{ and } w_{i+1} = b \), then iterate the inner star of the type 3 syllable, and use \( bb \).

   * Let \( i = i + 2 \).

   * If \( (w_i = a \text{ and } w_{i+1} = b) \), then choose \( ab \) in the last part of the type 3 syllable.

   * If \( (w_i = b \text{ and } w_{i+1} = a) \), then choose \( ba \) in the last part of the type 3 syllable.

   * Remarks:
      
      * We must eventually read in either \( ab \) or \( ba \), which balances out the previous unbalanced pair. This completes a syllable of type 3.
      
      * If we never read in the second unbalanced pair, then either the number of \( a \)'s is odd or the number of \( b \)'s is odd, which is a contradiction.

(d) Let \( i = i + 2 \).
This algorithm shows how to use the regular expression $E$ to generate any string in EVEN-EVEN; i.e., if $w \in$ EVEN-EVEN, then we can use the above algorithm to generate $w$ using $E$.

Thus, $L_2 \subset L_1$.

### 4.4 More Examples and Definitions

**Example:** $b^* (abb^*)^* (\Lambda + a)$ generates the language of all words without a double $a$.

**Example:** What is a regular expression for all valid variable names in C?

**Definition:** The set of regular expressions is defined by the following:

**Rule 1** Every letter of $\Sigma$ can be made into a regular expression by writing it in boldface; $\Lambda$ and $\emptyset$ are regular expressions.

**Rule 2** If $r_1$ and $r_2$ are regular expressions, then so are

1. $(r_1)$
2. $r_1r_2$
3. $r_1 + r_2$
4. $r_1^*$ and $r_1^+$

**Rule 3** Nothing else is a regular expression.

**Definition:** For a regular expression $r$, let $L(r)$ denote the language generated by (or associated with) $r$; i.e., $L(r)$ is the set of strings that can be generated by $r$.

**Definition:** The following rules define the *language associated* with (or generated by) any regular expression:

**Rule 1 (i)** If $\ell \in \Sigma$, then $L(\ell) = \{\ell\}$; i.e., the language associated with the regular expression that is just a single letter is that one-letter word alone.
(ii) $L(\Lambda) = \{\Lambda\}$; i.e., the language associated with $\Lambda$ is $\{\Lambda\}$, a one-word language.

(iii) $L(\emptyset) = \emptyset$; i.e., the language associated with $\emptyset$ is $\emptyset$, the language with no words.

**Rule 2** If $r_1$ is a regular expression associated with the language $L_1$ and $r_2$ is a regular expression associated with the language $L_2$, then

(i) The regular expression $(r_1)(r_2)$ is associated with the language $L_1$ concatenated with $L_2$:

$$\text{language}(r_1r_2) = L_1L_2.$$  

We define $\emptyset L_1 = L_1 \emptyset = \emptyset$.

(ii) The regular expression $r_1 + r_2$ is associated with the language formed by the union of the sets $L_1$ and $L_2$:

$$\text{language}(r_1 + r_2) = L_1 + L_2$$

(iii) The language associated with the regular expression $(r_1)^*$ is $L_1^*$, the Kleene closure of the set $L_1$ as a set of words:

$$\text{language}(r_1^*) = L_1^*$$

(iv) The language associated with the regular expression $(r_1)^+$ is $L_1^+$:

$$\text{language}(r_1^+) = L_1^+$$
Chapter 5

Finite Automata

5.1 Introduction

- Modern computers are often viewed as having three main components:
  1. the central processing unit (CPU)
  2. memory
  3. input-output devices (IO)
- The CPU is the “thinker”
  1. Responsible for such things as individual arithmetic computations and logical decisions based on particular data items.
  2. However, the amount of data the unit can handle at any one time is fixed forever by its design.
  3. To deal with more than this predetermined, limited amount of information, it must ship data back and forth, over time, to and from the memory and IO devices.
- Memory
  1. The memory may in practice be of several different kinds, such as magnetic core, semiconductor, disks, and tapes.
  2. The common feature is that the information capacity of the memory is vastly greater than what can be accommodated, at any one instant of time, in the CPU.
3. Therefore, this memory is sometimes called *auxiliary*, to distinguish it from the limited storage that is part of the CPU.

4. At least in theory, the memory can be expanded without limit, by adding more core boxes, more tape drives, etc.

- IO devices are the means by which information is communicated back and forth to the outside world; e.g.,
  
  1. terminals
  2. printers
  3. tapes

We now will study a severely restricted model of an actual computer called a *finite automaton* (FA).

- Like a real computer, it has a central processor with fixed finite capacity, depending on its original design.
- Unlike a real computer, it has no auxiliary memory at all.
- It receives its input as a string of characters.
- It delivers no output at all, except an indication of whether the input is considered acceptable.
- It is a language-recognition device.

Why should we study such a simple model of computer with no memory?

- Actually, finite automata do have memory, but the amount they have is fixed and cannot be expanded.
- Finite automata are applicable to the design of several common types of computer algorithms and programs.
  
  - For example, the lexical analysis phase of a compiler is often based on the simulation of a finite automaton.
  
  - The problem of finding an occurrence of one string within another — for example, a particular word within a large text file — can also be solved efficiently by methods originating from the theory of finite automata.
To introduce finite automata, consider the following scenario:

- Play a board game in which two players move pieces around different squares.
- Throw dice to determine where to move.
- Players have no choices to make when making their move. The move is completely determined by the dice.
- A player wins if after 10 throws of the dice, his piece ends up on a certain square.
- Note that no skill or choice is involved in the game.
- Each possible position of pieces on the board is called a state.
- Every time the dice are thrown, the state changes according to what came up on the dice.
- We call the winning square a final state (also known as a halting state, terminal state, or accepting state).
- There may be more than one final state.

Let’s look at another simple example

- Suppose you have a simple computer (machine), as described above.
- Your goal is to write a program to compute $3 + 4$.
- The program is a sequence of instructions that are fed into the computer one at a time.
- Each instruction is executed as soon as it is read, and then the next instruction is read.
- If the program is correct, then the computer outputs the number 7 and terminates execution.
- We can think of taking a snapshot of the internals (i.e., contents of memory, etc.) of the computer after every instruction is executed.
Each possible configuration of 0’s and 1’s in the cells of memory represents a different state of the system.

We say the machine ends in a final state (also called a halting, terminal, or accepting state) if when the program finishes executing, it outputs the number 7.

Two machines are in the same state if their output pages look the same and their memories look the same cell by cell.

The computer is deterministic, i.e., on reading one particular input instruction, the machine converts itself from one given state to some particular other state (which is possibly the same), where the resultant state is completely determined by the prior state and the input instruction. No choice is involved.

The success of the program (i.e., it outputs 7) is completely determined by the sequence of inputs (i.e., the lines of code).

We can think of the set of all computer instructions as the letters of an alphabet.

We can then define a language to be the set of all words over this alphabet that lead to success.

This is the language with words that are all programs that print a 7.

5.2 Finite Automata

Definition: A finite automaton (FA), also known as a finite acceptor, is a collection \( M = (K, \Sigma, \pi, s, F) \) where:

1. \( K \) is a finite set of states.

   - Exactly one state \( s \in K \) is designated as the initial state (or start state).
   
   - Some set \( F \subset K \) is the set of final states, where we allow \( F = \emptyset \) or \( F = K \) or \( F \) could be any other subset of \( K \).
2. An alphabet $\Sigma$ of possible input letters, from which are formed strings, that are to be read one letter at a time.

3. $\pi : K \times \Sigma \to K$ is the transition function.

   - In other words, for each state and for each letter of the input alphabet, the function $\pi$ tells which (one) state to go to next; i.e., if $x \in K$ and $\ell \in \Sigma$, then $\pi(x, \ell)$ is the state that you go to when you are in state $x$ and read in $\ell$.
   - For each state $x$ and each letter $\ell \in \Sigma$, there is exactly one arc leaving $x$ labeled with $\ell$.
   - Thus, there is no choice in how to process a string, and so the machine is deterministic.

An FA works as follows:

- It is presented with an input string of letters.
- It starts in the start state.
- It reads the string one letter at a time, starting from the left.
- The letters read in determine a sequence of states visited.
- Processing ends after the last input letter has been read.
- If after reading the entire input string the machine ends up in a final state, then the input string is accepted. Otherwise, the input string is rejected.

Example: Consider an FA with three states ($x$, $y$, and $z$) with input alphabet $\Sigma = \{a, b\}$.

Define the following transition table for the FA:

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>$x$</td>
<td>$y$</td>
</tr>
<tr>
<td></td>
<td>$y$</td>
<td>$x$</td>
</tr>
<tr>
<td>final</td>
<td>$z$</td>
<td>$z$</td>
</tr>
</tbody>
</table>

Input the string $aaaa$ to the FA:
Start in state \( x \) and read in first \( a \), which takes us to state \( y \).

- From state \( y \), read in second \( a \), which takes us to state \( x \).
- From state \( x \), read in third \( a \), which takes us to state \( y \).
- From state \( y \), read in fourth \( a \), which takes us to state \( x \).
- No more letters in input string so stop.

Note that on input \( aaaa \),

- We ended up in state \( x \), which is not a final state.
- We say that \( aaaa \) is not accepted or rejected by this FA.

Now consider the input string \( abab \):

- Start in state \( x \) and read in first \( a \), which takes us to state \( y \).
- From state \( y \), read in second letter, which is \( b \), which takes us to state \( z \).
- From state \( z \), read in third letter, which is \( a \), which takes us to state \( z \).
- From state \( z \), read in fourth letter, which is \( b \), which takes us to state \( z \).
- No more letters in input string so stop.

On the input string \( abab \):

- We ended up in state \( z \), which is a final state.
- We say that \( abab \) is accepted by this FA.

**Definition:** The set of all strings accepted is the *language associated* with or accepted by the FA.

Note that

- the above FA accepts all strings that have the letter \( b \) in them and no other strings.
the language accepted by this FA is the one defined by the regular expression

\[(a + b)^*b(a + b)^*\]

Can also draw \textit{transition diagram}:

- directed graph
- directed edge
- every state has as many \textit{outgoing edges} as there are letters in the alphabet.
- it is possible for a state to have no \textit{incoming edges}.
- the start state is labeled with a \textit{−}.
- final states are labeled with a \textit{+}.
- some states are neither labeled with \textit{−} or \textit{+}. 
Example: From before.

### 5.3 Examples of FA

Example: regular expression

\[(a + b)(a + b)^* = (a + b)^+\]

All strings over the alphabet \(\Sigma = \{a, b\}\) except \(\Lambda\).
Example: regular expression

\[(a + b)^*\]

This FA accepts all strings over the alphabet \(\Sigma = \{a, b\}\) including \(\Lambda\).
There are FA’s that accept the language having no words:

- FA has no final states

- Final state cannot be reached from start state because graph disconnected.

- Final state cannot be reached from start state because no path
Example: Build FA to accept all words in the language

\[ a(a + b)^* \]

\[ a \rightarrow \quad + \quad a, b \]

\[ - \rightarrow \quad b \quad a, b \]

or

\[ a \rightarrow \quad + \quad a, b \]

\[ - \rightarrow \quad b \quad a, b \]

Note that

- more than one possible FA for any given language
- can have more than one final state
Example:

Note that

- $ababa$ is not accepted.
- $baaba$ is accepted.
- FA accepts strings that have a double letter.
- Regular expression of language

$$(a + b)^*(aa + bb)(a + b)^*$$
Example:

- Only accepts words whose third and fourth letters are $ab$.
- Rejects all other words
- Regular expressions:
  1. $(aaab + abab + baab + bbab)(a + b)^*$
  2. $(a + b)(a + b)ab(a + b)^*$
Example: Only accepts the word $aba$.

Example: Only accepts the words $aba$ and $ba$. 
Example: Regular expression:

\[(a + ba^*ba^*b)^+\]

Language with words having at least one letter and the number of \(b\)'s divisible by 4.

Example: Only accepts the word \(\Lambda\).
Example: Regular expression:

\[(a + b)^* b\]

- Words that end with \(b\)
- does not include \(\Lambda\).

Example: Regular expression:

\[\Lambda + (a + b)^* b\]

Either \(\Lambda\) or words that end in \(b\); i.e., words that do not end in \(a\).
Example: Regular expression:

\[(a + b)^*aa + (a + b)^*bb\]

Words that end in a double letter.
Example: EVEN-EVEN

Note that

- Every $b$ moves us either left or right.
- Every $a$ moves us either up or down.
Chapter 6

Transition Graphs

6.1 Introduction

Each FA has the following properties (among others):

- For each state $x$ and each letter $\ell \in \Sigma$, there is exactly one arc leaving $x$ labeled with $\ell$.
- Can only read one letter at a time when traversing an arc.
- Exactly one start state.

Now we want a different kind of machine that relaxes the above requirements:

- For each state $x$ and each letter $\ell \in \Sigma$, we do not require that there is exactly one arc leaving $x$ labeled with $\ell$.
- Able to read any number of letters at a time when traversing an arc. Specifically, each arc is now labeled with a string $s \in \Sigma^*$, so the string $s$ might be $\Lambda$ or it might be a single letter $\ell \in \Sigma$.
- If an arc is labeled with $\Lambda$, we traverse the arc without reading any letters from the input string.
- If an arc is labeled with a non-empty string $s \in \Sigma^*$, we can traverse the arc if and only if the next unread letter(s) from the original input string are the string $s$. 
• Suppose that we are in a state and we cannot leave the state because there is no arc leaving the state labeled with a string that corresponds to the next unread letters from the input string. Then if there are still more unread letters from the original input string, the machine crashes.

• There may be more than one way to process a string on the machine, and so the machine may be nondeterministic.

• If there is at least one way of processing the string on the machine such that it ends in a final state with no unread letters left and without crashing, then the string is accepted; otherwise, the string is rejected.

• There can be more than one start state.

Example: Consider the following machine that processes strings over the alphabet $\Sigma = \{a, b\}$:

![Diagram](image-url)

Note that this machine is not a finite automaton:

• The arc from state 1 to state 2 is labeled with the string $aa$, which is not a single letter.

• There are two arcs leaving state 2 labeled with $b$.

• There is no arc leaving state 2 labeled with $a$.

• There is an arc from state 1 to state 3 labeled with $\Lambda$, which is not a letter from $\Sigma$.

• There is no arc leaving state 3 labeled with $b$. 

Example: Only accepts the word *aaba*

```
-  aaba  +
\(\text{a, b}\)  \(\text{a, b}\)
\(\text{a, b}\)
```

Example: Accepts all words that contain a doubled letter.

```
-  aa, bb  +
\(\text{a, b}\)
\(\text{a, b}\)
```

- Note that we must decide how many letters to read from the input string each time we go back for more.
- Depending on how we process the string *abb*, the machine may or may not accept it.
- Thus, we say that a string is *accepted* by a machine if there is some way (called a successful path) to process all of the letters in the string and end in a final state without having crashed.
- If there is no way to do this, then the string is not accepted.
- For example, consider the string *baba*, which is not accepted.

### 6.2 Definition of Transition Graph

**Definition:** A *transition graph* (TG) is a collection

\[ M = (K, \Sigma, \Pi, S, F) \]

where:
1. $K$ is a finite set of states.

   - $S \subset K$ is a set of start states with $S \neq \emptyset$ (but possibly with more than one state), where each start state is designated pictorially by $\ominus$.
   - $F \subset K$ is a set of final states (possibly empty, possibly all of $K$), where each final state is designated pictorially by $\oplus$.

2. An alphabet $\Sigma$ of possible input letters from which input strings are formed.

3. $\Pi \subset K \times \Sigma^* \times K$ is a finite set of transitions, where each transition (arc) from one state to another state is labeled with a string $s \in \Sigma^*$.

   - If an arc is labeled with $\Lambda$, we traverse the arc without reading any letters from the input string.
   - If an arc is labeled with an non-empty string $s \in \Sigma^*$, we can traverse the arc if and only if the next unread letter(s) from the original input string are the string $s$.
   - We allow for the possibility that for any state $x \in K$ and any string $s \in \Sigma^*$, there is more than one arc leaving $x$ labeled with string $s$.
   - Also, we allow for the possibility that for any state $x \in K$ and any letter $\ell \in \Sigma$, there is no arc leaving state $x$ labeled with $\ell$.

Remarks:

   - when an edge is labeled with $\Lambda$, we can take that edge without consuming any letters from the input string.
   - We can have more than one start state.
   - Note that every FA is also a TG.
   - However, not every TG is an FA.
6.3 Examples of Transition Graphs

Example: this TG accepts nothing, not even Λ.

\[ - \]

Example: this TG accepts only the string Λ.

\[ \pm \]

\[ \text{a, b} \]

\[ \rightarrow \]

\[ \pm \] or \[ \pm \]
Example: This TG accepts only the words Λ, aaa and bbbb.
Example: this TG accepts only words that end in \textit{aba}; i.e., the language generated by the regular expression

$$(a + b)^*aba$$

Example: this TG accepts the language of all words that begin and end with the same letter and have at least two letters.
Example: this TG accepts the language of all words in which the $a$'s occur in clumps of three and that end in four or more $b$'s.

Example: this is the TG for EVEN-EVEN

Example: Is the word $baaabab$ accepted by this machine?
Chapter 7

Kleene’s Theorem

7.1 Kleene’s Theorem

The following theorem is the most important and fundamental result in the theory of FA’s:

**Theorem 6** Any language that can be defined by either

- regular expression, or
- finite automata, or
- transition graph

*can be defined by all three methods.*

**Proof.** The proof has three parts:

**Part 1:** (FA ⇒ TG) Every language that can be defined by an FA can also be defined by a transition graph.

**Part 2:** (TG ⇒ RegExp) Every language that can be defined by a transition graph can also be defined by a regular expression.

**Part 3:** (RegExp ⇒ FA) Every language that can be defined by a regular expression can also be defined by an FA.
7.2 Proof of Part 1: FA ⇒ TG

- We previously saw that every FA is also a transition graph.
- Hence, any language that has been defined by a FA can also be defined by a transition graph.

7.3 Proof of Part 2: TG ⇒ RegExp

- We will give a constructive algorithm for proving part 2.
- Thus, we will describe an algorithm to take any transition graph $T$ and form a regular expression corresponding to it.
- The algorithm will work for any transition graph $T$.
- The algorithm will finish in finite time.

An overview of the algorithm is as follows:

- Start with any transition graph $T$.
- First, transform it into an equivalent transition graph having only one start state and one final state.
- In each following step, eliminate either some states or some arcs by transforming the TG into another equivalent one.
- We do this by replacing the strings labelling arcs with regular expressions.
- We can traverse an arc labelled with a regular expression using any string that can be generated by the regular expression.
- End up with a TG having only two states, start and final, and one arc going from start to final.
- The final TG will have a regular expression on its one arc
- Note that in each step we eliminate some states or arcs.
- Since the original TG has a finite number of states and arcs, the algorithm will terminate in a finite number of iterations.
Algorithm:

1. If $T$ has more than one start state, add a new state and add arcs labeled $\Lambda$ going to each of the original start states.

2. If $T$ has more than one final state, add a new state and add arcs labeled $\Lambda$ going from each of the original final states to the new state. Need to make sure the final state is different than the start state.
3. Now we give an iterative procedure for eliminating states and arcs

(a) If $T$ has some state with $n > 1$ loops circling back to itself, where the loops are labeled with regular expressions $r_1, r_2, \ldots, r_n$, then replace the $n$ loops with a single loop labeled with the regular expression $r_1 + r_2 + \cdots + r_n$.

(b) If two states are connected by $n > 1$ direct arcs in the same direction, where the arcs are labeled with the regular expressions $r_1, r_2, \ldots, r_n$, then replace the $n$ arcs with a single arc labeled with the regular expression $r_1 + r_2 + \cdots + r_n$. 
(c) *Bypass operation:*

i. If there are three states \(x, y, z\) such that
   - there is an arc from \(x\) to \(y\) labelled with the regular expression \(r_1\) and
   - an arc from \(y\) to \(z\) labelled with the regular expression \(r_2\),
then replace the two arcs and the state \(y\) with a single arc from \(x\) to \(z\) labelled with the regular expression \(r_1r_2\).

\[
\begin{array}{c}
\text{x} \xrightarrow[r_1]{ } \text{y} \xrightarrow[r_2]{ } \text{z} \\
\text{x} \xrightarrow[r_1r_2]{ } \text{z}
\end{array}
\]

\[
\begin{array}{c}
\text{x} \xrightarrow[r_1]{ } \text{y} \xrightarrow[r_2]{ } \text{z} \\
\text{x} \xrightarrow[r_1r_3r_2]{ } \text{z}
\end{array}
\]
ii. If there are
  - $n+2$ states $x, y, z_1, z_2, \ldots, z_n$ such that there is an arc from $x$ to $y$ labelled with the regular expression $r_0$, and
  - an arc from $y$ to $z_i$, $i = 1, 2, \ldots, n$, labelled with the regular expression $r_i$, and
  - an arc from $y$ back to itself labelled with regular expression $r_{n+1}$,
then replace the $n+1$ original arcs and the state $y$ with $n$ arcs from $x$ to $z_i$, $i = 1, 2, \ldots, n$, each labelled with the regular expression $r_0r_{n+1}r_i$.

iii. If any other arcs led directly to $y$, divert them directly to the $z_i$'s.
iv. Need to make sure that all paths possible in the original TG are still possible after the bypass operation.

• Example

\[
\begin{align*}
&\text{w} \quad \text{r}_1 \quad \text{x} \quad \text{r}_4 \\
&\text{w} \quad \text{r}_2 \quad \text{y} \quad \text{r}_3 \quad \text{z} \\
&\text{w} \quad \text{r}_4 \quad \text{x} \quad \text{r}_1 \quad \text{y} \quad \text{r}_5 \quad \text{z} \\
\end{align*}
\]

\[
\begin{align*}
&\text{r}_1 \text{r}_2 (\text{r}_3 \text{r}_2)^* \text{r}_4 \\
&\Rightarrow \quad \text{w} \quad \text{r}_2 (\text{r}_3 \text{r}_2)^* \text{r}_5 \\
&\Rightarrow \quad \text{w} \quad \text{r}_2 (\text{r}_3 \text{r}_2)^* \text{r}_5 \\
\end{align*}
\]

\[
\begin{align*}
&\text{r}_1 (\text{r}_2 \text{r}_3)^* \text{r}_4 + \text{r}_1 \text{r}_2 (\text{r}_3 \text{r}_2)^* \text{r}_5 \\
\end{align*}
\]
• Example:

Suppose we want to get rid of state $y$.

Need to account for all paths that go through state $y$.

There are arcs coming from $x$, $w$, and $z$ going into $y$.

There are arcs from $y$ to $x$ and $z$.

Thus, we need to account for each possible path from a state having an arc into $y$ (i.e., $x$, $w$, $z$) to each state having an arc from $y$ (i.e., $x$, $z$).

Thus, we need to account for the paths from

* $x$ to $y$ to $x$, which has regular expression $r_1r_2^*r_5$
* $x$ to $y$ to $z$, which has regular expression $r_1r_2^*r_3$
* $w$ to $y$ to $x$, which has regular expression $r_7r_2^*r_5$
* $w$ to $y$ to $z$, which has regular expression $r_7r_2^*r_3$
* $z$ to $y$ to $x$, which has regular expression $r_6r_2^*r_5$
* $z$ to $y$ to $z$, which has regular expression $r_6r_2^*r_3$

Thus, after eliminating state $y$, we get the following:

v. Never delete the unique start or final state.
Example:

```
1 - 2 3
4 5 +
1 -
5 +
1 -
5 +
a
=>
abba
abb
bb
a+b
=> a*(abba+abb+bb)(a+b)*
/ \bb abb
a, b
ba ab a
```
Example:

\[
1 - b \\
\downarrow a \\
2 - b \\
\downarrow a \\
3 + \\
\downarrow b \\
4 \\
\downarrow a, b \\
5 + \\
\downarrow a, b \\
\]

\[
\xrightarrow{\Rightarrow} \\
\]

\[
- \\
\downarrow a \\
2 \\
\downarrow bb* \\
4 \\
\downarrow a+b \\
5 + \\
\downarrow b \\
\]

\[
\xrightarrow{\Rightarrow} \\
\]

\[
- \\
\downarrow a \\
2 \\
\downarrow bb* \cdot a \\
4 \\
\downarrow a+b \\
5 + \\
\downarrow b \\
\]

\[
\xrightarrow{\Rightarrow} \\
\]

\[
- \\
\downarrow a \\
2 \\
\downarrow bb* \cdot a \cdot a+b \\
4 \\
\downarrow a \cdot a+b \\
5 + \\
\downarrow b \\
\]

\[
\xrightarrow{\Rightarrow} \\
\]
\[ a(ba)^*a(a+b)^* + ab(ab)^*bb^*(\lambda+a(a+b)^*) \]

\[ (\lambda+b)((ab)^*bb^*(\lambda+a(a+b)^*) + a(ba)^*a(a+b)^*) \]

\[ a(ba)^*a(a+b)^* + ab(ab)^*bb^*(\lambda+a(a+b)^*) \]
7.4 Proof of Part 3: RegExp $\Rightarrow$ FA

To show: every language that can be defined by a regular expression can also be defined by a FA.

We will do this by using a recursive definition and a constructive algorithm.

Recall

- every regular expression can be built up from the letters of the alphabet and $\Lambda$ and $\emptyset$.
- Also, given some existing regular expressions, we can build new regular expressions by applying the following operations:
  1. union (+)
  2. concatenation
  3. closure (Kleene star)
- We will not include $r^+$ in our discussion here, but this will not be a problem since $r^+ = rr^*$. 
Recall that we had the following recursive definition for regular expressions:

**Rule 1:** If \( x \in \Sigma \), then \( x \) is a regular expression. \( \Lambda \) is a regular expression. \( \emptyset \) is a regular expression.

**Rule 2:** If \( r_1 \) and \( r_2 \) are regular expressions, then \( r_1 + r_2 \) is a regular expression.

**Rule 3:** If \( r_1 \) and \( r_2 \) are regular expressions, then \( r_1 r_2 \) is a regular expression.

**Rule 4:** If \( r_1 \) is a regular expression, then \( r_1^* \) is a regular expression.

Based on the above recursive definition for regular expressions, we have the following recursive definition for FA’s associated with regular expressions:

**Rule 1:**
- There is an FA that accepts the language \( L \) defined by the regular expression \( x \); i.e., \( L = \{x\} \), where \( x \in \Sigma \), so language \( L \) consists of only a single word and that word is the single letter \( x \).
- There is an FA that accepts the language defined by regular expression \( \Lambda \); i.e., the language \( \{\Lambda\} \).
- There is an FA defined by the regular expression \( \emptyset \); i.e., the language with no words, which is \( \emptyset \).

**Rule 2:** If there is an FA called \( FA_1 \) that accepts the language defined by the regular expression \( r_1 \) and there is an FA called \( FA_2 \) that accepts the language defined by the regular expression \( r_2 \), then there is an FA called \( FA_3 \) that accepts the language defined by the regular expression \( r_1 + r_2 \).

**Rule 3:** If there is an FA called \( FA_1 \) that accepts the language defined by the regular expression \( r_1 \) and there is an FA called \( FA_2 \) that accepts the language defined by the regular expression \( r_2 \), then there is an FA called \( FA_3 \) that accepts the language defined by the regular expression \( r_1 r_2 \), which is the concatenation.

**Rule 4:** If there is an FA called \( FA_1 \) that accepts the language defined by the regular expression \( r_1 \), then there is an FA called \( FA_2 \) that accepts the language defined by the regular expression \( r_1^* \).
Let’s now show that each of the rules hold by construction:

**Rule 1:** There is an FA that accepts the language $L$ defined by the regular expression $x$; i.e., $L = \{x\}$, where $x \in \Sigma$. There is an FA that accepts language defined by the regular expression $\Lambda$. There is an FA that accepts the language defined by the regular expression $\emptyset$.

- If $x \in \Sigma$, then the following FA accepts the language $\{x\}$:

- An FA that accepts the language $\{\Lambda\}$ is

- An FA that accepts the language $\emptyset$ is
**Rule 2:** If there is an FA called $FA_1$ that accepts the language defined by the regular expression $r_1$ and there is an FA called $FA_2$ that accepts the language defined by the regular expression $r_2$, then there is an FA called $FA_3$ that accepts the language defined by the regular expression $r_1 + r_2$.

- Suppose regular expressions $r_1$ and $r_2$ are defined with respect to a common alphabet $\Sigma$.
- Let $L_1$ be the language generated by regular expression $r_1$.
- $L_1$ has finite automaton $FA_1$.
- Let $L_2$ be the language generated by regular expression $r_2$.
- $L_2$ has finite automaton $FA_2$.
- Regular expression $r_1 + r_2$ generates the language $L_1 + L_2$.
- Recall $L_1 + L_2 = \{ w \in \Sigma^* : w \in L_1$ or $w \in L_2 \}$.
- Thus, $w \in L_1 + L_2$ if and only if $w$ is accepted by either $FA_1$ or $FA_2$ (or both).
- We need $FA_3$ to accept a string if the string is accepted by $FA_1$ or $FA_2$ or both.
- We do this by constructing a new machine $FA_3$ that simultaneously keeps track of where the input would be if it were running on $FA_1$ and where the input would be if it were running on $FA_2$.
- Suppose $FA_1$ has states $x_1, x_2, \ldots, x_m$, and $FA_2$ has states $y_1, y_2, \ldots, y_n$.
- Assume that $x_1$ is the start state of $FA_1$ and that $y_1$ is the start state of $FA_2$.
- We will create $FA_3$ with states of the form $(x_i, y_j)$.
- The number of states in $FA_3$ is at most $mn$, where $m$ is the number of states in $FA_1$ and $n$ is the number of states in $FA_2$.
- Each state in $FA_3$ corresponds to a state in $FA_1$ and a state in $FA_2$.
- $FA_3$ accepts string $w$ if and only if either $FA_1$ or $FA_2$ accepts $w$.
- So final states of $FA_3$ are those states $(x, y)$ such that $x$ is a final state of $FA_1$ or $y$ is a final state of $FA_2$. 
We use the following algorithm to construct $F A_3$ from $F A_1$ and $F A_2$.

- Suppose that $\Sigma$ is the alphabet for both $F A_1$ and $F A_2$.
- Given $F A_1 = (K_1, \Sigma, \pi_1, s_1, F_1)$ with
  - Set of states $K_1 = \{x_1, x_2, \ldots, x_m\}$
  - $s_1 = x_1$ is the initial state
  - $F_1 \subseteq K_1$ is the set of final states of $F A_1$.
  - $\pi_1 : K_1 \times \Sigma \rightarrow K_1$ is the transition function for $F A_1$.
- Given $F A_2 = (K_2, \Sigma, \pi_2, s_2, F_2)$ with
  - Set of states $K_2 = \{y_1, y_2, \ldots, y_n\}$
  - $s_2 = y_1$ is the initial state
  - $F_2 \subseteq K_2$ is the set of final states of $F A_2$.
  - $\pi_2 : K_2 \times \Sigma \rightarrow K_2$ is the transition function for $F A_2$.
- We then define $F A_3 = (K_3, \Sigma, \pi_3, s_3, F_3)$ with
  - Set of states $K_3 = K_1 \times K_2 = \{(x, y) : x \in K_1, y \in K_2\}$
  - The alphabet of $F A_3$ is $\Sigma$.
  - $F A_3$ has transition function $\pi_3 : K_3 \times \Sigma \rightarrow K_3$ with
    \[ \pi_3((x, y), \ell) = (\pi_1(x, \ell), \pi_2(y, \ell)). \]
  - The initial state $s_3 = (s_1, s_2)$.
  - The set of final states
    \[ F_3 = \{(x, y) \in K_1 \times K_2 : x \in F_1 \text{ or } y \in F_2\}. \]
- Since $K_3 = K_1 \times K_2$, the number of states in the new machine $F A_3$ is $|K_3| = |K_1| \cdot |K_2|$.
  - But we can leave out a state $(x, y) \in K_1 \times K_2$ from $K_3$ if $(x, y)$ is not reachable from $F A_3$’s initial state $(s_1, s_2)$.
  - This would result in fewer states in $K_3$, but still we have $|K_1| \cdot |K_2|$ as an upper bound for $|K_3|$; i.e., $|K_3| \leq |K_1| \cdot |K_2|$.
Example: \( L_1 = \{ \text{words with } b \text{ as second letter} \} \)
with regular expression \( r_1 = (a + b)b(a + b)^* \)
\( L_2 = \{ \text{words with odd number of } a \text{'s} \} \)
with regular expression \( r_2 = b^a(b + ab^*a)^* \)

**FA1 for L1:**

```
\*a,b
/ x1- a,b x2 b
|    x3+
|    a
|x4  a,b
```

**FA2 for L2:**

```
\*b
/ y1- a
| b
| y2+
```

**FA3 for L1+L2:**

```
\*a
/ x1,y1- a b
/ x2,y1 b
/ x2,y2+ a
/ x3,y1+ b
/ x3,y2+ a
/ x4,y1 b
/ x4,y2+
```

**Rule 3:** If there is an FA called $FA_1$ that accepts the language defined by the regular expression $r_1$ and there is an FA called $FA_2$ that accepts the language defined by the regular expression $r_2$, then there is an FA called $FA_3$ that accepts the language defined by the regular expression $r_1r_2$.

For this part,

- we need $FA_3$ to accept a string if the string can be factored into two substrings, where the first factor is accepted by $FA_1$ and the second factor is accepted by $FA_2$.
- One problem is we don’t know when we reach the end of the first factor and the beginning of the second factor.

**Example:** $L_1 = \{\text{words that end with } aa\}$ with regular expression $r_1 = (a + b)^*aa$
$L_2 = \{\text{words with odd length}\}$ with regular expression $r_2 = (a + b)((a + b)(a + b))^*$

- Consider the string $baaab$.
- If we factor it as $(baa)(ab)$, then $baa \in L_1$ but $ab \notin L_2$.
- However, another factorization, $(baaa)(b)$, shows that $baaab \in L_1L_2$ since $baaa \in L_1$ and $b \in L_2$.

**FA1 for L1:**

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1-</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3+</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>
```

**FA2 for L2:**

```
<table>
<thead>
<tr>
<th></th>
<th>a, b</th>
</tr>
</thead>
<tbody>
<tr>
<td>y1-</td>
<td></td>
</tr>
<tr>
<td>y2+</td>
<td>a, b</td>
</tr>
</tbody>
</table>
```
• Basically idea of building $FA_3$ for $L_1L_2$ from $FA_1$ for $L_1$ and $FA_2$ for $L_2$:
  ■ Recall $L_1L_2 = \{w = w_1w_2 : w_1 \in L_1, w_2 \in L_2\}$.
  ■ So a string $w$ is in $L_1L_2$ if and only if we can factor $w = w_1w_2$ such that $w_1$ is accepted by $FA_1$ and $w_2$ is accepted by $FA_2$.
  ■ $FA_3$ initially acts like $FA_1$.
  ■ When $FA_3$ hits a $\oplus$ state of $FA_1$,
    * Start a version of $FA_2$.
    * Keep processing on $FA_1$ and any previous versions of $FA_2$.
  ■ We need to keep processing on $FA_1$ because we don’t know where the first factor $w_1$ ends and the second factor $w_2$ begins.
  ■ Final states of $FA_3$ are those states that have at least one final state from $FA_2$.

• More formally, we build machine $FA_3$ in following way:
  ■ Suppose that $FA_1$ and $FA_2$ have the same alphabet $\Sigma$.
  ■ Let $L_1$ be language generated by regular expression $r_1$ and having $FA_1 = (K_1, \Sigma, \pi_1, s_1, F_1)$.
  ■ Let $L_2$ be language generated by regular expression $r_2$ and having $FA_2 = (K_2, \Sigma, \pi_2, s_2, F_2)$.

  ■ Definition: For any set $S$, define $2^S$ to be the set of all possible subsets of $S$.

  Example: If $S = \{a, bb, ab\}$, then
  
  $2^S = \{\emptyset, \{a\}, \{bb\}, \{ab\}, \{a, bb\}, \{a, ab\}, \{bb, ab\}, \{a, bb, ab\}\}$.

  Fact: If $|S| < \infty$, then $|2^S| = 2^{|S|}$; i.e., there are $2^{|S|}$ different subsets of $S$.

  ■ Machine $FA_3 = (K_3, \Sigma, \pi_3, s_3, F_3)$ for $L_1L_2$ is as follows:
    * States
      
      $K_3 = \{\{x\} + Y : x \in K_1, Y \in 2^{K_2}\}$;

      i.e., each state of $FA_3$ is a set of states, where exactly one of the states is from $FA_1$ and the rest (possibly none) are from $FA_2$.
    * Initial state $s_3 = \{s_1\}$; i.e., the initial state of $FA_3$ is the set consisting of only the initial state of $FA_1$. 
* Transition function $\pi_3 : K_3 \times \Sigma \rightarrow K_3$ is defined as

$$
\pi_3(\{x, y_1, \ldots, y_n\}, \ell) = \begin{cases} 
\{\pi_1(x, \ell), \pi_2(y_1, \ell), \ldots, \pi_n(y_2, \ell)\} & \text{if } \pi_1(x, \ell) \notin F_1, \\
\{\pi_1(x, \ell), \pi_2(y_1, \ell), \ldots, \pi_n(y_2, \ell), s_2\} & \text{if } \pi_1(x, \ell) \in F_1,
\end{cases}
$$

where $\{x, y_1, \ldots, y_n\} \in K_3$, $n \geq 0$, $x \in K_1$, $y_i \in K_2$ for $i = 1, \ldots, n$, and $\ell \in \Sigma$.

* Final states

$$
F_3 = \{\{x, y_1, \ldots, y_n\} : n \geq 1, y_i \in F_2 \text{ for some } i = 1, \ldots, n\}.
$$

* The number of states in $FA_3$ is

$$
|K_3| = |K_1| \cdot 2^{|K_2|} = |K_1| \cdot 2^{|K_2|}.
$$

* Actually, we can leave out from $K_3$ any states $\{x, y_1, \ldots, y_n\}$ that are not reachable from the initial state $s_3$.

* In this case, $|K_1| \cdot 2^{|K_2|}$ still provides an upper bound for $|K_3|$; i.e., $|K_3| \leq |K_1| \cdot 2^{|K_2|}$.  

**Example:** \( L_1 = \{ \text{words that end with } aa \} \)
with regular expression \( r_1 = (a + b)^*aa \)
\( L_2 = \{ \text{words with odd length} \} \)
with regular expression \( r_2 = (a + b)((a + b)(a + b))^* \)
CHAPTER 7. KLEENE’S THEOREM

Rule 4: If there is an FA called $FA_1$ that accepts the language defined by the regular expression $r_1$, then there is an FA called $FA_2$ that accepts the language defined by the regular expression $r_1^*$.

Basic idea of how to build machine $FA_2$:

- Each state of $FA_2$ corresponds to one or more states of $FA_1$.
- $FA_2$ initially acts like $FA_1$.
- when $FA_2$ hits a $\bigoplus$ state of $FA_1$, then $FA_2$ simultaneously keeps track of how the rest of the string would be processed on $FA_1$ from where it left off and how the rest of the string would be processed on $FA_1$ starting in the start state.
- Whenever $FA_2$ hits a $\bigoplus$ state of $FA_1$, we have to start a new process starting in the start state of $FA_1$ (if no version of $FA_1$ is currently in its start state.)
- The final states of $FA_2$ are those states which have a correspondence to some final state of $FA_1$.
- We need to be careful about making sure that $FA_2$ accepts $\Lambda$.
- To have $FA_2$ accept $\Lambda$, we make the start state of $FA_2$ also a final state.
- But we need to be careful when there are arcs going into the start state of $FA_1$. 
Formally, we build the machine $FA_2$ for $L_1^*$ as follows:

- Let $L_1$ be language generated by regular expression $r_1$ and having finite automaton $FA_1 = (K_1, \Sigma, \pi_1, s_1, F_1)$.
- For now, assume that $FA_1$ does not have any arcs entering the initial state $s_1$.
- Know that language $L_1^*$ is generated by regular expression $r_1^*$.
- Define $FA_2 = (K_2, \Sigma, \pi_2, s_2, F_2)$ for $L_1^*$ with
  
  ■ States $K_2 = 2^{K_1}$.
  ■ Initial state $s_2 = \{s_1\}$.
  ■ Transition function $\pi_2 : K_2 \times \Sigma \to K_2$ with
    
    $$\pi_2(\{x_1, \ldots, x_n\}, \ell) = \begin{cases} 
    \{\pi_1(x_1, \ell), \ldots, \pi_1(x_n, \ell)\} & \text{if } \pi_1(x_k, \ell) \notin F_1 \text{ for all } k = 1, \ldots, n, \\
    \{\pi_1(x_1, \ell), \ldots, \pi_1(x_n, \ell), s_1\} & \text{if } \pi_1(x_k, \ell) \in F_1 \text{ for some } k = 1, \ldots, n,
    \end{cases}$$

    where $\{x_1, \ldots, x_n\} \in K_2$, $n \geq 1$, $x_i \in K_1$ for all $i = 1, \ldots, n$, and $\ell \in \Sigma$.
  ■ Final states
    
    $$F_2 = \{s_1\} + \{\{x_1, \ldots, x_n\} : n \geq 1, x_i \in F_1 \text{ for some } i = 1, \ldots, n\}.$$  

- The number of states in $FA_2$ is
  
  $$|K_2| = 2^{K_1} = 2^{|K_1|}.$$  

- Actually, we can leave out from $K_2$ any state $\{x_1, \ldots, x_n\}$ that is not reachable from the initial state $s_2$.
- In this case, $2^{|K_1|}$ still provides an upper bound for $|K_2|$; i.e., $|K_3| \leq 2^{|K_1|}$.  

**Example:** Consider language $L$ having regular expression

$$r = (a + bb^*ab^*a)((b + ab^*a)b^*a)^*$$

**FA for L:**

```
  x1-
    |   |
    v   v
  x3   x4
    |   |
    a   a
      |   |
      b   b
```

**FA for $L^*$:**

```
  x1-
    |   |
    v   v
  x3   x4
    |   |
    a   a
      |   |
      b   b
```

```
  x2+   x1+
    |     |
    v     v
  x3  x4
    |   |
    a   a
      |   |
      b   b
```

```
  x2,x1+  x4,x2,x1+
    |     |
    v     v
  x3,x4  x4,x2,x1+
    |     |
    v     v
  x1,x2,x3,x4+  x1,x2,x3,x4+
    |     |
    v     v
  a  a
```

Example: Consider language $L$ having regular expression

$$(a + b)^*b$$

Need to be careful since we can return to the start state.

**FA for L:**

![FA for L diagram]

If we blindly applied previous method for constructing $FA$ for $L^*$, we get the following:

![FA for L^* diagram]

Problem:
- Note that start state is final state.
- But this $FA$ accepts $a \not\in L^*$, and so this $FA$ is incorrect.
- Problem occurs because we can return to start state in original $FA$, and since we make the start state a final state in new $FA$. 
Solution:

- Given original FA $FA_1$ having arcs going into the initial state, create an equivalent FA $\tilde{FA}_1$ having no arcs going into the initial state by splitting the original start state $x_1$ of $FA_1$ into two states $x_{1.1}$ and $x_{1.2}$.
  - $x_{1.1}$ is the new start state of $\tilde{FA}_1$ and is never visited again after the first letter of the input string is read.
  - $x_{1.2}$ in $\tilde{FA}_1$ corresponds to $x_1$ after the first letter of the input string is read.
- Then run algorithm to create FA for $L^*$ from the new FA $\tilde{FA}_1$.

new FA for L:

FA for $L^*$:
7.5 Nondeterministic Finite Automata

Definition: A nondeterministic finite automaton (NFA) is given by $M = (K, \Sigma, \Pi, s, F)$, where

1. $K$ is a finite set of states.
   - $s \in K$ is the initial state, which is denoted pictorially by $\ominus$, and there is exactly one initial state.
   - $F \subset K$ is a set of final states (possibly empty), where each final state is denoted pictorially by $\bigoplus$.

2. An alphabet $\Sigma$ of possible input letters.

3. $\Pi \subset K \times \Sigma \times K$ is a finite set of transitions, where each transition (arc) from one state to another state is labeled with a letter $\ell \in \Sigma$. (We do not allow for $\Lambda$ to be the label of an arc since $\Lambda$ is a string and not a letter of $\Sigma$.) We allow for the possibility of more than one edge with the same label from any state and there may be a state (or states) for which certain input letters have no edge leaving that state.
Example:

Note that

- definition of NFA is different from that of FA since
  - a FA must have from each state an arc labeled with each letter of alphabet, while NFA does not.
  - a FA is deterministic, while a NFA may be nondeterministic.
  - An NFA can have repeated labels from any single state.

- NFA allows for human choice to become a factor in selecting a way to process an input string.

- The definition of NFA is different from that of TG since
  - a TG can have arcs labeled with substrings of letters while a NFA has arcs labeled with only letters.
  - a TG can have arcs labeled with Λ while a NFA cannot.
  - a TG can have more than one start state while a NFA can only have one.
• Can transform any NFA with repeated labels from any single state to an equivalent TG with no repeated labels from any single state.

7.6 Properties of NFA

Theorem 7 $FA = NFA$; i.e., any language definable by a NFA is also definable by a deterministic FA and vice versa.

Proof. Note that

• Every FA is an NFA since we can consider an FA to be an NFA without the extra possible features.

• Every NFA is a TG.

• Kleene’s theorem states that every TG has an equivalent FA.

NFA useful because

• applications in artificial intelligence (AI).
• given two FA’s for two languages with regular expressions $r_1$ and $r_2$, it is easy to construct an NFA to accept language corresponding to regular expression $r_1 + r_2$. 
Example:

FA1:

FA2:

This works when neither of the original FA’s has any arcs going into their original initial states.

If one or both of the original FA’s has an arc going into its original initial state, the newly constructed FA for the language corresponding
to regular expression $r_1 + r_2$ may be incorrect. This is because the new FA may process part of the word on one of the original FA’s and then process the rest of the word on the other FA, and then incorrectly accept the word.
Chapter 8

Finite Automata with Output

8.1 Moore Machines

**Definition:** A *Moore machine* is a collection of five things:

1. A finite set of states $q_0, q_1, q_2, \ldots, q_n$, where $q_0$ is designated as the start state.

2. A finite alphabet of letters for forming the input string
   \[
   \Sigma = \{a, b, c, \ldots\}
   \]

3. A finite alphabet of possible output characters
   \[
   \Gamma = \{x, y, z, \ldots\}
   \]

4. A transition table that shows for each state and each input letter what state is reached next.

5. An output table that shows what character from $\Gamma$ is printed by each state that is entered.
Example: Input alphabet: $\Sigma = \{a, b\}$
Output alphabet: $\Gamma = \{0, 1\}$
States: $q_0, q_1, q_2, q_3$

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_3$</td>
<td>$q_2$</td>
<td>0</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_0$</td>
<td>0</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_3$</td>
<td>1</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_0$</td>
<td>$q_1$</td>
<td>0</td>
</tr>
</tbody>
</table>

On input string $bababb$, the output is 01100100.

8.2 Mealy Machines

Definition: A Mealy machine is a collection of four things:

1. A finite set of states $q_0, q_1, q_2, \ldots, q_n$, where $q_0$ is designated as the start state.
2. A finite alphabet of letters $\Sigma = \{a, b, \ldots\}$.
3. A finite alphabet of output characters $\Gamma = \{x, y, z, \ldots\}$. 
4. A pictorial representation with states represented by small circles and directed edges indicating transitions between states. Each edge is labeled with a compound symbol of the form $i/o$, where $i$ is an input letter and $o$ is an output character. Every state must have exactly one outgoing edge for each possible input letter. The way we travel is determined by the input letter $i$. While traveling on the edge, we must print the output character $o$.

The key difference between Moore and Mealy machines:

- Moore machines print character when in state.
- Mealy machines print character when traversing an arc.

**Example:** Mealy machine

![Mealy machine diagram]
Example: Mealy machine prints out the 1’s complement of an input bit string.
$\Sigma = \Gamma = \{0, 1\}$. 

Diagram:

```
q_0 0/1, 1/0
```
8.3 Properties of Moore and Mealy Machines

Definition: Given a Mealy machine $Me$ and a Moore machine $Mo$, which automatically prints the character $x$ in the start state, we say these two machines are equivalent if for every input string the output string from $Mo$ is exactly $x$ concatenated with the output from $Me$.

Theorem 8 If $Mo$ is a Moore machine, then there is a Mealy machine $Me$ that is equivalent to it.

Proof.

- Consider any state $q_i$ of $Mo$.
- Suppose $Mo$ prints the character $t$ upon entering $q_i$.
- Hence, the label in state $q_i$ is $q_i/t$.
- Suppose that there are $n$ arcs entering $q_i$, with labels $a_1, a_2, \ldots, a_n$.
- We create the machine $Me$ by changing the labels on the incoming arcs to $q_i$ to $a_m/t$, $m = 1, 2, \ldots, n$.
- Change the label of state $q_i$ to be just $q_i$.
Example: Convert Moore machine into equivalent Mealy machine.

Mo:

\[
\begin{align*}
q_0 / 1 & \quad a \quad b \\
q_1 / 0 & \quad a \\
q_2 / 1 & \quad a \\
q_3 / 1 & \quad b \\
q_4 / 0 & \quad a
\end{align*}
\]

Me:

\[
\begin{align*}
q_0 & \quad a / 0 \\
q_1 & \quad a / 0 \\
q_2 & \quad q / 0 \\
q_3 & \quad b / 0 \\
q_4 & \quad b / 0
\end{align*}
\]
Theorem 9 For every Mealy machine Me, there is an equivalent Moore machine Mo.

Proof.

- Consider any state $q_i$ of $Me$.
- Suppose that there are $n$ arcs entering $q_i$, with labels $a_1/t_1, a_2/t_2, \ldots, a_n/t_n$.
- So if we enter state $q_i$ using the $k$th arc, we just read in $a_k$ and printed $t_k$.
- Suppose that among $\{t_1, t_2, \ldots, t_n\}$, there are $k$ different characters; call them $c_1, c_2, \ldots, c_k$.
- To create the Moore machine $Mo$, split the state $q_i$ into $k$ different states; call them $q_i^1, q_i^2, \ldots, q_i^k$.
- State $q_i^l$ will correspond to the output character $c_l$.
- For each arc going into $q_i$ in $Me$ which was labeled with the output character $c_l$, have that arc in $Mo$ go to the state $q_i^l/c_l$. Label that arc with its input letter.
- For any state in $Me$ which has no incoming edges, we arbitrarily assign it any output character in $Mo$.

\[
\text{Me:} \quad \begin{array}{c}
q_1 \\
\begin{array}{c}
a / 0 \\
\text{a / 1} \\
b / 0 \\
b / 1 \\
b / 0
\end{array}
\end{array} \quad \Rightarrow \quad \begin{array}{c}
q_1/0 \\
\begin{array}{c}
a \\
a / 0 \\
b \\
b / 1 \\
b / 1
\end{array}
\end{array} \quad \text{Mo:}
\]
Example: Convert Mealy machine into equivalent Moore machine.

Me:

Mo:
Chapter 9

Regular Languages

9.1 Properties of Regular Languages

Definition: A language that can be defined by a regular expression is a regular language.

Theorem 10 If \( L_1 \) and \( L_2 \) are regular languages, then \( L_1 + L_2 \), \( L_1L_2 \), and \( L_1^* \) are also regular languages.

Proof. (by regular expressions)

- If \( L_1 \) and \( L_2 \) are regular languages, then there are regular expressions \( r_1 \) and \( r_2 \) that define these languages.
- \( r_1 + r_2 \) is a regular expression that defines the language \( L_1 + L_2 \), and so \( L_1 + L_2 \) is a regular language.
- \( r_1r_2 \) is a regular expression that defines the language \( L_1L_2 \), and so \( L_1L_2 \) is a regular language.
- \( r_1^* \) is a regular expression that defines the language \( L_1^* \), and so \( L_1^* \) is a regular language.
Proof. (by machines)

- If $L_1$ and $L_2$ are regular languages, then there are transition graphs $TG_1$ and $TG_2$ that accept them by Kleene’s Theorem.
- We may assume that $TG_1$ has a unique start state and unique final state, and the same for $TG_2$.
- We construct the TG for $L_1 + L_2$ as follows:

![Diagram of $L_1 + L_2$]

- We construct the TG for $L_1L_2$ as follows:

![Diagram of $L_1L_2$]
• We construct the TG for $L_1^*$ as follows:
Remarks:

- The technique given in the tapes of lectures 11 and 12 is wrong.
- To see why, consider the following FA for the language
  \( L = \{ \text{words having an odd number of } b\text{'s } \} \)

![Diagram of FA for L](image)

- Note that \( L^* \) is the language consisting of \( \Lambda \) and all words having at least one \( b \), which has regular expression \( \Lambda + (a + b)^*b(a + b)^* \) (which was also wrong in the tape of lecture 12).

- If we use the (incorrect) technique to construct a TG for \( L^* \) given in the taped lecture, then we get the following:

![Diagram of TG for L^*](image)

- However, the above TG accepts the string \( a \notin L^* \).

- On the other hand, if we use the method presented above to construct a TG for \( L^* \), then we get the following correct TG:
Example: alphabet $\Sigma = \{a, b\}$
$L_1 = \text{all words ending with } a$
$L_2 = \text{all words containing the substring } aa.$

Regular expressions:
$r_1 = (a + b)^*a$
$r_2 = (a + b)^*aa(a + b)^*$

FA1:

```
 b  a  +  a
+  b  -  b
```

FA2:

```
 b  a  +  a
+  b  a, b
```

$r_1 + r_2 = (a+b)^*a + (a+b)^*aa(a+b)^*$

$\text{TG}_{1+2}$ for $r_1 + r_2$
CHAPTER 9. REGULAR LANGUAGES

TG for $r_1 r_2$

TG for $r_2^*$
CHAPTER 9. REGULAR LANGUAGES

9.2 Complementation of Regular Languages

Definition: If $L$ is a language over the alphabet $\Sigma$, we define $L'$ to be its complement, which is the language of all strings of letters from $\Sigma$ that are not words in $L$, i.e., $L' = \{w \in \Sigma^* : w \notin L\}$.

Example: alphabet $\Sigma = \{a, b\}$
$L = $ language of all words in $\Sigma^*$ containing the substring $abb$.
$L' = $ language of all words in $\Sigma^*$ not containing the substring $abb$.

Note that
\[(L')' = L\]

Theorem 11 If $L$ is a regular language, then $L'$ is also a regular language. In other words, the set of regular languages is closed under complementation.

Proof.

• If $L$ is a regular language, then there exists some FA that accepts $L$ by Kleene’s Theorem.
• Create new finite automaton $FA'$ from $FA$ as follows:
  • $FA'$ has same states and arcs as $FA$.
  • Every final state of $FA$ becomes a nonfinal state in $FA'$
  • Every nonfinal state of $FA$ becomes a final state in $FA'$
  • $FA'$ has same start state as $FA$.
• $FA'$ accepts the language $L'$.
• Kleene’s Theorem implies that $L'$ is a regular language.
Example: $\Sigma = \{a, b\}$

$L = \text{all words with length at least 2 and second letter } b$

$L' = \text{all words with length less than 2 or second letter } a$
9.3  Intersections of Regular Languages

Theorem 12  If \( L_1 \) and \( L_2 \) are regular languages, then \( L_1 \cap L_2 \) is a regular language. In other words, the set of regular languages is closed under intersection.

Proof.

• DeMorgan’s Law for sets states that

\[
L_1 \cap L_2 = (L'_1 + L'_2)'
\]

\[L'_1 + L'_2\] \hspace{1cm} \[(L'_1 + L'_2)'\]

• Since \( L_1 \) and \( L_2 \) are regular languages, Theorem 11 implies that \( L'_1 \) and \( L'_2 \) are regular languages.

• Theorem 10 then implies that \( L'_1 + L'_2 \) is a regular language.

• Theorem 11 then implies that \( (L'_1 + L'_2)' \) is a regular language.
Example: alphabet $\Sigma = \{a, b\}$

$L_1$ = all words with length $\geq 2$ and second letter $b$

$L_2$ = all words containing the substring $ab$. 
\[
r_1 = (a+b)b(a+b)^* \\
r_2 = (a+b)^*ab(a+b)^*
\]

**FA1:**

```
1- \ a, b \ 2 \ b \ 3+ \ a, b \\
\ \ \ \ a \ \ \ \ 4 \ \ a, b
```

**FA2:**

```
1- \ b \ 2 \ b \ 3+ \ a, b \\
\ a \ \ \ \ a 
```

**FA1'**

```
1+ \ a, b \ 2+ \ b \ 3 \ a, b \\
\ \ \ \ a \ \ \ \ 4+ \ a, b
```

**FA2'**

```
1- \ b \ 2+ \ b \ 3 \ a, b \\
\ a \ \ \ a 
```
FA for \((L_1'+L_2')\):

```
\begin{align*}
&\quad \quad x_2, y_1+ \quad b \quad x_1, y_1+ \quad a \quad x_2, y_2+ \\
&\quad \quad x_3, y_1+ \quad a \quad x_3, y_2+ \quad b \quad x_3, y_3 \quad a, b \\
&\quad \quad x_4, y_2+ \quad b \quad x_4, y_3+ \\
\end{align*}
```

FA for \((L_1'+L_2')'\):

```
\begin{align*}
&\quad \quad x_2, y_1 \quad b \quad x_3, y_1 \quad a \quad x_3, y_2 \quad b \quad x_3, y_3+ \quad a, b \\
&\quad \quad x_4, y_2 \quad b \quad x_4, y_3 \\
\end{align*}
```
As an exercise, we will now derive a regular expression for $L_1 \cap L_2$ using our FA for $(L'_1 + L'_2)'$ and our algorithm from Kleene’s theorem:
Proof. (another for Theorem 12)

- In proof of Kleene’s theorem, we showed how to construct $FA_3$ that is the union of $FA_1$ and $FA_2$.
- Suppose states of $FA_1$ are $x_1, x_2, \ldots$.
- Suppose states of $FA_2$ are $y_1, y_2, \ldots$.
- We do the same construction of $FA_3$ except we now make a state in $FA_3$ a final state only if both the corresponding $x$ and $y$ states are final states.
- Then $FA_3$ accepts only words that are accepted by both $FA_1$ and $FA_2$.

Example: alphabet $\Sigma = \{a, b\}$
$L_1 =$ all words with length $\geq 2$ and second letter $b$
$L_2 =$ all words containing the substring $ab$. 

![Diagram of FA3 accepting L1 and L2]
Chapter 10

Nonregular Languages

10.1 Introduction

Example: Consider the following FA having 5 states:

- Let’s process the string $ababbba$ on the FA:
  
  $1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{a} 3 \xrightarrow{b} 5 \xrightarrow{b} 4 \xrightarrow{a} 3 \xrightarrow{a} 2$
  
- Since 2 is a final state, we accept the string $ababbba$.
  
- In general,
  
  - We always start in initial state.
  - After reading first letter of input string,
CHAPTER 10. NONREGULAR LANGUAGES

* we end may go to another state or return to initial state.
* the maximum number of different states that we could have visited after reading the first letter is 2.

- After reading the first 2 letters of input string, the maximum number of different states that we could have visited is 3.
- In general, after reading the first \( m \) letters of input string, the maximum number of different states that we could have visited is \( m + 1 \).

- In our example above, after reading 5 letters, the maximum number of different states that we could have visited is \( 5 + 1 = 6 \). But since the FA has 5 states, we know that after reading in 5 letters, we must have visited some state twice.

- Consider the string \( aaabaa \).
  - The string has length 6, which is more than the number of states in the above FA.
  - We process the string as follows:
    
    \[
    1 \xrightarrow{a} 2 \xrightarrow{a} 1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{a} 3 \xrightarrow{a} 2
    \]
    
    and so it is accepted.
  - Notice that state 1 is the first state that we visit twice.

- In general, if we have an FA with \( N \) states and we process a string \( w \) with \( \text{length}(w) \geq N \), then there exists at least one state that we visit at least twice.
  - Let \( u \) be the first state that we visit twice.
  - Break up string \( w \) as \( w = xyz \), where \( x, y, \) and \( z \) are 3 strings such that
    * string \( x \) is the letters at the beginning of \( w \) that are read by the FA until the state \( u \) is hit for the first time.
    * string \( y \) is the letters used by the FA starting from the first time we are in state \( u \) until we hit state \( u \) the second time.
    * string \( z \) is the rest of the letters in \( w \).
  - For example, for the string \( w = ababbaa \) processed on the above FA, we have \( u = 2 \), and \( x = ab, y = abb, z = aa \).
For example, for the string \( w = aaabaa \) processed on the above FA, we have \( u = 1, \) and \( x = \Lambda, y = aa, z = abaa. \)

### 10.2 Definition of Nonregular Languages

**Definition:** A language that cannot be defined by a regular expression is called a *nonregular language*.

By Kleene’s Theorem, a nonregular language cannot be accepted by any FA or TG.

- Consider

\[
L = \{ \Lambda, ab, aabb, aaabbb, aaaaabbbb, \ldots \} = \{ a^n b^n : n = 0, 1, 2, \ldots \} \equiv \{ a^n b^n \}
\]

- We will show that \( L \) is a nonregular language by contradiction.
- Suppose that there is some FA that accepts \( L \).
- By definition, this FA must have a finite number of states, say 5.
- Consider the path the FA takes on the word \( a^6 b^6 \).
- The first 6 letters of the word are \( a \)’s.
- When processing the first 6 letters, the FA must visit some state \( u \) at least twice since there are only 5 states in the FA.
- We say that the path has a *circuit*, which consists of those edges that are taken from the first time \( u \) is visited to the second time \( u \) is visited.
- Suppose the circuit consists of 3 edges.
- After the first \( b \) is read, the path goes elsewhere and eventually we end up in a final state where the word \( a^6 b^6 \) is accepted.
- Now consider the string \( a^{6+3} b^6 \).
- When processing the \( a \) part of the string, the FA eventually hits state \( u \).
- From state \( u \), we can take the circuit and return to \( u \) by using up 3 \( a \)’s.
• From then on, we read in the rest of the a’s exactly as before and go on to read in the 6 b’s in the same way as before.

• Thus, when processing $a^{6+3}b^6$, we end up again in a final state.

• Hence, we are supposed to accept $a^9b^6$.

• However, $a^9b^6$ is not in $L$ since it does not have an equal number of a’s and b’s.

• Thus, we have a contradiction, and so $L$ must not be regular.

• We can use the same argument with any string $a^6(a^3)^kb^6$, for $k = 0, 1, 2, \ldots$.

10.3 First Version of Pumping Lemma

Theorem 13 (Pumping Lemma) Let $L$ be any regular language that has infinitely many words. Then there exists some three strings $x$, $y$, and $z$ such that $y$ is not the null string and that all strings of the form

$$xy^kz$$

for $k = 0, 1, 2, \ldots$

are words in $L$.

Proof.

• Since $L$ is a regular language, there exists some FA that accepts $L$ by Kleene’s theorem.

• FA must have a finite number of states $N$.

• Since $L$ is an infinite language and since alphabets are always finite, $L$ must consist of arbitrarily long words.

• Consider any word $w$ accepted by FA with $\text{length}(w) = m$, and assume that $m \geq N$.

• Since $\text{length}(w) = m$, when processing $w$ on the FA, we visit $m + 1$ states, not necessarily all unique.
• Since \( m + 1 \geq N + 1 \), when processing \( w \) on the FA, we visit at least \( N + 1 \) states.

• But since the FA has only \( N \) states in total, some state must be visited twice when processing \( w \) on the FA.

• Let \( u \) be the first state visited twice when processing the string \( w \) on the FA.

• Thus, there is a circuit in FA corresponding to state \( u \) for this string \( w \).

• We break up \( w \) into three substrings \( x, y, z \):
  
  1. \( x \) consists of the all letters starting at the beginning of \( w \) up to those consumed by the FA when state \( u \) is reached for the first time. Note that \( x \) may be null.
  
  2. \( y \) consists of the letters after \( x \) that are consumed by the FA as it travels around the circuit.
  
  3. \( z \) consists of the letters after \( y \) to the end of \( w \).

• Note that the following statements hold:
  
  1. \( w = xyz \).

  2. Note that \( y \) is not null since at least one letter is consumed by traveling around the circuit. The circuit starts in a particular state, and ends in the same state. Thus, traveling the circuit requires at least one transition, which means that at least one letter is consumed.

  3. The strings \( x \) and \( y \) satisfy \( \text{length}(x) + \text{length}(y) \leq N \), which we can show as follows. Let \( v \) be the string \( xy \) except for the last letter of \( xy \). By the way that we constructed \( x \) and \( y \), when we process \( v \) on the FA starting in the initial state, we never visit any state twice since it is only on reading the last letter of \( y \) do we first visit some state twice. Thus, processing \( v \) on the FA results in visiting at most \( N \) states, which corresponds to reading at most \( N - 1 \) letters. Since \( xy \) is the same as \( v \) with one more letter attached, we must have that \( xy \) has length at most \( N \).

• When processing \( w = xyz \),

  • the FA first processes substring \( x \) and ends in state \( u \).
• then it starts processing substring $y$ starting in state $u$ and ends in state $u$.
• then it starts processing substring $z$ starting in state $u$ and ends in some final state $v$.

- Now process the word $xyyz$ on FA.
  - For the substring $x$, the FA follows exactly the same path as when it processed the $x$-part of $w$.
  - For the first substring $u$, the FA starts in state $u$ and returns to state $u$.
  - For the second substring $u$, the FA starts in state $u$ and returns to state $u$.
  - For the substring $z$, the FA starts in $u$ and processes exactly as before for the word $w$, and so it ends in the final state $v$.
  - Thus, $xyyz$ is accepted by FA.

- Similarly, we can show that any string $xy^kz$, $k = 0, 1, 2, \ldots$, is accepted by FA.

\[ \square \]

### 10.4 Another Version of Pumping Lemma

**Theorem 14** Let $L$ be a language accepted by an FA with $N$ states. Then for all words $w \in L$ such that $\text{length}(w) \geq N$, there are strings $x$, $y$, and $z$ such that

P1. $w = xyz$;

P2. $y$ is not null;

P3. $\text{length}(x) + \text{length}(y) \leq N$;

P4. $xy^kz \in L$ for all $k = 0, 1, 2, \ldots$. 

**Proof.** The proof of Theorem 13 actually establishes Theorem 14.

**Remarks:**

- In the textbook Theorem 14 also assumes that $L$ is infinite. However, this additional assumption is not needed.

**Example:**

\[w = ababbaa\]
\[x = ab\]
\[y = abb\]
\[z = aa\]
Example: Prove $L = \text{PALINDROME}$ is nonregular.

We cannot use the first version of the Pumping Lemma (Theorem 13) since

$$x = a, \quad y = b, \quad z = a,$$

satisfy the lemma and do not contradict the language since all words of the form

$$xy^kz = ab^k a$$

are in PALINDROME.

We will instead apply Theorem 14 to show that PALINDROME is nonregular.

Proof.

- Suppose that PALINDROME is a regular language.
- Then by definition, PALINDROME must have a regular expression.
- Kleene’s Theorem then implies that there is a finite automaton for PALINDROME.
- Assume that the FA for PALINDROME has $N$ states, for some $N \geq 1$.
- Consider the string $w = a^N b a^N$ which is in PALINDROME.
- Note that $\text{length}(w) = 2N + 1 \geq N$.
- Thus, all of the assumptions of Theorem 14 hold, so the conclusions of Theorem 14 must hold; i.e., there exist strings $x, y,$ and $z$ such that

  \begin{itemize}
  \item \textbf{P1.} $w = xyz$;
  \item \textbf{P2.} $y$ is not null;
  \item \textbf{P3.} $\text{length}(x) + \text{length}(y) \leq N$;
  \item \textbf{P4.} $xy^kz \in L$ for all $k = 0, 1, 2, \ldots$.
  \end{itemize}

- \textbf{P1} of Theorem 14 says that $w = xyz$, so
  \begin{itemize}
  \item $x$ must be at the beginning of $w$,
  \item $y$ must be somewhere in the middle of $w$,
  \end{itemize}
• $z$ must be at the end of $w$.

• P2 of Theorem 14 says that $x$ and $y$ together have at most $N$ letters.

• Since $w$ has $N$ $a$’s in the beginning and $x$ and $y$ are at the beginning of $w$, $x$ and $y$ must consist solely of $a$’s.

• P1 and P3 of Theorem 14 imply that $x$ and $y$ must consist solely of $a$’s.

• Since $z$ is the rest of the string after $x$ and $y$, we must have that $z$ consists of zero or more $a$’s, followed by 1 $b$ and then $N$ $a$’s.

• In other words,

\[
\begin{align*}
x &= a^i \text{ for some } i \geq 0, \\
y &= a^j \text{ for some } j \geq 0, \\
z &= a^lba^N \text{ for some } l \geq 0.
\end{align*}
\]

• Since $y \neq \Lambda$ by P2 of Theorem 14, we must have $j \geq 1$.

• Also, since $w = xyz$ by P1 of Theorem 14, note that

\[
w = a^Nba^N = xyz = a^i a^j a^l ba^N = a^{i+j+l}ba^N,
\]

so $i + j + l = N$.

• Now consider the string $xyyz$, which is supposed to be in $L$ by P4 of Theorem 14.

• Note that

\[
xyyz = a^i a^j a^l ba^N = a^{i+2j+l}ba^N = a^{N+j}ba^N
\]

since $i + j + l = N$.

• But $a^{N+j}ba^N \not\in \text{PALINDROME}$ since $\text{reverse}(a^{N+j}ba^N) \neq a^{N+j}ba^N$.

• This is a contradiction, and so PALINDROME must be nonregular.

\[\]
• Suppose \( L \) is a regular language.

• Pumping Lemma says that there exist strings \( x, y, \) and \( z \) such that all words of the form \( xy^kz \) are in \( L \), where \( y \) is not null.

• All words in \( L \) are of the form \( a^n b^n \).

• How do we break up \( a^n b^n \) into substrings \( x, y, z \) with \( y \) nonempty?
  
  - If \( y \) consists solely of \( a \)'s, then \( xyyz \) has more \( a \)'s than \( b \)'s, and so it is not in \( L \).
  
  - If \( y \) consists solely of \( b \)'s, then \( xyyz \) has more \( b \)'s than \( a \)'s, and so it is not in \( L \).
  
  - If \( y \) consists of \( a \)'s and \( b \)'s, then all of the \( a \)'s in \( y \) must come before all of the \( b \)'s. However, \( xyyz \) then has some \( b \)'s appearing before some \( a \)'s, and so \( xyyz \) is not in \( L \).

• Thus, \( L \) is not a regular language.

Example:

• Let \( \Sigma = \{a, b\} \).

• For any string \( w \in \Sigma^* \), define \( n_a(w) \) to be the number of \( a \)'s in \( w \), and \( n_b(w) \) to be the number of \( b \)'s in \( w \).

• Define the language \( L = \{ w \in \Sigma^* : n_a(w) \geq n_b(w) \} \); i.e., \( L \) consists of strings \( w \) for which the number of \( a \)'s in \( w \) is at least as large as the number of \( b \)'s in \( w \).

• For example, \( abbaa \in L \) since the string has 3 \( a \)'s and 2 \( b \)'s, and \( 3 \geq 2 \).

• We can prove that \( L \) is a nonregular language using the pumping lemma.

• What string \( w \in L \) should we use to get a contradiction?

Example: Consider the language \( \text{EQUAL} = \{ \Lambda, ab, ba, aabb, abab, abba, baba, bbba, \ldots \} \), which consists of all words having an equal number of \( a \)'s and \( b \)'s. We now prove that \( \text{EQUAL} \) is a non-regular language.
We will prove this by contradiction, so suppose that EQUAL is a regular language.

Note that \( \{a^n b^n : n \geq 0\} = a^* b^* \cap \text{EQUAL} \)

Recall that the intersection of two regular languages is a regular language.

Note that \( a^* b^* \) is a regular expression, and so its language is regular.

If EQUAL were a regular language, then \( \{a^n b^n : n \geq 0\} \) would be the intersection of two regular languages.

This would imply that \( \{a^n b^n : n \geq 0\} \) is a regular language, which is not true.

Thus, EQUAL must not be a regular language.

### 10.5 Prefix Languages

**Definition:** If \( R \) and \( Q \) are languages, then \( \text{Pref}(Q \text{ in } R) \) is the language of “the prefixes of \( Q \) in \( R \),” which is the set of all strings of letters that can be concatenated to the front of some word in \( Q \) to produce some word in \( R \); i.e.,

\[
\text{Pref}(Q \text{ in } R) = \{ \text{strings } p : \exists q \in Q \text{ such that } pq \in R \} 
\]

**Example:** \( Q = \{aba, aaabb, baaba, bbaaaaabb, aaaa\} \)
\( R = \{baabaaba, aaabb, abbabaaaabb\} \)
\( \text{Pref}(Q \text{ in } R) = \{baaba, \Lambda, abbabba, abba\} \)

**Example:** \( Q = \{aba, aaabb, baaba, bbaaaaabb, aaaa\} \)
\( R = \{baab, ababb\} \)
\( \text{Pref}(Q \text{ in } R) = \emptyset \)

**Example:** \( Q = ab^* a \)
\( R = (ba)^* \)
\( \text{Pref}(Q \text{ in } R) = (ba)^* b \)
Theorem 16 If $R$ is a regular language and $Q$ is any language whatsoever, then the language
\[ P = \text{Pref}(Q \text{ in } R) \]
is regular.

Proof.

- Since $R$ is a regular language, it has some finite automaton $FA_1$ that accepts it.
- $FA_1$ has one start state and several (possibly none or one) final states.
- For each state $s$ in $FA_1$, do the following:
  - Using $s$ as the start state, process all words in the language $Q$ on $FA_1$.
  - When starting $s$, if some word in $Q$ ends in the final state of $FA_1$, then paint state $s$ blue.
- So for each state $s$ in $FA_1$ that is painted blue, there exists some word in $Q$ that can be processed on $FA_1$ starting from $s$ and end up in a final state.
- Now construct another machine $FA_2$:
  - $FA_2$ has the same states and arcs as $FA_1$.
  - The start state of $FA_2$ is the same as that of $FA_1$.
  - The final states of $FA_2$ are the ones that were previously painted blue (regardless if they were final states in $FA_1$).
- We will now show that $FA_2$ accepts exactly the prefix language
  \[ P = \text{Pref}(Q \text{ in } R) \].
- To prove this, we have to show two things:
  - Every word in $P$ is accepted by $FA_2$.
  - Every word accepted by $FA_2$ is in $P$.
- First, we show that every word accepted by $FA_2$ is in $P$. 
Consider any word \( w \) accepted by \( FA_2 \).

- Starting in the start state of \( FA_2 \), process the word \( w \) on \( FA_2 \), and we end up in a final state of \( FA_2 \).
- Final states of \( FA_2 \) were painted blue.
- Now we can start from here and process some word from \( Q \) and end up in a final state of \( FA_1 \).
- Thus, the word \( w \in P \).

- Now we prove that every word in \( P \) is accepted by \( FA_2 \).

  - Consider any word \( p \in P \).
  - By definition, there exists some word \( q \in Q \) and a word \( w \in R \) such that \( pq = w \).
  - This implies that if \( pq \) is processed on \( FA_1 \), then we end up in a final state of \( FA_1 \).
  - When processing the string \( pq \) on \( FA_1 \), consider the state \( s \) we are in just after finishing processing \( p \) and at the beginning of processing \( q \).
  - State \( s \) must be a blue state since we can start here and process \( q \) and end in a final state.
  - Hence, by processing \( p \), we must start in the start state and end in state \( s \).
  - Thus, \( p \) is accepted by \( FA_2 \).
Chapter 11

Decidability for Regular Languages

11.1 Introduction

We have three basic questions to answer:

1. How can we tell if two regular expressions define the same language?
2. How can we tell if two FA’s are equivalent?
3. How can we tell if the language defined by an FA has finitely many or infinitely many words in it?

Note that questions 1 and 2 are essentially the same by Kleene’s Theorem.

11.2 Decidable Problems

Definition: A problem is effectively solvable if there is an algorithm that provides the answer in a finite number of steps, no matter what the particular inputs are (but may depend on the size of the problem).

The maximum number of steps the algorithm will take must be predictable before we begin executing the procedure.
**Example:** Problem: find roots of quadratic equation $ax^2 + bx + c = 0$.
Solution: use quadratic equation

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

No matter what the coefficients $a$, $b$, and $c$ are, we can compute the solution using the following operations:

- four multiplications
- two subtractions
- one square root
- one division

Another solution: keep guessing until we find a root.
This approach is not guaranteed to find root in a fixed number of steps.

**Example:** Find the maximum of $n$ numbers. An effective solution for this is to scan through the list once while updating the maximum observed thus far. This takes $O(n)$ steps.

**Definition:** An effective solution to a problem that has a yes or no answer is called a *decision procedure*. A problem that has a decision procedure is called *decidable*.

11.2.1 Is $L_1 = L_2$?

Determine if two languages $L_1$ and $L_2$ are the same:

- Method 1: Check if the language

  $$L_3 \equiv (L_1 \cap L_2') + (L_1' \cap L_2)$$

  has any words (even $\Lambda$).
- If $L_1 = L_2$, then $L_3 = \emptyset$. 
• If \( L_1 \neq L_2 \), then \( L_3 \neq \emptyset \).

**Example:** Suppose \( L_1 = \{a, aa\} \) and \( L_2 = \{a, aa, aaa\} \). Then \( L_1 \cap L_2' = \emptyset \), but \( L_1' \cap L_2 = \{aaa\} \). Thus, \( L_1 \neq L_2 \).

• So now we have reduced the problem of determining if \( L_1 = L_2 \) to determining if \( L_3 = \emptyset \).

### 11.2.2 Is \( L = \emptyset? \)

• So we need a method for determining if a regular language is empty.

• Since the language is regular, it has a regular expression and a FA.

• Given a regular expression, check if there is any part that is not concatenated with \( \emptyset \).

• Specifically, use the following algorithm to determine if \( L = \emptyset \) given a regular expression \( r \) for \( L \):

• **Method 1** (for deciding if a language \( L = \emptyset \) given regular expression \( r \) for \( L \)):
  
  - Write \( r \) as
    
    \[
    r = r_1 + r_2 + \cdots + r_n,
    \]
    
    where for each \( i = 1, 2, \ldots, n \), \( r_i = r_{i,1}r_{i,2}\cdots r_{i,j_i} \) for some \( j_i \geq 1 \); i.e., \( r \) is written as a "sum" of other regular expressions \( r_i \), \( i = 1, 2, \ldots, n \), where each \( r_i \) is a concatenation of regular expressions. It is always possible to write any regular expression \( r \) in this form.

  - If there exists some \( i = 1, 2, \ldots, n \) such that \( r_{i,j} \neq \emptyset \) for all \( 1 \leq j \leq j_i \), then \( L \neq \emptyset \). In other words, if one of the summands has none of its "factors" being \( \emptyset \), then the language \( L \) is not empty.

  - If for each \( i = 1, 2, \ldots, n \), at least one of \( r_{i,1}, r_{i,2}, \ldots, r_{i,j_i} \) is \( \emptyset \), then \( L = \emptyset \). In other words, if each of the summands has at least one "factor" being \( \emptyset \), then the language \( L \) is empty.

**Example:** The regular expression

\[
\emptyset(b + a)^* + b
\]
has the last $b$ not concatenated with $\emptyset$ so the language is not empty.

**Example:** The regular expression

$$\emptyset(b + a)^* + \emptyset b$$

has all parts concatenated with $\emptyset$ so the language is empty.

**Remarks:** The algorithm in the book for determining if $L = \emptyset$ given a regular expression for $L$ is incorrect.

- **Method 2** (for deciding if a language $L = \emptyset$): Given an FA, we check if there are any paths from $-$ to some $+$ state by using the “blue paint algorithm”:

  1. Paint the start state blue.
  2. From every blue state, follow each edge that leads out of it and paint the connecting state blue, then delete this edge from the machine.
  3. Repeat Step 2 until no new state is painted blue, then stop.
  4. When the procedure has stopped, if any of the final states are painted blue, then the machine accepts some words, and if not, the machine accepts no words.

Remarks on Method 2:

- The above algorithm will iterate Step 2 at most $N$ times, where $N$ is the number of states in the machine.
- Thus, it is a decision procedure.
Example:
Theorem 17 Let $F$ be an FA with $N$ states. Then if $F$ accepts any strings at all, it accepts some string with $N - 1$ or fewer letters.

Proof.

- Consider any string $w$ that is accepted by $F$.
- Let $s = w$ and DONE = NO.
- Do while (DONE == NO)
  * Trace path of $s$ through $F$.
  * If no circuits in path, then set DONE = YES.
  * If there are circuits in the path, then
    - Eliminate first circuit in the path.
    - Let $s$ be the string resulting from the new path.
- Resulting path:
  * Starts in initial state.
  * Ends in a final state.
  * Has no circuits, so visits at most $N$ states.
  * This corresponds to a string of at most $N - 1$ letters.
  * String is accepted by FA.

Method 3 (for deciding if a language $L = \emptyset$): Test all words with $N - 1$ or fewer letters by running them on the FA.
Example: Consider the languages $L_1$ and $L_2$ with FA's:

FA1:

FA2:

FA for both $L_1 \cap L_2'$ and $L_1' \cap L_2$
Theorem 18 There are effective procedures to decide whether:

1. A given FA accepts any words.
2. Two FA’s are equivalent; i.e., the two FA’s accept the same language.
3. Two regular expressions are equivalent; i.e., the two regular expressions generate the same language.

Remarks:

• We can establish part 3 of Theorem 18 by first converting the regular expressions into FA’s.
• We previously saw an effective procedure for doing this in the proof of Kleene’s Theorem.
• Then we just developed an effective procedure to decide whether two FA’s are equivalent.

11.2.3 Is $L$ infinite?

Determining if a language $L$ is infinite

• If we have a regular expression for $L$, then all we need to do is check if the $\ast$ is applied to some part of the regular expression that is not $\Lambda$ nor $\emptyset$.
• Note that $\Lambda^* = \Lambda$ and $\emptyset^* = \Lambda$.
• Note that $a^*$ is infinite.

Theorem 19 Let $F$ be an FA with $N$ states. Then

1. If $F$ accepts an input string $w$ such that
   
   \[ N \leq \text{length}(w) < 2N \]

   then $F$ accepts an infinite language.

2. If $F$ accepts infinitely many words, then $F$ accepts some word $w$ such that

   \[ N \leq \text{length}(w) < 2N \]
CHAPTER 11. DECIDABILITY FOR REGULAR LANGUAGES

Proof.

1. • Assume that $F$ accepts an input string $w$ such that

\[ N \leq \text{length}(w) < 2N \]

• Since $\text{length}(w) \geq N$, the second version of the pumping lemma (Theorem 14) implies that there exist substrings $x$, $y$, and $z$ such that $y \neq \text{}$ and $xy^nz$, $n = 0, 1, 2, \ldots$, are all accepted by $F$.

• Thus, the FA accepts infinitely many words.

2. • Assume that $F$ accepts infinitely many words.

• This implies that there exists some word $u$ accepted by $F$ that has a circuit (possibly more than one). Why?

• Each circuit can consist of at most $N$ states since $F$ has only $N$ states.

• Iteratively eliminate the first circuit in the path until only one circuit left (as in the proof of Theorem 17).

• Let $v$ correspond to the word from this one-circuit path, and note that $v$ is accepted by $F$.

• We can write $v$ as the concatenation of three strings $x$, $y$, and $z$, i.e.,

\[ v = xyz, \]

such that

- $x$ consists of the letters read before the circuit.
- $y$ consists of the letters read along the circuit.
- $z$ consists of the letters read after the circuit

• We can show that

\[ 0 < \text{length}(y) \leq N \]

as follows:

- Since we have eliminated all but the first circuit, the circuit starts and ends in the same state and all of the other states are unique.

- Thus, the circuit can visit at most $N + 1$ states (with at most one state repeated).

- This corresponds to reading at most $N$ letters.
Also, since a circuit corresponds to at least one transition and each transition in an FA uses up exactly one letter, we see that 
\[ \text{length}(y) > 0. \]

- We can show that 
\[ \text{length}(x) + \text{length}(z) < N \]
as follows:
  - Since we constructed the string \( v \) by eliminating all but the first circuit, the paths followed by processing \( x \) and \( z \) have no circuits.
  - Thus, all of the states visited along the paths followed by processing \( x \) and \( z \) are unique.
  - Hence, the paths followed by processing \( x \) and \( z \) visit at most \( N \) states.
  - This means that 
\[ \text{length}(x) + \text{length}(z) \leq N - 1 < N. \]
- Thus,
\[ \text{length}(v) = \text{length}(x) + \text{length}(y) + \text{length}(z) \leq N - 1 + N < 2N. \]

- If \( v \) has at least \( N \) letters, then we are done.
- If \( v \) has less than \( N \) letters, then we can pump up the cycle some number of times to obtain a word that has the desired characteristics since \( 0 < \text{length}(y) \leq N. \)
Example:

Consider the word $w = abaaaababbabb$

- $\text{length}(w) = 13 > 2N = 12$.
- $w$ is accepted by the FA.
- Processing $w$ on FA takes the path
  
  $1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 6$

  circuit 1 circuit 2 circuit 3 circuit 4

- Bypassing all but the first circuit yields the path
  
  $1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 6$

  which corresponds to the word $abaaab$, which has length 6.
- Thus, Theorem 19 implies that the FA accepts an infinite language.

Consider the word $w = bbaabb$

- $\text{length}(w) = 6 = N$
- $w$ is accepted by the FA.
• Processing $w$ on FA takes the path

$$1 \rightarrow \text{circuit 1} \rightarrow 3 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow \text{circuit 2} \rightarrow 6 \rightarrow 6$$

• Bypassing all but the first circuit yields the path

$$1 \rightarrow \text{circuit 1} \rightarrow 3 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 6$$

which corresponds to the word $bbaab$, which has length 5.

• However, we can go around the circuit one more time, yielding the path

$$1 \rightarrow \text{circuit 1} \rightarrow 3 \rightarrow \text{circuit 1} \rightarrow 3 \rightarrow 3 \rightarrow \text{circuit 1} \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 6$$

which corresponds to the word $bbbaab$, which has length 6.
Theorem 20 There is an effective procedure to decide whether a given FA accepts a finite or an infinite language.

Proof.

- Suppose that the FA has $N$ states.
- Suppose that the alphabet consists of $m$ letters.
- Then by Theorem 19, we only need to check all strings $w$ with
  \[ N \leq \text{length}(w) < 2^N \]
to determine if FA accepts an infinite language.
- If any of these are accepted, then the FA accepts an infinite language. Otherwise, it accepts a finite language.
- The number of strings $w$ satisfying
  \[ N \leq \text{length}(w) < 2^N \]
is
  \[ m^N + m^{N+1} + m^{N+2} + \cdots + m^{2^N-1} \]
  which is finite.
- Thus, checking all of these strings is an effective procedure.
Chapter 12

Context-Free Grammars

12.1 Introduction

English grammar has rules for constructing sentences; e.g.,

1. A sentence can be a subject followed by a predicate.
2. A subject can be a noun-phrase.
3. A noun-phrase can be an adjective followed by a noun-phrase.
4. A noun-phrase can be an article followed by a noun-phrase.
5. A noun-phrase can be a noun.
6. A predicate can be a verb followed by a noun-phrase.
7. A noun can be: person fish stapler book
8. A verb can be: buries touches grabs eats
9. An adjective can be: big small
10. An article can be: the a an
These rules can be used to construct the following sentence:

*The small person eats the big fish*

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>sentence ⇒ subject predicate</td>
<td>Rule 1</td>
</tr>
<tr>
<td>⇒ noun-phrase predicate</td>
<td>Rule 2</td>
</tr>
<tr>
<td>⇒ noun-phrase verb noun-phrase</td>
<td>Rule 6</td>
</tr>
<tr>
<td>⇒ article noun-phrase verb noun-phrase</td>
<td>Rule 4</td>
</tr>
<tr>
<td>⇒ article adjective noun-phrase verb noun-phrase</td>
<td>Rule 3</td>
</tr>
<tr>
<td>⇒ article adjective noun verb noun-phrase</td>
<td>Rule 5</td>
</tr>
<tr>
<td>⇒ article adjective noun verb article noun-phrase</td>
<td>Rule 4</td>
</tr>
<tr>
<td>⇒ article adjective noun verb article adjective noun-phrase</td>
<td>Rule 3</td>
</tr>
<tr>
<td>⇒ article adjective noun verb article adjective noun</td>
<td>Rule 5</td>
</tr>
<tr>
<td>⇒ the adjective noun verb article adjective noun</td>
<td>Rule 10</td>
</tr>
<tr>
<td>⇒ the small noun verb article adjective noun</td>
<td>Rule 9</td>
</tr>
<tr>
<td>⇒ the small person verb article adjective noun</td>
<td>Rule 7</td>
</tr>
<tr>
<td>⇒ the small person eats article adjective noun</td>
<td>Rule 8</td>
</tr>
<tr>
<td>⇒ the small person eats the adjective noun</td>
<td>Rule 10</td>
</tr>
<tr>
<td>⇒ the small person eats the big noun</td>
<td>Rule 9</td>
</tr>
<tr>
<td>⇒ the small person eats the big fish</td>
<td>Rule 7</td>
</tr>
</tbody>
</table>

**Definition:** The things that cannot be replaced by anything are called *terminals*.

**Definition:** The things that must be replaced by other things are called *nonterminals*.

In the above example,

- *small* and *eats* are terminals.
- *noun-phrase* and *verb* are nonterminals.
Example: restricted class of arithmetic expressions on integers.

\[
\begin{align*}
\text{start} & \rightarrow \text{AE} \\
\text{AE} & \rightarrow \text{AE} + \text{AE} \\
\text{AE} & \rightarrow \text{AE} - \text{AE} \\
\text{AE} & \rightarrow \text{AE} \times \text{AE} \\
\text{AE} & \rightarrow \text{AE} / \text{AE} \\
\text{AE} & \rightarrow \text{AE}^{\text{AE}} \\
\text{AE} & \rightarrow (\text{AE}) \\
\text{AE} & \rightarrow -\text{AE} \\
\text{AE} & \rightarrow \text{ANY-NUMBER}
\end{align*}
\]

- nonterminals: \textit{start}, \textit{AE}
- Can generate the arithmetic expression

\[
\text{ANY-NUMBER} + (\text{ANY-NUMBER} - \text{ANY-NUMBER}) / \text{ANY-NUMBER}
\]

as follows:

\[
\begin{align*}
\text{start} & \Rightarrow \text{AE} \\
& \Rightarrow \text{AE} + \text{AE} \\
& \Rightarrow \text{AE} + \text{AE} / \text{AE} \\
& \Rightarrow \text{AE} + (\text{AE}) / \text{AE} \\
& \Rightarrow \text{AE} + (\text{AE} - \text{AE}) / \text{AE} \\
& \Rightarrow \text{ANY-NUMBER} + (\text{AE} - \text{AE}) / \text{AE} \\
& \Rightarrow \text{ANY-NUMBER} + (\text{ANY-NUMBER} - \text{AE}) / \text{AE} \\
& \Rightarrow \text{ANY-NUMBER} + (\text{ANY-NUMBER} - \text{ANY-NUMBER}) / \text{AE} \\
& \Rightarrow \text{ANY-NUMBER} + (\text{ANY-NUMBER} - \text{ANY-NUMBER}) / \text{ANY-NUMBER}
\end{align*}
\]
CHAPTER 12. CONTEXT-FREE GRAMMARS

Could also make \texttt{ANY-NUMBER} a nonterminal:

Rule 1 \texttt{ANY-NUMBER} \rightarrow \texttt{FIRST-DIGIT}
Rule 2 \texttt{FIRST-DIGIT} \rightarrow \texttt{FIRST-DIGIT \ OTHER-DIGIT}
Rule 3 \texttt{FIRST-DIGIT} \rightarrow 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9
Rule 4 \texttt{OTHER-DIGIT} \rightarrow 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9

In this case,

- nonterminals: \texttt{ANY-NUMBER}, \texttt{FIRST-DIGIT}, \texttt{OTHER-DIGIT}
- terminals: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Can produce the number 90210 as follows:

Rule 1 \texttt{ANY-NUMBER} \Rightarrow \texttt{FIRST-DIGIT}
Rule 2 \Rightarrow \texttt{FIRST-DIGIT \ OTHER-DIGIT}
Rule 2 \Rightarrow \texttt{FIRST-DIGIT \ OTHER-DIGIT \ OTHER-DIGIT}
Rule 2 \Rightarrow \texttt{FIRST-DIGIT \ OTHER-DIGIT \ OTHER-DIGIT \ OTHER-DIGIT}
Rule 2 \Rightarrow \texttt{FIRST-DIGIT \ OTHER-DIGIT \ OTHER-DIGIT \ OTHER-DIGIT \ OTHER-DIGIT}
Rule 4 \Rightarrow 9 \ \texttt{OTHER-DIGIT \ OTHER-DIGIT \ OTHER-DIGIT}
Rule 4 \Rightarrow 9 \ 0 \ \texttt{OTHER-DIGIT \ OTHER-DIGIT}
Rule 4 \Rightarrow 9 \ 0 \ 2 \ \texttt{OTHER-DIGIT}
Rule 4 \Rightarrow 9 \ 0 \ 2 \ 1 \ \texttt{OTHER-DIGIT}

Note that we had rules of the form:

one nonterminal \rightarrow\ string of nonterminals

or

one nonterminal \rightarrow\ choice of terminals

**Definition:** The sequence of applications of the rules that produces the finished string of terminals from the starting symbol is called a \textit{derivation} or \textit{production}.
12.2 Context-Free Grammars

Example: terminals: $\Sigma = \{a\}$
nonterminal: $\Omega = \{S\}$
productions:

\[
S \rightarrow aS \\
S \rightarrow \Lambda
\]

- Can generate $a^4$ as follows:

\[
S \Rightarrow aS \\
\Rightarrow aaS \\
\Rightarrow aaaS \\
\Rightarrow aaaaS \\
\Rightarrow aaaa\Lambda = aaaa
\]

Example: terminal: $a$
nonterminal: $S$
productions:

\[
S \rightarrow SS \\
S \rightarrow a \\
S \rightarrow \Lambda
\]

Can write this in more compact notation:

\[
S \rightarrow SS \mid a \mid \Lambda
\]

which is called the Backus Normal Form or Backus-Naur Form (BNF).

- CFL is $a^*$
- Can generate $a^2$ as follows:

\[
S \Rightarrow SS \\
\Rightarrow SSS \\
\Rightarrow SSa \\
\Rightarrow SSSa \\
\Rightarrow SaSa \\
\Rightarrow \Lambda aSa \\
\Rightarrow \Lambda a\Lambda a = aa
\]
- In previous example, unique way to generate any word.
- Here, each word in CFL has infinitely many derivations.

**Definition:** A *context-free grammar* (CFG) is a collection $G = (\Sigma, \Omega, R, S)$, with

1. A (finite) alphabet $\Sigma$ of letters called *terminals* from which we make strings that will be the words of the language.
2. A finite set $\Omega$ of symbols called *nonterminals*, one of which is the symbol $S$ (i.e., $S \in \Omega$), standing for “start here.”
3. A finite set $R$ of *productions*, with $R \subseteq \Omega \times (\Sigma + \Omega)^*$. If a production $(N, U) \in R$ with $N \in \Omega$ and $U \in (\Sigma + \Omega)^*$, then we write the production as $N \rightarrow U$.

Thus, each production is of the form

one nonterminal $\rightarrow$ finite string of terminals and/or nonterminals

where the strings of terminals, nonterminals can consist of only terminals or of only nonterminals, or any mixture of terminals and nonterminals or even the empty string. We require that at least one production has the nonterminal $S$ as its left side.

**Convection:**
- Terminals will typically be smallcase letters.
- Nonterminals will typically be uppercase letters.

**Definition:** The *language generated* (defined, derived, produced) by a CFG $G$ is the set of all strings of terminals that can be produced from the start symbol $S$ using the productions as substitutions. A language generated by a CFG $G$ is called a *context-free language* (CFL) and is denoted by $L(G)$.

**Example:** terminals: $\Sigma = \{a\}$
nonterminal: $\Omega = \{S\}$
productions:

\[
S \rightarrow aS \\
S \rightarrow \Lambda
\]
Let $L_1$ the language generated by this CFG, and let $L_2$ be the language generated by regular expression $a^*$. 

**Claim:** $L_1 = L_2$. 

**Proof:**

* We first show that $L_2 \subseteq L_1$.
  · Consider $a^n \in L_2$ for $n \geq 1$. We can generate $a^n$ by using first production $n$ times, and then second production.
  · Can generate $\Lambda \in L_2$ by using second production only.
  · Hence $L_2 \subseteq L_1$.
* We now show that $L_1 \subseteq L_2$.
  · Since $a$ is the only terminal, CFG can only produce strings having only $a$’s.
  · Thus, $L_1 \subseteq L_2$.

Note that

- Two types of arrows:
  → used in statement of productions
  ⇒ used in derivation of word

- in the above derivation of $a^4$, there were many unfinished stages that consisted of both terminals and nonterminals. These are called working strings.
- $\Lambda$ is neither a nonterminal (since it cannot be replaced with something else) nor a terminal (since it disappears from the string).

### 12.3 Examples

**Example:** terminals: $a, b$
nonterminals: $S$
productions:

$$
S \rightarrow aS \\
S \rightarrow bS \\
S \rightarrow a \\
S \rightarrow b
$$
More compact notation:

\[ S \to aS \mid bS \mid a \mid b \]

- Can produce the word \textit{abbab} as follows:

\[
\begin{align*}
S & \Rightarrow aS \\
& \Rightarrow abS \\
& \Rightarrow abbS \\
& \Rightarrow abbaS \\
& \Rightarrow abbab
\end{align*}
\]

- Let \( L_1 \) be the CFL, and let \( L_2 \) be the language generated by the regular expression \((a + b)^+\).

**Claim:** \( L_1 = L_2 \).

**Proof:**

- First we show that \( L_2 \subset L_1 \).
  
  * Consider any string \( w \in L_2 \).
  * Read letters of \( w \) from left to right.
  * For each letter read in, if it is not the last, then
    - use the production \( S \to aS \) if the letter is \( a \) or
    - use the production \( S \to bS \) if the letter is \( b \)
  * For the last letter of the word,
    - use the production \( S \to a \) if the letter is \( a \) or
    - use the production \( S \to b \) if the letter is \( b \)
  * In each stage of the derivation, the working string has the form

    \[(\text{string of terminals})S\]

  - Hence, we have shown how to generate \( w \) using the CFG, which means that \( w \in L_1 \).
  - Hence, \( L_2 \subset L_1 \).

- Now we show that \( L_1 \subset L_2 \).
  
  * To show this, we need to show that if \( w \in L_1 \), then \( w \in L_2 \).
  * This is equivalent to showing that if \( w \notin L_2 \), then \( w \notin L_1 \).
Note that the only string \( w \notin L_2 \) is \( w = \Lambda \).

But note that \( \Lambda \) cannot be generated by the CFG, so \( \Lambda \notin L_1 \).

Hence, we have proven that \( L_1 \subset L_2 \).

**Example:** terminals: \( a, b \)
nonterminals: \( S, X, Y \)
productions:

\[
\begin{align*}
S & \to X \mid Y \\
X & \to \Lambda \\
Y & \to aY \mid bY \mid a \mid b
\end{align*}
\]

- Note that if we use first production \( (S \to X) \), then the only word we can generate is \( \Lambda \).
- The second production \( (S \to Y) \) leads to a collection of productions identical to the previous example.
- Thus, the second production produces \( (a+b)^+ \).
- CFL is \( (a+b)^+ \)

**Example:** terminals: \( a, b \)
nonterminals: \( S \)
productions:

\[
S \to aS \mid bS \mid a \mid b \mid \Lambda
\]

- CFL is \( (a+b)^+ \)
- For this CFG, the sequence of productions to generate any word is not unique.
- e.g., can generate \( bab \) using

\[
\begin{align*}
S & \Rightarrow bS \\
& \Rightarrow baS \\
& \Rightarrow babS \\
& \Rightarrow bab\Lambda = bab
\end{align*}
\]
or

\[ S \Rightarrow bS \]
\[ \Rightarrow baS \]
\[ \Rightarrow bab \]

**Example:** terminals: \( a, b \)
nonterminals: \( S, X \)
productions:

\[
\begin{align*}
S & \rightarrow XaaX \\
X & \rightarrow aX \mid bX \mid \Lambda
\end{align*}
\]

- The last set of productions generates any word from \( \Sigma^* \).
- CFL is \((a + b)^*aa(a + b)^*\)
- Can generate \( abbaaba \) as follows:

\[
\begin{align*}
S & \Rightarrow XaaX \\
& \Rightarrow aXaaX \\
& \Rightarrow abXaaX \\
& \Rightarrow abbXaaX \\
& \Rightarrow abb\Lambda aaX = abbaaX \\
& \Rightarrow abbaabX \\
& \Rightarrow abbaabaX \\
& \Rightarrow abbaaba\Lambda = abbaaba
\end{align*}
\]

**Example:** terminals: \( a, b \)
nonterminals: \( S, X, Y \)
productions:

\[
\begin{align*}
S & \rightarrow XY \\
X & \rightarrow aX \mid bX \mid a \\
Y & \rightarrow Ya \mid Yb \mid a
\end{align*}
\]

- \( X \) productions can produce words ending with \( a \).
• Y productions can produce words starting with a.
• CFL is \((a + b)^*aa(a + b)^*\)
• Can generate \(abbaaba\) as follows:

\[
\begin{align*}
S & \Rightarrow XY \\
& \Rightarrow aXY \\
& \Rightarrow abXY \\
& \Rightarrow abbXY \\
& \Rightarrow abbaY \\
& \Rightarrow abbaYa \\
& \Rightarrow abbaYba \\
& \Rightarrow abbaaba
\end{align*}
\]

Example: Give CFGs for each of the following languages over the alphabet \(\Sigma = \{a, b\}\):

1. \(\{a^n b^n : n \geq 0\}\)
2. PALINDROME
3. EVEN-PALINDROME
4. ODD-PALINDROME

Example: terminals: \(a, b\)
nonterminals: \(S, B, U\)
productions:

\[
\begin{align*}
S & \rightarrow SS | BS | SB | \Lambda | USU \\
B & \rightarrow aa | bb \\
U & \rightarrow ab | ba
\end{align*}
\]

Show that this generates EVEN-EVEN

• Note that starting from \(B\), we can generate a balanced pair, i.e., either \(aa\) or \(bb\).
• Starting from \( U \), we can generate an unbalanced pair, i.e., either \( ab \) or \( ba \).

• First show that every word in EVEN-EVEN can be generated using these productions.

  Recall that EVEN-EVEN has regular expression
  \[
  [aa + bb + (ab + ba)(aa + bb)^*(ab + ba)]^*
  \]

  Three types of syllables:
  1. \( aa \),
  2. \( bb \),
  3. \((ab + ba)(aa + bb)^*(ab + ba)\)

  Consider any word generated from the regular expression for EVEN-EVEN. Let’s examine the way it was generated using the regular expression, and show how to generate the same word using our CFG.

  Start our derivation using the CFG from \( S \).

  Every time we iterate the outer star in the regular expression, we choose one of the three syllables.

  1. If we choose a syllable of type 1, then first use the production \( S \to BS \) and then the production \( B \to aa \). Thus, we end up with a working string of \( aaS \) for this iteration of the outer star.

  2. If we choose a syllable of type 2, then first use the production \( S \to BS \) and then the production \( B \to bb \). Thus, we end up with a working string of \( bbS \) for this iteration of the outer star.

  3. If we choose a syllable of type 3, then
    (a) First use the production \( S \to SS \).
    (b) Then change the first \( S \) using the production \( S \to USU \), resulting in \( USUS \).
    (c) If the first \((ab+ba)\) in the syllable \((ab+ba)(aa+bb)^*(ab+ba)\) is used to generate \( ab \), then replace the first \( U \) in \( USUS \) using the production \( U \to ab \), resulting in \( abSUS \). If the first \((ab + ba)\) in \((ab + ba)(aa + bb)^*(ab + ba)\) is used to generate \( ba \), then replace the first \( U \) in \( USUS \) using the production \( U \to ba \), resulting in \( baSUS \). Do the same for the second \((ab + ba)\) in \((ab + ba)(aa + bb)^*(ab + ba)\). Thus,
we now have $xSyS$ as a working string for this iteration of the outer star of the regular expression, where $x$ is either $ab$ or $ba$, and $y$ is either $ab$ or $ba$.

(d) Now suppose the $(aa + bb)^*$ is iterated $n$ times, $n \geq 0$. If $n = 0$, then change the first $S$ in $xSyS$ using the production $S \rightarrow \Lambda$, resulting in $x\Lambda yS = xyS$. If $n \geq 1$, then change the first $S$ in $xSyS$ using the production $S \rightarrow BS$ and do this $n$ times, resulting in $xBBB \cdots BSyS$, where there are $n$ $B$’s in the clump of $B$’s. Then change the first $S$ using the production $S \rightarrow \Lambda$, resulting in $xBBB \cdots B\Lambda yS = xBBB \cdots ByS$, where there are $n$ $B$’s in the clump of $B$’s. Finally, if on the $k$th iteration, $k \leq n$, of the $*$ in $(aa + bb)^*$ we generated $aa$, then replace the $k$th $B$ using the production $B \rightarrow aa$. If on the $k$th iteration, $k \leq n$, of the $*$ in $(aa + bb)^*$ we generated $bb$, then replace the $k$th $B$ using the production $B \rightarrow bb$.

After completing all of the iterations of the outer star in the regular expression, use the production $S \rightarrow \Lambda$.

- e.g., for word $babbaba \in$ EVEN-EVEN,

\[
S \Rightarrow SS \\
\quad \Rightarrow USUS \\
\quad \Rightarrow baSUS \\
\quad \Rightarrow baBSUS \\
\quad \Rightarrow babbSUS \\
\quad \Rightarrow babb\Lambda US = babbUS \\
\quad \Rightarrow babbabS \\
\quad \Rightarrow babbabaS \\
\quad \Rightarrow babbabaa\Lambda = babbabaa
\]

- Now show that all words generated by these productions are in EVEN-EVEN.

- all words derived from $S$ can be decomposed into two-letter syllables.

- unbalanced syllables ($ab$ and $ba$) come into working string in pairs, which adds two $a$’s and two $b$’s.
balanced syllables add two of one letter and none of the other
thus, the sum total of a’s will be even, and the sum total of b’s will be even
Thus, word generated by productions will be in EVEN-EVEN.

Example: terminals: a, b
nonterminals: S, A, B
productions:

\begin{align*}
S & \rightarrow aB \mid bA \\
A & \rightarrow a \mid aS \mid bAA \\
B & \rightarrow b \mid bS \mid aBB
\end{align*}

This generates the language EQUAL, which consists of all strings of positive length and that have an equal number of a’s and b’s.

Proof. Need to show two things:

1. every word in EQUAL can be generated using our productions.
2. every word generated by our productions is in EQUAL.

First we show 1.

- We make three claims:
  
  **Claim 1**: All words in EQUAL can be generated by some sequence of productions beginning with the start symbol S.
  
  **Claim 2**: All words that have one more a than b’s can be generated from these productions by starting with the nonterminal A.
  
  **Claim 3**: All words that have one more b than a’s can be generated from these productions by starting with the nonterminal B.

- We will prove that these three claims hold by contradiction.

- Assume that one of the three claims does not hold.

- Then there is some smallest word w that violates one of the claims.

- All words shorter than w must satisfy the three claims.
• First assume that \( w \) violates Claim 1.
  
  * This means that \( w \) is in EQUAL but cannot be generated starting with \( S \).
  * Assume that \( w \) starts with \( a \) and that \( w = aw_1 \).
  * Since \( w \in \text{EQUAL} \), \( w_1 \) must have exactly one more \( b \) than \( a \)'s.
  * However, \( w_1 \) is shorter than \( w \).
  * Thus, we must be able to generate \( w_1 \) starting with \( B \); i.e.,
    \[
    B \Rightarrow \cdots \Rightarrow w_1
    \]
  * But then
    \[
    S \Rightarrow aB \Rightarrow \cdots \Rightarrow aw_1 = w
    \]
    which is a contradiction.
  * We similarly reach a contradiction when the first letter of \( w \) is \( b \).
  * Thus, \( w \) cannot violate Claim 1.

• Now assume that \( w \) violates Claim 2.
  
  * This means that \( w \) has one more \( a \) than \( b \)'s but cannot be generated starting with \( A \).
  * First assume that \( w \) starts with \( a \).
    * Then \( w = aw_1 \), where \( w_1 \in \text{EQUAL} \).
    * Since \( w_1 \) is shorter than \( w \), we must be able to generate \( w_1 \) starting with \( S \); i.e.,
      \[
      S \Rightarrow \cdots \Rightarrow w_1
      \]
    * But then
      \[
      A \Rightarrow aS \Rightarrow \cdots \Rightarrow aw_1 = w
      \]
      which is a contradiction.
  * Now assume that \( w \) starts with \( b \).
    * Then if we write \( w = bw_1 \), then \( w_1 \) has two more \( a \)'s than \( b \)'s.
    * We now split \( w_1 = w_{11}w_{12} \), where \( w_{11} \) is the part of \( w_1 \) scanning from left to right until there is exactly one more \( a \) than \( b \)'s, and let \( w_{12} \) be the rest of \( w_1 \).
    * Note that \( w_{12} \) also has exactly one more \( a \) than \( b \)'s.
CHAPTER 12. CONTEXT-FREE GRAMMARS

* Since \( w_{11} \) and \( w_{12} \) are both shorter than \( w \), we must be able to
generate each of them starting with \( A \); i.e.,

\[
A \Rightarrow \cdots \Rightarrow w_{11}
\]

and

\[
A \Rightarrow \cdots \Rightarrow w_{12}
\]

* But then

\[
A \Rightarrow bAA \Rightarrow \cdots \Rightarrow bw_{11}w_{12} = bw_1 = w
\]

which is a contradiction.

- Thus we have shown that Claim 2 must hold.

- We can similarly show that Claim 3 must hold.

- Thus, all 3 claims hold, and so in particular, Claim 1 holds: all words in
EQUAL can be generated starting from \( S \).

Now we show 2 holds: every word generated by our productions is in EQUAL.

- We again make 3 claims
  
  Claim 4 All words generated from \( S \) are in EQUAL.
  Claim 5 All words generated from \( A \) have one more \( a \) than \( b \)'s.
  Claim 6 All words generated from \( B \) have one more \( b \) than \( a \)'s.

- We will show that these 3 claims hold by contradiction.

- Assume that one of the three claims does not hold.

- Then there is some smallest word \( w \) generated from \( S, A, \) or \( B \) that does
not have the required property.

- All words shorter than \( w \) must satisfy the three claims.

- First assume that \( w \) violates Claim 4.

  - We have assumed that \( w \) can be generated from \( S \) but is not in
EQUAL.

  - Assume that the first letter of \( w \) is \( a \).
Then $w$ was generated by first using the production $S \rightarrow aB$.

To generate $w$, this $B$ generates a word $w_1$ which is shorter than $w$ and by assumption $w_1$ has one more $b$ than $a$'s.

This implies that $w$ has an equal number of $a$'s and $b$'s, which is a contradiction.

We get a similar contradiction if the first letter of $w$ is $b$.

• Now assume that $w$ violates Claim 5.

We have assumed that $w$ can be generated from $A$ but does not have exactly one more $a$ than $b$'s.

$w$ could not have been generated by $A \rightarrow a$ since $w = a$, which satisfies the requirement.

Suppose $w$ was generated by first using the production $A \rightarrow aS$.

* Then to generate the rest of $w$, we would have to start from $S$ to generate $w_1$, where $w = aw_1$.
* However, since $w_1$ is shorter than $w$ and $w_1$ is generated starting with $S$, we must have that $w_1 \in \text{EQUAL}$.
* This implies that $w$ has exactly one more $a$ than $b$'s, which is a contradiction.

Suppose $w$ was generated by first using the production $A \rightarrow bAA$.

* To generate the rest of $w$, each of the $A$'s need to generate strings $w_1$ and $w_2$ which are shorter than $w$ such that $w = bw_1w_2$.
* However, since $w_1$ and $w_2$ are shorter than $w$, we must have that $w_1$ and $w_2$ each have exactly one more $a$ than $b$'s.
* Hence, $w = bw_1w_2$ must have exactly one more $a$ than $b$'s, which is a contradiction.

Thus, we have shown that Claim 5 must hold.

• We can similarly show that Claim 6 must hold.

• Thus, all of the claims hold, and in particular, Claim 4: all words generated from $S \in \text{EQUAL}$. 
12.4 Trees

Can use a tree to illustrate how a word is derived from a CFG.

**Definition:** These trees are called *syntax trees, parse trees, generation trees, production trees,* or *derivation trees.*

**Example:** CFG:
- terminals: $a, b$
- nonterminals: $S, A$
- productions:
  
  \[
  S \rightarrow AAA \mid A \\
  A \rightarrow AA \mid aA \mid Ab \mid a \mid b
  \]

String $abaaba$ has the following derivation:

\[
S \Rightarrow AAA \\
\Rightarrow aAAA \\
\Rightarrow abAA \\
\Rightarrow abAbA \\
\Rightarrow abaAbA \\
\Rightarrow abaabA \\
\Rightarrow abaaba
\]

which corresponds to the following derivation tree:
Example: CFG for simplified arithmetic expressions.

terminals: +, *, 0, 1, 2, ..., 9

nonterminals: S

productions:
\[ S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \cdots \mid 9 \]

- Consider the expression 2 * 3 + 4.
- Ambiguous how to evaluate this:
- Does this mean \((2 * 3) + 4 = 10\) or \(2 * (3 + 4) = 14\)?
- Can eliminate ambiguity by examining the two possible derivation trees

Eliminate the S’s as follows:
Note that we can construct a new notation for mathematical expressions:

- start at top of tree
- walk around tree keeping left hand touching tree
- first time hit each terminal, print it out.

This gives us a string which is in operator prefix notation or Polish notation. In above examples,

- first tree yields \[ + \ast 2 3 4 \]
- second tree yields \[ \ast 2 + 3 4 \]

To evaluate the string:

1. scan string from left to right.
2. the first time we read a substring of the form “operator-operand-operand” (o-o-o), replace the three symbols with the one result of the indicated arithmetic calculation.
3. go back to step 1
Example: (from above)

<table>
<thead>
<tr>
<th>string</th>
<th>first o-o-o substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ * 2 3 4</td>
<td>* 2 3</td>
</tr>
<tr>
<td>+ 6 4</td>
<td>+ 6 4</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

- first tree yields:

<table>
<thead>
<tr>
<th>string</th>
<th>first o-o-o substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>* 2 + 3 4</td>
<td>+ 3 4</td>
</tr>
<tr>
<td>* 2 7</td>
<td>* 2 7</td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

- second tree yields:

Example: Consider the arithmetic expression:

\[ 3 + 4 \times 6 + 2 + 8 + 1 \times 5 + 9 \times 7 \]

There are many ways to evaluate this expression, one of which is as

\[ ((3 + 4) \times (6 + 2) + ((8 + 1) \times 5) + 9) \times 7 \]

This interpretation has

- derivation tree:
• prefix notation:

\[ * + + * + 3 4 + 6 2 * + 8 1 5 9 7 \]

• can evaluate prefix notation expression:

<table>
<thead>
<tr>
<th>string</th>
<th>first o-o-o substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ * + + * + 3 4 + 6 2 * + 8 1 5 9 7 ]</td>
<td>3 4</td>
</tr>
<tr>
<td>[ * + + * + 7 + 6 2 * + 8 1 5 9 7 ]</td>
<td>6 2</td>
</tr>
<tr>
<td>[ * + + * + 7 8 * + 8 1 5 9 7 ]</td>
<td>7 8</td>
</tr>
<tr>
<td>[ * + + 56 * + 8 1 5 9 7 ]</td>
<td>8 1</td>
</tr>
<tr>
<td>[ * + + 56 * + 9 5 9 7 ]</td>
<td>9 5</td>
</tr>
<tr>
<td>[ * + + 56 45 9 7 ]</td>
<td>56 45</td>
</tr>
<tr>
<td>[ * + 101 9 7 ]</td>
<td>101 9</td>
</tr>
<tr>
<td>[ * 110 7 ]</td>
<td>110 7</td>
</tr>
<tr>
<td>[ 770 ]</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**
terminals: \( a, b \)
nonterminals: \( S, A, B \)
productions:

\[
S \rightarrow AB \\
A \rightarrow a \\
B \rightarrow b
\]

Can produce word \( ab \) in two ways:

1. \( S \Rightarrow AB \Rightarrow aB \Rightarrow ab \)
2. \( S \Rightarrow AB \Rightarrow Ab \Rightarrow ab \)

However, both derivations have the same syntax tree:

```
  S
 / \  
A B   
|   |
a b
```
**Definition:** A CFG is *ambiguous* if for at least one word in its CFL there are two possible derivations of the word that correspond to two different syntax trees.

**Example:** PALINDROME

terminals: $a$, $b$

nonterminals: $S$

productions:

\[ S \rightarrow aSa \mid bSb \mid a \mid b \mid \Lambda \]

Can generate the word $babbab$ as follows:

\[
S \Rightarrow bSb \\
\quad \Rightarrow baSab \\
\quad \Rightarrow babSbab \\
\quad \Rightarrow babbab
\]

which has derivation tree:

```
S
  /\   \\
 b S b
  /\  \\
 a S a
  /\  \\
 b S b
```

Can show that this CFG is *unambiguous*. 
Example:
terminals:  a, b
nonterminals:  S
productions:

\[ S \rightarrow aS \mid Sa \mid a \]

The word \( aa \) can be generated by two different trees:

```
S   S
/ \ / \
a S S a
|   |
a a
d```

Therefore, this CFG is ambiguous.

Example:  terminals:  a, b
nonterminals:  S
productions:

\[ S \rightarrow aS \mid a \]

The CFL for this CFG is the same as above.
The word \( aa \) can now be generated by only one tree:

```
S
/ \
a S
|   |
a```

Therefore, this CFG is unambiguous.
Example:
terminals: $a, b$
nonterminals: $S, X$
productions:

\[
S \rightarrow aS \mid aSb \mid X \\
X \rightarrow Xa \mid a
\]

The word $aa$ has two different derivations that correspond to different syntax trees:

1. $S \Rightarrow aS \Rightarrow aX \Rightarrow aa$

```
S
/ \a S
| X
| a
```

2. $S \Rightarrow X \Rightarrow Xa \Rightarrow aa$

```
S
| X
/ \X a
/ \ a
```

Thus, this CFG is ambiguous.
Definition: For a given CFG, the total language tree is the tree

- with root $S$,
- whose children are all the productions of $S$,
- whose second descendents are all the working strings that can be constructed by applying one production to the leftmost nonterminal in each of the children,
- and so on.

Example:
terminals: $a$, $b$
nonterminals: $S$, $X$
productions:

\[
S \rightarrow aX \mid Xa \mid aXBXa \\
X \rightarrow ba \mid ab
\]

This CFG has total language tree as follows:

```
S   
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
aba aab baa aba ababXa aabbXa 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
/ | \ 
ababbaa abababa aabbbaa aababa
```

The CFL is finite.
Example:
terminals: $a$, $b$
nonterminals: $S$, $X$
productions:

$$S \rightarrow aSb \mid aX$$
$$X \rightarrow bX \mid a$$

Total language tree:

```
S
  / \  /
 /   \ /  \
/     / \
|     |   \
aSb   aX
  |     |   |
  |     |   |
aaSbb aaXb abX aa
     |       |
     |       |
     |       |
    aaaSbbb aaaXbb aabXb aaab abbX aba
```

CFL is infinite.
Example: terminals: $a$
nonterminals: $S, X$
productions:

$$S \rightarrow X \mid a$$
$$X \rightarrow aX$$

Total language tree:

```
S
 / \
X   a
 |  |
aX  |
 | 
aaX
```

Tree is infinite, but CFL = \{a\}. 
Chapter 13

Grammatical Format

13.1 Regular Grammars

We previously saw that

- CFG’s can generate some regular languages.
- CFG’s can generate some nonregular languages.

We will see that

- all regular languages can be generated by CFG’s.
- some nonregular languages cannot be generated by CFG’s.

Can turn FA into a CFG as follows:
Example: \( L = \) all words ending in \( a \).
FA:

```
S-  A+  B
a
b
a
b
b a
b
a
```

Definition: The path development of a word processed on a machine:

- Start in starting state \( S \).
- For each state visited, print out the input letters used thus far and the current state.

The word \( ababba \) has following path development on the FA:

- \( S \)
- \( aA \)
- \( abB \)
- \( abaA \)
- \( ababB \)
- \( ababbB \)
- \( ababbaA \)
- \( ababba \)

Now we define the following productions:

\[
S \rightarrow aA \mid bB
\]
The CFG has a production

\[ X \rightarrow cY \]

if and only if in the FA, there is an arc from state \( X \) to state \( Y \) labeled with \( c \).

- The CFG has a production

\[ X \rightarrow \Lambda \]

if and only if state \( X \) in the FA is a final state.

Derivation of \( ababba \) using the CFG:

\[
\begin{align*}
S & \Rightarrow aA \\
& \Rightarrow abB \\
& \Rightarrow abaA \\
& \Rightarrow ababB \\
& \Rightarrow ababbB \\
& \Rightarrow ababbaA \\
& \Rightarrow ababba
\end{align*}
\]

There is a one-to-one correspondence between path developments on the FA and derivations in the CFG; i.e., we can use the pigeonhole principle.

The derivation of the word \( ababba \) using the CFG is exactly the same as the path development given above.

**Theorem 21** All regular languages are CFL’s.
Example:
FA:

```
productions:
S  →  aS | bA
A  →  aC | bB | Λ
B  →  aB | bC
C  →  aA | bB | Λ
```

Consider a CFG $G = (\Sigma, \Omega, R, S)$, where

- $\Sigma$ is the set of terminals
- $\Omega$ is the set of nonterminals, and $S \in \Omega$ is the starting nonterminal
- $R \subset \Omega \times (\Sigma + \Omega)^*$ is the set of productions, where a production $(N, U) \in R$ with $N \in \Omega$ and $U \in (\Sigma + \Omega)^*$ is written as $N \rightarrow U$

**Definition:** For a given CFG $G = (\Sigma, \Omega, R, S)$, $W$ is a *semiword* if $W \in \Sigma^*\Omega$; i.e., $W$ is a string of terminals (maybe none) concatenated with exactly one nonterminal (on the right).

**Example:** $aabaN$ is a semiword if $N$ is a nonterminal and $a$ and $b$ are terminals.
Definition: \( G = (\Sigma, \Omega, R, S) \) is a regular grammar if \((N, U) \in R\) implies \(U \in (\Sigma^*\Omega) + \Sigma^*\); i.e., each production has one of the following two forms:

1. nonterminal \( \rightarrow \) semiword
2. nonterminal \( \rightarrow \) word

where “word” \( \in \Sigma^* \) is a string of terminals, possibly \( \Lambda \).

**Theorem 22** If a CFG is a regular grammar, then the language generated by this CFG is regular.

**Proof.**

- will prove theorem by showing that there is a TG that accepts the language generated by the CFG.
- Suppose CFG is as follows:

\[
\begin{align*}
N_1 & \rightarrow w_1M_1 \\
N_2 & \rightarrow w_2M_2 \\
& \vdots \\
N_n & \rightarrow w_nM_n \\
N_{n+1} & \rightarrow w_{n+1} \\
N_{n+2} & \rightarrow w_{n+2} \\
& \vdots \\
N_{n+m} & \rightarrow w_{n+m}
\end{align*}
\]

where \( N_i \) and \( M_i \) are nonterminals (not necessarily distinct) and \( w_i \in \Sigma^* \) are strings of terminals.

- Thus, \( w_iM_i \) is a semiword.
- At least one of the \( N_i = S \). Assume that \( N_1 = S \).
- Create a state of the TG for each nonterminal \( N_i \) and for each nonterminal \( M_j \).
• Also create a state +.
• Make the state for nonterminal $S$ the initial state of the transition graph.
• Draw an arc labeled with $w_i$ from state $N_i$ to state $M_i$ if and only if there is a production $N_i \rightarrow w_i M_i$.
• Draw an arc labeled with $w_i$ from state $N_i$ to state + if and only if there is a production $N_i \rightarrow w_i$.
• Thus, we have created a TG.
• By considering the path developments of words accepted by the TG, we can show that there is a one-to-one correspondence between words accepted by TG and words in CFL.
• Thus, these are the same language.
• Kleene’s Theorem implies that the language has a regular expression.
• Thus, language is regular.

Remarks:

• all regular languages can be generated by some regular grammars (Theorem 21)
• all regular grammars generate some regular language.
• a regular language may have many CFG’s that generate it, where some of the CFG’s may not be regular grammars.

Example: CFG
productions:

\[ S \rightarrow aaS \mid abS \mid baS \mid bb \]

Corresponding TG:
Below is another CFG that is not a regular grammar for the same language:

\[
S \rightarrow AaS \mid AbS \mid bAS \mid bb \\
A \rightarrow a
\]
Example: CFG
productions:

\[
S \rightarrow aB \mid bA \mid abA \mid baB \mid \Lambda \\
A \rightarrow abaA \mid bb \\
B \rightarrow baA \mid ab
\]

Corresponding TG:
Example: CFG productions:

\[
S \rightarrow aB \mid bA \mid abA \mid baB \\
A \rightarrow abaA \mid bb \\
B \rightarrow baA \mid ab
\]

Corresponding TG (note that CFG does not generate \( \Lambda \)):
**Definition:** A production \((N, U) \in R\) is a \(\Lambda\)-production if \(U = \Lambda\), i.e., the production is \(N \rightarrow \Lambda\).

If CFG does not contain a \(\Lambda\)-production, then \(\Lambda \notin \text{CFL}\).

However, CFG may have \(\Lambda\)-production and \(\Lambda \notin \text{CFL}\).

**Example:** productions:

\[
\begin{align*}
S & \rightarrow aX \\
X & \rightarrow \Lambda
\end{align*}
\]

### 13.2 Chomsky Normal Form

#### 13.2.1 \(\Lambda\) Productions and Nullable Nonterminals

Recall we previously defined \(\Lambda\)-production:

\(N \rightarrow \Lambda\)

where \(N\) is some nonterminal.

Note that

- If some CFL contains the word \(\Lambda\), then the CFG must have a \(\Lambda\)-production.
- However, if a CFG has a \(\Lambda\)-production, then the CFL does not necessarily contain \(\Lambda\); e.g.,

\[
\begin{align*}
S & \rightarrow aX \\
X & \rightarrow \Lambda
\end{align*}
\]

which defines the CFL \(\{a\}\).

**Definition:** For a given CFG with \(\Omega\) as its set of nonterminals and \(\Sigma\) as its set of terminals, a working string \(W \in (\Sigma + \Omega)^*\) is any string of nonterminals and/or terminals that can be generated from the CFG starting from any nonterminal.
Example: CFG:

$$
S \rightarrow a \mid Xb \mid aYa \\
X \rightarrow Y \mid \Lambda \\
Y \rightarrow X \mid a
$$

Then in the derivation

$$
S \Rightarrow aYa \\
\Rightarrow aXa \\
\Rightarrow aa
$$

we have that $aYa$, $aXa$, and $aa$ are all working strings.

**Definition:** For a given CFG having a nonterminal $X$ and $W$ a possible working string, we use the notation

$$X \Rightarrow^* W$$

if there is some derivation in the CFG starting from $X$ that can result in the working string $W$.

**Example:** CFG:

$$
S \rightarrow a \mid Xb \mid aYa \\
X \rightarrow Y \mid \Lambda \\
Y \rightarrow X \mid a
$$

Since we have the following derivation

$$
S \Rightarrow aYa \\
\Rightarrow aXa \\
\Rightarrow aa
$$

we can write $S \Rightarrow aYa$ and $S \Rightarrow aXa$ and $S \Rightarrow aa$.

**Definition:** In a given CFG, a nonterminal $X$ is *nullable* if

1. There is a production $X \rightarrow \Lambda$, or
2. $X \Rightarrow \Lambda$; i.e., there is a derivation that starts at $X$ and leads to $\Lambda$:

$$X \Rightarrow \cdots \Rightarrow \Lambda$$

**Example:** CFG:

$$S \rightarrow a \mid Xb \mid aYa$$

$$X \rightarrow Y \mid \Lambda$$

$$Y \rightarrow X \mid a$$

has nullable nonterminals $X, Y$.

**Example:** CFG:

$$S \rightarrow X \mid XY \mid Z$$

$$X \rightarrow Z \mid \Lambda$$

$$Y \rightarrow Wa \mid a$$

$$Z \rightarrow WX \mid aZ \mid Zb$$

$$W \rightarrow XYZ \mid bXa \mid \Lambda$$

has nullable nonterminals $S, X, Z, W$.

**Definition:** For any language $L$, define the language $L_0$ as follows:

1. if $\Lambda \not\in L$, then $L_0$ is the entire language $L$, i.e., $L_0 = L$.
2. if $\Lambda \in L$, then $L_0$ is the language $L - \{\Lambda\}$; i.e., if we let $T = \{\Lambda\}$, then $L_0 = L \cap T'$, so $L_0$ is all words in $L$ except $\Lambda$.

**Theorem 23** If $L$ is a CFL generated by a CFG $G_1$ that includes $\Lambda$-productions, then there is another CFG $G_2$ with no $\Lambda$-productions that generates $L_0$.

**Basic Idea.**

- We give constructive algorithm to convert CFG $G_1$ with $\Lambda$-productions into equivalent CFG $G_2$ with no $\Lambda$-productions:

  1. Delete all $\Lambda$-productions.
2. For each production
\[ X \rightarrow \text{something} \]
with at least one nullable nonterminal on the right-hand side, do the following for each possible nonempty subset of nullable nonterminals on the RHS:
(a) create a new production
\[ X \rightarrow \text{new something} \]
where the new RHS is the same as the old RHS except with the entire current subset of nullable nonterminals removed.
(b) do not create the production
\[ X \rightarrow \Lambda \]

**Example:** CFG \( G_1 \)

\[
\begin{align*}
S & \rightarrow a \mid Xb \mid aYa \\
X & \rightarrow Y \mid \Lambda \\
Y & \rightarrow X \mid a
\end{align*}
\]

has nullable nonterminals \( X, Y \).

We create new productions:

<table>
<thead>
<tr>
<th>Original Production</th>
<th>New Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow Xb )</td>
<td>( S \rightarrow b )</td>
</tr>
<tr>
<td>( S \rightarrow aYa )</td>
<td>( S \rightarrow aa )</td>
</tr>
<tr>
<td>( X \rightarrow Y )</td>
<td>Nothing</td>
</tr>
<tr>
<td>( Y \rightarrow X )</td>
<td>Nothing</td>
</tr>
</tbody>
</table>

New CFG \( G_2 \):

\[
\begin{align*}
S & \rightarrow a \mid Xb \mid aYa \mid b \mid aa \\
X & \rightarrow Y \\
Y & \rightarrow X \mid a
\end{align*}
\]
**Example:** CFG $G_1$

\[
\begin{align*}
S & \to X \mid XY \mid Z \\
X & \to Z \mid \Lambda \\
Y & \to Wa \mid a \\
Z & \to WX \mid aZ \mid Zb \\
W & \to XYZ \mid bXa \mid \Lambda
\end{align*}
\]

has nullable nonterminals $S$, $X$, $Z$, $W$.

We create new productions:

<table>
<thead>
<tr>
<th>Original Production</th>
<th>New Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \to X$</td>
<td>Nothing</td>
</tr>
<tr>
<td>$S \to XY$</td>
<td>$S \to Y$</td>
</tr>
<tr>
<td>$S \to Z$</td>
<td>Nothing</td>
</tr>
<tr>
<td>$X \to Z$</td>
<td>Nothing</td>
</tr>
<tr>
<td>$Y \to Wa$</td>
<td>$Y \to a$</td>
</tr>
<tr>
<td>$Z \to WX$</td>
<td>$Z \to W$ and $Z \to X$</td>
</tr>
<tr>
<td>$Z \to aZ$</td>
<td>$Z \to a$</td>
</tr>
<tr>
<td>$Z \to Zb$</td>
<td>$Z \to b$</td>
</tr>
<tr>
<td>$W \to XYZ$</td>
<td>$W \to YZ$, $W \to XY$, and $W \to Y$</td>
</tr>
<tr>
<td>$W \to bXa$</td>
<td>$W \to ba$</td>
</tr>
</tbody>
</table>

New CFG $G_2$:

\[
\begin{align*}
S & \to X \mid XY \mid Z \mid Y \\
X & \to Z \\
Y & \to Wa \mid a \\
Z & \to WX \mid aZ \mid Zb \mid W \mid X \mid a \mid b \\
W & \to XYZ \mid bXa \mid YZ \mid XY \mid Y \mid ba
\end{align*}
\]

- We need to show two things:
  1. all non-$\Lambda$ words generated using original CFG $G_1$ can be generated using new CFG $G_2$.
  2. all words generated using new CFG $G_2$ can be generated using original CFG $G_1$.  

First we show that all non-$\Lambda$ words generated using original CFG $G_1$ can be generated using new CFG $G_2$.

- Suppose our CFG $G_1$ included the productions $A \rightarrow bBb$ and $B \rightarrow \Lambda$.
- Suppose we had the following derivation of a word:

$$S \Rightarrow \ldots$$
$$\Rightarrow baAaAa$$
$$\Rightarrow babBbaAa \quad \text{from } A \rightarrow bBb$$
$$\Rightarrow \ldots$$
$$\Rightarrow babBbaabAa$$
$$\Rightarrow bbabbaabAa \quad \text{from } B \rightarrow \Lambda$$
$$\Rightarrow \ldots$$

There would have been no difference if we had applied the production $A \rightarrow bb$ rather than $A \rightarrow bBb$ in the third line.

- More generally, we can see that any non-$\Lambda$ word generated using original CFG $G_1$ can be generated using new CFG $G_2$.

Now show that all words generated using new CFG $G_2$ can be generated using original CFG $G_1$.

- Note that each new production is just a combination of old productions (e.g., $X \rightarrow aYa$ and $Y \rightarrow \Lambda$).
- Can show that any derivation using $G_2$ has a corresponding derivation using $G_1$ that possibly uses a $\Lambda$-production.
- Hence, all words generated using new CFG $G_2$ can be generated using original CFG $G_1$.

13.2.2 Unit Productions

**Definition:** A production $(N, U) \in R$ is a *unit production* if $U \in \Omega$; i.e., the production is of the form

$$\text{one nonterminal } \rightarrow \text{ one nonterminal}$$
Theorem 24 If a language $L$ is generated by a CFG $G_1$ that has no $\Lambda$-productions, then there is also a CFG $G_2$ for $L$ with no $\Lambda$-productions and no unit productions.

Basic Idea.

- Use the following rules to create new CFG:

- For each pair of nonterminals $A$ and $B$ such that there is a production $A \rightarrow B$

  or a chain of productions (unit derivation)

  $A \Rightarrow B$,  

introduce the following new productions:

  - if the non-unit productions from $B$ are

    $B \rightarrow s_1 | s_2 | \ldots | s_n$

    where the $s_i \in (\Sigma + \Omega)^*$ are strings of terminals and nonterminals, then create the new productions

    $A \rightarrow s_1 | s_2 | \ldots | s_n$

  - Do the same for all such pairs $A$ and $B$ simultaneously.

  - Remove all unit productions.

- Can show that $G_1$ and $G_2$ generate the same language.

Example: CFG $G_1$:

\[
\begin{align*}
S & \rightarrow X | Y | \text{bb} \\
X & \rightarrow Z | aXY \\
Y & \rightarrow Xa | a \\
Z & \rightarrow YX | S | Zb
\end{align*}
\]
has unit productions and unit derivations

\[
\begin{align*}
S & \rightarrow X \\
S & \rightarrow Y \\
X & \rightarrow Z \\
Z & \rightarrow S \\
S & \Rightarrow X \Rightarrow Z \\
X & \Rightarrow Z \Rightarrow S \\
Z & \Rightarrow S \Rightarrow X \\
Z & \Rightarrow S \Rightarrow Y \\
X & \Rightarrow Z \Rightarrow S \Rightarrow Y
\end{align*}
\]

We create new productions:

<table>
<thead>
<tr>
<th>Original Unit Production (Derivation)</th>
<th>New Productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S \rightarrow X)</td>
<td>(S \rightarrow aXY)</td>
</tr>
<tr>
<td>(S \rightarrow Y)</td>
<td>(S \rightarrow Xa \mid a)</td>
</tr>
<tr>
<td>(S \Rightarrow X \Rightarrow Z)</td>
<td>(S \rightarrow YX \mid Zb)</td>
</tr>
<tr>
<td>(X \Rightarrow Z)</td>
<td>(X \rightarrow YX \mid Zb)</td>
</tr>
<tr>
<td>(X \Rightarrow Z \Rightarrow S)</td>
<td>(X \rightarrow bb)</td>
</tr>
<tr>
<td>(X \Rightarrow Z \Rightarrow S \Rightarrow Y)</td>
<td>(X \rightarrow Xa \mid a)</td>
</tr>
<tr>
<td>(Z \Rightarrow S)</td>
<td>(Z \rightarrow bb)</td>
</tr>
<tr>
<td>(Z \Rightarrow S \Rightarrow X)</td>
<td>(Z \rightarrow aXY)</td>
</tr>
<tr>
<td>(Z \Rightarrow S \Rightarrow Y)</td>
<td>(Z \rightarrow Xa \mid a)</td>
</tr>
</tbody>
</table>

New CFG \(G_2\):

\[
\begin{align*}
S & \rightarrow bb \mid aXY \mid Xa \mid a \mid YX \mid Zb \\
X & \rightarrow aXY \mid YX \mid Zb \mid bb \mid Xa \mid a \\
Y & \rightarrow Xa \mid a \\
Z & \rightarrow YX \mid Zb \mid bb \mid aXY \mid Xa \mid a
\end{align*}
\]

**Theorem 25** Consider a CFG \(G_1 = (\Sigma, \Omega_1, R_1, S_1)\), which generates language \(L_1 = L(G_1)\). Then there exists another CFG \(G_2 = (\Sigma, \Omega_2, R_2, S_2)\) such that

- \(L(G_2) = L(G_1) - \{\Lambda\}\) and
- \((N, u) \in R_2\) implies \(u \in \Omega^+ + \Sigma;\)
i.e., $G_2$ generates all non-$\Lambda$ strings of $L_1$ and each production in $G_2$ is of one of two basic forms:

1. Nonterminal $\rightarrow$ string of only Nonterminals
2. Nonterminal $\rightarrow$ one terminal

Basic Idea. We will give a constructive proof:

- Assume that the nonterminals in the CFG $G_1$ are $S, X_1, X_2, \ldots, X_n$.
- Assume that the terminals in the CFG $G_1$ are $a$ and $b$.
- Introduce two new nonterminals $A$ and $B$.
- Introduce two new productions:
  
  $A \rightarrow a$
  $B \rightarrow b$

- For each original production involving terminals,
  
  - replace each $a$ with the nonterminal $A$
  - replace each $b$ with the nonterminal $B$

**Example:** Original production in $G_1$:  

$$X_5 \rightarrow X_1abaX_3bbX_2$$

becomes new production in $G_2$:  

$$X_5 \rightarrow X_1ABAAX_3BBX_2$$

which is string of only Nonterminals.

**Example:** Production in $G_1$:  

$$X_2 \rightarrow abaab$$

becomes new production in $G_2$:  

$$X_2 \rightarrow ABAAB$$

which is string of only nonterminals.
• So now all original productions have been transformed into new productions that have only nonterminals on the RHS.
• Also, we have two new productions for A and B.
• Note that any derivation starting from S to produce the word 
  \[\text{ababba}\]
  will now follow same sequence of (new) productions to derive the string 
  \[\text{ABABBA}\]
  starting from S.
• Then apply \(A \rightarrow a\) and \(B \rightarrow b\) the proper number of times to get the word ababba.
• Hence, any word generated by the original CFG \(G_1\) can be generated by the new CFG \(G_2\).
• Need to show that any word generated by the new CFG \(G_2\) can also be generated by the original CFG \(G_1\).
  
  ■ Consider new CFG \(G_2\) without the two productions \(A \rightarrow a\) and \(B \rightarrow b\).
  ■ Applying the new productions numerous times will result in a string of A’s and B’s.
  ■ Applying corresponding original productions in same order will result in the same string with a’s and b’s.
  ■ Then change string of A’s and B’s into a’s and b’s using \(A \rightarrow a\) and \(B \rightarrow b\).
  ■ Thus, every word generated using new CFG \(G_2\) can also be generated using original CFG \(G_1\).

Example: CFG \(G_1\):

\[
\begin{align*}
  S & \rightarrow abSba \mid bX_1aX_1 \mid X_2 \mid bb \\
  X_1 & \rightarrow aa \mid aSX_1b \\
  X_2 & \rightarrow X_1a \mid a
\end{align*}
\]
can be transformed into new CFG $G_2$:

\[
S \rightarrow ABSBA \mid BX_1AX_1 \mid X_2 \mid BB \\
X_1 \rightarrow AA \mid ASX_1B \\
X_2 \rightarrow X_1A \mid A \\
A \rightarrow a \\
B \rightarrow b
\]

### 13.2.3 Chomsky Normal Form

**Definition:** A CFG $G = (\Sigma, \Omega, R, S)$ is in *Chomsky Normal Form* (CNF) if $(N, U) \in R$ implies $U \in (\Omega \Omega) + \Sigma$; i.e., each of its productions has one of the two forms:

1. Nonterminal $\rightarrow$ string of exactly two Nonterminals
2. Nonterminal $\rightarrow$ one terminal

**Theorem 26** For any CFL $L$, the non-$\Lambda$ words of $L$ can be generated by a CFG in CNF.

**Basic Idea.** By construction:

- Let $L_0 = L$ if $\Lambda \notin L$, and $L_0 = L - \{\Lambda\}$ if $\Lambda \in L$.
- By Theorem 23, we know there is a CFG for $L_0$ that has no $\Lambda$-productions.
- By Theorem 24, we know there is a CFG for $L_0$ that has no unit productions.
- By Theorem 25, we know there is a CFG for $L_0$ for which each of its productions are of one of two forms:
  1. Nonterminal $\rightarrow$ string of only nonterminals
  2. Nonterminal $\rightarrow$ one terminal
- So now assume that our CFG for $L_0$ has the above three properties.
• Do nothing to the productions of the form
  Nonterminal → one terminal

• For each production of the form
  Nonterminal → string of Nonterminals
we expand it into a collection of productions as follows:
  - Suppose we have the production
    \[ X_4 \rightarrow X_2X_5X_3X_2X_1 \]
  - Replace the production with the new productions
    \[
    \begin{align*}
    X_4 & \rightarrow X_2R_1 \\
    R_1 & \rightarrow X_5R_2 \\
    R_2 & \rightarrow X_3R_3 \\
    R_3 & \rightarrow X_2X_1
    \end{align*}
    \]
    where the \( R_i \) are new nonterminals.
  - For each transformation of original productions, introduce new nonterminals \( R_i \).

• This transformation creates a new CFG in CNF.

• Now we have to show that the language generated by the new CFG is the same as that generated by the original CFG.

• First show that any word that can be generated by original CFG can also be generated by new CFG:
  - In any derivation of a word using the original CFG, we just replace any production of the form
    \[ X_4 \rightarrow X_2X_5X_3X_2X_1 \]
    with the new productions
    \[
    \begin{align*}
    X_4 & \rightarrow X_2R_1 \\
    R_1 & \rightarrow X_5R_2 \\
    R_2 & \rightarrow X_3R_3 \\
    R_3 & \rightarrow X_2X_1
    \end{align*}
    \]
This gives us a derivation of the word using the new CFG.

- Now show that any word that can be generated by the new CFG can also be generated by the original CFG:
  - Note that the nonterminal $R_3$ is only used in the RHS of the production
    \[ R_2 \rightarrow X_3R_3 \]
  - Thus, that is the only way $R_3$ would arise.
  - Similarly, the nonterminal $R_2$ is only used in the RHS of the production
    \[ R_1 \rightarrow X_5R_2 \]
  - Thus, that is the only way $R_2$ would arise.
  - We can similarly show the same for all new nonterminals $R_i$
  - Thus, since we use different $R_i$’s in the expansion of each production, the new nonterminals $R_i$ cannot interact to create new words.

**Example:** CFG

\[
\begin{align*}
S & \rightarrow abSba | bX_1aX_2 | bb \\
X_1 & \rightarrow aa | aSX_1b \\
X_2 & \rightarrow X_1a | abb
\end{align*}
\]

can be transformed into new CFG

\[
\begin{align*}
S & \rightarrow ABSBA | BX_1AX_2 | BB \\
X_1 & \rightarrow AA | ASX_1B \\
X_2 & \rightarrow X_1A | ABB \\
A & \rightarrow a \\
B & \rightarrow b
\end{align*}
\]

which can then be transformed into a CFG in CNF:

\[
\begin{align*}
S & \rightarrow AR_1 \\
R_1 & \rightarrow BR_2
\end{align*}
\]
13.3 Leftmost Nonterminals and Derivations

**Definition:** The leftmost nonterminal (LMN) in a working string is the first nonterminal that we encounter when we scan the string from left to right.

**Example:** In the string $bbabXbaYSbXbY$, the LMN is $X$.

**Definition:** If a word $w$ is generated by a CFG by a certain derivation and at each step in the derivation, a rule of production is applied to the leftmost nonterminal in the working string, then this derivation is called a *leftmost derivation* (LMD).
Example: CFG:

\[
S \rightarrow baXaS \mid ab \\
X \rightarrow Xab \mid aa
\]

The following is a LMD:

\[
S \Rightarrow baXaS \\
\Rightarrow baXabaS \\
\Rightarrow baXababaS \\
\Rightarrow baaaaababaS \\
\Rightarrow baaaaababaab
\]
Example: CFG:

\[
S \rightarrow XY \\
X \rightarrow Yb \mid Xa \mid aa \mid YY \\
Y \rightarrow XbbX \mid ab
\]

The word \textit{abbaaabbabab} has the following derivation tree:

```
S
 /\ \\
 / \ \\
 / \ \\
 / \ \\
 / \ \\
/ \ \\
X _ Y _
 /\ /\ /\ /\ \\
Y b / | | / \\
 / \ X b b X \\
/ \ a b \\
X a / \\
/ \ Y Y \\
/ \ a a \\
/ \ a b a b
```

Note that if we walk around the tree starting down the left branch of the root with our left hand always touching the tree, then the order in which we first visit each nonterminal corresponds to the order in which the nonterminals are replaced in LMD.

This is true for any derivation in any CFG.

**Theorem 27** Any word that can be generated by a given CFG by some derivation also has a LMD.
Chapter 14

Pushdown Automata

14.1 Introduction

- Previously, we saw connection between
  1. Regular languages
  2. Finite automata

- We saw that certain languages generated by CFG’s could not be accepted by FA’s.

14.2 Pushdown Automata

- Now we will introduce new kind of machine: pushdown automaton (PDA).

- Will see connection between
  1. context-free languages
  2. pushdown automata

- Pushdown automata and FA’s share some features, but a PDA can have one extra key feature: STACK.
  - infinitely long INPUT TAPE on which input is written.
• INPUT TAPE is divided into cells, and each cell holds one input letter or a blank Δ.

| a | b | b | Δ | Δ | Δ | Δ | Δ | ... |

• Once blank Δ is encountered on INPUT TAPE, all of the following cells also contain Δ.

• Read TAPE one cell at a time, from left to right. Cannot go back.

• START, ACCEPT, and REJECT states.

• Once enter either ACCEPT or REJECT state, cannot ever leave.

• READ state to read input letter from INPUT TAPE.

• Also, have an infinitely tall PUSHDOWN STACK, which has last-in-first-out (LIFO) discipline.

• Always start with STACK empty.

• STACK can hold letters of STACK alphabet (which can be same as input alphabet) and blanks Δ.

• Once we encounter a Δ in stack, everything below it is also a Δ.

PUSH and POP states alter contents of STACK.

* PUSH adds something to the top of the STACK.
* POP takes off the thing on the top of the STACK.
Example: Convert FA to PDA

FA:

PDA:

For this example, no STACK.
Example: Convert FA to PDA

FA:

```
+ b
a a
a, b
```

PDA:

```
START
REJECT
READ
a
b
```

For this example, no STACK.
Example: PDA with STACK

Suppose we had the INPUT TAPE

\[ a \ a \ b \ b \ \Delta \ \Delta \ \ldots \]

Stack is initially empty:

\[
\begin{array}{c}
\Delta \\
\Delta \\
\vdots
\end{array}
\]

See what happens when we process it:
The language accepted by the PDA is

\[ \{a^n b^n : n = 0, 1, 2, \ldots \} \]

which is a nonregular language.

**Proof.** see pages 295–299 of text.

So, why can PDA’s accept certain nonregular languages?

- STACK is memory with unlimited capacity.
- FA’s only had fixed amount of memory built in.
14.3 Determinism and Nondeterminism

Definition: A PDA is *deterministic* if each input string can only be processed by the machine in one way.

Definition: A PDA is *nondeterministic* if there is some string that can be processed by the machine in more than one way.

A nondeterministic PDA

- may have more than one edge with the same label leading out of a certain READ state or POP state.
- may have more than one arc leaving the START state.

Both deterministic and nondeterministic PDAs

- may have no edge with a certain label leading out of a certain READ state or POP state.
- if we are in a READ or POP state and encounter a letter for which there is no out-edge from this state, the PDA crashes.

Remarks:

- For FA’s, nondeterminism does not increase power of machines.
- For PDA’s, nondeterminism does increase power of machines.
14.4 Examples

Example: Language PALINDROMEX, which consists of all words of the form

$$sX\text{reverse}(s)$$

where $$s$$ is any string generated by $$(a + b)^*$$. 

$$\text{PALINDROMEX} = \{X, aXa, bXb, aaXaa, abXba, baXab, bbXbb, \ldots\}$$

- Each word in PALINDROMEX has odd length and $$X$$ in middle.
- When processing word on PDA, first read letters from TAPE and PUSH letters onto STACK until read in $$X$$.
- Then POP letters off STACK, and check if they are the same as rest of input string on TAPE.

PDA:

- Input alphabet $$\Sigma = \{a, b, X\}$$
- Stack alphabet $$\Gamma = \{a, b\}$$
Example: Language ODDPALINDROME, which consists of all words over $\Sigma = \{a, b\}$ having odd length that are the same forwards and backwards.

$$ODDPALINDROME = \{a, b, aaa, aba, bab, bbb, aaaaa, \ldots\}$$

Remarks:

- For PALINDROMEX, easy to detect when at middle of word when reading TAPE since marked by $X$.
- For ODDPALINDROME, impossible to detect when at middle of word when reading TAPE.
- Need to use nondeterminism.
Example: Language $\text{EVENPALINDROME}$, which consists of all words over $\Sigma = \{a, b\}$ having even length that are the same forwards as backwards.

$$\text{EVENPALINDROME} = \{s \text{ reverse}(s) : s \text{ can be generated by } (a+b)^*\}$$

$$\ = \{\Lambda, aa, bb, aaaa, abba, baab, bbbb, aaaaaa, \ldots \}$$
Suppose we had the INPUT TAPE

| b | a | a | b | Δ | Δ | ⋮ |

Stack is initially empty:

Δ
Δ
⋮

See what happens when we process it:

<table>
<thead>
<tr>
<th>STATE</th>
<th>STACK</th>
<th>TAPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>Δ⋯</td>
<td>baabΔ⋯</td>
</tr>
<tr>
<td>READ₁</td>
<td>Δ⋯</td>
<td≯baabΔ⋯</td>
</tr>
<tr>
<td>PUSH b</td>
<td>bΔ⋯</td>
<td≯baabΔ⋯</td>
</tr>
<tr>
<td>READ₁</td>
<td>bΔ⋯</td>
<td≯ybabΔ⋯</td>
</tr>
<tr>
<td>PUSH a</td>
<td>abΔ⋯</td>
<td≯ybΔabΔ⋯</td>
</tr>
<tr>
<td>READ₁</td>
<td>abΔ⋯</td>
<td≯ybΔybΔ⋯</td>
</tr>
<tr>
<td>POP₁</td>
<td>bΔ⋯</td>
<td≯ybΔybΔ⋯</td>
</tr>
<tr>
<td>READ₂</td>
<td>bΔ⋯</td>
<td≯ybΔybΔ⋯</td>
</tr>
<tr>
<td>POP₂</td>
<td>Δ⋯</td>
<td≯ybΔybΔ⋯</td>
</tr>
<tr>
<td>READ₂</td>
<td>Δ⋯</td>
<td≯ybΔybΔΔ⋯</td>
</tr>
<tr>
<td>POP₃</td>
<td>Δ⋯</td>
<td≯ybΔybΔΔ⋯</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Δ⋯</td>
<td≯ybΔybΔΔ⋯</td>
</tr>
</tbody>
</table>

Alternatively, we could have processed it as follows:

<table>
<thead>
<tr>
<th>STATE</th>
<th>STACK</th>
<th>TAPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>Δ⋯</td>
<td>baabΔ⋯</td>
</tr>
<tr>
<td>READ₁</td>
<td>Δ⋯</td>
<td≯baabΔ⋯</td>
</tr>
<tr>
<td>PUSH b</td>
<td>bΔ⋯</td>
<td≯baabΔ⋯</td>
</tr>
<tr>
<td>READ₁</td>
<td>bΔ⋯</td>
<td≯ybabΔ⋯</td>
</tr>
<tr>
<td>POP₁</td>
<td≯ybΔ</td>
<td≯ybΔybΔ⋯</td>
</tr>
<tr>
<td>CRASH</td>
<td>Δ⋯</td>
<td≯ybΔybΔ⋯</td>
</tr>
</tbody>
</table>

This time the PDA crashes.

But since there is at least one way of processing the string baaaab which leads to an ACCEPT state, the string is accepted by the PDA.
14.5 Formal Definition of PDA and More Examples

**Definition:** A pushdown automaton (PDA) is a collection of eight things:

1. An alphabet $\Sigma$ of input letters.
2. An input TAPE (infinite in one direction), which initially contains the input string to be processed followed by an infinite number of blanks $\Delta$.
3. An alphabet $\Gamma$ of STACK characters.
4. A pushdown STACK (infinite in one direction), which initially contains all blanks $\Delta$.
5. One START state that has only out-edges, no in-edges. Can have more than one arc leaving the START state. There are no labels on arcs leaving the START state.
6. Halt states of two kinds:
   - (a) zero or more ACCEPT states
   - (b) zero or more REJECT states
   Each of which have in-edges but no out-edges.
7. Finitely many nonbranching PUSH states that introduce characters from $\Gamma$ onto the top of the STACK.
8. Finitely many branching states of two kinds:
   - (a) READ states, which read the next unused letter from TAPE and may have out-edges labeled with letters from $\Sigma$ or a blank $\Delta$. (There is no restriction on duplication of labels and no requirement that there be a label for each letter of $\Sigma$, or $\Delta$.)
   - (b) POP states, which read the top character of STACK and may have out-edges labeled with letters of $\Gamma$ and the blank character $\Delta$, with no restrictions.

Remarks:
• The definition for PDA allows for nondeterminism.

• If we want to consider a PDA that does not have nondeterminism, then we will call it a deterministic PDA.
Example: CFG:

\[ S \rightarrow S + S \mid S \ast S \mid 3 \]

- terminals: +, *, 3
- nonterminals: S

(Nondeterministic) PDA:

Process \( 3 \ast 3 + 3 \) on PDA, where we now erase input TAPE as we read in letters:
### 14.6 Some Properties of PDA

**Theorem 28** For every regular language $L$, there is some PDA that accepts it.
Note that PDA can reach ACCEPT state and still have non-blank letters on TAPE and/or STACK.

Example:

```
START

PUSH X

READ

REJECT

b

ACCEPT

a

START

PUSH S

PUSH X
```

**Theorem 29** Given any PDA, there is another PDA that accepts exactly the same language with the additional property that whenever a path leads to ACCEPT, the STACK and the TAPE contain only blanks.

**Proof.** Can convert above PDA into equivalent one below:
Chapter 15

CFG = PDA

15.1 Introduction

We will now see that the following are equivalent:

1. the set of all languages accepted by PDA’s
2. the set of all languages generated by CFG’s.

15.2 CFG ⊂ PDA

Theorem 30 Given a language $L$ generated by a particular CFG, there is a PDA that accepts exactly $L$.

Proof. By construction

• By Theorem 26, we can assume that the CFG is in CNF.
Example: CFG in CNF:

\[
\begin{align*}
S & \rightarrow AS \\
S & \rightarrow BC \\
B & \rightarrow AA \\
A & \rightarrow a \\
C & \rightarrow b
\end{align*}
\]

Propose following (nondeterministic) PDA for above CFG:

- STACK alphabet: $\Gamma = \{S, A, B, C\}$
- Input TAPE alphabet: $\Sigma = \{a, b\}$
• Consider following leftmost derivation of word $aaaab$:

$$
S \Rightarrow AS \\
\Rightarrow aS \\
\Rightarrow aAS \\
\Rightarrow aaS \\
\Rightarrow aaBC \\
\Rightarrow aaAAC \\
\Rightarrow aaaAC \\
\Rightarrow aaaaC \\
\Rightarrow aaaaab
$$

• Now process string $aaaab$ on PDA:
### Leftmost derivation

<table>
<thead>
<tr>
<th>Leftmost derivation</th>
<th>STATE</th>
<th>TAPE</th>
<th>STACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>aaaab</td>
<td>Δ</td>
<td>aaaaab</td>
</tr>
<tr>
<td>( S )</td>
<td>PUSH S</td>
<td>aaaab</td>
<td>S</td>
</tr>
<tr>
<td>POP (S)</td>
<td>aaaab</td>
<td>Δ</td>
<td>S</td>
</tr>
<tr>
<td>PUSH S</td>
<td>aaaab</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>( \Rightarrow AS )</td>
<td>PUSH A</td>
<td>aaaab</td>
<td>AS</td>
</tr>
<tr>
<td>POP (A)</td>
<td>aaaab</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>( \Rightarrow aS )</td>
<td>READ (_2)</td>
<td>(\delta)aaaab</td>
<td>S</td>
</tr>
<tr>
<td>POP (S)</td>
<td>(\delta)aaaab</td>
<td>Δ</td>
<td>S</td>
</tr>
<tr>
<td>PUSH S</td>
<td>(\delta)aaaab</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>( \Rightarrow aAS )</td>
<td>PUSH A</td>
<td>(\delta)aaaab</td>
<td>AS</td>
</tr>
<tr>
<td>POP (A)</td>
<td>(\delta)aaaab</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>( \Rightarrow aaS )</td>
<td>READ (_2)</td>
<td>(\delta)(\delta)aaaab</td>
<td>S</td>
</tr>
<tr>
<td>POP (S)</td>
<td>(\delta)(\delta)aaaab</td>
<td>Δ</td>
<td>S</td>
</tr>
<tr>
<td>PUSH C</td>
<td>(\delta)(\delta)aaaab</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>( \Rightarrow aaBC )</td>
<td>PUSH B</td>
<td>(\delta)(\delta)aaaab</td>
<td>BC</td>
</tr>
<tr>
<td>POP (B)</td>
<td>(\delta)(\delta)aaaab</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>PUSH A</td>
<td>(\delta)(\delta)aaaab</td>
<td>AC</td>
<td>AC</td>
</tr>
<tr>
<td>( \Rightarrow aaAAC )</td>
<td>PUSH A</td>
<td>(\delta)(\delta)aaaab</td>
<td>AAC</td>
</tr>
<tr>
<td>POP (A)</td>
<td>(\delta)(\delta)aaaab</td>
<td>AC</td>
<td>AC</td>
</tr>
<tr>
<td>( \Rightarrow aaaAC )</td>
<td>READ (_2)</td>
<td>(\delta)(\delta)(\delta)aaaab</td>
<td>AC</td>
</tr>
<tr>
<td>POP (A)</td>
<td>(\delta)(\delta)(\delta)aaaab</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>( \Rightarrow aaaaAC )</td>
<td>READ (_2)</td>
<td>(\delta)(\delta)(\delta)(\delta)aaaab</td>
<td>C</td>
</tr>
<tr>
<td>POP (C)</td>
<td>(\delta)(\delta)(\delta)(\delta)aaaab</td>
<td>Δ</td>
<td>Δ</td>
</tr>
<tr>
<td>( \Rightarrow aaaaab )</td>
<td>READ (_1)</td>
<td>(\delta)(\delta)(\delta)(\delta)(\delta)aaaab</td>
<td>Δ</td>
</tr>
<tr>
<td>POP (Δ)</td>
<td>(\delta)(\delta)(\delta)(\delta)(\delta)aaaab</td>
<td>Δ</td>
<td>Δ</td>
</tr>
<tr>
<td>READ (_3)</td>
<td>(\delta)(\delta)(\delta)(\delta)(\delta)aaaab</td>
<td>Δ</td>
<td>Δ</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>(\delta)(\delta)(\delta)(\delta)(\delta)aaaab</td>
<td>Δ</td>
<td>Δ</td>
</tr>
</tbody>
</table>

- Note that just before entering the POP state, the current working string in the LMD is the same as the cancelled letters on the TAPE concatenated with current contents of the STACK.

- Before the first time we enter POP,
  - working string = \( S \)
  - letters cancelled = none
  - string of nonterminals in STACK = \( S \)

- Just before entering POP for the last time,
  - working string = whole word
• letters cancelled = all
• string of nonterminals in STACK = Δ
Consider the following CFG in CNF:

\[
\begin{align*}
X_1 & \rightarrow X_2X_3 \\
X_1 & \rightarrow X_1X_3 \\
X_4 & \rightarrow X_2X_5 \\
& \vdots \\
X_2 & \rightarrow a \\
X_3 & \rightarrow a \\
X_4 & \rightarrow b \\
& \vdots 
\end{align*}
\]

where start symbol \( S = X_1 \).

- Terminals: \( a, b \)
- Nonterminals: \( X_1, X_2, \ldots, X_n \)
- Construction of PDA will correspond to leftmost derivation of words.
- PDA will have only one POP and will be nondeterministic.
- Begin constructing PDA by starting with
• For each production of the form

\[ X_i \rightarrow X_jX_k \]

we include this circuit from the POP back to itself:

![Diagram](image-url)
• For all productions of the form

\[ X_i \rightarrow b \]

we add the following circuit to the above POP:

\[ b \rightarrow \text{READ} \]

\[ X_i \rightarrow \text{POP} \]

• Finally, add the following to the above POP:

\[ \text{POP} \rightarrow \text{READ} \rightarrow \text{ACCEPT} \]
• Recall that languages that include the word \( \Lambda \) cannot be put into CNF.

  To take care of this, we need to add loop to the above POP when \( \Lambda \) is in the language:

  ![Diagram of S and POP]

  This last loop will kill nonterminal \( S \) without replacing it with anything.
Example: Let $L_0$ be the language of the following CFG in CNF:

$S \rightarrow AB$

$S \rightarrow SB$

$A \rightarrow CA$

$A \rightarrow a$

$B \rightarrow b$

$C \rightarrow b$

We now want a PDA for the language $L = L_0 + \{\Lambda\}$.

Propose following (nondeterministic) PDA for above CFG:

- STACK alphabet: $\Gamma = \{S, A, B, C\}$
CHAPTER 15. CFG = PDA

- Input TAPE alphabet: $\Sigma = \{a, b\}$

Consider following leftmost derivation of word $babb$:

$$
S \Rightarrow SB \\
\Rightarrow ABB \\
\Rightarrow CABB \\
\Rightarrow bABB \\
\Rightarrow baBB \\
\Rightarrow babB \\
\Rightarrow babb
$$

Now process string $babb$ on PDA:

<table>
<thead>
<tr>
<th>Leftmost derivation</th>
<th>STATE</th>
<th>TAPE</th>
<th>STACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>babb</td>
<td>$\Delta$</td>
<td></td>
</tr>
<tr>
<td>PUSH S</td>
<td>babb</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>POP (S)</td>
<td>babb</td>
<td>$\Delta$</td>
<td></td>
</tr>
<tr>
<td>PUSH B</td>
<td>babb</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>$S \Rightarrow SB$</td>
<td>PUSH S</td>
<td>babb</td>
<td>SB</td>
</tr>
<tr>
<td>POP (S)</td>
<td>babb</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>PUSH B</td>
<td>babb</td>
<td>BB</td>
<td></td>
</tr>
<tr>
<td>$\Rightarrow ABB$</td>
<td>PUSH A</td>
<td>babb</td>
<td>ABB</td>
</tr>
<tr>
<td>POP (A)</td>
<td>babb</td>
<td>BB</td>
<td></td>
</tr>
<tr>
<td>PUSH A</td>
<td>babb</td>
<td>ABB</td>
<td></td>
</tr>
<tr>
<td>$\Rightarrow CABB$</td>
<td>PUSH C</td>
<td>babb</td>
<td>CABB</td>
</tr>
<tr>
<td>POP (C)</td>
<td>babb</td>
<td>ABB</td>
<td></td>
</tr>
<tr>
<td>$\Rightarrow bABB$</td>
<td>READ3</td>
<td>$\gamma abb$</td>
<td>ABB</td>
</tr>
<tr>
<td>POP (A)</td>
<td>$\gamma abb$</td>
<td>BB</td>
<td></td>
</tr>
<tr>
<td>$\Rightarrow baBB$</td>
<td>READ1</td>
<td>$\gamma a\gamma b$</td>
<td>BB</td>
</tr>
<tr>
<td>POP (B)</td>
<td>$\gamma a\gamma b$</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>$\Rightarrow babB$</td>
<td>READ2</td>
<td>$\gamma a\gamma b\gamma b$</td>
<td>B</td>
</tr>
<tr>
<td>POP (B)</td>
<td>$\gamma a\gamma b\gamma b$</td>
<td>$\Delta$</td>
<td></td>
</tr>
<tr>
<td>$\Rightarrow babb$</td>
<td>READ2</td>
<td>$\gamma a\gamma b\gamma b$</td>
<td>$\Delta$</td>
</tr>
<tr>
<td>POP ((\Delta))</td>
<td>$\gamma a\gamma b\gamma b$</td>
<td>$\Delta$</td>
<td></td>
</tr>
<tr>
<td>READ4</td>
<td>$\gamma a\gamma b\gamma b$</td>
<td>$\Delta$</td>
<td></td>
</tr>
<tr>
<td>ACCEPT</td>
<td>$\gamma a\gamma b\gamma b$</td>
<td>$\Delta$</td>
<td></td>
</tr>
</tbody>
</table>
15.3 PDA ⊂ CFG

**Theorem 31** Given a language $L$ that is accepted by a certain PDA, there exists a CFG that generates exactly $L$.

**Proof.** Strategy of proof:

1. Start with any PDA
2. Put the PDA into a standardized form, known as conversion form.
3. The purpose of putting a PDA in conversion form is that since the PDA now has a standardized form, we can easily convert the pictorial representation of the PDA into a table. This table will be known as a summary table. Number the rows in the summary table.
   - The summary table and the pictorial representation of the PDA will contain exactly the same amount of information. In other words, if you are only given a summary table, you could draw the PDA from it.
   - The correspondence between the pictorial representation of the PDA and the summary table is similar to the correspondence between a drawing of a finite automaton and a tabular representation of the FA.
4. Processing and accepting a string on the PDA will correspond to a particular sequence of rows from the summary table. But not every possible sequence of rows from the summary table will correspond to a processing of a string on the PDA. So we will come up with a way of determining if a particular sequence of rows from the summary table corresponds to a valid processing of a string on the PDA.
5. Then we will construct a CFG that will generate all valid sequences of rows from the summary table. We call the collection of all valid sequences of rows the row-language.
6. Convert this CFG for row-language into CFG that generates all words of a’s and b’s in original language of PDA.

We now begin by showing how to transform a given PDA into conversion form:
• first introduce new state HERE in PDA.
  • HERE state does not read TAPE nor push or pop the STACK.
  • HERE is just used as a marker.

**Definition:** A PDA is in *conversion form* if it meets all of the following conditions:

1. there is only one ACCEPT state.
2. there are no REJECT states.
3. Every READ or HERE is followed immediately by a POP.
4. POP’s must be separated by READ’s or HERE’s.
5. All branching occurs at READ or HERE states, none at POP states, and every edge has only one label.
6. The STACK is initially loaded with the symbol $ on top. If the symbol is ever popped in processing, it must be replaced immediately. The STACK is never popped beneath this symbol. Right before entering ACCEPT, this symbol is popped and left out.
7. The PDA must begin with the sequence:

\[
\text{START} \rightarrow \text{POP} \rightarrow \text{PUSH} \rightarrow \text{HERE or READ}
\]

8. The entire input string must be read before the machine can accept a word.
Note that we can convert any PDA into an equivalent PDA in conversion form as follows:

1. There is only one ACCEPT state:
   If there is more than one ACCEPT state, then delete all but one and have all the edges that formerly went into the others feed into the remaining one:

\[
\begin{array}{c}
\text{ACCEPT} \\
\rightarrow \text{ACCEPT} \\
\end{array}
\]

becomes

\[
\begin{array}{c}
\text{ACCEPT} \\
\end{array}
\]
2. There are no REJECT states:
   If there were previously any REJECT states in the original PDA, just delete them from the new PDA. This will just lead to a crash, which is equivalent to going to a REJECT state.

\[ \text{READ} \xrightarrow{b} \text{REJECT} \]

becomes

\[ \text{READ} \xrightarrow{a} \]

3. Every READ or HERE is followed immediately by a POP:

```
READ1   b
   a

READ2
```

becomes

```
READ1   b
   a

POP      a
  b
PUSH b

READ2
```

becomes (by property 5)

```
POP      a
  b
PUSH a

READ1   b
   a

POP      b
  b
PUSH b

READ2
```

```
POP      $     
  b
PUSH $    

READ1   b
   a

POP      $     
  b
PUSH $    
```
4. POP’s must be separated by READ’s or HERE’s:

\[
\text{POP}_1 \xrightarrow{b} \text{POP}_2
\]

becomes

\[
\text{POP}_1 \xrightarrow{b} \text{HERE} \xrightarrow{} \text{POP}_2
\]
5. All branching occurs at READ or HERE states, none at POP states, and every edge has only one label.

becomes

\[ \text{READ}_1 \xrightarrow{\ b\ } \text{POP} \xrightarrow{\ a\ } \text{READ}_2 \]

\[ \text{READ}_3 \xrightarrow{\ b\ } \text{POP} \xrightarrow{\ b\ } \text{READ}_3 \]
6. The STACK is initially loaded with the symbol $ on top. If the symbol is ever popped in processing, it must be replaced immediately. The STACK is never popped beneath this symbol. Right before entering ACCEPT, this symbol is popped and left out.

```
| $   |
| △   |
| △   |
```

7. The PDA must begin with the sequence:

```
START $\rightarrow$ POP $\rightarrow$ PUSH $\rightarrow$ HERE or READ
```

Simple.

8. The entire input string must be read before the machine can accept a word:

Use algorithm of Theorem 29.
Example: PDA for language \( \{a^{2n}b^n : n = 1, 2, 3, \ldots \} \):

PDA in conversion form:
Example: PDA for language \{ab\}:
PDA in conversion form:

From | To | READ what | POP what | PUSH what | Row number
---|---|---------|---------|----------|----------
START | READ₁ | $\Lambda$ | $\$$ | $\$$ | 1
READ₁ | READ₁ | $a$ | $\$$ | $a\$$ | 2
READ₁ | READ₁ | $a$ | $a$ | $aa$ | 3
READ₁ | READ₂ | $b$ | $a$ | $-$ | 4
READ₂ | ACCEPT | $\Delta$ | $\$$ | $-$ | 5
• Purpose of conversion form is to decompose machine into path segments, each of the form:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Reading</th>
<th>Popping</th>
<th>Pushing</th>
</tr>
</thead>
<tbody>
<tr>
<td>START or READ</td>
<td>READ or HERE or ACCEPT</td>
<td>One or no input letters</td>
<td>Exactly one STACK character</td>
<td>Any string onto the STACK</td>
</tr>
</tbody>
</table>

• The states START, READ, HERE, and ACCEPT are called joints.

• We can break up any PDA in conversion form into a collection of joint-to-joint segments.

• Each joint-to-joint segment has the following form:

1. It starts with a joint.
2. The first joint is immediately followed by exactly one POP.
3. The one POP is immediately followed by zero or more PUSHes.
4. The PUSHes are immediately followed by another JOINT.

• Summary table describes the entire PDA as list of all joint-to-joint segments:

<table>
<thead>
<tr>
<th>From where</th>
<th>To where</th>
<th>READ what</th>
<th>POP what</th>
<th>PUSH what</th>
<th>Row number</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>READ₁</td>
<td>Λ</td>
<td>$</td>
<td>$</td>
<td>1</td>
</tr>
<tr>
<td>READ₁</td>
<td>READ₁</td>
<td>a</td>
<td>$</td>
<td>a$</td>
<td>2</td>
</tr>
<tr>
<td>READ₁</td>
<td>READ₁</td>
<td>a</td>
<td>a</td>
<td>aa</td>
<td>3</td>
</tr>
<tr>
<td>READ₁</td>
<td>READ₂</td>
<td>b</td>
<td>a</td>
<td>–</td>
<td>4</td>
</tr>
<tr>
<td>READ₂</td>
<td>ACCEPT</td>
<td>Δ</td>
<td>$</td>
<td>–</td>
<td>5</td>
</tr>
</tbody>
</table>

• Consider processing string $ab$ on PDA:
• Every path through PDA corresponds to a sequence of rows of the summary table

• Not every sequence of rows of the summary table corresponds to a path through PDA.
  
  ■ Need to make sure joint consistent; i.e., last STATE of one row is same as first STATE of next row in sequence.
  
  ■ Need to make sure STACK consistent; i.e., when a row pops a character, it should be at the top of the STACK.

• Define row-language of PDA represented by a summary table:
  
  ■ Alphabet letters:

  \[ \Sigma = \{ \text{Row}_1, \text{Row}_2, \ldots, \text{Row}_5 \} \]

  i.e., terminals

  ■ All valid words are sequences of alphabet letters that correspond to paths from START to ACCEPT that are joint consistent and STACK consistent.

  ■ All valid words begin with Row_1 and end with Row_5.

  ■ The string

  \[ \text{Row}_1 \text{Row}_4 \text{Row}_3 \text{Row}_3 \]

  is not a valid word
• Does not end with Row_5
• not joint consistent since Row_4 ends in state READ_2, and Row_3 begins in state READ_1
• not STACK consistent since Row_1 ends with $ on the top of the STACK, and Row_4 tries to pop a from the top of the STACK

• We will develop a CFG for row-language and then transform it into another CFG for the original language accepted by the PDA.

• Recall the strategy of our proof:
  1. Start with any PDA
  2. Redraw PDA in conversion form.
  3. Build summary table and number the rows.
  4. Define row-language to be set of all sequences of rows that correspond to paths through PDA. Make sure STACK consistent.
  5. Determine a CFG that generates all words in row-language.
  6. Convert this CFG for row-language into CFG that generates all words of a’s and b’s in original language of PDA.

• We are now up to Step 5.

• So for Step 5, we want to determine a CFG for the row-language.

• Define nonterminal $S$ to be used to start any derivation in row-language grammar.

• Nonterminals in the row-language grammar:
  $$Net(X, Y, Z)$$

where

- $X$ and $Y$ are specific joints (START, READ, HERE, ACCEPT)
- $Z$ is any character from stack alphabet $\Gamma$.
- Interpretation: There is some path going from joint $X$ to joint $Y$ (possibly going through other joints) that has the net effect on the STACK of removing the symbol $Z$ from top of STACK.
- STACK is never popped below the initial $Z$ on the top, but may be built up along the path, and eventually ends with the $Z$ popped.
Example:

```
READ₁
  a
  Z
POP    PUSH b    PUSH a   POP   POP
           a
```

has net effect of popping $Z$, and is a $\text{Net}(\text{READ₁}, \text{READ₂}, Z)$.

Example:

```
READ₁
  a
POP   Z   POP   a
   a
PUSH a
```

does not have net effect of popping $Z$ since STACK went below the initial $Z$. Hence, this is not a $\text{Net}(\text{READ₁}, \text{READ₂}, Z)$. 
• Productions in the CFG for row-language will typically have
  • a nonterminal $Net(\cdot, \cdot, \cdot)$ on the LHS
  • and on the RHS, there will be a terminal $Row_i$ followed by zero or more nonterminals $Net(\cdot, \cdot, \cdot)$.

• The LHS and RHS of each production will have the same net effect on the STACK.

• Recall that the summary table for our example is

<table>
<thead>
<tr>
<th>From where</th>
<th>To where</th>
<th>READ what</th>
<th>POP what</th>
<th>PUSH what</th>
<th>Row number</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>READ$_1$</td>
<td>$\Lambda$</td>
<td>$$</td>
<td>$$</td>
<td>1</td>
</tr>
<tr>
<td>READ$_1$</td>
<td>READ$_1$</td>
<td>$a$</td>
<td>$$</td>
<td>$a$</td>
<td>2</td>
</tr>
<tr>
<td>READ$_1$</td>
<td>READ$_1$</td>
<td>$a$</td>
<td>$a$</td>
<td>$aa$</td>
<td>3</td>
</tr>
<tr>
<td>READ$_1$</td>
<td>READ$_2$</td>
<td>$b$</td>
<td>$a$</td>
<td>$-$</td>
<td>4</td>
</tr>
<tr>
<td>READ$_2$</td>
<td>ACCEPT</td>
<td>$\Delta$</td>
<td>$$</td>
<td>$-$</td>
<td>5</td>
</tr>
</tbody>
</table>

**Example:** Production:

$$Net(\text{READ}_1, \text{READ}_2, a) \rightarrow \text{Row}_4$$

**Example:** Production:

$$Net(\text{READ}_1, \text{ACCEPT}, \$)$$

$$\rightarrow \text{Row}_2 \ Net(\text{READ}_1, \text{READ}_2, a) \ Net(\text{READ}_2, \text{ACCEPT}, \$)$$
In last example, note that

- Row 2 POPs the $ off the stack, then PUSHes $ and then a, and ends in state READ₁.

- Then, Net(READ₁, READ₂, a) starts in state READ₁, has the net effect of POPping the a off the top of the STACK, and ends in state READ₂.

- Then, Net(READ₂, ACCEPT, $) starts in state READ₂, has the net effect of POPping the $ off the top of the STACK, and ends in state ACCEPT.

- The above three steps can be summarized by Net(READ₁, ACCEPT, $).

More generally, use following rules to create productions:

**Rule 1:** Create production

\[ S \rightarrow \text{Net(START, ACCEPT, $)} \]

**Rule 2:** For every row of summary table that has no PUSH entry, such as

<table>
<thead>
<tr>
<th>FROM</th>
<th>TO</th>
<th>READ</th>
<th>POP</th>
<th>PUSH</th>
<th>ROW</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>anything</td>
<td>Z</td>
<td>–</td>
<td>i</td>
</tr>
</tbody>
</table>

we include the production:

\[ \text{Net}(X, Y, Z) \rightarrow \text{Row}_i \]
CHAPTER 15. CFG = PDA

**Rule 3:** For every row that pushes \( n \geq 1 \) characters onto the STACK, such as

<table>
<thead>
<tr>
<th>FROM</th>
<th>TO</th>
<th>READ</th>
<th>POP</th>
<th>PUSH</th>
<th>ROW</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X )</td>
<td>( Y )</td>
<td>anything</td>
<td>( Z )</td>
<td>( m_1 m_2 \cdots m_n )</td>
<td>( j )</td>
</tr>
</tbody>
</table>

for all sets of \( n \) READ, HERE, or ACCEPT states \( S_1, S_2, \ldots, S_n \), we create the productions:

\[ \text{Net}(X, S_n, Z) \rightarrow \text{Row}_j \text{Net}(Y, S_1, m_1) \text{Net}(S_1, S_2, m_2) \cdots \text{Net}(S_{n-1}, S_n, m_n) \]
Some productions generated may never be used in a derivation of a word. This is analogous to the following:

**Example:** CFG:

\[
\begin{align*}
S & \rightarrow X \mid Y \\
X & \rightarrow aX \\
Y & \rightarrow ab
\end{align*}
\]

Production \(S \rightarrow X\) doesn’t lead to a word.

- Applying Rule 1 gives
  \[
  \text{PROD 1} \quad S \rightarrow Net(\text{START}, \text{ACCEPT}, \$)
  \]

- Applying Rule 2 to Rows 4 and 5 gives
  \[
  \begin{align*}
  \text{PROD 2} & \quad Net(\text{READ}_1, \text{READ}_2, a) \rightarrow \text{Row}_4 \\
  \text{PROD 3} & \quad Net(\text{READ}_2, \text{ACCEPT}, \$) \rightarrow \text{Row}_5
  \end{align*}
  \]

- Applying Rule 3 to Row 1 gives
  \[
  \begin{align*}
  Net(\text{START}, S_1, \$) & \rightarrow \text{Row}_1 \quad Net(\text{READ}_1, S_1, \$)
  \end{align*}
  \]

where \(S_1\) can take on values \(\text{READ}_1, \text{READ}_2, \text{ACCEPT}\).

\[
\begin{align*}
\text{PROD 4} & \quad Net(\text{START}, \text{READ}_1, \$) \rightarrow \text{Row}_1 \quad Net(\text{READ}_1, \text{READ}_1, \$) \\
\text{PROD 5} & \quad Net(\text{START}, \text{READ}_2, \$) \rightarrow \text{Row}_1 \quad Net(\text{READ}_1, \text{READ}_2, \$) \\
\text{PROD 6} & \quad Net(\text{START}, \text{ACCEPT}, \$) \rightarrow \text{Row}_1 \quad Net(\text{READ}_1, \text{ACCEPT}, \$)
\end{align*}
\]
CHAPTER 15. CFG = PDA

• Applying Rule 3 to Row 2 gives

\[ Net(\text{READ}_1, S_2, \$) \rightarrow \text{Row}_2 \ Net(\text{READ}_1, S_1, a) \ Net(S_1, S_2, \$) \]

where \( S_2 \) can be any joint except START
and \( S_1 \) can be any joint except START or ACCEPT.

**PROD 7** \( Net(\text{READ}_1, \text{READ}_1, \$) \)
\[ \rightarrow \text{Row}_2 \ Net(\text{READ}_1, \text{READ}_1, a) \ Net(\text{READ}_1, \text{READ}_1, \$) \]

**PROD 8** \( Net(\text{READ}_1, \text{READ}_1, \$) \)
\[ \rightarrow \text{Row}_2 \ Net(\text{READ}_1, \text{READ}_2, a) \ Net(\text{READ}_2, \text{READ}_1, \$) \]

**PROD 9** \( Net(\text{READ}_1, \text{READ}_2, \$) \)
\[ \rightarrow \text{Row}_2 \ Net(\text{READ}_1, \text{READ}_1, a) \ Net(\text{READ}_1, \text{READ}_2, \$) \]

**PROD 10** \( Net(\text{READ}_1, \text{READ}_2, \$) \)
\[ \rightarrow \text{Row}_2 \ Net(\text{READ}_1, \text{READ}_2, a) \ Net(\text{READ}_2, \text{READ}_2, \$) \]

**PROD 11** \( Net(\text{READ}_1, \text{ACCEPT}, \$) \)
\[ \rightarrow \text{Row}_2 \ Net(\text{READ}_1, \text{READ}_1, a) \ Net(\text{READ}_1, \text{ACCEPT}, \$) \]

**PROD 12** \( Net(\text{READ}_1, \text{ACCEPT}, \$) \)
\[ \rightarrow \text{Row}_2 \ Net(\text{READ}_1, \text{READ}_2, a) \ Net(\text{READ}_2, \text{ACCEPT}, \$) \]
Applying Rule 3 to Row 3 gives

\[ \text{Net}(\text{READ}_1, S_2, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, S_1, a) \text{ Net}(S_1, S_2, a) \]

where \( S_2 \) can be any joint except START
and \( S_1 \) can be any joint except START or ACCEPT.

**PROD 13**

\[ \text{Net}(\text{READ}_1, \text{READ}_1, a) \rightarrow \text{Row}_3 \text{ Net}(\text{READ}_1, \text{READ}_1, a) \text{ Net}(\text{READ}_1, \text{READ}_1, a) \]

**PROD 14**

\[ \text{Net}(\text{READ}_1, \text{READ}_1, a) \rightarrow \text{Row}_3 \text{ Net}(\text{READ}_1, \text{READ}_2, a) \text{ Net}(\text{READ}_2, \text{READ}_1, a) \]

**PROD 15**

\[ \text{Net}(\text{READ}_1, \text{READ}_2, a) \rightarrow \text{Row}_3 \text{ Net}(\text{READ}_1, \text{READ}_1, a) \text{ Net}(\text{READ}_1, \text{READ}_2, a) \]

**PROD 16**

\[ \text{Net}(\text{READ}_1, \text{READ}_2, a) \rightarrow \text{Row}_3 \text{ Net}(\text{READ}_1, \text{READ}_2, a) \text{ Net}(\text{READ}_2, \text{READ}_2, a) \]

**PROD 17**

\[ \text{Net}(\text{READ}_1, \text{ACCEPT}, a) \rightarrow \text{Row}_3 \text{ Net}(\text{READ}_1, \text{READ}_1, a) \text{ Net}(\text{READ}_1, \text{ACCEPT}, a) \]

**PROD 18**

\[ \text{Net}(\text{READ}_1, \text{ACCEPT}, a) \rightarrow \text{Row}_3 \text{ Net}(\text{READ}_1, \text{READ}_2, a) \text{ Net}(\text{READ}_2, \text{ACCEPT}, a) \]

• Our CFG for the row-language has

  - 5 terminals:
    \( \text{Row}_1, \text{Row}_2, \ldots, \text{Row}_5 \)
  - 16 nonterminals:
    \( S, 9 \) of the form \( \text{Net}(\cdot, \cdot, \$) \), \( 6 \) of the form \( \text{Net}(\cdot, \cdot, a) \).
  - 18 productions:
    PROD 1, \ldots, PROD 18
Can derive word in row-language using left-most derivation:

\[ S \Rightarrow Net(START, ACCEPT, \$) \quad \text{PROD 1} \]
\[ \Rightarrow Row_1 Net(READ_1, ACCEPT, \$) \quad \text{PROD 6} \]
\[ \Rightarrow Row_1 Row_2 Net(READ_1, READ_2, a) Net(READ_2, ACCEPT, \$) \quad \text{PROD 12} \]
\[ \Rightarrow Row_1 Row_2 Row_4 Net(READ_2, ACCEPT, \$) \quad \text{PROD 2} \]
\[ \Rightarrow Row_1 Row_2 Row_4 Row_5 \quad \text{PROD 3} \]

Not all productions in CFG will be used in derivations of actual words.

Our CFG doesn’t generate words having \( a \)'s and \( b \)'s. It generates words using terminals \( Row_1, Row_2, \ldots, Row_5 \).

Need to transform this CFG into another CFG that has terminals \( a \) and \( b \).
To convert previous CFG for row-language into CFG for original language of a’s and b’s,

- Change the terminals Rowᵢ into nonterminals
- Add new terminals a, b.
- Also use Λ
- Create more productions as below:

**Rule 4:** For every row

<table>
<thead>
<tr>
<th>FROM</th>
<th>TO</th>
<th>READ</th>
<th>POP</th>
<th>PUSH</th>
<th>ROW</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>EFGH</td>
<td>i</td>
</tr>
</tbody>
</table>

create the production

Rowᵢ → C

- Applying Rule 4 gives

<table>
<thead>
<tr>
<th>PROD</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Row₁ → Λ</td>
</tr>
<tr>
<td>20</td>
<td>Row₂ → a</td>
</tr>
<tr>
<td>21</td>
<td>Row₃ → a</td>
</tr>
<tr>
<td>22</td>
<td>Row₄ → b</td>
</tr>
<tr>
<td>23</td>
<td>Row₅ → Δ</td>
</tr>
</tbody>
</table>

- We can continue with the previous derivation in the row-language grammar to get a word in the original language:

\[
S \Rightarrow Net(START, ACCEPT, \$) \quad \text{PROD 1} \\
\Rightarrow \cdots \\
\Rightarrow Row₁ \ \text{Row₂} \ \text{Row₄} \ \text{Row₅} \quad \text{PROD 3} \\
\Rightarrow \Lambda \ \text{Row₂} \ \text{Row₄} \ \text{Row₅} \quad \text{PROD 19} \\
\Rightarrow \Lambda \ a \ \text{Row₄} \ \text{Row₅} \quad \text{PROD 20} \\
\Rightarrow \Lambda \ a \ b \ \text{Row₅} \quad \text{PROD 22} \\
\Rightarrow \Lambda \ a \ b \ \Lambda \quad \text{PROD 23}
\]

giving us the word ab.

- The word ab can be accepted by the PDA in conversion form by following the path:

Row₁ Row₂ Row₄ Row₅
Chapter 17

Context-Free Languages

17.1 Closure Under Unions

We will now prove some properties of CFLs.

**Theorem 36** If $L_1$ and $L_2$ are CFLs, then their union $L_1 + L_2$ is a CFL.

**Proof.** By grammars.

- $L_1$ CFL implies that $L_1$ has a CFG, $\text{CFG}_1$, that generates it.
- Assume that the nonterminals in $\text{CFG}_1$ are $S, A, B, C, \ldots$.
- Change the nonterminals in $\text{CFG}_1$ to $S_1, A_1, B_1, C_1, \ldots$.
- Do not change the terminals in the $\text{CFG}_1$.
- $L_2$ CFL implies that $L_2$ has a CFG, $\text{CFG}_2$, that generates it.
- Assume that the nonterminals in $\text{CFG}_2$ are $S, A, B, C, \ldots$.
- Change the nonterminals in $\text{CFG}_2$ to $S_2, A_2, B_2, C_2, \ldots$.
- Do not change the terminals in the $\text{CFG}_2$.
- Now $\text{CFG}_1$ and $\text{CFG}_2$ have nonintersecting sets of nonterminals.
- We create a CFG for $L_1 + L_2$ as follows:
Include all of the nonterminals $S_1, A_1, B_1, C_1, \ldots$ and $S_2, A_2, B_2, C_2, \ldots$.

Include all of the productions from CFG$_1$ and CFG$_2$.

Create a new nonterminal $S$ and a production

$$S \rightarrow S_1 \mid S_2$$

To see that this new CFG generates $L_1 + L_2$,

- note that any word in language $L_i$, $i = 1, 2$, can be generated by first using the production $S \rightarrow S_i$
- also, since there is no overlap in the use of nonterminals in CFG$_1$ and CFG$_2$, once we start a derivation with the production $S \rightarrow S_1$, we can only use the productions originally in CFG$_1$ and cannot use any of the productions from CFG$_2$, and so we can only produce words in $L_1$.
- Similar situation occurs when we start a derivation with the production $S \rightarrow S_2$.

Example:

CFG$_1$ for $L_1$

- $S \rightarrow SS \mid AaAb \mid BBB \mid \Lambda$
- $A \rightarrow SaS \mid bBb \mid abba$
- $B \rightarrow SSS \mid baab$

CFG$_2$ for $L_2$

- $S \rightarrow aS \mid aAba \mid BbB \mid \Lambda$
- $A \rightarrow aSa \mid abab$
- $B \rightarrow BabaB \mid bb$

To construct CFG for $L_1 + L_2$

- transform CFG$_1$

  - $S_1 \rightarrow S_1S_1 \mid A_1aA_1b \mid B_1B_1B_1 \mid \Lambda$
  - $A_1 \rightarrow S_1aS_1 \mid bB_1b \mid abba$
  - $B_1 \rightarrow S_1S_1S_1 \mid baab$
• transform CFG

\[ S_2 \rightarrow aS_2 \mid aA_2ba \mid Bb_2B_2 \mid \Lambda \]
\[ A_2 \rightarrow aS_2a \mid abab \]
\[ B_2 \rightarrow B_2abaB_2 \mid bb \]

• construct CFG for \( L_1 + L_2 \):

\[ S \rightarrow S_1 \mid S_2 \]
\[ S_1 \rightarrow S_1S_1 \mid A_1aA_1b \mid B_1B_1B_1 \mid \Lambda \]
\[ A_1 \rightarrow S_1aS_1 \mid bB_1b \mid abba \]
\[ B_1 \rightarrow S_1S_1S_1 \mid baab \]
\[ S_2 \rightarrow aS_2 \mid aA_2ba \mid Bb_2B_2 \mid \Lambda \]
\[ A_2 \rightarrow aS_2a \mid abab \]
\[ B_2 \rightarrow B_2abaB_2 \mid bb \]

**Proof.** (of Theorem 36 by machines)

• Since \( L_1 \) is CFL, Theorem 30 implies that there exists some PDA, PDA\(_1\), that accepts \( L_1 \).

• Since \( L_2 \) is CFL, Theorem 30 implies that there exists some PDA, PDA\(_2\), that accepts \( L_2 \).

• Construct new PDA\(_3\) to accept \( L_1 + L_2 \) by combining PDA\(_1\) and PDA\(_2\) into one machine by coalescing START states of PDA\(_1\) and PDA\(_2\) into a single START state.

• Note that once we leave the START state of PDA\(_3\), we can never come back to the START state.

• Also, there is no way to cross over from PDA\(_1\) to PDA\(_2\).

• Hence, any word accepted by PDA\(_3\) must also be accepted by either PDA\(_1\) or PDA\(_2\).

• Also, it is obvious that any word accepted by either PDA\(_1\) or PDA\(_2\) will be accepted by PDA\(_3\).
Example:
PDA_1 for \( L_1 \):
CHAPTER 17. CONTEXT-FREE LANGUAGES

PDA3 for \( L_1 + L_2 \):

17.2 Closure Under Concatenations

Theorem 37 If \( L_1 \) and \( L_2 \) are CFLs, then \( L_1 L_2 \) is a CFL.

Proof. By grammars.

- \( L_1 \) CFL implies that \( L_1 \) has a CFG, \( \text{CFG}_1 \), that generates it.
- Assume that the nonterminals in \( \text{CFG}_1 \) are \( S, A, B, C, \ldots \).
- Change the nonterminals in \( \text{CFG}_1 \) to \( S_1, A_1, B_1, C_1, \ldots \).
- Do not change the terminals in the \( \text{CFG}_1 \).
- \( L_2 \) CFL implies that \( L_2 \) has a CFG, \( \text{CFG}_2 \), that generates it.
- Assume that the nonterminals in \( \text{CFG}_2 \) are \( S, A, B, C, \ldots \).
- Change the nonterminals in \( \text{CFG}_2 \) to \( S_2, A_2, B_2, C_2, \ldots \).
Do not change the terminals in the CFG$_2$.

Now CFG$_1$ and CFG$_2$ have nonintersecting sets of nonterminals.

We create a CFG for $L_1 L_2$ as follows:

- Include all of the nonterminals $S_1, A_1, B_1, C_1, \ldots$ and $S_2, A_2, B_2, C_2, \ldots$.
- Include all of the productions from CFG$_1$ and CFG$_2$.
- Create a new nonterminal $S$ and a production $S \to S_1 S_2$.

To see that this new CFG generates $L_1 L_2$,

- Obviously, we can generated any word in $L_1 L_2$ using our new CFG.
- also, since there is no overlap in the use of nonterminals in CFG$_1$ and CFG$_2$, once we start a derivation with the production $S \to S_1 S_2$, the $S_1$ part will generate a word from $L_1$ and the $S_2$ part will generate a word from $L_2$.
- hence, any word generated by the new CFG will be in $L_1 L_2$.

Example:

CFG$_1$ for $L_1$

\[
S \to SS \mid AaAb \mid BBB \mid \Lambda \\
A \to SaS \mid bBb \mid abba \\
B \to SSS \mid baab
\]

CFG$_2$ for $L_2$

\[
S \to aS \mid aAba \mid BbB \mid \Lambda \\
A \to aSa \mid abab \\
B \to BabaB \mid bb
\]

To construct CFG for $L_1 L_2$
• transform CFG$_1$

\[
\begin{align*}
S_1 & \rightarrow S_1S_1 \mid A_1aA_1b \mid B_1B_1B_1 \mid \Lambda \\
A_1 & \rightarrow S_1aS_1 \mid bB_1b \mid abba \\
B_1 & \rightarrow S_1S_1S_1 \mid baab
\end{align*}
\]

• transform CFG$_2$

\[
\begin{align*}
S_2 & \rightarrow aS_2 \mid aA_2ba \mid Bb_2B_2 \mid \Lambda \\
A_2 & \rightarrow aS_2a \mid abab \\
B_2 & \rightarrow B_2abaB_2 \mid bb
\end{align*}
\]

• construct CFG for $L_1L_2$:

\[
\begin{align*}
S & \rightarrow S_1S_2 \\
S_1 & \rightarrow S_1S_1 \mid A_1aA_1b \mid B_1B_1B_1 \mid \Lambda \\
A_1 & \rightarrow S_1aS_1 \mid bB_1b \mid abba \\
B_1 & \rightarrow S_1S_1S_1 \mid baab \\
S_2 & \rightarrow aS_2 \mid aA_2ba \mid Bb_2B_2 \mid \Lambda \\
A_2 & \rightarrow aS_2a \mid abab \\
B_2 & \rightarrow B_2abaB_2 \mid bb
\end{align*}
\]

Remarks:

• Difficult to prove Theorem 37 by machines.

• Cannot just combine PDA$_1$ and PDA$_2$ by removing the ACCEPT state of PDA$_1$ and replacing it with the START state of PDA$_2$.

• Problem is we can reach the ACCEPT state of PDA$_1$ while there are still unread characters on the input TAPE and there are still characters on the STACK.

• Thus, when we go to PDA$_2$, we may process the last part of the word in $L_1$ and the entire word in $L_2$ and incorrectly accept or reject the entire word.
17.3 Closure Under Kleene Star

Theorem 38 If $L$ is a CFL, then $L^*$ is a CFL.

Proof.

- Since $L$ is a CFL, by definition there is some CFG that generates $L$.
- Suppose CFG for $L$ has nonterminals $S, A, B, C, \ldots$.
- Change the nonterminal $S$ to $S_1$.
- We create a new CFG for $L^*$ as follows:
  - Include all the nonterminals $S_1, A, B, C, \ldots$ from the CFG for $L$.
  - Include all of the productions from the CFG for $L$.
  - Add the new nonterminal $S$ and the new production
    $$S \rightarrow S_1S | \Lambda$$
- We can repeat last production
  $$S \rightarrow S_1S \rightarrow S_1S_1S \rightarrow S_1S_1S_1S \rightarrow S_1S_1S_1S_1 \rightarrow S_1S_1S_1S_1$$
- Note that any word in $L^*$ can be generated by the new CFG.
- To show that any word generated by the new CFG is in $L^*$, note that each of the $S_1$ above generates a word in $L$.
- Also, there is no interaction between the different $S_1$’s.

Example: CFG for $L$:

\[
\begin{align*}
S & \rightarrow AaAb | BBB | \Lambda \\
A & \rightarrow SaS | bBb | abba \\
B & \rightarrow SSS | baab
\end{align*}
\]
CHAPTER 17. CONTEXT-FREE LANGUAGES

Convert CFG for \( L \):

\[
S_1 \rightarrow AaAb \mid BBB \mid \Lambda \\
A \rightarrow S_1aS_1 \mid bBb \mid abba \\
B \rightarrow S_1S_1 \mid baab
\]

New CFG for \( L^* \):

\[
S \rightarrow S_1S \mid \Lambda \\
S_1 \rightarrow AaAb \mid BBB \mid \Lambda \\
A \rightarrow S_1aS_1 \mid bBb \mid abba \\
B \rightarrow S_1S_1 \mid baab
\]

17.4 Intersections

- We now will give an example showing that the intersection of two CFLs may not be a CFL.
- To show this, we will need to assume that the language \( L_3 = \{a^n b^n a^n : n = 0, 1, 2, \ldots \} \) is a non-context-free language. This is shown in the textbook in Chapter 16. \( L_3 \) is the set of words with some number of \( a \)'s, followed by an equal number of \( b \)'s, and ending with the same number of \( a \)'s.

Example:

- Let \( L_1 \) be generated by the following CFG:

\[
S \rightarrow XY \\
X \rightarrow aXb \mid \Lambda \\
Y \rightarrow aY \mid \Lambda
\]

Thus, \( L_1 = \{a^n b^n a^m : n, m \geq 0\} \), which is the set of words that have a clump of \( a \)'s, followed by a clump of \( b \)'s, and ending with another clump of \( a \)'s, where the number of \( a \)'s at the beginning is the same as the number of \( a \)'s in the middle. The number of \( a \)'s at the end of the word is arbitrary, and does not have to equal the number of \( a \)'s and \( b \)'s that come before it.
Let $L_2$ be generated by the following CFG:

\[
S \rightarrow WZ \\
W \rightarrow aW | \Lambda \\
Z \rightarrow bZa | \Lambda
\]

Thus, $L_2 = \{a^i b^k a^k : i, k \geq 0\}$, which is the set of words that have a clump of $a$’s, followed by a clump of $b$’s, and ending with another clump of $a$’s, where the number of $b$’s in the middle is the same as the number of $a$’s at the end. The number of $a$’s at the beginning of the word is arbitrary, and does not have to equal the number of $b$’s and $a$’s that come after it.

Note that $L_1 \cap L_2 = L_3$, where $L_3 = \{a^n b^n a^n : n = 0, 1, 2, \ldots\}$, which is a non-context-free language.

However, sometimes the intersection of two CFLs is a CFL.

For example, suppose that $L_1$ and $L_2$ are regular languages. Then Theorem 21 implies that $L_1$ and $L_2$ are CFLs. Also, Theorem 12 implies that $L_1 \cap L_2$ is a regular language, and so $L_1 \cap L_2$ is also a CFL by Theorem 21. Thus, here is an example of 2 CFLs whose intersection is a CFL.

Thus, in general, we cannot say if the intersection of two CFLs is a CFL.

### 17.5 Complementation

If $L$ is a CFL, then $L'$ may or may not be a CFL.

We first show that the complement of a CFL may be a CFL:

- If $L$ is regular, then $L'$ is also regular by Theorem 11.
- Also, Theorem 21 implies that both $L$ and $L'$ are CFLs.

We now show that the complement of a CFL may not be a CFL by contradiction:
Suppose that it is always true that if $L$ is a CFL, then $L'$ is a CFL.

Suppose that $L_1$ and $L_2$ are CFLs.

Then by our assumption, we must have that $L_1'$ and $L_2'$ are CFLs.

Theorem 36 implies that $L_1' + L_2'$ is a CFL.

Then by our assumption, we must have that $(L_1' + L_2')'$ is a CFL.

But we know that $(L_1' + L_2')' = L_1 \cap L_2$ by DeMorgan’s Law.

However, we previously showed that the intersection of two CFLs is not always a CFL, which contradicts the previous two steps.

So our assumption that CFLs are always closed under complementation must not be true.

Thus, in general, we cannot say if the complement of a CFL is a CFL.
Chapter 18

Decidability for CFLs

18.1 Membership – The CYK Algorithm

We want to determine if a given string $x$ can be generated from a particular CFG $G$.

**Theorem 45** Let $L$ be a language generated by a CFG $G$ with alphabet $\Sigma$. Given a string $s \in \Sigma^*$, we can decide whether or not $s \in L$.

**Proof.** We will use a constructive algorithm known as the CYK algorithm, developed by Cocke, Younger and Kasami.

- First suppose $s = \Lambda$.
  - The proof of Theorem 23 gives an algorithm to find all of the nullable nonterminals in a CFG.
  - If the starting nonterminal $S$ is a nullable nonterminal, then $\Lambda \in L$.

- Now suppose $s \neq \Lambda$.
  - Assume that $\Lambda \not\in L$, so we can transform CFG $G$ into another CFG $G_1$ in Chomsky Normal Form by Theorem 26.
Let $s = s_1 s_2 \cdots s_n$ be a string of length $n \geq 1$, so $s_i$ is the $i$th letter of $s$.

Let $s_{ik} = s_i s_{i+1} \cdots s_k$, the substring of $s$ from the $i$th letter to the $k$th letter.

The algorithm will determine for each $i$ and $k$ with $0 < i \leq k \leq n$ and each nonterminal $X$ whether $X \Rightarrow s_{ik}$.

* We denote the answer to this question by $T[i, k, X]$.

First consider the case when $i = k$ so $s_{ik} = s_{ii} = s_i$, a one-character string.

* Then $T[i, k, X]$ is true if and only if the CFG $G_1$ includes the production

$$X \rightarrow s_i$$

Now suppose $i < k$, so that length($s_{ik}$) $\geq 2$.

* Then $T[i, k, X]$ is true if and only if

  * $G_1$ includes a production

$$X \rightarrow YZ$$

  * $s_{ik} = uv$, i.e., can split $s_{ik}$ into substrings $u$ and $v$ such that their concatenation gives $s_{ik}$.

  * $Y \Rightarrow u$

  * $Z \Rightarrow v$

* Formally, $T[i, k, Z]$ is true if and only if

  * $G_1$ includes a production

$$X \rightarrow YZ$$

  * there exists $j$ with $i \leq j < k$ such that

$$Y \Rightarrow s_{ij}$$

$$Z \Rightarrow s_{j+1,k}$$

* Thus, we get the following recurrence:

$$T[i, k, X] = \begin{cases} 
  \text{true} & \text{if } i = k \text{ and } G_1 \text{ has production } X \rightarrow s_{ii} \\
  \text{true} & \text{if } i < k \text{ and } G_1 \text{ has production } X \rightarrow YZ \text{ such that } \\
 & \exists j \text{ with } i \leq j < k \text{ and } T[i, j, Y] \text{ and } T[j+1, k, Z] \\
  \text{false} & \text{otherwise}
\end{cases}$$
Can solve recursion using *dynamic programming*.

* Store the values of $T$ in an array that is initialized to false everywhere.
* Need to go through the array in such an order that $T[i, j, Y]$ and $T[j + 1, k, Z]$ are evaluated before $T[i, k, X]$ for $i \leq j < k$.
* Can do this by going through the array for increasing values of $k$ and, subject to that, decreasing the values of $i$.

**CYK Algorithm**: to determine if $s \in L$, where $L$ is generated by CFG $G_1$ in Chomsky normal form.

/* initialization */

\[ n = \text{length}(s); \]

for every nonterminal $X$, do begin

\[ i = 1 \text{ to } n \]

\[ k = i \text{ to } n \]

\[ T[i, k, X] = \text{false}; \]

\[ i = 1 \text{ to } n \]

\[ \text{if } G_1 \text{ has production } X \rightarrow s_{ii}, \text{ then} \]

\[ T[i, i, X] = \text{true}; \]

end;

\[ k = 2 \text{ to } n \]

\[ i = k - 1 \text{ down to } 1 \]

\[ \text{for all productions in } G_1 \text{ of the form } X \rightarrow YZ \text{ do} \]

\[ j = i \text{ to } k - 1 \]

\[ \text{if } T[i, j, Y] \text{ and } T[j + 1, k, Z] \text{ then} \]

\[ T[i, k, X] = \text{true}; \]

$s \in L$ iff $T[1, n, S] = \text{true};$
Chapter 19

Turing Machines

19.1 Introduction

- Turing machines will be our ultimate model for computers, so they need output capabilities.
- But computers without output statements can tell us something.
- Consider the following program

1. READ X
2. IF X=1 THEN END
3. IF X=2 THEN DIVIDE X BY 0
4. IF X>2 THEN GOTO STATEMENT 4

- If we assume that the input is always a positive integer, then
  - if program terminates naturally, then we know X was 1.
  - if program terminates with error message saying there is an overflow (i.e., crashes), then we know X was 2.
  - if the program does not terminate, then we know X was greater than 2.

**Definition:** A *Turing machine* (TM) \( T = (\Sigma, \Theta, \eta, \Gamma, K, s, H, \Pi) \), where

1. An alphabet \( \Sigma \) of input letters, and assume that the blank \( \Delta \notin \Sigma \).
2. A Tape $\Theta$ divided into a sequence of numbered cells, each containing one character or a blank.

- The input word is presented to the machine on the tape with one letter per cell beginning in the leftmost cell, called cell $i$.
- The rest of the Tape is initially filled with blanks $\Delta$.
- The Tape is infinitely long in one direction.

\[
\begin{array}{cccccc}
\text{cell } i & \text{cell } ii & \text{cell } iii & \text{cell } iv & \text{cell } v & \ldots \\
\text{Tape Head}
\end{array}
\]

3. A Tape Head $\eta$ that can in one step read the contents of a cell on the Tape, replace it with some other character, and reposition itself to the next cell to the right or to the left of the one it has just read.

- At the start of the processing, the Tape Head always begins by reading the input in cell $i$.
- The Tape Head can never move left from cell $i$. If it is given orders to do so, the machine crashes.
- The location of the Tape Head is indicated as in the above picture.

4. An alphabet $\Gamma$ of characters that can be printed on the Tape $\Theta$ by the Tape Head $\eta$.

- Assume that $\Delta \notin \Gamma$, and we may have that $\Sigma \subset \Gamma$.
- The Tape Head may erase a cell, which corresponds to writing $\Delta$ in the cell.

5. A finite set $K$ of states including

- Exactly one START state $s \in K$ from which we begin execution (and which we may reenter during execution).
• $H \subset K$ is a set of HALT states, which cause execution to terminate when we enter any of them. There are zero or more HALT states.

• The other states have no function, only names such as $q_1, q_2, q_3, \ldots$ or $1, 2, 3, \ldots$.

6. A program $\Pi$, which is a finite set of rules that, on the basis of the state we are in and the letter the Tape Head has just read, tells us

(a) how to change states,
(b) what to print on the Tape,
(c) where to move the Tape Head.

The program

$$\Pi \subset K \times K \times (\Sigma + \Gamma + \{\Delta\}) \times (\Gamma + \{\Delta\}) \times \{L, R\},$$

with the restriction that

• if $(q_1, q_2, \ell, c, d) \in \Pi$ and $(q'_1, q'_2, \ell', c', d') \in \Pi$ with $q_1 = q'_1$ and $\ell = \ell'$, then $q_2 = q'_2$, $c = c'$ and $d = d'$;

• i.e., for any state $q_1$ and any character $\ell \in \Sigma + \Gamma + \{\Delta\}$, there is only one arc leaving state $q_1$ corresponding to reading character $\ell$ from the Tape.

• This restriction means that TMs are deterministic.

We depict the program as a collection of directed edges connecting the states. Each edge is labeled with the triplet of information:

(character, character, direction) $\in (\Sigma + \Gamma + \{\Delta\}) \times (\Gamma + \{\Delta\}) \times \{L, R\}$

where

• The first character (either $\Delta$ or from $\Sigma$ or $\Gamma$) is the character the Tape Head reads from the cell to which it is pointing.

  - From any state, there can be at most one arc leaving that state corresponding to $\Delta$ or any given letter of $\Sigma + \Gamma$;

  - i.e., there cannot be two arcs leaving a state both with the same first letter (i.e., a Turing machine is deterministic).

• The second character (either $\Delta$ or from $\Gamma$) is what the Tape Head prints in the cell before it leaves.
• The third component, the direction, tells the Tape Head whether to move one cell to the right, \( R \), or one cell to the left, \( L \).

Remarks:

• The above definition does not require that every state has an edge leaving it corresponding to each letter of \( \Sigma + \Gamma \).

• If we are in a state and read a letter for which there is no arc leaving that state corresponding to that letter, then the machine crashes. In this case, the machine terminates execution unsuccessfully.

• To terminate execution successfully, machine must be led to a HALT state. In this case, we say that the word on the input tape is accepted by the TM.

• If Tape Head is currently in cell \( i \) and the program tells the Tape Head to move left, then the machine crashes.

• Our definition of TM’s requires them to be deterministic. There are also non-deterministic TM’s. When we say just “TM”, then we mean our above definition, which means it is deterministic.

**Definition:** A string \( w \in \Sigma^* \) is accepted by a Turing machine if the following occurs: when \( w \) is loaded onto the Tape and the machine is run, the TM ends in a Halt state.

**Definition:** The language accepted by a Turing machine is the set of accepted strings \( w \in \Sigma^* \).
Example: Consider the following TM with input alphabet $\Sigma = \{a, b\}$ and tape alphabet $\Gamma = \{a, b\}$:

![Turing Machine Diagram]

and input tape containing input $aba$

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>ii</th>
<th>iii</th>
<th>iv</th>
<th>v</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
</tr>
</tbody>
</table>

- We start in state START 1 with the Tape Head reading cell i, and we denote this by

\[
\begin{array}{c}
1 \\
aba
\end{array}
\]

The number on top denotes the state we are currently in. The things below represent the current contents of the tape, with the letter about to be read underlined.

- After reading in $a$ in state 1, the TM then takes the top arc from state 1 to state 2, and so it prints $a$ into the contents of cell i and the Tape Head moves to the right to cell ii. We record this action by writing

\[
\begin{array}{c}
1 \\
aba
\end{array} \rightarrow \begin{array}{c}
2 \\
aba
\end{array}
\]
The tape now looks like

```
  i  ii  iii  iv  v  vi
  a  b  a  △  △  △
```

• Now we are in state 2, and the Tape Head is pointing to cell ii. Since cell ii contains $b$, we will take the arc from state 2 to state 3, print $b$ in cell ii, and move the Tape Head to the right to cell iii. We record this action by writing

```
1  aba → 2  aba → 3  aba
```

The tape now looks like

```
  i  ii  iii  iv  v  vi
  a  b  a  △  △  △
```

• Now we are in state 3, and the Tape Head is pointing to cell iii. Since cell iii contains $a$, we will take the arc labeled $(a, a, R)$ from state 3 back to state 3, print $a$ in cell iii, and move the Tape Head to the right to cell iv, which contains a blank $\triangle$. We record this action by writing

```
1  aba → 2  aba → 3  aba → 3  aba\triangle
```

The tape now looks like
Now we are in state 3, and the Tape Head is pointing to cell iv. Since cell iv contains ∆, we will take the arc labeled (Δ, Δ, R) from state 3 to state HALT 4, print Δ in cell iv, and move the Tape Head to the right to cell v, which contains a blank ∆. We record this action by writing

\[
aba \rightarrow aba \rightarrow aba \rightarrow aba \Delta \rightarrow \text{HALT}
\]

Since we reached a HALT state, the string on the input tape is accepted.

• Note that if an input string has a as its second letter, then the TM crashes, and so the string is not accepted.

• This TM accepts the language of all strings over the alphabet \( \Sigma = \{a, b\} \) whose second letter is \( b \).
Example: Consider the following TM with input alphabet $\Sigma = \{a, b\}$ and tape alphabet $\Gamma = \{a, b\}$:

- (\(\Delta, \Delta, R\))
  - (b,b,R)

```
START 1
```

- (a,a,R)
- (a,a,R)

```
HALT 3
```

- (b,b,R)

```
(b,b,R)
```

- (a,a,R)

```
HALT 3
```

- (b,b,R)

```
(b,b,R)
```

Consider processing the word \(baab\) on the TM

- Note that the first cell on the TAPE contains \(b\), and so upon reading this, the TM writes \(b\) in cell i, moves the tape head to the right to cell ii, and then the TM loops back to state 1,
- The second cell on the TAPE contains \(a\), and so upon reading this, the TM moves to state 2, writes \(a\) in cell ii, and moves the tape head to the right to cell iii.
- The third cell on the TAPE contains \(a\), and so upon reading this, the TM writes \(a\) in cell iii, moves the tape head to the right to cell iv, and moves to state 3, which is a HALT state
- The TM now halts, and so the string is accepted. Note that the input tape still has a letter \(b\) that has not been read.

Consider processing on the TM the word \(bba\).

- Note that each of the first two \(b\)'s results in the TM looping back to state 1 and moving the tape head to the right one cell.
- The third letter \(a\) makes the TM go to state 2 and moves the tape head to the right one cell.
• The fourth cell of the TAPE has a blank, and so the TM then crashes. Thus, $bba$ is not accepted.

• Consider processing on the TM the word $bab$.

  • Note that the first letter $b$ results in the TM looping back to state 1 and moving the tape head to the right one cell.
  • The tape head then reads the $a$ in the second cell, which causes the TM to move to state 2 and moves the tape head to the right one cell.
  • The tape head then reads the $b$ in the third cell, which causes the TM to move back to state 1 and moves the tape head to the right one cell.
  • The fourth cell of the TAPE has a blank, and so the TM returns to state 1, and the tape head moves one cell to the right.
  • All of the other cells on the TAPE are blank, and so the TM will keep looping back to state 1 forever.
  • Since the TM never reaches a HALT state, the string $bab$ is not accepted.

• In general, we can divide the set of all possible strings into three sets:
  1. Strings that contain the substring $aa$, which are accepted by the TM since the TM will reach a HALT state.
  2. Strings that do not contain substring $aa$ and that end in $a$. For these strings, the TM crashes, and so they are not accepted.
  3. Strings that do not contain substring $aa$ and that do not end in $a$. For these strings, the TM loops forever, and so they are not accepted.

  Note: The videotaped lecture contains an error about this point.

  • Let $S_1$ be the set of strings that do not contain the substring $aa$ and that do not end in $a$.
  • Let $S_2$ be the set of strings that do not contain the substring $aa$ and that end in $b$.
  • In the videotaped lecture, I said that $S_2$ is the set of strings for which the TM loops forever, but actually, $S_1$ is the set of strings for which the TM loops forever.
CHAPTER 19. TURING MACHINES

• Note that $S_1 \neq S_2$ since $\Lambda \in S_1$ but $\Lambda \notin S_2$.

• This TM accepts the language having regular expression $(a + b)^*aa(a + b)^*$.

Definition: Every Turing machine $T$ over the alphabet $\Sigma$ divides the set of input strings into three classes:

1. $\text{ACCEPT}(T)$ is the set of all strings $w \in \Sigma^*$ such that if the Tape initially contains $w$ and $T$ is then run, $T$ ends in a HALT state. This is the language accepted by $T$.

2. $\text{REJECT}(T)$ is the set of all strings $w \in \Sigma^*$ such that if the Tape initially contains $w$ and $T$ is then run, $T$ crashes (by either moving left from cell $i$ or by being in a state that has no exit edge labeled with the letter that the Tape Head is currently pointing to).

3. $\text{LOOP}(T)$ is the set of all strings $w \in \Sigma^*$ such that if the Tape initially contains $w$ and $T$ is then run, $T$ loops forever.

So for our last example,

• $\text{ACCEPT}(T) = \text{set of strings generated by the regular expression } (a + b)^*aa(a + b)^*$.

• $\text{REJECT}(T) = \text{strings in } \Sigma^* \text{ that do not contain the substring } aa \text{ and that end in } a$, where $\Sigma = \{a, b\}$.

• $\text{LOOP}(T) = \text{strings in } \Sigma^* \text{ that do not contain the substring } aa \text{ and that do not end in } a$, where $\Sigma = \{a, b\}$.
Example: Below is a TM for the language $L = \{a^n b^n a^n : n = 0, 1, 2, \ldots \}$:

$\Sigma = \{a, b\}, \Gamma = \{a, b, *\}$

We now examine why this TM accepts the language $L$ defined above.

**Step 1.** Presume that we are in state 1, and we are reading the first letter of what remains on the input.

- So initially, we are reading the first letter on the input tape, but as we progress, we may find ourselves back in state 1 reading the first letter of what remains on the tape.
- If we read a blank, then we go to HALT.
- If what we read is $a$, then change it to $*$, and move the tape head to the right.
- If we read anything else, we crash.
Step 2. In state 2, we skip over the rest of the $a$’s in the initial clump of $a$’s, looking for the first $b$.

- When we find the first $b$, we move to state 3.
- As long as we keep reading $b$’s, we keep returning to state 3.
- When we find the first $a$ in the second clump of $a$’s, we then go to state 4, and move the tape head to the left back to the last $b$ in the clump of $b$’s.
- We then change the last $b$ to an $a$, move the tape head to the right, and go to state 5. So now the number of $b$’s has reduced by 1, and the number of $a$’s in the second clump of $a$’s has increased by 1.
- We did all of these last few steps to find the last $b$ in the clump of $b$’s.

Step 3. Now we are in state 5 with the tape head pointing to the first $a$ in the second clump of $a$’s, and we want to find the last $a$ in the second clump of $a$’s.

- Each $a$ that we now read makes us return back to state 5, and move the tape head to the right.
- If we read $b$, then the machine crashes.
- When we finally encounter $\Delta$, then the tape contains no more characters to the right, and the TM goes to state 6.
- We then move to state 7 and then to state 8, and change the last two $a$’s to $\Delta$’s.
- Thus, the number of $a$’s in the second clump has decreased by 2. But in Step 2, we increased the number of $a$’s in the second clump by 1, and so now the number of $a$’s in the second clump has decreased by 1 since we started in Step 1.
- Recall that Step 2, we also reduced the number of $b$’s by 1.
- Recall that in Step 1, we changed the first $a$ in the first clump of $a$’s to $*$.

Step 4. Now we are in state 8 with the tape head pointing to the last $a$ currently in the second clump of $a$’s, and we want to get back to the first $a$ that is currently in the first clump of $a$’s.
• In state 8, as long as we keep reading a’s and b’s, we move the tape head to the left, and return back to state 8.
• Recall that in Step 1, we changed the first a in the first clump of a’s into *.
• So when the tape head finally reaches the rightmost * by moving left, the TM goes to state 1, and we move the tape head to the right, and we repeat our 4 steps.

19.2 Stupid TM Tricks

There are several tricks that one can do with Turing Machines:

1. Storing information in states; e.g., check if first letter of input string appears later in the string.
2. Multiple tracks on tape.
3. Checking off symbols on tape.
4. Inserting a character anywhere onto the input tape and shifting over the rest of the contents of the tape.

**Example:** TM to

- insert c at the beginning of Tape,
- shift entire original contents of Tape one cell to the right, and
- finish with Tape Head pointing at cell i.

\[ \Sigma = \{a, b\}, \Gamma = \{a, b, A, B, c\}. \]
The language of this TM is all strings over $\Sigma = \{a, b\}$ since starting the TM with any string in $\Sigma^*$ loaded on the Tape and running the TM will lead to the Halt state.

5. Can design TM to delete contents of specific cell and shift contents of all cells to the right one cell to the left.


7. Can convert any FA into a TM.

8. Can convert any PDA into a TM.

**Theorem 46** If $L$ is a regular language, then there exists a TM for $L$. 
Chapter 23

TM Languages

23.1 Recursively Enumerable Languages

Definition: A language $L$ over an alphabet $\Sigma$ is called *recursively enumerable* if there is a TM that accepts every word in $L$ and either rejects (crashes) or loops forever for every word in $L'$; i.e.,

\[
\begin{align*}
\text{accept}(T) &= L, \\
\text{reject}(T) + \text{loop}(T) &= L'.
\end{align*}
\]

In other words, the class of languages that are accepted by a TM is exactly those languages that are recursively enumerable.

Definition: A language $L$ over an alphabet $\Sigma$ is called *recursive* if there is a TM that accepts every word in $L$ and rejects every word in $L'$; i.e.,

\[
\begin{align*}
\text{accept}(T) &= L, \\
\text{reject}(T) &= L', \\
\text{loop}(T) &= \emptyset
\end{align*}
\]

23.2 Church-Turing Thesis

There is an effective procedure to solve a decision problem if and only if there is a Turing machine that halts for all input strings and solves the problem.
### 23.3 Encoding of Turing Machines

Can take any pictorial representation of a TM and represent it as two tables of information.

**Example:** For the following TM

\[
\begin{align*}
(Δ, Δ, R) \\
(b, b, R)
\end{align*}
\]

\[
\begin{array}{c}
\text{START 1} \\
\text{2} \\
\text{HALT 3}
\end{array}
\]

\[
(b, b, R)
\]

we can represent it as the following tables:

<table>
<thead>
<tr>
<th>State</th>
<th>Start?</th>
<th>Halt?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Read</th>
<th>Write</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Δ</td>
<td>Δ</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>b</td>
<td>b</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>a</td>
<td>a</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>b</td>
<td>b</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>a</td>
<td>a</td>
<td>R</td>
</tr>
</tbody>
</table>

**Remarks:**

- We can do this encoding for any TM. We call this an *encoded Turing machine.*
The encoding can be written as just a string of characters.

• For example, we can write the above encoding as

\[110200301\%11\Delta\Delta R11bbR12aaR21bbR23aaR\]

where we use the % to denote where the first table ends and the second one begins.

• The textbook converts the above string into a string of a’s and b’s, which we won’t do.

• Thus, we can represent any TM as a string of characters, which we can think of as a program.

• We can use the encoded TM as an input string to another TM, just as a C++ program is an input string to a C++ compiler, which itself is just a program.

• In particular, a copy of a program may be passed to itself as input.

• For our above example, the string

\[110200301\%11\Delta\Delta R11bbR12aaR21bbR23aaR\]

is rejected by the TM since it crashes on the first letter.

### 23.4 Non-Recursively Enumerable Language

**Theorem 64** Not all languages are recursively enumerable.

**Proof.**

• Let \( L_N \) be the set of strings \( w \) that are encoded TMs for which \( w \) is not accepted by its own TM.

• For example, the string \( 110200301\%11\Delta\Delta R11bbR12aaR21bbR23aaR \) is in \( L_N \) since it was not accepted by its own TM.

• We will prove by contradiction that \( L_N \) is not recursively enumerable.

• Suppose that \( L_N \) is recursively enumerable.
• Then there exists a TM $T_N$ for $L_N$.
• Let $P$ be the encoded TM of TM $T_N$.
• There are 2 possibilities: either TM $T_N$ accepts $P$ or TM $T_N$ doesn’t accept $P$.
• If $T_N$ accepts $P$,
  • then $P \not\in L_N$ since $L_N$ consists of strings $w$ that are encoded TMs such that $w$ is not accepted by its own TM.
  • But this is a contradiction since the TM $T_N$ is only supposed to accept those strings in $L_N$.
• If $T_N$ doesn’t accept $P$,
  • then $P \in L_N$.
  • But this is a contradiction since $T_N$ should accept $P$ since $P \in L_N$.
• Therefore, $L_N$ is not recursively enumerable.

23.5 Universal Turing Machine

Definition: A universal Turing machine (UTM) is a TM that can be fed as input a string composed of 2 parts:

1. The first is any encoded TM $P$, followed by a marker, say $\$. 
2. The second part is a string $w$ called the data.

The UTM reads the input, and then simulates $P$ with input $w$.

Theorem 65 UTMs exist.

Remarks about UTMs
The reason that UTMs are important is that they allow one to write programs; i.e., UTMs are programmable, just like real computers.

We don’t have to build a new Turing machine for each problem.

For a proof of Theorem 65, see pp. 554–557 of Cohen.

23.6 Halting Problem

Theorem 69 There is no TM that can accept any encoded TM \( P \) and any input string \( w \) for \( P \) and always decide correctly whether \( P \) halts on \( w \); i.e., the halting problem cannot be decided by a TM.

Basic Idea:

- Define halting function \( H(P, w) \), where
  - \( P \) is encoding of program (i.e., encoded Turing machine)
  - \( w \) is intended input for \( P \).
- Let \( H(P, w) = \text{yes} \) if \( P \) halts on input \( w \).
  Let \( H(P, w) = \text{no} \) if \( P \) does not halt on input \( w \).
- Assume that a program computing \( H(P, w) \) exists.
- Construct a program \( Q(P) \) with input \( P \):
  1. \( x = H(P, P) \)
  2. While \( x = \text{yes} \), goto step 2.
- Now run program \( Q \) with input \( P = Q \).
- Suppose \( Q(Q) \) halts. Then \( H(Q, Q) = \text{yes} \), but \( Q \) is stuck in infinite loop and so it doesn’t halt.
- Suppose that \( Q(Q) \) doesn’t halt. Then \( H(Q, Q) = \text{no} \), while in fact \( Q(Q) \) halts.
- Therefore \( H(P, w) \) cannot exist.
Proof. (of Theorem 69)

- Suppose there is a TM, call it \( H \), to solve the halting problem; i.e., \( H \) works as follows:
  - Recall that all TM’s take a TAPE loaded with an input string.
  - Our TM \( H \) takes as its input an encoded TM \( P \) and an input string \( w \) to be used with \( P \).
  - So we have to specify how \( P \) and \( w \) can be specified as an input string to \( H \).
  - We do this by taking \( P \) and first concatenating it with a special character, say \#, and then concatenating this with the input string \( w \). We use the \# to mark the end of the encoded TM and the beginning of the input string.
  - Thus, we now have a single long string \( P\#w \).
  - If we feed the string \( P\#w \) into \( H \), then
    - * if \( P \) halts on \( w \), then \( H \) prints “yes” somewhere on the TAPE.
    - * if \( P \) does not halt on \( w \), then \( H \) prints “no” somewhere on the TAPE.
    - * See p. 449 of the textbook to see how to print characters on the TAPE.

- Now suppose that we create another encoded TM \( Q \) that takes an encoded TM \( P \) as input and uses \( H \) as a subroutine as follows:
  - Since \( P \) is the input, the TAPE initially contains \( P \).
  - First modify the TAPE so that it now contains \( P\#P \). (See p. 449 of the textbook to see how this can be done.)
  - Then run \( H \) using input \( P\#P \).
    - if TM \( H \) prints “yes” on input \( P\#P \), then loop forever;
      - if TM \( H \) prints “no” on input \( P\#P \), then halt.

- Now run \( Q \) with input \( P = Q \).
- Suppose \( Q \) halts on input \( Q \).
This means that $H$ prints “no” on input $Q\#Q$.

- But this means that the encoded TM $Q$ does not halt on input $Q$, which is a contradiction.

- Suppose $Q$ does not halt on input $Q$.

  - This means that $H$ prints “yes” on input $Q\#Q$.
  - But this means that the encoded TM $Q$ halts on input $Q$, which again is a contradiction.

- Therefore, $H$ cannot exist.

23.7 Does TM Accept $\Lambda$?

**Theorem 70** There is no TM that can decide for every encoded TM $T$ whether or not $T$ accepts the word $\Lambda$; i.e., the blank-tape problem for TMs is undecidable.

**Proof.**

- We will prove this by contradiction.

- Suppose that there is a TM, call it $B$, that can decide for every encoded TM $T$ whether or not $T$ accepts the word $\Lambda$; i.e., whether $T$ halts when it starts with a blank tape.

  - Function $B(T)$, where $T$ is encoding of program (i.e., encoded Turing machine)
    - $B(T) = \text{yes}$ if $T$ halts on input $\Lambda$.
    - $B(T) = \text{no}$ if $T$ does not halt on input $\Lambda$.

- Define a new program $M(P, w)$, with input $P$ and $w$, where $P$ is any encoded Turing machine and $w$ is any input string:
First construct new program $P_w$ that starts with blank input tape and works as follows:

* First $P_w$ writes $w$ on the input tape.
* Then $P_w$ positions the tape head back to the beginning of the tape.
* Finally $P_w$ simulates program $P$ with $w$ on the input tape.

Call $B(P_w)$, and return $M(P, w) = B(P_w)$.

- Since we started $P_w$ with a blank tape, we can apply program $B$ to $P_w$ to see if it halts.
- Clearly, $P_w$ will halt on a blank tape if and only if $P$ halts on $w$.
  - If $P_w$ halts on blank tape (i.e., if $B(P_w) = \text{yes}$), then $P$ halts on $w$.
  - If $P_w$ does not halt on blank tape (i.e., if $B(P_w) = \text{no}$), then $P$ does not halt on $w$.
- Note that $M(P, w)$ solves the halting problem.
- But the halting problem is undecidable, and so $B$ cannot exist.

\section*{23.8 Does TM Accept Any Words?}

\textbf{Theorem 71} There is no TM that can decide for every encoded TM $T$ whether or not $T$ accepts any words at all; i.e., the emptiness problem for TMs is undecidable.

\textbf{Proof.}

- We will prove this by contradiction.
- Suppose that there is a TM, call it $N$, that can decide for every encoded TM $T$ whether or not $N$ accepts any words at all; i.e., whether the language $L$ of $T$ is $L \neq \emptyset$.
• Function $N(T)$, where $T$ is encoding of program (i.e., encoded Turing machine)
  - $N(T) = \text{yes}$ if $T$ accepts language $L \neq \emptyset$.
  - $N(T) = \text{no}$ if $T$ accepts language $L = \emptyset$.

• Define a new program $E(P)$, with input $P$, which is any encoded Turing machine:
  - First construct new program $P'$ that works as follows:
    * First $P'$ erases contents of input tape.
    * Then $P'$ positions the tape head back to the beginning of the tape.
    * Finally $P'$ simulates program $P$ with blank input tape.
  - Call $N(P')$, and return $E(P) = N(P')$

• Suppose $E(P) = \text{yes}$. Then
  - $N(P') = \text{yes}$.
  - This implies $P'$ accepts at least one word.
  - But since $P'$ always erases whatever is on tape and then simulates $P$, this means that $P$ accepts $\Lambda$.

• Suppose $E(P) = \text{no}$. Then
  - $N(P') = \text{no}$.
  - This implies $P'$ does not accept any words; i.e., the language of $P'$ is $\emptyset$.
  - But since $P'$ always erases whatever is on tape and then simulates $P$, this means that $P$ does not accept $\Lambda$.

• Therefore, $E(P)$ solves the blank-tape problem.

• But Theorem 70 says this is impossible, and so $N$ cannot exist.
Chapter 24

Review

24.1 Topics Covered

1. Languages
   - $\Sigma$ is alphabet with finite number of symbols.
   - Languages are sets of strings over $\Sigma$
   - For sets $S_1$ and $S_2$, can define
     - union $S_1 + S_2 = \{w : w \in S_1 \text{ or } w \in S_2\}$
     - intersection $S_1 \cap S_2 = \{w : w \in S_1 \text{ and } w \in S_2\}$
     - product $S_1S_2 = \{w = w_1w_2 : w_1 \in S_1, w_2 \in S_2\}$
     - subtraction $S_1 - S_2 = \{w : w \in S_1, w \notin S_2\}$
   - For any set $S$ of strings over an alphabet $\Sigma$, can define
     - Kleene closure $S^* = \{w = w_1w_2\cdots w_n : n \geq 0, w_i \in S \forall i = 1, 2, \ldots, n\}$
     - complement $S' = \{w \in \Sigma^* : w \notin S\}$
     - $S^{**} = S^*$

2. Regular expressions

3. FA = $(K, \Sigma, \pi, s, F)$, where
   - $K$ is finite set of states
   - $\Sigma$ is the alphabet
CHAPTER 24. REVIEW

• $\pi : K \times \Sigma \rightarrow K$ is the transition function
• $s$ is the initial state
• $F$ is the set of final states.

4. $TG = (K, \Sigma, \Pi, S, F),$ where

• $K$ is finite set of states
• $\Sigma$ is the alphabet
• $\Pi \subseteq K \times \Sigma \times K$ is the transition relation
• $S$ is the set of initial states
• $F$ is the set of final states.

5. Kleene’s Theorem

• Any language that can be defined by a
  ■ regular expression
  ■ FA
  ■ TG
can be defined by all three methods.
• Given FA for $L_1$ and $L_2$, can construct FA for
  ■ $L_1 + L_2$
  ■ $L_1L_2$
  ■ $L_1^*$
• Algorithm for generating regular expression from FA.
• Nondeterminism

6. FA with output

• Moore machine
• Mealy machine
• These are equivalent.

7. Regular languages
   If $L_1$ and $L_2$ are RL, then so are

• $L_1 + L_2$
8. Nonregular languages
  Pumping lemma.

9. Decidability
  - Can tell if two FA’s, $FA_1$ and $FA_2$, generate the same language by checking if either of the following accepts any words:
    - $FA'_1 \cap FA_2$
    - $FA_1 \cap FA'_2$
  - There are effective procedures to decide if
    - an FA accepts a finite or infinite language
    - a regular expression generates an infinite language
    - an FA has language $\emptyset$
    - a regular expression generates language $\emptyset$

10. CFG
  - CFG $G = (\Sigma, \Omega, R, S)$, where
    - $\Sigma$ is the finite set of terminals, i.e., the alphabet
    - $\Omega$ is finite set of nonterminals
    - $R \subseteq \Omega \times (\Sigma + \Omega)^*$ is finite set of productions, where $(N, U) \in R$
      is written $N \rightarrow U$.
    - $S \in \Omega$ is the starting nonterminal.
  - CFG used to generate languages.
  - If $L$ is regular, then $L$ is CFL.
  - Some, but not all, nonregular languages are CFL.
  - Trees
    - Can eliminate ambiguity in meaning by using tree to show how word was derived using CFG.

11. Grammatical Format
• **Definition:** A CFG $G$ is a *regular grammar* if every production $N \rightarrow U$ in $G$ has $U \in \Sigma^*\Omega + \Sigma^*$.

• If a CFG is a regular grammar, then the CFL is a regular language. (Can do this by converting regular grammar into TG.)

• Chomsky Normal Form: A CFG $G$ is in CNF if every production $N \rightarrow U$ in $G$ has $U \in \Omega\Omega + \Sigma$.

12. PDA
   Every regular language $L$ is accepted by some PDA.

13. CFG = PDA

14. CFL’s
   If $L_1$ and $L_2$ are CFLs, then so are
   
   • $L_1 + L_2$
   • $L_1L_2$
   • $L_1^*$

   However, CFLs are not closed under intersection or complements; i.e., there are examples of CFLs $L_1$ and $L_2$ such that $L_1 \cap L_2$ is not context-free, and there are examples of CFLs $L$ such that $L'$ is not context-free.

15. Decidability for CFLs
   
   • Membership is decidable for CFLs; i.e., for any CFG $G$ and string $w$, can decide if $G$ generates $w$ (using CYK algorithm).

16. Turing Machines
   Following problems are undecidable:
   
   • Halting problem
   • Whether arbitrary TM halts on a blank tape
   • Whether arbitrary TM accepts any words
   • Whether arbitrary TM accepts finite or infinite language.