Theory of Computation (ToC)
Automata Theory (AT)

Rupak R. Gupta (RRG)

Aaditya Joil (Jojo)

2025

Contents

Chapter 1

Fundamentals and Finite Automata

1.1 Terminologies

1.1.1 Symbol

A *symbol* is a user-defined entity. Some examples of symbols are a, b, c, 0, 1, 2, etc.

1.1.2 String

A *string* is defined as a sequence of symbols of a finite length.

A string is denoted as w while the length of a string is denoted as $|w|$. For example, if $w = 000111$ is a binary string then $|w| = |000111| = 6$.

The Empty String

The empty string is denoted as ϵ. it is defined as a string whose length is zero, i.e. $|\epsilon| \coloneqq 0$.

1.1.3 Alphabet

An *alphabet* is a finite set of symbols. An alphabet is denoted as Σ. For example, $\{0, 1\}$ is a binary alphabet while $\{a, b, c, \ldots, z, A, B, C, \ldots, Z\}$ is an alphabet set for the English language.

Powers of Sigma$^{\text{TM}}$

For an alphabet Σ, Σ^n denotes the set of all strings of length n.

For example, if $\Sigma = \{0, 1\}$, then:

$$\Sigma^0 = \{\epsilon\},$$
$$\Sigma^1 = \{0, 1\},$$
$$\Sigma^2 = \{00, 01, 10, 11\} \text{ and so on.}$$

Corollary 1. $|\Sigma^n| = |\Sigma|^n$, *where* $|\Sigma|$ *denotes the* cardinality *of the set* Σ.

The set of all possible strings for an alphabet Σ is denoted by Σ^*. It is defined as:

$$\Sigma^* := \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots = \bigcup_{i=0}^{\infty} \Sigma^i \tag{1.1}$$

1.1.4 Language

A *language* is a collection of strings. It is denoted as L. For example, $L = \{00, 01, 10, 11\}$ is a language comprising all binary strings of length 2.

E.g., if L denotes the language consisting of all binary strings that begin with a '0', then

$$L = \left\{ \begin{array}{ccccc} 0, & 01, & 010, & 011, & \dots \\ 00, & 001, & 0010, & 0011, & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} \right\}$$

.

1.2 Finite State Machines

A Finite State Machine (FSM) or Finite Automaton (FA) is the machine format of regular expressions. It has a very limited memory. It cannot store or count strings.

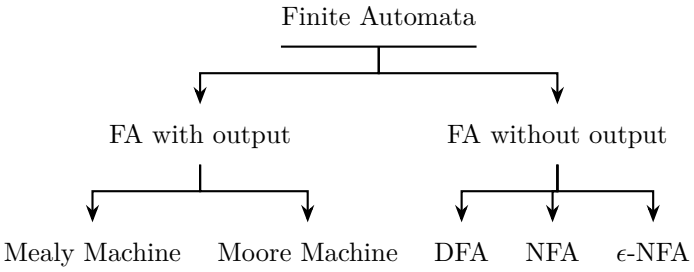


Figure 1.1: Classification of Finite Automata

1.2.1 State

A finite automaton, at any given point of time during it's working, is always in a *state*. This state is chosen from a set of states which is specified for a given automaton. Information is conveyed to the outside world through this state.

1.2.2 Transition

Upon encountering a specific input, a finite automaton may choose to move from it's current state to a new one, or choose to stay in the same state. This process of change of state is known as *transitioning* from one state to another.

The specific state transitions are usually specified in terms of a mathematical function called the **transition function**.

1.3 Deterministic Finite Automata

A Deterministic Finite Automaton (DFA) is the simplest model of computation.

Any DFA can be completely described by the following 5-tuple:

$$(Q, \Sigma, q_0, F, \delta) \tag{1.2}$$

where,

$$\begin{aligned} Q &= \text{Finite set of states} \\ \Sigma &= \text{Set of input symbols} \\ q_0 &= \text{Initial state} \\ F &= \text{Set of final states} \\ \delta &= \text{Transition function, i.e. } \delta : Q \times \Sigma \to Q \end{aligned}$$

For example, consider the following FSM:

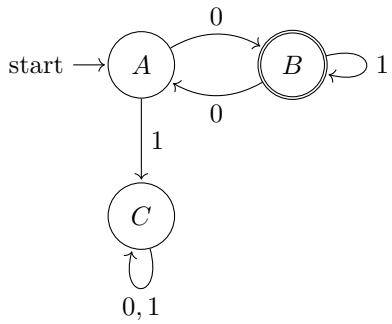


Figure 1.2: FSM

Here,

$$\begin{aligned} Q &= \{A, B, C\} \\ \Sigma &= \{0, 1\} \\ q_0 &= A \\ F &= \{B\} \end{aligned}$$

δ	0	1
A	B	C
B	A	B
C	C	C

$$\delta : \{A, B, C\} \times \{0, 1\} \to \{A, B, C\}$$

1.3.1 Acceptance of a String by a DFA

A string w is said to be accepted by a given DFA if the following two conditions are satisfied:

- The entire string is traversed completely from start to finish.
- The automaton goes to a final state after complete traversal of w.

1.3.2 Minimisation of DFA

A DFA always corresponds to a unique language but the reverse is not always true i.e. a single language may correspond to multiple different DFAs.

The running time and space requirements of the implementation of DFA depend on it's states. Hence, in order to reduce these requirements, we perform minimisation of the DFA.

There are two methods to perform minimisation of a DFA:

- K-Equivalency Method
- Myhill-Nerode Theorem (Table Filling Method)

1.3.3 K-equivalency Method of reducing DFA

K-equivalency

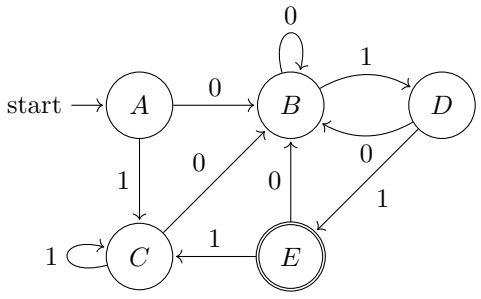
A pair of states (q_i, q_j) are said to be k-equivalent if $\delta(q_i, w)$ and $\delta(q_j, w)$ both produce final states or both produce non-final states for all strings $w \in \Sigma^*$ of length k or less.

Process of minimisation

Considering the DFA to be defined as in eq. (1.2)

1. All the states are 0-equivalent. Mark this partition of Q as S_0.
2. Now partition Q as follows: $\{Q - F, F\}$. This is the first equivalency S_1.
3. Apply all the inputs separately on the two subsets and find the next state combinations. If it happens that, for applying input on one set of states, the next states belong to different subsets, then separate the states which produce next states belonging to different subsets forming new partition.
4. Continue these steps $n + 1$ times, where n is the number of states of DFA i.e. $|Q|$.
5. If S_n and S_{n+1} are the same, then stop and declare S_n as an equivalent partition.
6. The equivalent partition will contain subsets whose contents on combining will form the new states of the minimised DFA

e.g. Minimise the following DFA:



We first find the table for the transition function for this automaton.

δ	0	1
$\rightarrow A$	B	C
B	B	D
C	B	C
D	B	E
E^*	B	C

Let us find the k-equivalencies of this DFA

$$0\textbf{-equivalency} = \{\{A, B, C, D, E\}\}$$
$$1\textbf{-equivalency} = \{\{A, B, C, D\}, \{E\}\}$$

Since A and D go to different subsets of the partition on the input 1, D is to be separated. $\delta(A, 1) \to C$ and $\delta(D, 1) \to E$.

$$2\textbf{-equivalency} = \{\{A, B, C\}, \{D\}, \{E\}\}$$

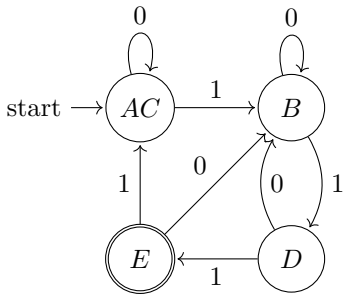
Since A and B now go to different subsets of the partition on the input 1, B is to be separated. $\delta(A, 1) \to C$ and $\delta(B, 1) \to D$.

$$3\textbf{-equivalency} = \{\{A, C\}, \{B\}, \{D\}, \{E\}\}$$

Now, no more changes in the partition occurs after this, so we can end the process early and say that S_3 is an equivalent partition.

To find the minimised DFA, we must first find the transition table and then convert it to a DFA. **(Ans.)**

δ	0	1
$\to AC$	B	AC
B	B	D
D	B	E
E^*	B	AC



1.3.4 Myhill-Nerode Theorem

Theorem 1. *A language L is regular if and only if the number of equivalence classes of the relation R is finite.*

Where R is the equivalence relation such that:

$$\forall x, y \in \Sigma^*, xRy \iff \forall z \in \Sigma^*, (xz \in L \iff yz \in L)$$

We now use this theorem in order to minimise a DFA.

e.g. Minimise the DFA from the previous example:

Let us construct a $|Q| \times |Q| = 5 \times 5$ matrix while only keeping entries below the principal diagonal.

Let us mark off the squares which correspond to any pair if the form $(Q \setminus F, F)$ with a $\times$.

Now that all such pairs have been marked with an $\times$, we can now go on to the next step:

- Find all pairs (α, β), which lead to a marked pair when the transition function δ is applied to the constituents.

- Mark these pairs with an $\times$ as well.
- Repeat until no new pairs are marked.

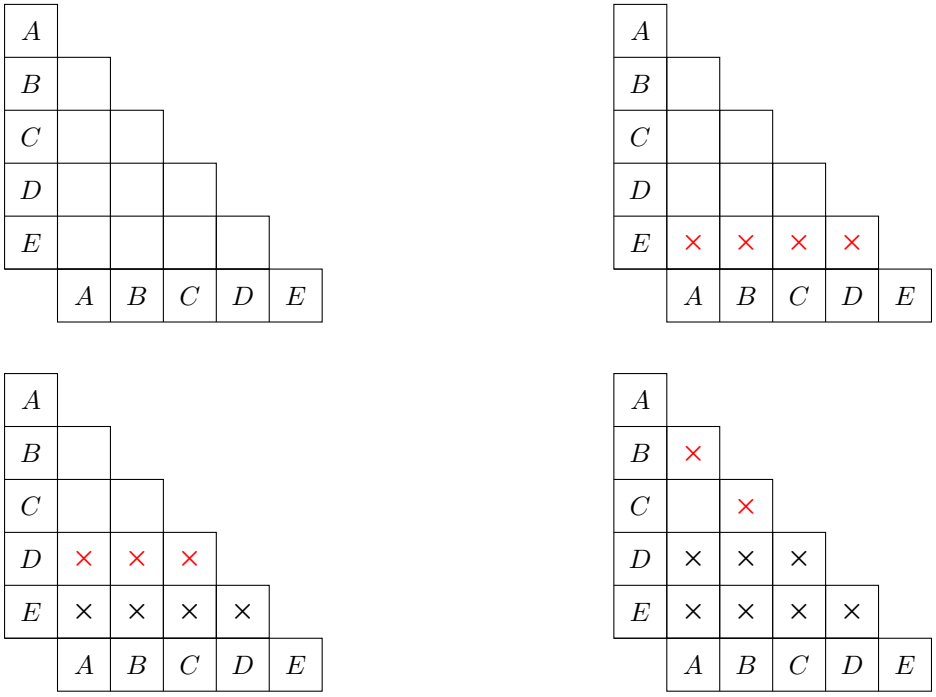


Figure 1.3: Table Filling Method

Now find the squares corresponding to the pairs which are not marked yet, the states corresponding to these squares are to be combined in a single state.

We get the set of states in the minimised DFA to be $\{AC, B, D, E\}$.

We now continue ahead in the same way as we did for the k-equivalency method. Finding the transition table for the system and then finding the minimised version of the DFA.

1.4 Regular Languages

A language L is said to be regular if and only if some FSM can recognize it. Regular Languages are the languages formed from "Regular Grammar".

1.4.1 Operations on Regular Languages

We have the following operations on regular languages:

Union $A \cup B \coloneqq \{x | x \in A \vee x \in B\}$.

Concatenation $A \circ B \coloneqq \{xy | x \in A \wedge y \in B\}$.

Kleene's Closure $A^* \coloneqq \{x_1 x_2 \dots x_k | k \geq 0 \wedge x \in A\}$.

For example, Let $A = \{rq, s\}$ and $B = \{mn, o\}$

$$\therefore A \cup B = \{rq, s, mn, o\}$$
$$\therefore A \circ B = \{rqmn, rqo, smn, so\}$$
$$\therefore A^* = \{\epsilon, rq, s, rqrq, rqs, srq, ss, \dots\}$$

Theorem 2. *The class of all regular languages is closed under* union*, i.e.*

$$\forall A \in L \ \forall B \in L \ A \cup B \in L$$

Theorem 3. *The class of all regular languages is closed under* concatenation*, i.e.*

$$\forall A \in L \ \forall B \in L \ A \circ B \in L$$

1.5 Non-Deterministic Finite Automata

Any Non-Deterministic Finite Automaton (NFA) can be completely described by the following 5-tuple:

$$(Q, \Sigma, q_0, F, \delta)$$

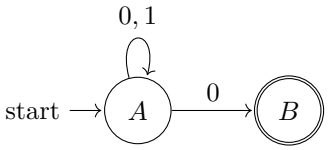
where,

$$\begin{aligned} Q &= \text{Set of states} \\ \Sigma &= \text{Set of inputs} \\ q_0 &= \text{Initial state} \\ F &= \text{Set of final states} \\ \delta &= \text{Transition function, i.e. } \delta : Q \times \Sigma \to \mathcal{P}(Q) \end{aligned}$$

The main difference between an NFA and a DFA is the fact that the next state from a given state and input is random. In a DFA, the next state for a given pair of current state and input symbol is clearly defined and is *deterministic* in nature.

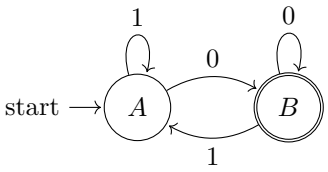
On the other hand, NFAs are *non-deterministic* by nature. By definition, there are a variety of states it can choose to go to from any given state and input.

e.g. Consider the below NFA. It accepts all strings which end with an 0. As we can see, there are two possible states where we can go to from the state A on the input of 0, A itself and B. We can also see that from B, there is no state to go to.



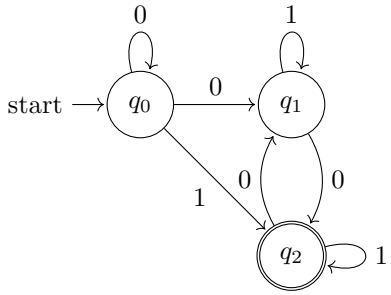
δ	0	1
A	$\{A, B\}$	A
B	ϕ	ϕ

The corresponding DFA for the above NFA is given below

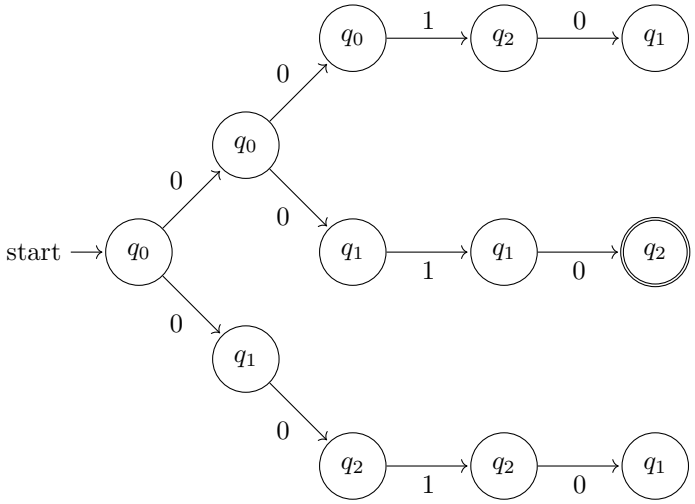


It is not really intuitive to understand what this DFA does at a glance. This is the main advantage of an NFA, specifying automata for languages in an easier and faster way.

e.g. Find out if the string "0010" is accepted by the given NFA.



We can determine whether the given string is accepted by the NFA or not. We first draw all the possible paths which are taken by the string in a "transaction diagram".



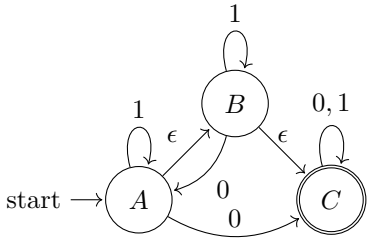
After drawing the diagram, we check the set of final states. If even one of the state in this set is an accepting state, then the string is accepted by the NFA. In this case, the set of final states is $\{q_0, q_1, q_2\}$, the accepting state q_2 is a part of this set, so it means that "0010" is accepted by the NFA. **(Ans.)**

1.5.1 ϵ-NFA

If any finite automaton contains a null move or an ϵ-transaction, then that finite automaton is called NFA with ϵ moves.

Note:

An ϵ move is a move which does not contain any input characters. For representation purposes, we make use of ϵ to show the fact that there are no characters in the input. An epsilon move would be represented by "" in a programming language like C.



An ϵ-NFA can be defined as a regular NFA but with the input alphabet containing an ϵ character as well.

1.5.2 Conversion to DFA

An NFA can be easily converted to a DFA as follows:

1. Find the transition table for the given NFA
2. Create a new table for the DFA to be formed.
3. The first entry in the new table will be the initial state of the NFA. Copy this row as it is from the NFA to the DFA.
4. If any new states are added in the *next state* column then add them as an entry to the current state column. If the next state is a compound state i.e. a set of states, then the next state for this compound states will be a union of the states in that column.
5. Repeat the previous step until no new combination of states appear in the next state column.
6. The accepting states are those which contain at least one of the accepting states of the NFA in them.

1.6 FSM with output

These are finite automata without output but with output. A major difference between the two is that these automata do not "accept" strings, instead they produce output strings of their own. Consequently, these machines do not have a set of "final" or "accepting" states.

1.6.1 Mealy Machines

The Mealy Machine was proposed by *George H. Mealy* at the Bell Labs in 1960.

Mealy Machines are one type of finite automata with output, which produce a distinct output character for a given pair of current state and input character on top of producing a next state.

Any Mealy Machine can be completely described by the following 6-tuple:

$$(Q, \Sigma, \Delta, q_0, \delta, \lambda)$$

where,

$$\begin{aligned}
Q &= \text{Finite set of states} \\
\Sigma &= \text{Set of input symbols} \\
\Delta &= \textbf{Set of output symbols} \\
q_0 &= \text{Initial state} \\
\delta &= \text{Transition function, i.e.} && \delta : Q \times \Sigma \to Q \\
\lambda &= \textbf{Output function, i.e.} && \lambda : Q \times \Sigma \to \Delta
\end{aligned}$$

1.6.2 Moore Machines

The Moore Machine was proposed by *Edward F. Moore* in IBM around 1960.

Moore Machines are one type of finite automata with output, which produce a distinct output character for a given state.

Any Moore Machine can be completely described by the following 6-tuple:

$$(Q, \Sigma, \Delta, q_0, \delta, \lambda)$$

where,

Q = Finite set of states

Σ = Set of input symbols

Δ = **Set of output symbols**

q_0 = Initial state

δ = Transition function, i.e. $\delta : Q \times \Sigma \to Q$

λ = **Output function, i.e.** $\lambda : Q \to \Delta$

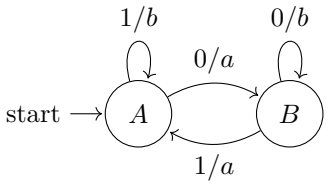
1.6.3 Representations of Mealy and Moore Machines

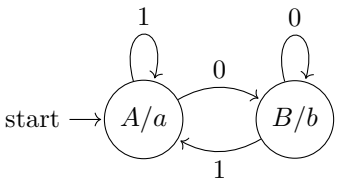
The tabular representation of Mealy and Moore Machines are pretty straightforward.

δ	0		1	
	Q	Δ	Q	Δ
A	B	a	A	b
B	B	b	A	a

δ	0	1	
	Q		Δ
A	B	A	a
B	B	A	b

The graphical representation of Mealy and Moore Machines contain the output symbol/character written next to the state transition or the state itself.





1.6.4 Interconversion of Mealy and Moore Machines

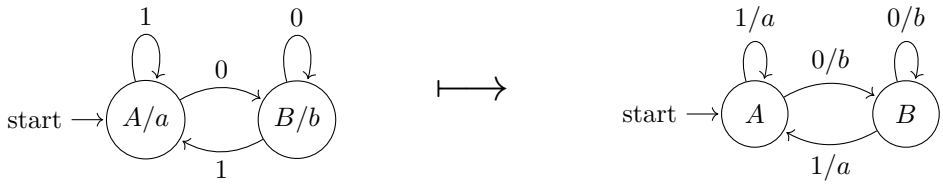
We can perform interconversions between Mealy and Moore Machines in such a way that the output string produced when an input string is given remains the same for both the machines.

Moore to Mealy

We can convert a Moore Machine to a corresponding Mealy Machine very easily from it's graphical representation.

We take the output character of the state and put it on the transitions coming to that state.

e.g. Convert the given Moore Machine to a Mealy Machine



(Ans.)

Mealy to Moore

We can convert a Mealy Machine to a corresponding Moore Machine easily using it's tabular representation.

We try to coalesce the outputs from the transitions to the output of the states, and if any discrepencies arise, we create new states, duplicating the outgoing transitions from these states.

e.g. Convert the given Mealy Machine to a Moore Machine

δ	0		1	
	Q	Δ	Q	Δ
A	B	a	C	a
B	B	b	C	a
C	A	a	B	b

$\longmapsto$

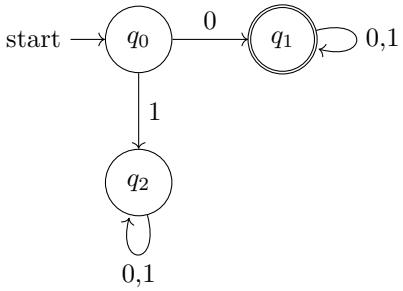
δ	0	1	
	Q		Δ
A_a	B_a	C	a
A_b	B_a	C	b
B_a	B_b	C	a
B_b	B_b	C	b
C	A_a	B_b	a

(Ans.)

Solved Examples

Q.1. Construct a DFA that only accepts (binary) strings which start with the symbol '0'.

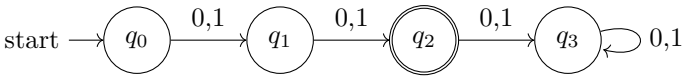
$$\therefore L = \{0, 00, 01, 000, 001, 010, 011, \dots\}$$



(Ans.)

Q.2. Construct a DFA that accepts all (binary) strings of length 2 only.

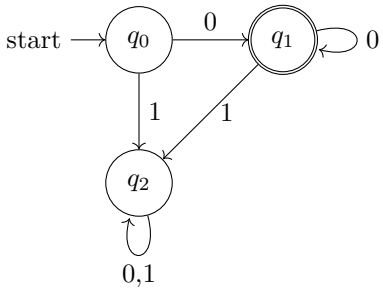
$$\therefore L = \{00, 01, 10, 11\}$$



(Ans.)

Q.3. Construct a DFA accepting words from a language with words only containing '0' over the alphabet $\Sigma = \{0, 1\}$.

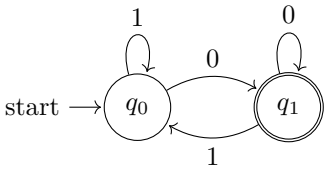
$$\therefore L = \{0, 00, 000, 0000, \dots\}$$



(Ans.)

Q.4. Draw a DFA which will accept even binary number strings.

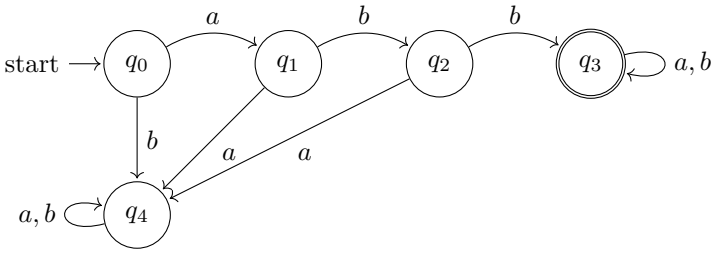
$$\therefore L = \{0, 10, 100, 110, 1000, \dots\}$$



(Ans.)

Q.5. Draw a DFA to accept all strings starting with abb over the alphabet $\{a, b\}$.

$$\therefore L = \{abb, abba, abbb, abbaa, \dots\}$$

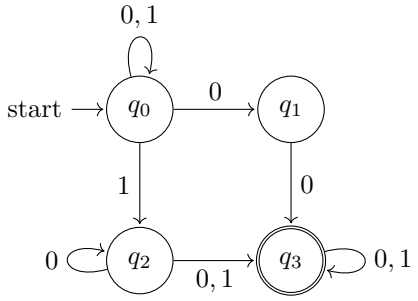


(Ans.)

Q.6. Design an NFA for the transition table as given below.

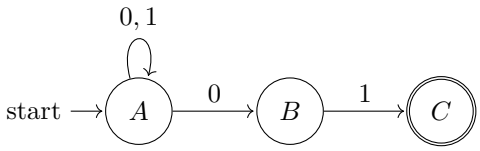
δ	0	1
q_0	q_0, q_1	q_0, q_2
q_1	q_3	ϵ
q_2	q_2, q_3	q_3
q_3	q_3	q_3

where q_0 is the initial state and q_3 is the accepting state.



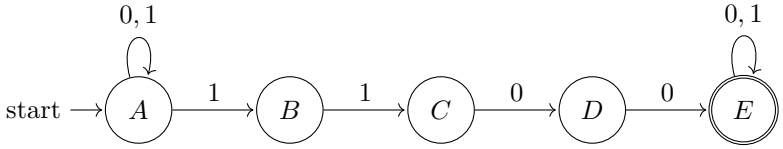
(Ans.)

Q.7. Design an NFA over $\Sigma = \{0, 1\}$ which accepts all strings ending with "01".



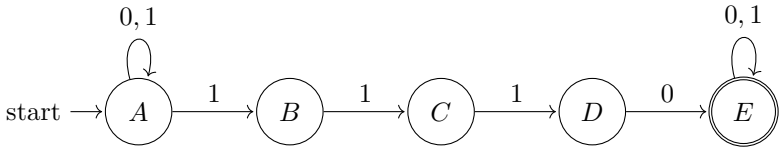
(Ans.)

Q.8. Design an NFA which accepts strings over $\{0, 1\}$ where double 1 is followed by double 0.



(Ans.)

Q.9. Design an NFA in which all the string contain a substring "1110".



(Ans.)

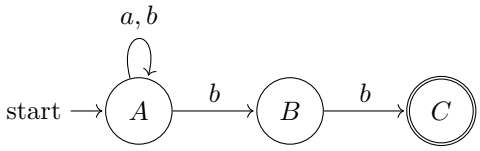
Practice Problems

1. Create the following NFAs which:

(a) Accepts strings starting with '0'.

(b) Accepts strings ending with "01".

2. Convert the given NFA to a DFA:



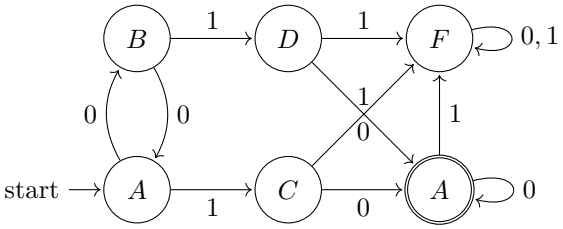
3. Minimise the following DFAs:

(a) The transition table for the DFA is as follows:

δ	0	1
q_0	q_1	q_5
q_1	q_6	q_2
q_2	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

Where q_0 is the initial state and q_2 is the final state.

(b) The graphical representation of the DFA is as follows:



4. Construct the following Mealy Machines:

(a) Produces the 1's-complement of any binary string.

(b) That prints 'a' whenever the sequence "01" is encountered in any input binary string and convert it to a Moore Machine.

5. Construct the following Moore Machines:

(a) That counts the occurrences of the sequence "abb" in any input strings over $\{a, b\}$.

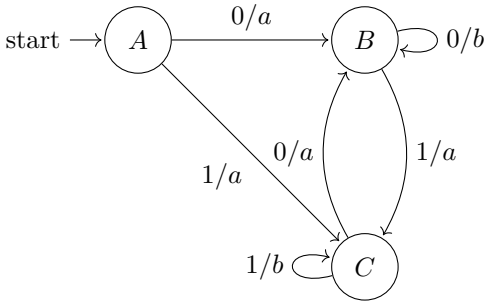
(b) That prints 'a' whenever the sequence "01" is encountered over strings from $\Sigma = \{0, 1\}$ and then convert it to a Mealy Machine.

6. Convert the given Moore Machine to a Mealy Machine.

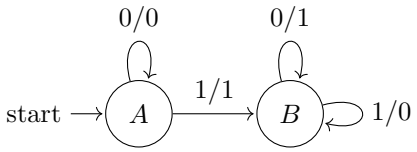
δ	0	1	
	Q		Δ
q_0	q_1	q_2	1
q_1	q_3	q_2	0
q_2	q_2	q_1	1
q_3	q_0	q_3	1

Where q_0 is the initial state.

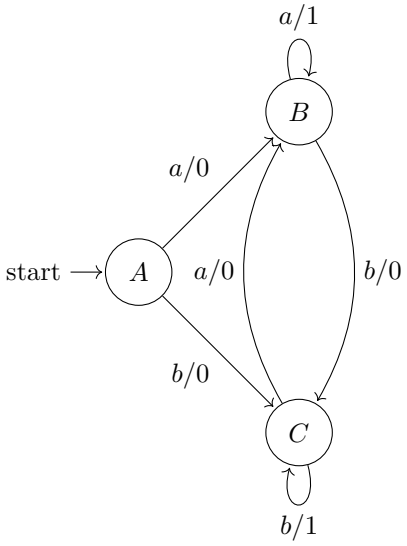
7. Convert the given Mealy Machines to equivalent Moore Machines



(a)



(b)



(c)

Chapter 2

Language and Grammar

2.1 Formal Grammar

A *grammar* is a set of rules to define a valid string in any language. A language is not complete without a grammar.

Grammar is defined as the 4-tuple:

$$G \coloneqq (V, \Sigma, P, S)$$

where,

V = Set of non-terminal symbols

Σ = Set of terminal symbols

P = Set of prouction rules

S = Start symbol (the symbol from which the grammar starts generating strings)

A language is generated from the rules of a grammar. A language contains only terminal symbols ($\in \Sigma$). Let a language L be generated from a grammar G. The language is written as $L(G)$ and read as the language generated by the grammar G. The grammar is called $G(L)$ and read as the grammar for the language L.

The production rules of a grammar consist of two parts. The left side of a production rule mainly contains the non-terminal symbols to be replaced. The right side of a production rule may contain any combination of terminal and non-terminal symbols (even null). To be a valid grammar, at least one production rule must contain the start symbol S on its left side.

In most of the cases, a grammar is represented by only the production rules, as the symbols are understood. Sometimes in production rules, two or more rules are grouped into one. For those cases, the left side of the production must remain the same and the right side rules are separated

by the '|' symbol. As an example,

$$\left\{\begin{array}{l} A \to aA, \\ A \to bCa, \\ A \to ab \end{array}\right\} \text{ are grouped as } A \to aA \,|\, bCa \,|\, ab.$$

2.1.1 Chomsky Hierarchy

The Chomsky hierarchy is an important contribution in the field of formal language and automata theory.

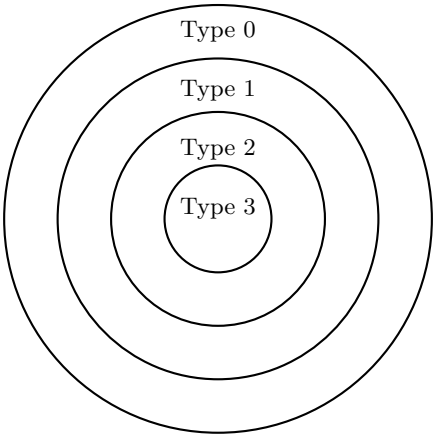


Figure 2.1: Chomsky Hierarchy

Chomsky classified the grammar into four types, depending on the production rules.

Type 0 Type 0 grammar is a phase structure grammar without any restriction. All grammars are type 0 grammar.

For type 0 grammar, the production rules are in the format of

$$\{(\mathrm{L_c})(\mathrm{NT})(\mathrm{R_c})\} \to \{\text{String of terminals and/or non-terminals}\}$$

where $\mathrm{L_c}$ = Left context, $\mathrm{R_c}$ = Right context and NT = Non-terminal.

Type 1 Type 1 grammar is called *context-sensitive grammar*.

For type 1 grammar, all production rules are context-sensitive if all rules in P are of the form

$$\alpha A \beta \to \alpha \gamma \beta$$

where $A \in \mathrm{NT}$, $\alpha, \beta \in (\mathrm{NT} \cup \Sigma)^*$, and $\gamma \in (\mathrm{NT} \cup \Sigma)^+$.

Type 2 Type 2 grammar is called *context-free grammar*. On the left side of each production, there will be no left or right context.

For type 2 grammar, all the production rules are of in the format of

$$\mathrm{NT} \to \alpha$$

where $|\mathrm{NT}| = 1$ and $\alpha \in (\mathrm{NT} \cup \mathrm{T})^*$ (T refers to the set of terminal symbols).

Type 3 Type 3 grammar is called *regular grammar*. Here, all the productions will be in the

following forms:

$$A \to \alpha \text{ or } A \to \alpha B$$

where $A, B \in \text{NT}$ and $\alpha \in \text{T}$.

The table below shows the different machine formats for different languages.

Grammar	**Language**	**Machine Format**
Type 0	Unrestricted language	Turing machine
Type 1	Context-sensitive language	Linear bounded automaton
Type 2	Context-free language	Pushdown automaton
Type 3	Regular language	Finite automaton

Table 2.1: Machine Formats for Grammars in the Chomsky hierarchy

2.1.2 Regular Grammar

Regular grammar is the type of grammar which can be evaluated by the Finite Automata. They have the production rules in the form of: $A \to \alpha$ or $A \to \alpha B$.

2.2 Regular Sets

A Regular Set or Regular Language is the language formed by a Regular Grammar.

2.2.1 Basic Operations

Consider two regular grammars R_1 and R_2 with the set of terminal symbols Σ.

- Union: The language generated by the set $L(R_1) \cup L(R_2)$ is denoted by $L(R_1 + R_2)$. The strings in this set are generated either by R_1 or R_2.
- Concatenation: The language generated by the set $L(R_1) \cap L(R_2)$ is denoted by $L(R_1 R_2)$. The strings in this set are the strings generated by R_1 followed by the strings generated by R_2.
- Kleene's Closure: The Kleene's Closure (or Kleene's Star) is defined as the operation of repeated union and concatenation as follows:

$$L(R^*) = L\left(\epsilon + R + R^2 + R^3 + \cdots\right) = L\left(\sum_{n=0}^{\infty} R^n\right)$$

where

$$\begin{aligned} L(R^*) &= \text{The Kleene's Closure of } R \\ \epsilon &= \text{The empty string.} \\ R^i &= \text{The repeated concatenation of } R \text{ with itself } i \text{ times.} \end{aligned}$$

We define the operator precedence as follows to avoid having to write brackets everywhere as follows:

$$\text{Kleene Star} \succ \text{Concatenation} \succ \text{Union}$$

2.2.2 Closure Properties

A set is said to be closed under some operation if the output of the operation on elements of the set produces an element in the set itself.

1. Closure under Union: Consider two regular sets $L(R_1)$ and $L(R_2)$ and two DFA M_1 and M_2 which accept them respectively.

   Proof. Let us construct a finite automaton M' which will accept strings from both languages.

   $$\begin{aligned}
   &Q' = Q_{1(1)} \cup Q_{2(2)} \cup \{q_0'\} \\
   &\Sigma' = \Sigma_1 = \Sigma_2 \\
   &F' = F_1 \cup F_2 \\
   &q_0' \text{ is a new state which is initial state of either machine.} \\
   &\delta' : Q' \times \Sigma' \mapsto F' \text{ such that} \\
   &\qquad \delta'(q_0', \epsilon) \to \{q_{01}, q_{02}\} \\
   &\qquad \delta'(q_{01}, \Sigma') = \delta_1(q_0, \Sigma_1) \\
   &\qquad \delta'(q_{02}, \Sigma') = \delta_2(q_0, \Sigma_2)
   \end{aligned}$$

   The above defined automaton M' accepts strings either from $L(R_1)$ as well as $L(R_2)$.

   Since there exists an automaton which accepts strings from the union of the regular sets, the regular sets are closed under the Union operation. □

   Note:

   Here, the smaller indices in brackets are used to indicate that the states are being relabelled to avoid confusion.

---

2. Closure under Complement: Consider a regular set $L(R)$ and a DFA M which accepts it.

   Proof. Let us construct a finite automaton which will accept the complement of this set i.e. $L(R)^{\complement} = \Sigma^* - L(R)$

   If we just flip the states which are accepting in nature and which are not, then we get the finite automaton which accepts $L(R)^{\complement}$ i.e. $F_{M'} = Q - F_M$

   Since there exist a finite automaton which accepts strings from the complement of regular sets, the regular sets are closed under the Complement operation. □

3. Closure under Concatenation: Consider two regular sets $L(R_1)$ and $L(R_2)$.

   Proof. By D'Morgan's Theorems, we can prove that:

   $$\begin{aligned}
   L(R_1R_2) &= L(R_1) \cap L(R_2) \\
   &= \left(L(R_1)^{\complement} \cup L(R_2)^{\complement}\right)^{\complement}
   \end{aligned}$$

Since the complement and union operations on a regular language form regular languages as well, the result of a concatenation operation is also a regular language. □

4. Closure under Kleene Star: Consider a regular set $L(R)$ and a DFA M which accepts it

   Proof. Let us construct a finite automaton which will accept the Kleene Star of this set. i.e.

$$\begin{aligned} L(R)^* &= L\left(\sum_{n=0}^{\infty} R^n\right) \\ &= \bigcup_{n=0}^{\infty} L(R)^n \end{aligned}$$

   Since the union and the concatenation operations on a regular language form languages as well, the result of a Kleene Star operation is also a regular language. □

2.3 Regular Expressions

Regular Expressions (RE) provide a way to represent regular languages and their corresponding grammar in a compact way using a single expression.

e.g. Let $L(R_1)$ be a regular set which denotes the set of strings over $\{a, b\}$ which contains an odd number of a's followed by an even number of b's.

Instead of specifying each element of the set in roster form or using set builder form, we can make use of a regular expression such as $a(aa)^*(bb)^*$

e.g. Let R_2 be a regular grammar which denotes the set of even length palindromes over $\{0, 1\}$

Instead of stating the tuple for R_2 or only it's production rules, we can instead use the regular expression: $ww^{\mathcal{R}}$, where $w \coloneqq (a + b)^*$

2.3.1 Identites

An *identity* is a relation that is tautologically true. The following are some identities that are true for all RE.

1. $\varnothing + R = R + \varnothing = R$

   Proof.

$$\begin{aligned} \varnothing + R &= \varnothing \cup R \\ &= \{\ \} \cup R = R \end{aligned}$$

   □

   Similarly, $R + \varnothing = R$.

2. $\varnothing R = R\varnothing = \varnothing$

   Proof.

$$\begin{aligned} \varnothing R &= \varnothing \cap R \\ &= \{\ \} \cap R = \varnothing \end{aligned}$$

□

Similarly, $R\varnothing = \varnothing$.

3. $\epsilon R = R\epsilon = R$

Proof.

$$\begin{aligned} \epsilon R &= \text{Empty string concatenated with any symbol of } R \\ &= \text{That same symbol of } R \implies \epsilon R = R \end{aligned}$$

□

Similarly, $R\epsilon = R$.

4. $\epsilon^* = \epsilon$

Proof.

$$\begin{aligned} \epsilon^* &= \{\epsilon, \epsilon\epsilon, \epsilon\epsilon\epsilon, \dots\} \\ &= \{\epsilon, \epsilon, \epsilon, \dots\} && (\because \epsilon R = R) \\ &= \{\epsilon\} = \epsilon \end{aligned}$$

□

$\varnothing^* = \epsilon$

Proof.

$$\begin{aligned} \varnothing^* &= \varnothing^0 + \varnothing^1 + \varnothing^2 + \cdots \\ &= \epsilon + \varnothing + \varnothing + \cdots \\ &= \epsilon && (\because R + \varnothing = R) \end{aligned}$$

□

5. $R + R = R$

Proof.

$$\begin{aligned} R + R &= R\epsilon + R\epsilon && (\because P\epsilon = P) \\ &= R(\epsilon + \epsilon) \\ &= R\epsilon && (\because \epsilon + P = P) \\ &= R \end{aligned}$$

□

6. $R^*R^* = R^*$

Proof.

$$\begin{aligned} R^*R^* &= \{\epsilon, R, RR, \dots\}\{\epsilon, R, RR, \dots\} \\ &= \{\epsilon\epsilon, \epsilon R, \epsilon RR, \dots, R\epsilon, RR, RRR, \dots\} \\ &= \{\epsilon, R, RR, \dots\} && (\because \epsilon P = P\epsilon = P) \\ &= R^* \end{aligned}$$

□

7. $R^*R = RR^*$

   Proof.

$$\begin{aligned} R^*R &= \{\epsilon, R, RR, \dots\}R \\ &= \{\epsilon R, RR, RRR, \dots\}R \\ &= \{R, RR, RRR, \dots\}R && (\because \epsilon P = P) \\ &= R\{\epsilon, R, RR, \dots\} \\ &= RR^* \end{aligned}$$

□

8. $(R^*)^* = R^*$

   Proof.

$$\begin{aligned} (R^*)^* &= \{\epsilon, R^*, R^*R^*, \dots\} \\ &= \{\epsilon, R^*, R^*, \dots\} && (\because P^*P^* = P^*) \\ &= \{\epsilon, R^*\} \\ &= \{\epsilon\} \cup \{R^*\} \\ &= \epsilon + R^* \\ &= R^* && (\because \epsilon + P = P) \end{aligned}$$

□

9. $\epsilon + RR^* = \epsilon + R^*R = R^*$

   Proof.

$$\begin{aligned} \epsilon + RR^* &= \epsilon + R\{\epsilon, R, RR, \dots\} \\ &= \epsilon + \{R\epsilon, RR, RRR, \dots\} \\ &= \epsilon + \{R, RR, RRR, \dots\} && (\because P\epsilon = P) \\ &= \{\epsilon, R, RR, RRR, \dots\} \\ &= R^* \end{aligned}$$

□

   Similarly, $\epsilon + R^*R = R^*$.

10. $(PQ)^*P = P(QP)^*$

    Proof.

$$\begin{aligned} (PQ)^*P &= \{\epsilon, PQ, PQPQ, PQPQPQ, \dots\}P \\ &= \{\epsilon P, PQP, PQPQP, PQPQPQP, \dots\} \\ &= \{P, PQP, PQPQP, PQPQPQP, \dots\} && (\because \epsilon R = R) \\ &= P\{\epsilon, QP, QPQP, QPQPQP, \dots\} \\ &= P(QP)^* \end{aligned}$$

□

11. $(P+Q)^* = (P^*Q^*)^* = (P^* + Q^*)^*$ (De Morgan's Theorem)
12. $(P+Q)R = PR + QR$

2.3.2 Arden's Theorem

The Arden's theorem is used to construct the RE from a transitional system of FA.

Theorem 4. *Let P and Q be two RE over Σ. If $\epsilon \notin P$, then the equation has a unique solution $R = QP^*$.*

Proof. Given that,

- P and Q are two RE.
- $\epsilon \notin P$.

We need to prove,

- $R = QP^*$ is a solution to $R = Q + RP$.
- $R = QP^*$ is the only solution to $R = Q + RP$.

Substituting $R = QP^*$ in $Q = Q + RP$, we get

$$\begin{aligned} R &= Q + RP \\ \therefore QP^* &= Q + QP^*P \\ \therefore QP^* &= Q(\epsilon + P^*P) \\ \therefore QP^* &= QP^* \qquad (\because \epsilon + R^*R = R^*) \end{aligned}$$

Therefore, $R = QP^*$ solves the equation $R = Q + RP$.

Given that $R = Q + RP$,

$$\begin{aligned} R &= Q + RP \\ \therefore R &= Q + (Q + RP)P \\ &= Q + QP + RPP \\ &= Q(\epsilon + P) + RP^2 \\ &= Q(\epsilon + P) + (Q + RP)P^2 \\ &= Q(\epsilon + P) + QP^2 + RPP^2 \\ &= Q(\epsilon + P + P^2) + RP^3 \end{aligned}$$

After several steps, we obtain

$$R = Q(\epsilon + P + P^2 + \cdots + P^n) + RP^{n+1}.$$

Consider a string $w \in R$. If $w = \epsilon$, then $w \in Q(\epsilon + P + P^2 + \cdots + P^n)$ or $w \in RP^{n+1}$.

$\because \epsilon \notin P$, w must belong to $Q(\epsilon + P + P^2 + \cdots + P^n)$, which is effectively QP^* as $n \to \infty$.

As any string w belongs to only one part, R and QP^* represent the same set. This means that $R = QP^*$ is the unique solution of the equation $R = Q + RP$. □

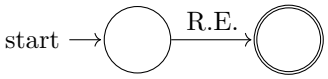
2.3.3 Conversion to Finite Automata

Regular expressions are another way to represent the Regular Grammar. We can use them to form Finite Automata.

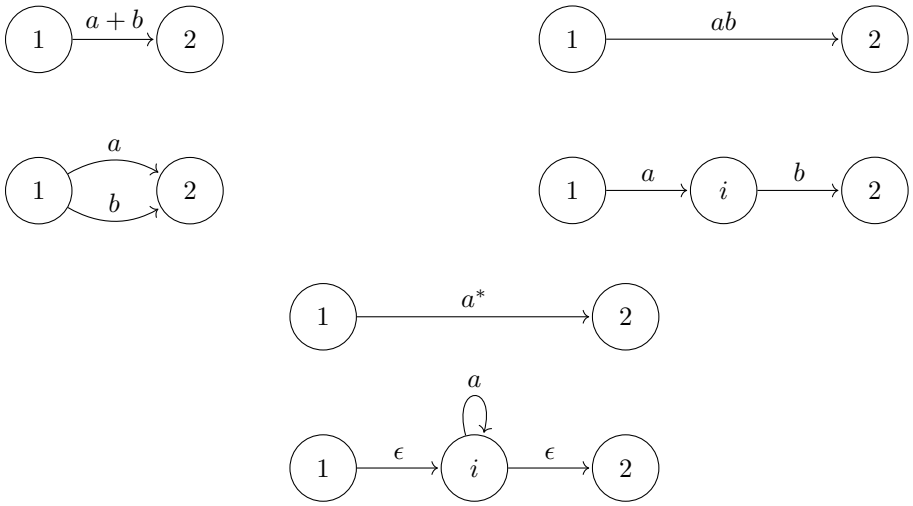
There are two major methods to forming finite automata from regular expressions: the top-down and the bottom-up approach. We only discuss the top-down approach here.

Top Down Approach

In this method we start from a simple finite automaton accepting the entire RE and keep breaking it down.



The different operators are broken down in the following ways



Using these three methods, we can break down a complex RE to it's constituent parts and form an ϵ-NFA. We then use the ϵ-closure method to form a DFA.

2.3.4 Conversion from Finite Automata

Regular expressions are another way to represent the Regular Grammar. We can form them from Finite Automata.

Each state in a finite automaton forms a regular expression.

Suppose a given state q_a has the predecessors q_{a-1} coming to it from the transition $\delta(q_{a-1}, \vartheta) \to q_a$. Then we can find the regular expression for the state q_a as follows:

$$RE(q_a) = \sum RE(q_{a-1}) \cdot \vartheta$$

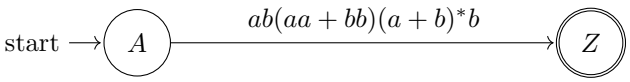
To avoid notational complexity, we directly write the name of the states in the R.H.S. of the equation.

$$RE(q_a) = \sum q_{a-1} \cdot \vartheta \tag{2.1}$$

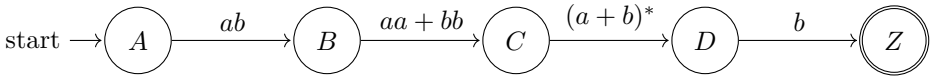
e.g. Construct a Finite Automaton equivalent to the Regular Expression:

$$L = ab(aa + bb)(a + b)^*b$$

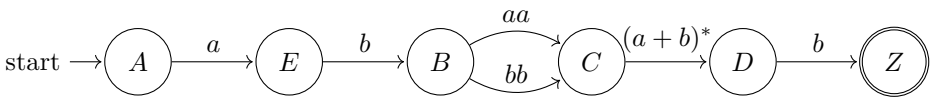
Step I: Take a beginning state A and a final state Z. Between the beginning and final state, place the given regular expression.



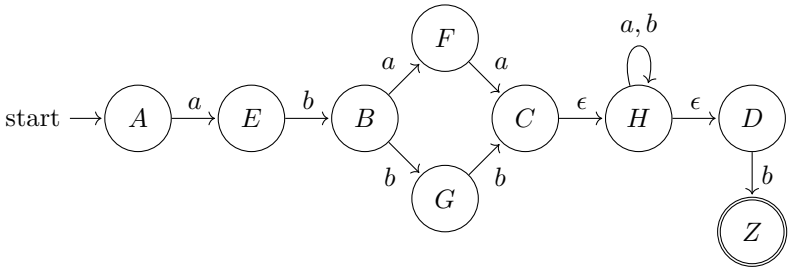
Step II: Since there are three $(\cdot)$s in between, we add three intermediate states.



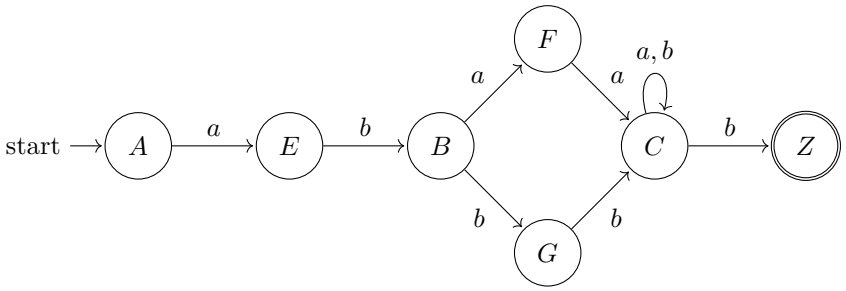
Step III: Between a and b there is another $(\cdot)$, so we add another intermediate state, we also add parallel paths between B and C in order to remove the $+$.



Step IV: We now break up the double letters aa and bb taking intermediate states. We also break the Kleene Star operation.

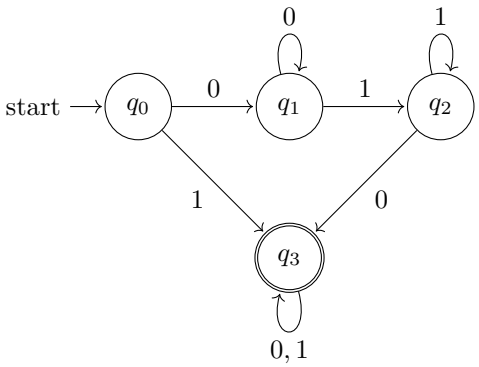


Step V: We can now use the ϵ-closure method or the transitional method in order to remove the ϵ transitions.



We have formed an NFA, we can easily convert it to a regular DFA and find the finite automaton which accpets the given RE. **(Ans.)**

e.g. Given the following FA give the regular expression which represents the language accepted by it.



For the above given DFA, the equations for each state are as follows:

- $RE(q_0) = q_0 = \epsilon$
- $RE(q_1) = q_1 = q_0 \cdot 0 + q_1 \cdot 0$
- $RE(q_2) = q_2 = q_1 \cdot 1 + q_2 \cdot 1$
- $RE(q_3) = q_3 = q_0 \cdot 1 + q_2 \cdot 0 + q_3 \cdot 0 + q_3 \cdot 1$

For q_0, it only depends on the start input which is ϵ so the RE for this state is ϵ.

For q_1, it depends on q_0 and q_1 (by a self loop) . Let us substitute the value of q_0 over here.

$$\begin{aligned} q_1 &= q_0 \cdot 0 + q_1 \cdot 0 \\ &= \epsilon \cdot 0 + q_1 \cdot 0 \\ q_1 &= 0 + q_1 \cdot 0 \\ q_1 &= 00^* \end{aligned} \quad \text{From (theorem 4)}$$

Similarly for q_2, it depends on q_1 and q_2 (by self loop). Let us substitute the value of q_1 over here.

$$\begin{aligned} q_2 &= q_1 \cdot 1 + q_2 \cdot 1 \\ &= 00^* \cdot 1 + q_2 \cdot 1 \\ q_2 &= 00^*1 + q_2 \cdot 1 \\ q_2 &= 00^*11^* \end{aligned} \quad \text{From (theorem 4)}$$

Finally, for q_3, it depends on q_0, q_2 and q_3 itself by a self loop. Let us substitute the values of q_0 and q_2 over here.

$$\begin{aligned} q_3 &= q_0 \cdot 1 + q_2 \cdot 0 + q_3 \cdot (0+1) \\ &= \epsilon \cdot 1 + 00^*11^* \cdot 0 + q_3 \cdot (0+1) \\ q_3 &= (1 + 00^*11^*0) + q_3 \cdot (0+1) \\ q_3 &= (1 + 00^*11^*0)(0+1)^* \end{aligned} \quad \text{From (theorem 4)}$$

Since, q_3 is the accepting state, the regular expression of q_3 is the representation of the language accepted by the FA. $\boxed{q_3 = (1 + 00^*11^*0)(0+1)^*}$ **(Ans.)**

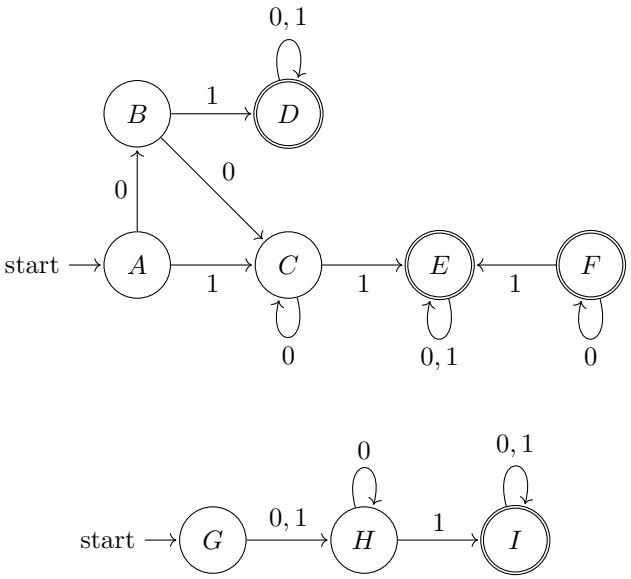
2.4 Equivalence of two Finite Automata

Two Finite Automata are said to be equivalent in in nature if they generate the same equivalent RE. Alternatively, two FA are said to be equivalent in nature if on minimisation, they produce the same number of states and same transitional function δ.

We can easily check if two finite automata are equivalent in nature by the following procedure:

1. Given two FA, M_1 and M_2, with the start symbols s_1 and s_2, ensure that they have the same number of input symbols.
2. Draw a table with $|\Sigma| + 1$ columns.
3. The row of the first column will contain the ordered pair (s_1, s_2). The columns will contain the ordered pairs of states which we will go to after applying the transition $\delta(s_i, x)$ to the states in the first column in that row.
4. If any new ordered pairs appear in the final $|\Sigma|$ columns, then place it in the first column of the next row and repeat the process until no new ordered pairs arrive.
5. If at any point in the process, an ordered pair of the form $Q_1 - F_1, F_2$ or $F_1, Q_2 - F_2$ appears then declare the two FA to not be equivalent. If this does not happen then declare the two FA to be equivalent.

e.g. Find out whether the given FA are equivalent or not.



We start the process of finding if the two automata are equivalent or not by creating a table with 3 columns.

$q_c = (q, q')$	$\delta(q_c, 0)$	$\delta(q_c, 1)$
(A, G)	(B, H)	(C, H)
(B, H)	(C, H)	(D, I)
(C, H)	(C, H)	(E, I)
(E, I)	(E, I)	(E, I)
(D, I)	(D, I)	(D, I)

As no new combinations appear, we stop the process over here. We do not encounter any

ordered pairs of the form where one of the entries is non-final state and the other is a final state. This means that the two FA are equivalent in nature. **(Ans.)**

2.5 Pumping Lemma

The pumping lemma is a necessary[1] condition for a string to belong to a regular set. Therefore, it can be used to disprove a language to be regular if it does not fulfill the conditions of the lemma.

Lemma 1. *Assume that set L is regular. Let n be the number of states of the FA accepting L.*

1. *Every $w \in L$ with $|w| \geq n$ can be written as $w \coloneqq xyz$, for some strings xyz.*
2. *$|y| \geq 1$.*
3. *$|xy| \leq n$.*
4. *$xy^iz \in L \ \forall i \geq 0$.*

Q.1. Show that $L = \{a^p \mid p \text{ is prime}\}$ is not regular.

Proof. Assume that L is regular. Let n be the number of states in the FA accepting L.

Suppose p is a prime number greater than n. Let the string $w \coloneqq a^p \in L$. One deduction is that the length of every string in L is prime, i.e. $\forall w \in L|w|$ is prime.

By the pumping lemma, we can write $w = xyz$ with $|xy| \leq n$ and $|y| > 0$. Suppose $y = a^m$ for some m with $1 \leq m \leq n$.

Now, for xy^iz,

$$\begin{aligned} \left|xy^iz\right| &= |xyz| + \left|y^{i-1}\right| \\ &= p + (i-1)|y| \\ \therefore \left|xy^iz\right| &= p + (i-1)m \end{aligned}$$

Suppose the case for $i = p + 1$. Then, $\left|xy^iz\right| = p + pm = p(1 + m)$.

However, $p(1 + m)$ is not a prime number as it has factors p and $(1 + m)$ other than 1 and $p(1 + m)$ itself. This implies $xy^iz \notin L$ and by contradiction, L cannot be a regular language. □

---

[1] The pumping lemma is not sufficient for a language to be regular. If a language fulfils the conditions of the pumping lemma, it still might not be regular.

Chapter 3

Context-Free Grammar

3.1 Definition

3.1.1 Context

According to the Chomsky hierarchy, Context-Free Grammar (CFG) is **type 2 grammar**.

Before describing why this type of grammar is called context-free, we have to know the definition of *context*. Production rules are usually of the following form:

{String comprising at least one non-terminal} → {String of terminals and/or non-terminals}

If any symbol is present with the producing non-terminal on the left side of the production rule, then that extra symbol is called *context*. In CFG, at the left side of each production rule, there is only one non-terminal (no context is added with it). For this reason, this type of grammar is called context-free.

3.1.2 Formal Definition

In formal language theory, a Context-Free Language (CFL) is a language generated by some CFG. The set of all CFLs is identical to the set of languages accepted by a Pushdown Automaton (PDA).

A CFG is defined by the following 4-tuple:

$$G \coloneqq (V, \Sigma, P, S)$$

where,

V = Set of variables or non-terminal symbols

Σ = Set of terminal symbols

P = Set of Prouction rules

S = Start symbol

3.1.3 Production Rules

A *production rule* (or *production*) defines how terminal and non-terminal symbols can be combined to form strings. It is given by:

$$A \to \alpha$$

where,

- A is a non-terminal symbol from the set V.
- α is a string of symbols from the set $\{V \cup \Sigma\}^*$, meaning it could be a combination of terminal and non-terminal symbols.

For example, consider a language that generates equal numbers of a's and b's in the form $a^n b^n$. The CFG can be defined as:

$$G \coloneqq (\{S, A\}, \{a, b\}, \{S \to aAb, A \to aAb \,|\, \epsilon\})$$

3.1.4 Differences from Regular Grammar

Grammar Type	Regular Grammar	Context-Free Grammar
Class	Generates Regular Languages, which can be recognized by FA. Regular grammars are limited in complexity and cannot generate languages that require memory beyond an FSM.	Generates CFLs, which can be recognized by a PDA and can handle more complex languages that require a stack-based memory model, such as matching parentheses.
Production Rules	The production rules are more restrictive. They take the form: $A \to aB \quad \text{or} \quad A \to a$ where A is a non-terminal, and a is a terminal symbol. The non-terminal is allowed only on the right side of the production rule, and it is limited to right-linear or left-linear forms.	The production rules are more flexible and can take the form: $A \to \alpha$ where A is a non-terminal, and α can be any string of terminals and non-terminals. This allows CFGs to generate more complex structures like nested or recursive patterns.
Complexity	Only handles simpler language structures, such as strings that can be recognized by FSM. For example, regular languages cannot handle matching numbers of parentheses or nested structures.	Can handle more complex constructs like balanced parentheses, palindromes, and other recursive patterns that require stack-like memory. CFGs can generate programming languages' syntax and expressions.
Acceptance	Accepted by FA, which have no memory other than the current state.	Accepted by PDA, which has a stack for memory and can thus handle more complex language constructs.

3.2 Derivation Tree

A Derivation Tree (or Parse Tree) is a diagram that represents how a string of symbols can be derived from a CFG. It visually illustrates the process of applying the production rules of a grammar to form a string.

Root Vertex The root of the tree must always be labeled with the start symbol S.

Vertices These nodes represent non-terminal symbols. They are symbols used to define other symbols or structures in the grammar.

Leaves The leaves of the tree are labeled with terminal symbols or ϵ. Terminal symbols are the basic symbols that cannot be replaced further and are part of the final derived string.

3.2.1 Left Derivation Tree

A *Left Derivation Tree* is a tree formed by applying production rules to the *leftmost variable* (non-terminal symbol) in each step.

A derivation is called a *leftmost derivation* if we replace only the leftmost non-terminal by some production rule at each step of the generating process of the language from the grammar.

3.2.2 Right Derivation Tree

A *Right Derivation Tree* is a tree formed by applying production rules to the *rightmost variable* (non-terminal symbol) in each step.

A derivation is called a *rightmost derivation* if we replace only the rightmost non-terminal by some production rule at each step of the generating process of the language from the grammar.

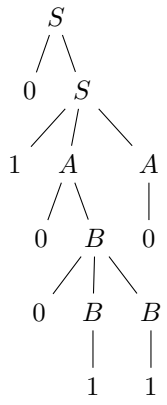
Q.1. Find the parse tree for generating the string 0100110 from the following grammar.

$$S \to 0S \,|\, 1AA$$
$$A \to 0 \,||1A \,|\, 0B$$
$$B \to 1 \,|\, 0BB$$

For generating the string 0100110 form the given CFG, the leftmost derivation will be

$$S \to 0\underline{S} \to 01\underline{A}A \to 010\underline{B}A \to 0100\underline{B}BA \to 01001\underline{B}A \to 010011\underline{A} \to 0100110$$

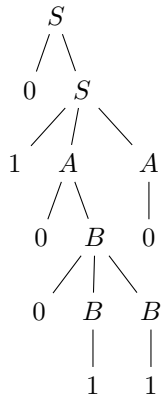
.



For generating the string 0100110 form the given CFG, the rightmost derivation will be

$$S \to 0\underline{S} \to 01A\underline{A} \to 01\underline{A}0 \to 010\underline{B}0 \to 0100B\underline{B}0 \to 0100\underline{B}10 \to 0100110$$

.



3.2.3 Ambiguous Grammar

A grammar is said to be *ambiguous* if there exist two or more derivation trees for a string w (i.e., two or more *left* derivation trees).

Consider generating the string aaa from the following grammar:

$$S \to aS \,|\, AS \,|\, A$$
$$A \to AS \,|\, a$$

We have two ways to generate the string as follows (using leftmost derivation):

1. $S \to aS \to aAS \to aaS \to aaA \to aaa$
2. $S \to A \to AS \to ASS \to aAS \to aaS \to aaA \to aaa$

In relation to ambiguity in CFG, there are few more definitions.

Ambiguous CFL A CFG G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct parse trees.

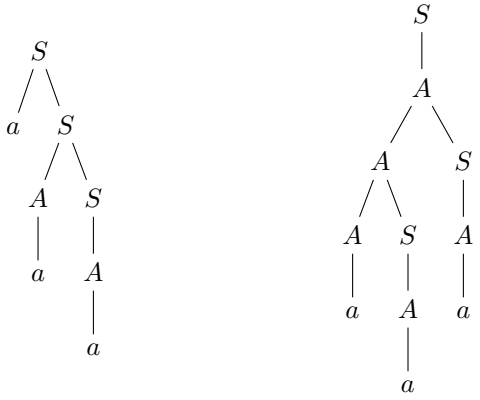


Figure 3.1: Ambiguous grammar

Inherently Ambiguous CFL A CFL L is said to be *inherently ambiguous* if all its grammars are ambiguous.

Unambiguous CFL If L is a CFL for which there exists an unambiguous grammar, then l is said to be *unambiguous*.

3.3 Simplification of CFG

When working with CFG, some rules and symbols are unnecessary for generating valid strings. These unnecessary parts make the grammar more complicated than it needs to be.

To make the grammar simpler and more efficient, we remove:

1. Useless rules or symbols that do not help in generating valid strings.
2. Null (ϵ) productions, which allow a variable to disappear without contributing to the output.
3. Unit productions, which are rules that simply replace one variable with another without adding new information.

3.3.1 Elimination of useless variables

A variable Y is *useful* iff both the following hold:

1. Y must produce at least one terminal.
2. Y must be reachable from S via some series of productions ($S \rightsquigarrow Y$).

Even if one of the above conditions are false, the variable Y is declared *useless* and discarded as follows:

- If Y is present on the **right** of a production, then discard the sub-production.
- If Y is present on the **left** of a production, then discard the entire production.

3.3.2 Removal of Unit Productions

A unit production is of the form $A \to B$, where $A \in V \land B \in V$ (non-terminals). The procedure for removal of unit productions is as follows:

To remove $A \to B$,

1. Add a production $A \to x$ to the grammar rules for all occurrences of $B \to x$ in the grammar, where $x \in \{V \cup \Sigma\}^*$.
2. Delete $A \to B$ from the grammar.

3.3.3 Removal of Null Productions

In a CFG, a non-terminal symbol A is said to be *nullable* if there exists a production $A \to \epsilon \in P$ or there is a derivation that starts at A and eventually leads to ϵ ($A \to \cdots \to \epsilon$ or $A \rightsquigarrow \epsilon$).

To remove $A \rightsquigarrow \epsilon$,

1. For all productions whose right side contains A, replace each occurrence of A with ϵ.
2. Add the resultant productions to the grammar.

3.3.4 Chomsky Normal Form

A CFG is said to be in Chomsky Normal Form (CNF) when the elements on the right side of each production in the grammar are either two variables or a terminal.

$$\forall p \in P,\ p \equiv A \to BC \vee p \equiv A \to a$$

where $A, B, C \in V$ and $a \in \Sigma$.

Converting a CFG to CNF

A CFG can be converted to a CNF by the following algorithm:

Algorithm 1: Converting a CFG to a CNF

Step 1: If the start symbol S appears on the right-hand side of any production, create a new start symbol S' and add the production $S' \to S$.

Step 2: Remove all null (ϵ) productions from the grammar.

Step 3: Remove all unit productions (productions of the form $A \to B$ where both A and B are non-terminals).

Step 4: For each production of the form $A \to B_1 B_2 \dots B_n$ where $n > 2$, replace it by:

- $A \to B_1 C$,
- where $C \to B_2 \dots B_n$,

and repeat this step until all productions have at most two non-terminals on their right.

Step 5: For any production of the form $A \to aB$, where a is a terminal and A, B are non-terminals, replace it by:

- $A \to XB$, and
- $X \to a$,

and repeat for all such productions.

3.3.5 Greibach Normal Form

A grammar is said to be in Greibach Normal Form (GNF) if every production of the grammar is of the form

$$A \to aX$$

where $a \in \Sigma$ and $X \in V^*$.

In other words, each production must contain exactly one terminal followed by any combination of non-terminals on its right side.

If a CFG can be converted into a GNF, then the PDA accepting the CFL can be easily designed.

Before going into the details about the process of converting a CFG into GNF, we have to define two lemmas which are useful for the conversion process.

Lemma 2. *If $G \coloneqq (V, \Sigma, P, S)$ is a CFG, and if $A \to A\alpha$ and $A \to \beta_1 \,|\, \beta_2 \,|\, \dots \,|\, \beta_n$ belong to the production rules (P) of G, then a new grammar $G' \coloneqq (V, \Sigma, P', S)$ can be constructed by replacing $A \to \beta_1 \,|\, \beta_2 \,|\, \dots \,|\, \beta_n$ in $A \to A\alpha$, which will produce*

$$A \to \beta_1\alpha \,|\, \beta_2\alpha \,|\, \dots \,|\, \beta_n\alpha$$

This production belongs to P' in G'. It can be proven that $L(G) = L(G')$.

Lemma 3. *Let $G := (V, \Sigma, P, S)$ be a CFG and the 'A' productions which belong to P be $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \ldots \mid \beta_n$.*

Introduce a new non-terminal X. Let $G' := (V', \Sigma, P', S)$, where $V' := V \cup \{X\}$ and P' can be formed by replacing the A productions by

1. $\left.\begin{array}{lcl} A & \to & \beta_i \\ A & \to & \beta_i X \end{array}\right\} 1 \leq i \leq n$

2. $\left.\begin{array}{lcl} X & \to & \alpha_j \\ X & \to & \alpha_j X \end{array}\right\} 1 \leq j \leq m$

It can be proven that $L(G) = L(G')$.

Converting a CFG to GNF

A CFG can be converted to a GNF by the following algorithm:

Algorithm 2: Converting a CFG to a GNF

Step 1: Convert the grammar into CNF by algorithm 1.

Step 2: Rename all the non-terminals of V as $(A_1, A_2, \ldots, A_n)$ with start symbol A_1.

Step 3: Using lemma 2 modify the productions such that the variable subscript on the left side of the production is less than that of the starting variable on the right side of the production. In mathematical notation, it can be said that all the productions will be in the form $A_i \to A_j V$ where $i \leq j$.

Step 4: By repeating applications of lemma 2 and lemma 3, all the productions of the modified grammar will come into GNF.

3.4 Pumping Lemma

The pumping lemma for a CFL is used to prove that certain sets are not context-free. If a language fulfills all the properties of the pumping lemma for a CFL, it cannot be said that the language is context-free. However, the reverse is true: if a language breaks the properties then it can be said that the language is not context-free.

Lemma 4. *Let L be a CFL. Then, we can find a natural number n such that*

1. *Every $z \in L$ with $|z| \geq n$ can be written as $z := uvwxy$, for some strings u, v, w, x, y.*
2. $|vx| \geq 1$.
3. $|vwx| \leq n$.
4. $uv^k wx^k y \in L \ \forall k \geq 0$.

Q.2. Prove $L = \{a^n b^n c^n \mid n > 0\}$ is not a CFL.

Proof. Assume that L is a CFL. Let n be a natural number obtained by using the pumping lemma.

Let $z := a^n b^n c^n$. It can be observed that $|z| = 3n \geq n$. According to the pumping lemma, we can write $z := uvwxy$, where $|vx| \geq 1$ and $|vwx| \leq n$.

Consider the case for $n = 5$, where $u := aaa, v := a, w := a, x := b, y := bbbbccccc$.

If L is a CFL, then by the pumping lemma $uv^k wx^k y \in L \ \forall k \geq 0$.

When $k = 2$, $z = \underbrace{aaa}_{u}\underbrace{aa}_{v^2}\underbrace{a}_{w}\underbrace{bb}_{x^2}\underbrace{bbbbccccc}_{y} = a^6b^6c^5 \notin L$, which contradicts the pumping lemma.

$\therefore L$ is not a CFL. □

Q.3. Show that $L = \{ww \mid w \in \{0,1\}^*\}$ is not context-free.

Proof. Assume that L is a CFL. Let n be the pumping length of this language obtained by the pumping lemma.

Let $S \coloneqq 0^n1^n0^n1^n$. It can be observed that $|S| = 4n \geq n$. According to the pumping lemma, we can write $S \coloneqq uvxyz$, where $|vy| \geq 1$ and $|vxy| \leq n$.

Consider the case for $n = 3$, where $S = 000111000111$ and $u \coloneqq 0, v \coloneqq 0, x \coloneqq 0, y \coloneqq 1, z \coloneqq 11000111$.

If L is a CFL, then by the pumping lemma $uv^kwx^ky \in L\ \forall k \geq 0$.

When $k = 2$, $S = \underbrace{0}_{u}\underbrace{00}_{v^2}\underbrace{0}_{x}\underbrace{11}_{y^2}\underbrace{11000111}_{z} = 0^41^40^31^3 \notin L$, which contradicts the pumping lemma.

$\therefore L$ is not context-free. □

3.5 Closure Properties

A set is closed under an operation iff the operation on any two elements of the set produces another element of the set. If an element is produced that does not belong in the set, then the operation is not closed.

We shall discuss the closure of a CFL under the following operations:

3.5.1 Closed under Union

Let L_1 be a CFL for the CFG $G_1 \coloneqq (V_1, \Sigma_1, P_1, S_1)$ and L_2 be a CFL for the CFG $G_2 \coloneqq (V_2, \Sigma_2, P_2, S_2)$. We have to prove that $L_1 \cup L_2$ is also a CFL.

Proof. Construct a grammar $G \coloneqq (V, \Sigma, P, S)$ using the grammars G_1 and G_2 as follows:

$$V \coloneqq V_1 \cup V_2 \cup \{S\}$$
$$\Sigma \coloneqq \Sigma_1 \cup \Sigma_2$$
$$P = P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}$$

It is clear that the language set generated from the grammar G contains all the strings that are derived from S_1 as well as S_2. Therefore, $L_1 \cup L_2$ is also context-free. □

3.5.2 Closed under Concatenation

Let L_1 be a CFL for the CFG $G_1 \coloneqq (V_1, \Sigma_1, P_1, S_1)$ and L_2 be a CFL for the CFG $G_2 \coloneqq (V_2, \Sigma_2, P_2, S_2)$. We have to prove that L_1L_2 is also a CFL.

Proof. Construct a grammar $G \coloneqq (V, \Sigma, P, S)$ using the grammars G_1 and G_2 as follows:

$$V \coloneqq V_1 \cup V_2 \cup \{S\}$$
$$\Sigma \coloneqq \Sigma_1 \cup \Sigma_2$$
$$P \coloneqq P_1 \cup P_2 \cup \{S \to S_1 S_2\}$$

It is clear that the language set generated from the grammar G contains all the strings that are derived from S_1 as well as S_2. Therefore, $L_1 L_2$ is also context-free. □

3.5.3 Closed under Kleene's Closure

Let L be a CFL for the CFG $G \coloneqq (V, \Sigma, P, S)$. We have to prove that L^* is also a CFL.

Proof. Construct a grammar $G' \coloneqq (V', \Sigma, P', S')$ using the grammars G as follows:

$$V' \coloneqq V \cup \{S\}$$
$$P \coloneqq P \cup \{S' \to SS' \mid \epsilon\}$$

We have already proved that CFLs are closed under union and concatenation[1], hence L^* is also context-free. □

3.5.4 Not Closed under Intersection

Proof. Consider the following CFLs:

1. $L_1 \coloneqq \{a^n b^n c^m \mid n, m \geq 0\}$
2. $L_2 \coloneqq \{a^m b^n c^n \mid n, m \geq 0\}$

The intersection of these languages is $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$, which is not context-free (as it cannot be recognized by a PDA). Therefore, CFL is not closed under intersection. □

3.5.5 Not Closed under Complementation

Proof. Consider two CFLs L_1 and L_2. For contradiction, let us assume CFLs are closed under complement.

$\implies$ $L_1^{\complement}$ and $L_2^{\complement}$ are also CFLs.

From De Morgan's law in set theory,

$$L_1 \cap L_2 = \left(L_1^{\complement} \cup L_2^{\complement}\right)^{\complement}$$

$\because$ CFLs are closed under union, and $L_1^{\complement}$ and $L_2^{\complement}$ are CFLs by assumption, $L_1^{\complement} \cup L_2^{\complement}$ is also a CFL.

By our assumption, $\left(L_1^{\complement} \cup L_2^{\complement}\right)^{\complement}$ must also be a CFL $\implies$ $L_1 \cap L_2$ must also necessarily be a CFL.

However, we already proved that CFLs are not closed under intersection. Hence, our assumption must be incorrect and thus CFLs are not closed under complementation. □

[1] Kleene's closure is basically repeated union and concatenation with the empty string.

3.5.6 Every Regular Language is Context-free

From the recursive definition of a regular set, we know that $\varnothing$ and ϵ are regular expressions. If R is a regular expression, then $R + R$, RR and R^* are also regular expressions. A RE R is a string of terminal symbols. $\varnothing$ and ϵ are also CFLs, and we know that CFLs are closed under union, concatenation and Kleene's closure. Therefore, we can say that every regular language is a CFL.

Chapter 4

Pushdown Automata

Stack

A stack is a Last-In-First-Out (LIFO) data structure that allows sequential representation of input symbols. A stack allows two fundamental operations:

Push A new element is added on top of the stack.

Pop The top element of the stack is read and removed.

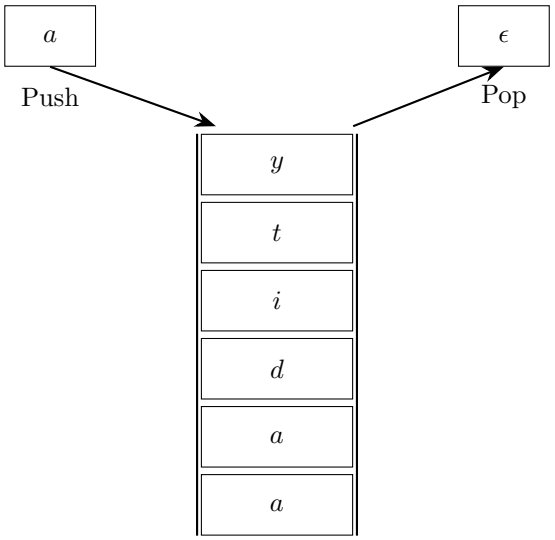


Figure 4.1: Stack

4.1 Definition of a PDA

A PDA is a way to implement a CFG. Unlike an FSM, a PDA is more powerful as it has more memory. A PDA is effectively an FSM with a stack memory.

4.1.1 Components of a PDA

A PDA has the following components:

1. An input tape,
2. A finite control unit,
3. A stack with infinite size.

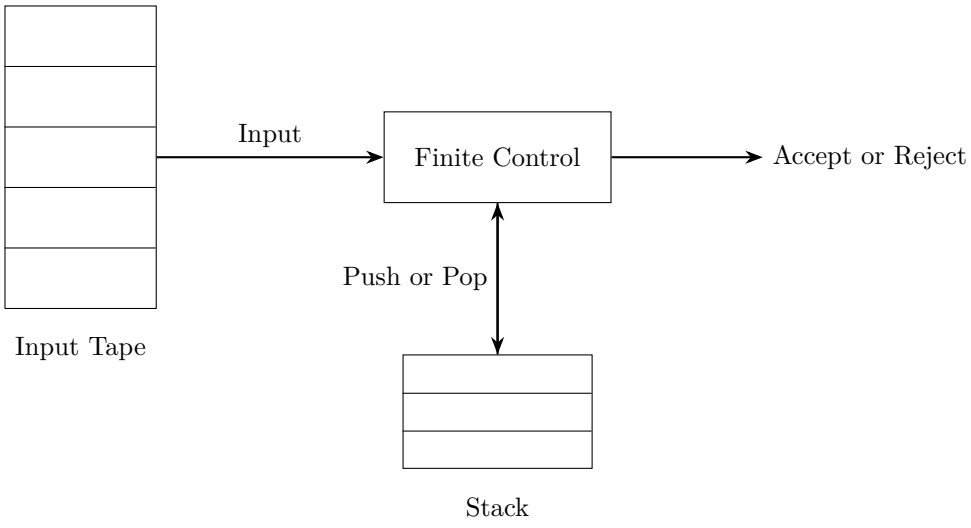


Figure 4.2: PDA

4.1.2 Formal Definition

A PDA is formally defined by the following 7-tuple:

$$P := (Q, \Sigma, \Gamma, \delta, q_0, \$, F)$$

where,

$$\begin{aligned} Q &= \text{Finite set of states} \\ \Sigma &= \text{Set of input symbols} \\ \Gamma &= \text{Finite stack alphabet} \\ \delta &= \text{Transition function} \\ q_0 &= \text{Initial state} \\ \$ &= \text{Initial stack symbol} \\ F &= \text{Set of final states} \end{aligned}$$

The transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^*$, i.e. δ takes in a triple $\delta(q, a, X)$ where,

- q is the input state in Q.
- a is either an input symbol in Σ or null (ϵ).
- X is a stack symbol in Γ[1].

The output of δ is a finite set of pairs (p, γ) where,

- p is the output state in Q.

[1] If $X = \epsilon$, then nothing is pushed onto the stack.

- γ is a string of stack symbols that replaces[2] X on top of the stack.

4.1.3 Instantaneous Description

The Instantaneous Description (ID) describes the configuration of the PDA at a given instance. ID remembers the information of the state and the stack contents at a given instance of time.

Formally, an ID is a format of triple (q, w, κ), where $q \in Q$ (finite set of states), $w \in \Sigma$ (finite set of input alphabets), and $\kappa \in \Gamma$ (finite set of stack symbols).

4.1.4 Acceptance by a PDA

In each PDA there is a stack attached. At the bottom of the stack, there is a sym,bol called the initial stack symbol, say \$. The symbol \$ remains in the stack for most operations.

A string w may be declared *accepted* by an empty stack after processing all the symbols of w, if the stack is empty after reading the rightmost input character of the string w. In mathematical notation, we can say that if $M = (Q, \Sigma, \Gamma, \delta, q_0, \$, F)$ is a PDA, the string w is declared accepted by the empty stack if

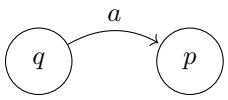
$$\exists q \in Q \, : \, (q_0, w, \$) \rightsquigarrow (q, \epsilon, \epsilon) \text{ where } w \in \Sigma^*$$

In general, we can say a string is said to be the accepted by a PDA if both the following hold:

- The string ends at a final state.
- The stack is empty.

4.1.5 Graphical Notation

The graphical notation of a PDA differs from that of an FSM.



FSM Graphical Notation

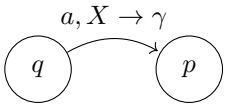


Figure 4.3: PDA Graphical Notation

A PDA can be represented by both its graphical notation or the set of transitions. Drawing the graphical notation is preferred, but Overleaf does not provide enough compile time for me to draw a diagram for every automaton (sed).

4.2 DPDA and NPDA

4.2.1 Deterministic Pushdown Automata

A PDA is said to be a Deterministic Pushdown Automaton (DPDA) if all derivations in the design give only a single move. In other words, if a PDA being in a state with a single input and a single stack symbol gives a single move, then the PDA is called a DPDA.

As an example, for $L = \{a^n b^n \,|\, n \geq 1\}$, a DPDA can be designed if the transitional functions

[2] If $\gamma = \epsilon$, the stack is popped with no replacement.

are as follows:

$$\begin{aligned}
\delta(q_0, a, \$) &\rightarrow (q_0, z_1\$) \\
\delta(q_0, a, z_1) &\rightarrow (q_0, z_1 z_1) \\
\delta(q_0, b, z_1) &\rightarrow (q_0, \epsilon) \\
\delta(q_1, b, z_1) &\rightarrow (q_1, \epsilon) \\
\delta(q_1, \epsilon, \$) &\rightarrow (q_1, \epsilon) \text{ accepted by the empty stack.} \\
\delta(q_1, \epsilon, \$) &\rightarrow (q_f, \$) \text{ accepted by the final state.}
\end{aligned}$$

In the above PDA, in a single state with a single input and a single stack symbol, there is only one move. So, the PDA is deterministic.

A CFL is said to be a *deterministic CFL* if it is accepted by a DPDA.

4.2.2 Non-Deterministic Pushdown Automata

A PDA is said to be a Non-Deterministic Pushdown Automaton (NPDA) if one of the derivations generates more than one move. If a PDA being in a state with a single input and a single stack symbol gives more than one move or any of its transitional functions, then the PDA is non-deterministic.

Consider the following example,

Q.1. Design an NPDA for accepting the string $\left\{ww^{\mathcal{R}} \,\middle|\, w \in \{a,b\}^+\right\}$.

The PDA can be defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \$, F)$, where:

$$\begin{aligned}
Q &= \{q_0, q_1, q_2, q_f\} \\
\Sigma &= \{a, b\} \\
\Gamma &= \{\$, z_1, z_2\} \\
q_0 &= q_0 \\
\$ &= \$ \\
F &= \{q_f\}
\end{aligned}$$

The transitional function will be as follows:

$$\begin{aligned}
\delta(q_0, a, \$) &\rightarrow \{(q_1, z_1\$)\} \\
\delta(q_0, b, \$) &\rightarrow \{(q_1, z_2\$)\} \\
\delta(q_1, a, z_1) &\rightarrow \{(q_1, z_1 z_1), (q_2, \epsilon)\} \\
\delta(q_1, b, z_1) &\rightarrow \{(q_1, z_1 z_1)\} \\
\delta(q_1, a, z_2) &\rightarrow \{(q_1, z_1 z_2)\} \\
\delta(q_1, b, z_2) &\rightarrow \{(q_1, z_2 z_2), (q_2, \epsilon)\} \\
\delta(q_2, a, z_1) &\rightarrow \{(q_2, \epsilon)\} \\
\delta(q_2, b, z_2) &\rightarrow \{(q_2, \epsilon)\} \\
\delta(q_2, \epsilon, \$) &\rightarrow \{(q_2, \epsilon), (q_f, \$)\} \text{ accepted by the empty stack and the final state respectively.}
\end{aligned}$$

A CFL is said to be a *non-deterministic CFL* if it is accepted by a NPDA.

4.3 Equivalence between CFG and PDA

4.3.1 Construction of a PDA from a CFG

PDA is the machine accepting a CFL, which is generated from a CFG. A PDA can be constructed from a CFG as follows:

1. Convert the CFG to GNF.
2. The start symbol S of the CFG is put to the stack by the transition

$$\delta(q_0, \epsilon, \$) \rightarrow (q_1, S\$)$$

3. For a production in the form {Non-terminal $(\mathrm{NT})_i$} $\rightarrow$ {Single Terminal (T)}{String of Non-Terminals}, the transition will be

$$\delta(q_i, \mathrm{T}, \mathrm{NT}_i) \rightarrow (q_i, \text{String of NT})$$

4. For a production in the form $\{\mathrm{NT}_i\} \rightarrow \{\text{Single T}\}$, the transition will be

$$\delta(q_i, \mathrm{T}, \mathrm{NT}_i) \rightarrow (q_i, \epsilon)$$

5. For accepting a string, two transitions are added, one for being accepted by the empty stack and one for being accepted by the final state.

Q.2. Construct a PDA that accepts the language generated by the following grammar.

$$S \rightarrow aB$$
$$B \rightarrow bA \,|\, b$$
$$A \rightarrow aB$$

First, the start symbol S is pushed onto the stack by the following production

$$\delta(q_0, \epsilon, \$) \rightarrow \{(q_1, S\$)\}$$

For the production $S \rightarrow aB$, the transition is

$$\delta(q_1, a, S) \rightarrow \{(q_1, B)\}$$

For the prouction $A \rightarrow aB$, the transition is

$$\delta(q_1, a, A) \rightarrow \{(q_1, B)\}$$

For the production $B \rightarrow bA \,|\, b$, the transition is

$$\delta(q_1, b, B) \rightarrow \{(q_1, A), (q_1, \epsilon)\}$$

So,, the PDA for the given CFG is

$$
\begin{aligned}
Q &= \{q_0, q_1, q_f\} \\
\Sigma &= \{a, b\} \\
\Gamma &= \{\$, S, A, B\} \\
q_0 &= q_0 \\
\$ &= \$ \\
F &= \{q_f\}
\end{aligned}
$$

δ transitions are defined as follows:

$$
\begin{aligned}
\delta(q_0, \epsilon, \$) &\rightarrow \{(q_1, S\$)\} \\
\delta(q_1, a, S) &\rightarrow \{(q_1, B)\} \\
\delta(q_1, a, A) &\rightarrow \{(q_1, B)\} \\
\delta(q_1, b, B) &\rightarrow \{(q_1, A), (q_1, \epsilon)\} \\
\delta(q_1, \epsilon, \$) &\rightarrow \{(q_f, \$)\} \text{ accepted by the final state.} \\
\delta(q_1, \epsilon, \$) &\rightarrow \{(q_1, \epsilon)\} \text{ accepted by the empty stack.}
\end{aligned}
$$

4.3.2 Construction of a CFG from a PDA

Consider a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, \$, F)$ which accepts a language L. A CFG $G = (V, \Sigma, P, S)$, equivalent to M can be constructed by the following rules.

Let S be the start symbol.

1. For $q_i \in Q$, add a production rule $S \rightarrow [q_0, \$, q_i]$ to P in G. If a PDA contains n states, then there will be n productions from the start symbol.
2. For the δ function $\delta(q, a, X) \rightarrow (p, \epsilon)$ where $q, p \in Q$, $a \in \Sigma$ and $X \in \Gamma$, add a production rule $[qXp] \rightarrow a$ in P.
3. For the transitional function $\delta(q, a, X) \rightarrow (p, X_1, X_2, \ldots, X_k)$ where $q, p \in Q$, $a \in \Sigma$ and $X, X_1, X_2, \ldots, X_k \in \Gamma$, then for each choice of $q_1, q_2 \ldots, q_k \in Q$, add production $[qXq_k] \rightarrow a[pX_1q_1][q_1X_2q_2] \cdots [q_{k-1}X_kq_k]$ to P in the grammar.

Refer page 403 of the reference book for a solved example (it is too lengthy).

4.4 Two-Stack PDA

A PDA is able to recognize CFLs due to the additional stack memory. However, a PDA is helpless if it needs to recognize a Context-Sensitive Language (CSL) like $\{a^n b^n c^n \mid n \geq 0\}$ because of only one auxiliary storage. This is why two (or more) stacks are used alongside a PDA to allow recognition of higher languages in the Chomsky hierarchy.

A PDA may be non-deterministic, but a two-stack PDA is *always deterministic*. A two-stack PDA is equivalent to a Turing Machine.

Chapter 5

Turing Machine

5.1 Definition

Turing Machines (TM) are abstract machines that could perform any computational process carried out by the present day's computer. The TM is the machine format for unrestricted language; all types of languages are accepted by the Turing machine.

5.2 Formal Definition

A TM is defined by the following 7-tuple:

$$(Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$$

where,

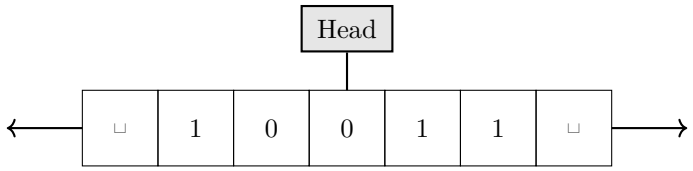
$$\begin{aligned} Q &= \text{Finite set of states} \\ \Sigma &= \text{Set of input symbols} \\ \Gamma &= \text{Finite set of allowable tape symbol} \\ \delta &= \text{Transitional function} \\ q_0 &= \text{Initial state} \\ \sqcup &= \text{A symbol of } \Gamma \text{ called } \textit{blank} \\ F &= \text{Final state} \end{aligned}$$

The transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, H\}$, i.e., from one state, by getting one input from the input tape, the machine moves to a state, writing a symbol on the tape and moves to left, right, or halts.

A string is said to be accepted by a TM if the machine halts at an accepting (final) state.

5.3 Mechanical Diagram

A TM consists of an input tape, a read-write head, and finite control. The input tape contains the input alphabets, with an infinite number of blanks at the left and the right-hand side of the input symbols. The mechanical diagram of a TM is illustrated in figure *figure*.



State: q_1

Figure 5.1: Mechanical model for a TM

The read-write head reads an input symbol from the input tape and sends it to the finite control. In the finite control, the transitional functions are written. According to the present state and the present input, a suitable transitional function is executed.

Upon execution of a suitable transitional function, the following operations are performed.

- The machine goes into some state.
- The machine writes a symbol in the cell of the input tape from where the input symbol was scanned.
- The machine moves the reading head to the left or right or halts.

5.4 Instantaneous Description

The ID of the TM remembers the following at a given instance of time:

- The contents of all the cells of the tape, theoretically infinite but usually non-blank cells, are shown.
- The cell currently being scanned by the read-write head.
- The state of the machine.

Q.1. Design a TM to accept the language

$$L = \left\{ ww^{\mathcal{R}} \,\middle|\, w \in \{a, b\}^+ \right\}$$

The transitions are given in a tabular format as follows.

State	a	b	␣	X	Y
q_0	(q_1, X, R)	(q_4, Y, R)		(q_f, X, H)	(q_f, Y, H)
q_1	(q_1, a, R)	(q_1, b, R)	$(q_2, ␣, L)$	(q_2, X, L)	(q_2, Y, L)
q_2	(q_3, X, L)				
q_3	(q_3, a, L)	(q_3, b, L)		(q_0, X, R)	
q_4	(q_4, a, R)	(q_4, b, R)	$(q_5, ␣, L)$	(q_5, X, L)	(q_5, Y, L)

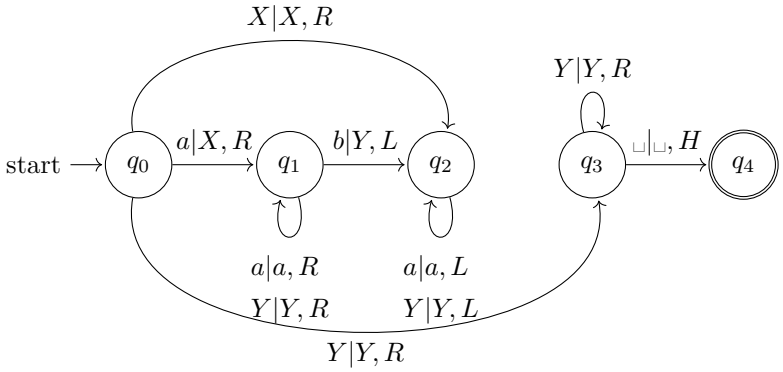
q_5		(q_6, Y, L)			
q_6	(q_6, a, L)	(q_6, b, L)			(q_0, Y, R)

5.5 Transitional Representation

The states and transitions of a TM are represented similar to other automata. The label of a transition consists of the input symbol, the symbol written on the tape and the movement of the read-write head (left, right, or halt).

Q.2. Design a TM by the transitional notation for the language $L = \{a^n b^n \mid n > 0\}$.

When the machine traverses 'a', replace that 'a' with 'X' and traverse right to find the leftmost 'b'. Replace the 'b' with 'Y' and traverse left to find the second 'a'. By this process, when n symbols of 'a' and n symbols of 'b' are traversed and replaced by 'X' and 'Y' respectively, then by getting a blank (␣) the machine halts. **(Ans.)**



Refer the Reference book for illustrated solved examples.

Acronyms

AT Automata Theory 1

CFG Context-Free Grammar 32–40, 42, 46, 47

CFL Context-Free Language 32, 33, 35–41, 45–47

CNF Chomsky Normal Form 37, 38

CSL Context-Sensitive Language 47

DFA Deterministic Finite Automaton 6–10, 12, 14, 15, 17, 22, 23, 27–29

DPDA Deterministic Pushdown Automaton 44, 45

FA Finite Automaton 5, 26, 28–31, 33

FSM Finite State Machine 5, 6, 9, 33, 42, 44

GNF Greibach Normal Form 37, 38, 46

ID Instantaneous Description 44, 49

Jojo Aaditya Joil 1

LIFO Last-In-First-Out 42

NFA Non-Deterministic Finite Automaton 10–12, 15–17, 27, 28

NPDA Non-Deterministic Pushdown Automaton 45, 46

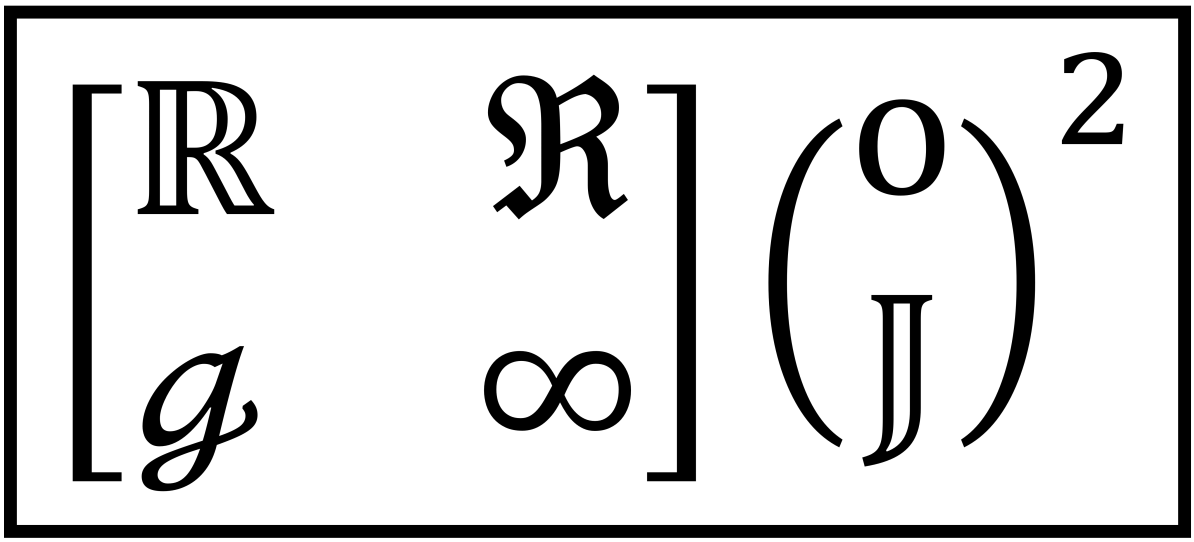
PDA Pushdown Automaton 32, 33, 37, 40, 42–47

RE Regular Expressions 23, 26–30, 41

RRG Rupak R. Gupta 1

TM Turing Machines 48–50

ToC Theory of Computation 1



The End.