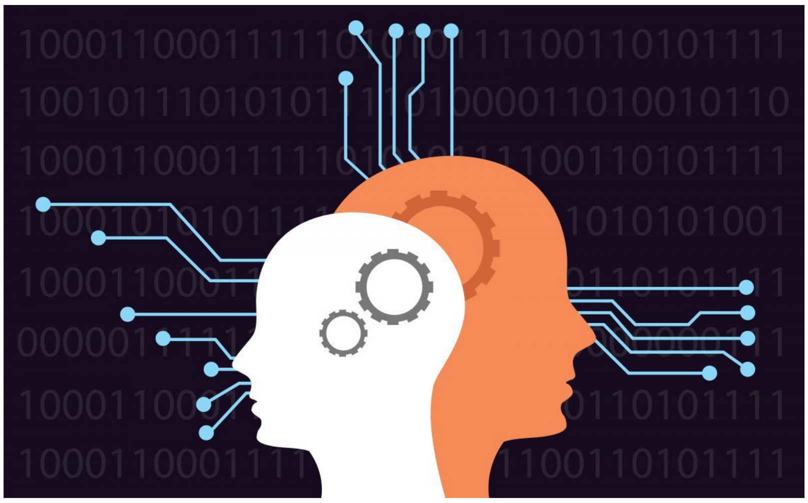
Indian Institute of Technology Bombay



# Automata Theory

**Liza Dahiya**
**Roll No.:** 190050063
Computer Science & Engineering
**Mentor:** Shrey Singhai

# Contents

# Brief Introduction

I, Liza Dahiya, began learning with the introduction of Propositional Logic and Natural Deduction since it is good to have a strong knowledge of logic in computer science before delving into the Automata theory.

After which I, formally begin with the Automata theory and describe the Finite Automaton. Various examples are included for understanding the different types of finite automaton. Then we deal with regular expressions which are used for describing regular languages for finite automaton. Then Context-Free Grammar and Pushdown Automaton are introduced which are another way for describing languages to broaden the spectrum of understanding the Automata Theory.

Final chapters are dedicated for the understanding of digital circuits before trying to code those in Intel Quartus (in VHDL) for having a sense of practical application for the theory.

# Chapter 1

# Propositional Logic

## 1.1 Introduction

**Declarative Sentences** or *propositions* are the ones which in principle can be argued as being true or false. Propositional Logic deals with such propositions. **Examples: Goldbach's conjecture-** Every even natural number $> 2$ is the sum of two prime numbers.

### Syntax

- $\neg$ : Negation
- $\vee$ : At least one of p or q is true
- $\wedge$ : Both p and q are true
- $\rightarrow$ : If this, then this

**Binding Proprieties:** $\neg$ binds more tightly than $\vee$ or $\wedge$, which binds tighter than $\rightarrow$. In the absence of parentheses, $p \rightarrow q \vee r$ is read as $p \rightarrow (q \vee r)$.

## 1.2 Natural Deduction

Natural deduction allows us to infer *conclusions* from some formulae called as *premise.* The following expression is called as *sequent*:

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$$

### 1.2.1 Rules of Natural Deduction

Rules of natural deduction allows us to conclude various formulas from some given formulas and hence provide proof for a sequent:

- **Rule for Conjugation:**
  - **Rule for and-introduction:** It allows us to conclude $\phi \wedge \psi$, given that $\phi$ and $\psi$ are concluded separately:

$$\frac{\phi \quad \psi}{\phi \wedge \psi} \wedge i$$

  - **Rules for and-elimination:** If you have a proof of $\phi \wedge \psi$ then, $\wedge e_1$ rule allows you to get a proof of $\phi$:

$$\frac{\phi \wedge \psi}{\phi}\ \wedge e_1 \qquad\qquad \frac{\phi \wedge \psi}{\psi}\ \wedge e_2$$

- **Rules for Double Negation** Intuitively, $\neg\neg\phi$ expresses same meaning as $\phi$ does. The following are rules for introduction & elimination respectively.
  - **Introduction:** $\dfrac{\phi}{\neg\neg\phi}\ \neg\neg i$
  - **Elimination:** $\dfrac{\neg\neg\phi}{\phi}\ \neg\neg e$
- **Modulus Ponens:**is the rule for eliminating-implication, popularly known as its Latin name, *Modulus Ponens.*

$$\frac{\phi \quad \phi \to \psi}{\psi} \to e$$

- **Modulus Toneus:** This rule is derived from other rules.

$$\frac{\phi \to \psi \quad \neg\psi}{\neg\phi}\ MT$$

- **Introduction implication:** In this rule, we open a box by assuming $\phi$, and then applying laws of natural deduction, we derive $\psi$. Finally, when we close a box, we have $\phi \to \psi$.

$$\frac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \psi\end{array}}}{\phi \to \psi} \to i$$

- **Rules for Disjunction**
  - **Introduction:** We may conclude $\phi \vee \psi$ given either of $\phi$ or $\psi$ is concluded before:

$$\frac{\phi}{\phi \vee \psi}\ \vee i_1 \qquad\qquad \frac{\psi}{\phi \vee \psi}\ \vee i_2$$

  - **Elimination:** The following steps need to be followed while implementing elimination rule for disjunction:
    * First, we assume $\phi$ is true and have to come up with a proof of $\chi$.
    * Next, we assume $\psi$ is true and need to give a proof of $\chi$ as well.
    * Given these two proofs, we can infer $\chi$ from the truth of $\phi \vee \psi$, since our case analysis above is exhaustive.

  These steps are summarised in the formula below:

$$\frac{\phi \vee \psi \quad \boxed{\begin{array}{c}\phi\\ \vdots\\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi}\ \vee e$$

- **Negation**
  - **Bottom Elimination:** The fact that $\perp$ can prove anything is encoded in our calculus by the proof rule bottom-elimination:

$$\frac{\bot}{\phi}\ \bot e$$

- **Not- Elimination:** The fact that $\bot$ itself represents a contradiction is encoded by the proof rule not-elimination:

$$\frac{\phi \quad \neg\phi}{\bot}\ \neg e$$

### 1.2.2 Proof by Contradiction

If we assume $\neg\phi$ (while opening a box) and then obtain $\bot$ i.e. contradiction in the box, we are entitled to deduce $\phi$.

### 1.2.3 Law of Excluded Middle (LEM)

The law just says that $\phi \vee \neg\phi$ is true whatever $\phi$ is, true or false.

$$\frac{}{\phi \vee \neg\phi}\ LEM$$

## 1.3 Semantics

**Definition 1.1.** *A valuation of a formula $\phi$ is an assignment of each propositional atom in $\phi$ to a truth value (i.e. True or False).*

$\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ iff whenever $\phi_1, \phi_2, \ldots, \phi_n$ evaluates to T (true) so does $\psi$. $\vDash$ is called **semantics entails**.

### 1.3.1 Soundness of Propositional Logic

**Theorem 1.** *Let $\phi_1, \phi_2, \ldots, \phi_n, \psi$ be propositional logic formulas. If $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid, then $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds.*

### 1.3.2 Completeness of Propositional Logic

**Theorem 2.** *If $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is, then there exists a natural deduction proof for the sequent $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds.*

In the process of proving completeness of propositional logic, we use definitions and theorems, which are stated below:

**Definition 1.2.** *A formula of propositional logic $\phi$ is called a tautology, iff it evaluates T under all its valuations, i.e. iff $\vDash \phi$*

**Theorem 3.** *If $\vDash \eta$ holds, then $\vdash \eta$ is valid. In other words, if $\eta$ is valid, then $\eta$ is theorem.*

### 1.3.3 Combining soundness and completeness

Let $\phi_1, \phi_2, \ldots, \phi_n, \psi$ be propositional logic formulas. This $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ hols **iff** $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds.

## 1.4 Normal Forms

**Definition 1.3.** *$\phi$ and $\psi$ are two propositional logic. We say that $\phi$ and $\psi$ are semantically equivalent iff $\phi \vDash \psi$ and $\psi \vDash \phi$. We write this as $\phi \equiv \psi$*

**Definition 1.4.** *A **literal** L is either an atom p or the negation of an atom $\neg p$. A formula C is in **conjunctive normal form (CNF)** if it is a conjugation of clauses, where each clauses D is*

***disjunction*** *of literals.*

$$L ::= p|\neg p$$
$$D ::= L|L \vee D$$
$$C ::= D|D \wedge C$$

**Examples:**

- $(\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$: This is in CNF.
- $(\neg(q \vee p) \vee r) \wedge (q \vee r)$: This is not in CNF.

**Proposition 1.4.1.** *Let $\phi$ be a formula of propositional logic. Then $\phi$ is satisfiable i.e. there is a valuation where $\phi$ evaluates to $T$, iff $\neg\phi$ is not valid.*

### 1.4.1 The CNF Algorithm

The CNF algorithm computes CNF as an output for a given input $\phi$. The following are the three requirements that CNF needs to satisfy:

1. CNF terminates for all formulas of propositional logic as input;
2. for each such input, CNF outputs an equivalent formula; and
3. all output computed by CNF is in CNF.

The algorithm involves three steps:

#### IMP_FREE($\phi$)

First task is to remove all of the sub formulas in the form $\psi \to \eta$ by $\neg\psi \vee \eta$. This process has to be done recursively, for there might be implications in $\psi$ or $\eta$. Hence, finally the entire formula is made implication free.

#### NNF(IMP_FREE($\phi$))

Negation Normal Forms (NNF) are the formulas which only *negate* atoms. NNF is applied after making sure that the formula is implication free. Following is the code for applying NNF on a formula $\phi$:

```
def function (NNF(($\phi$)):
    #precondition: $\phi$ is implication free
    #postcondition: NNF($\phi$) computes NNF of $\phi$
    begin function
    case
    $\phi$ is literal: return $\phi$
    $\phi$ is $\neg\neg\phi_1$: return NNF($\phi_1$)
    $\phi$ is $\phi_1 \wedge \phi_2$: return NNF($\phi_1$) ∧ NNF($\phi_2$)
    $\phi$ is $\phi_1 \vee \phi_2$: return NNF($\phi_1$) ∨ NNF($\phi_2$)
    $\phi$ is $\neg(\phi_1 \wedge \phi_2)$: return NNF($\neg\phi_1$) ∨ NNF($\neg\phi_2$)
    $\phi$ is $\neg(\phi_1 \vee \phi_2)$: return NNF($\neg\phi_1$) ∧ NNF($\neg\phi_2$)
    end case
    end function
```

After applying both of the above, we obtain NNF(IMPL_FREE($\phi$)) $= \phi'$; and now $\phi' \equiv \phi$. We now apply CNF($\phi'$), which is discussed in the subsection below:

#### CNF($\phi$)

We begin by defining the function for computing CNF of $\phi$:

```
def function CNF($\phi$):
    #precondition:$\phi$ is implication-free and in NNF
    #postcondition: CNF($\phi$) computes equivalent CNF for $\phi$
    begin function
    case
    $\phi$ is literal: return $\phi$
    $\phi$ is $\phi_1 \wedge \phi_2$: return CNF($\phi_1$) ∧ CNF($\phi_2$)
    $\phi$ is $\phi_1 \vee \phi_2$: return DISTR(CNF($\phi_1$), CNF($\phi_2$))
    end case
    end function
```

Here in case 3, when $\phi$ is in the form of $\phi_1 \vee \phi_2$, if we call CNF recursively on $\phi_i$ to get $\eta_i$; but in this case we cannot return $\eta_1 \vee \eta_2$, since the formula will not be in CNF form and the 3rd requirement will not be fulfilled.

Hence we introduce a new function DISTR($\eta_1$,$\eta_2$) which uses the distributivity of conjugations and disjunctions, and whose code is provided below:

```
def function DISTR($\eta_1, \eta_2$):
    #precondition:$\eta_1, \eta_2$ are in CNF
    #postcondition: DISTR($\eta_1, \eta_2$) computes a CNF for $\eta_1 \vee \eta_2$
    begin function
    case
    $\eta_1$ is $\eta_{11} \wedge \eta_{12}$: return DISTR($\eta_{11}, \eta_2$) ∧ DISTR($\eta_{12}, \eta_2$)
    $\eta_2$ is $\eta_{21} \wedge \eta_{22}$: return DISTR($\eta_1, \eta_{21}$) ∧ DISTR($\eta_1, \eta_{22}$)
    otherwise: return $\eta_1 \vee \eta_2$
    end case
    end function
```

## 1.5 Horn Formula

**Definition 1.5.** *A **Horn formula** is a formula of $\phi$ of propositional logic if it can be generated as an instance of H in this grammar:*

$$\begin{gathered} P ::= \bot | \top | p \\ A ::= P | A \wedge P \\ C ::= A \to P \\ H ::= C | C \wedge H \end{gathered}$$

**Example:**

- $(p \wedge q \wedge s \to p) \wedge (q \wedge r \to p) \wedge (p \wedge s \to s)$: This is a Horn Formula.
- $(p \wedge q \wedge s \to \bot) \wedge (\neg q \wedge r \to p) \wedge (\top \to s)$: This is not a Horn Formula, since $(\neg q \wedge r)$ is not a conjugation of atoms, $\top$ or $\bot$.

### 1.5.1 Satisfiability of Horn Formula

The following algorithm is used to check satisfiability of a Horn Formula.

```
def function HORN($\phi$):
    #precondition:$\phi$ is a Horn formula
    #postcondition: HORN($\phi$) decides satisfiability of $\phi$
    begin function
    mark all occurances of $\top$ in $\phi$
    while there is a conjuct $P_1 \ldots P_{ki} \wedge P'$ of $\phi$ such that all $P_j$          are marked
     but $P'$ is not if mark $P'$
    end while
    if $\bot$ is marked then return 'unsatisfiable'
    else return 'satisfiable'
    end function
```

**Theorem 4.** *The algorithm HORN is correct for the satisfiability decision problem of Horn formulas and has no more than n +1 cycles in its whilestatement if n is the number of atoms in $\phi$. In particular, HORN always terminates on correct input.*

Hence, according to the above theorem this algorithm terminates on all Horn formulas $\phi$ as input and that its output (= its decision) is always correct.

## 1.6 Resolution

Resolution is a technique used to check if a formula in CNF is satisfiable.
Let $C_1$ and $C_2$ are two clauses. Assume $A \in C_1$ and $\neg A \in C_2$. Then the clause:

$$\text{R}=(C_1 - A) \cup (C_2 - \neg A) \text{ is a resolvent of } C_1 \text{ and } C_2.$$

### 1.6.1 3 Rules of Resolution

1. Let $G$ be any formula. Let $F$ be the CNF formula resulting from the CNF algorithm applied to $G$. Then $G \vdash F$.
2. Let $F$ be a formula in CNF, and let $C$ be a clause in $F$. Then $F \vdash C$.
3. Let $F$ be a formula in CNF. Let $R$ be a resolvent of two clauses of $F$. Then $F \vdash R$

### 1.6.2 Completeness

There is an $m$ such that $Res^m(F) = Res^{m+1}(F)$, where $F$ is any formula in CNF. This is denoted by $Res^*(F)$.

**Theorem 5.** *Let $\psi$ be any formula, convert it into CNF form, say $\zeta$, then $\phi$ is satisfiable* ***iff*** $\emptyset \notin Res^*(F)$ .

# Chapter 2

# Finite Automata

## 2.1 Introduction

A finite automaton is a formal system that stores only finite amount of information. It can not be used when indefinite amount of data is required. Finite automaton can be used to tell whether two automaton perform the same task or not or whether there exists any smaller automaton that does the same thing.

- Information is stored in the form of ***states***, $Q$.
- State changes in response to ***inputs***.
- Rules that tell how the state changes in response to inputs are called ***transitions***, $\delta$.
- In the automaton, we designate some special states to be ***accepting states***. If an input word moves forward the automaton to an accepting state, we say that it is ***accepted***. Otherwise, ***rejected***.

### 2.1.1 Alphabets

- The domain of strings is called ***alphabet*** usually denoted by $\Sigma$.
- The elements of alphabet are called as ***letters***.
- Inputs are sequences of letters, usually called *words* (or string).

  $\Sigma^0$ is a string of length 0. There is only one single empty string, denoted by $\epsilon$.

$$\Sigma^0 \triangleq \epsilon$$

- $\Sigma^*$ is used to specify the union of all lengths of alphabets.

$$\Sigma^* \triangleq \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$$

- $\Sigma^+$ doesn't contain the empty string.

### 2.1.2 Languages

The set of strings accepted by an automaton A is the ***language*** of that automaton. Let $\Sigma$ be an alphabet. Then a subset of $\Sigma^*$ is called language. **Example:**

1. $L_{eq} = \{0^n 1^n | n \geqslant 0\}$

2. $L_{prime} = w|w$ is binary encoding of a prime number
   Consider a problem to check if a number n is a prime number? Let binary(n) be the binary encoding of n, then we need to check:

$$binary(n) \in L_{prime}$$

## 2.2 Deterministic Finite Automaton

The word deterministic means that there is a unique transition going on for every state and input symbol. A deterministic finite automaton(DFA) A is a five-tuple represented by:

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

where,

- $Q$ is a finite set of states.
- $\Sigma$ is a finite set of input symbols.
- $\delta$ is a transition function.
- $q_0 \in Q$ is the start/initial state, and
- $F \subseteq Q$ is a set of accepting states.

### 2.2.1 Transition Function

It takes up two inputs: a state and an input symbol and returns the next state. $\delta(q, a)$= the state that the DFA goes to when it is in state q and an input a is received.

- Note: always a next state – add a **dead state** if no transition.

### 2.2.2 Graphs

DFA can be represented as graphs with nodes represented the states $q$ and the transition $\delta$ is represented by the loops.
**Example:**

In the figure shown, state $q_1$ represent string that doesn't have any 11's and doesn't end in 1, while state $q_2$ represents string ending in 1 but with no consecutive 11's. And finally state $q_3$ represents string with consecutive 11's and hence a **dead state** since on any further input no transition can be made to any other state since we've already detected at least one pair of consecutive 11's.

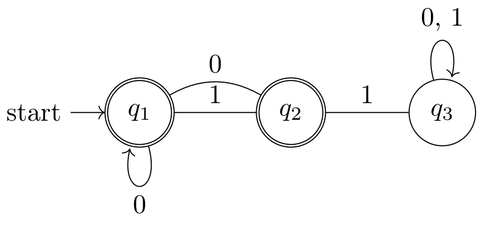


Figure 2.1: Strings ending with no 11's

### 2.2.3 Transition table

The graph represented in the above section can also be presented as a transition table with rows represented states and columns represented input symbols and the final states being represented as $*$.

| | 0 | 1 |
|---|---|---|
| $*$A | A | B |
| $*$B | A | C |
| C | C | C |

### 2.2.4 Extending Transition Function

Let $(Q, \delta, \Sigma, q_0, F)$ be a DFA. An extended transition function also takes two arguments. The first argument is a state q and the second argument is a string w. It returns a state just like the transition function discussed in the previous sections. It can be defined as the state in which the FA ends up, if it begins in state q and receives string x of input symbols. Let $\hat{\delta} : QX\Sigma^* \rightarrow Q$ be defined as follows:

$$\hat{\delta}(q, \epsilon) \triangleq q$$

$$\hat{\delta}(q, wa) \triangleq \delta(\hat{\delta}(q, wa), a)$$

### 2.2.5 Regular languages

A language L is regular if it is the language accepted by some DFA.
**Example of a Regular language**

- $L_2 = \{w|w \text{ in } \{0,1\}^*\}$ and w is viewed as binary integer is divisible by 23.
- $L_2 = \{w|w \text{ in } \{0,1\}^*\}$ and w is viewed as **reverse** of binary integer is divisible by 23.

**Theorem 6.** *The reverse of a regular language is also regular.*

**Example of a Non-Regular language**

- $L_1 = \{0^n 1^n | n \geqslant 0\}$
- $L_2 = \{w|w \text{ in } \{(,)\}^*\}$ and w is **balanced**

  Balanced parentheses are those sequences of parentheses that can appear in an arithmetic expression. Ex: (),()(),(()), etc.

### 2.2.6 Worked Out Example

**Find a DFA for** $w|$ **01s occur somewhere**. In the figure above, $q_0$ is the initial start state,

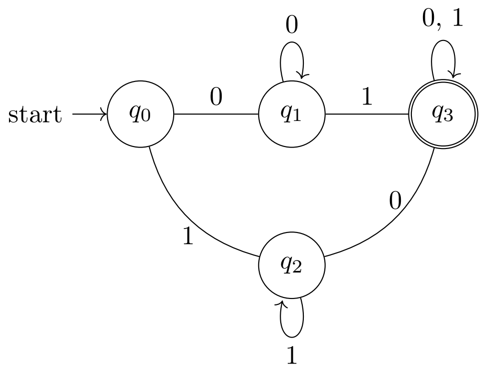


Figure 2.2: Strings with 01's somewhere

the automaton moves to the state $q_1$ on receiving an input symbol 0 and to state $q_2$ on receiving an input symbol 1. We mark $q_3$ as the accepting state & when $q_1$ will receive an input 1, it'll go to the final state $q_3$ as we can say that 01 was present somewhere in the string of input symbols. Other transitions can also be explained similarly.

## 2.3 Non-Deterministic Automaton

A non-deterministic finite automaton has the ability to be in several states at once. Transitions from a state on an input symbol can be to any set of states. Start in one of the start state. Accept if any sequence of choices leads to a final state. The formal symbols used in NFA are quite similar to DFA with the only difference being the **transition function**, $\delta(q, a)$ is a **set** of states rather than a single state as in DFA. Formally, it can be written as:

$$\delta : QX\Sigma \to \mathfrak{P}(Q),$$

where $\mathfrak{P}(Q)$ is the set of all subsets of Q.

- A string w is accepted by an NFA if $\delta(q_0, w)$ contains at least one final state.
- The language of the NFA is the set of strings it accepts.
- For any NFA there is a DFA that accepts the same language.

### 2.3.1 Example of a NFA

This example represents the previous example with a NFA. Observe that two transitions are possible on an input 0 while being in state $q_0$ and it is much easier to represent the example with a NFA with fewer states.

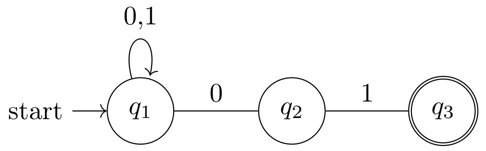


Figure 2.3: Strings with 01's somewhere

**Theorem 7.** *For a DFA $A = (Q, \delta, \Sigma, q_0, F)$, there is an NFA $A = (Q, \delta', \Sigma, q_0, F)$ such that $L(A) = L(A')$.*

### 2.3.2 Subset Construction

Considering the reverse theorm as stated in the previous sub section:

**Theorem 8.** *For a NFA $A = (Q, \delta, \Sigma, q_0, F)$, there is an DFA $A = (Q, \delta', \Sigma, q_0, F)$ such that $L(A) = L(A')$.*

The proof of above theorem is called as **subset construction.** Given an NFA with states $Q$, inputs $\Sigma$, transition function $\delta_N$, state state $q_0$, and final states $F$, construct equivalent DFA with:

- States $2^Q$
- Inputs $\Sigma$
- Start state $q_0$
- Final states = all those with a member of $F$.

The transition function $\delta_D$ is defined by: $\delta_D(q_1, \ldots, q_n, a)$ is the union of all $i = 1, \ldots, k$ of $\delta_N(q_i, a)$

## 2.4 NFA's $\epsilon$ transition

We can allow state-to-state transitions on $\epsilon$ input. These transitions are done spontaneously, without looking at the input string.

### 2.4.1 Example of $\epsilon$-transition

In the example given below, observe if we are in a state E, we can directly reach the final accepting state D without the requirement of any input symbols using the $\epsilon$ transition while two alternate ways to achieve same also exists. Similarly, if we're in state C we can directly reach to state D on $\epsilon$ transition.

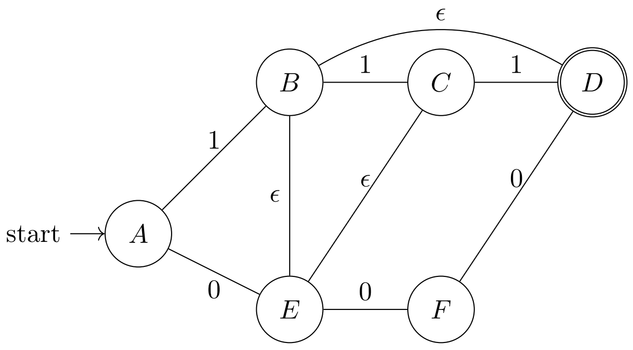


Figure 2.4: $\epsilon$ transition NFA

### 2.4.2 Closure of States

$CL(q)$ = set of states you can reach from state q following only arcs labeled $\epsilon$. **Example:** $CL(A) = A; CL(E) = B, C, D, E$. Closure of a set of states = union of the closure of each state. Language of an $\epsilon$-NFA is the set of strings w such that $\delta(q_0, w)$ contains a final state.

### 2.4.3 Equivalence

Every NFA can be represented as an $\epsilon$-NFA and vice-versa. Similarly, any DFA can be represented as a NFA and vice-versa. We can conclude that all three types of automaton accept same languages i.e. the regular languages.

# Chapter 3

# Regular Expressions

Regular expressions describe regular languages by an algebra. Regular expressions are widely used to find patterns in texts. All languages provide libraries to handle regular expressions. If $E$ is a regular expression, then $L(E)$ is the language it defines. Operations used:

- **Union**, consider language 01, 10, 0, $\epsilon$

$$L(01 + 10 + 0 + \epsilon) = 01, 10, 0, \epsilon$$

- The **concatenation** of languages $L$ and $M$ is denoted $LM$. For example: $01, 111, 1000, 01 = 0100, 0101, 11100, 11101, 1000, 1001$

**Definition 3.1.** *For regular expressions $R_1$ and $R_2$, let regular expression $R_1R_2$ define the language consists of words obtained by concatenating a word from L1 with a word from L2.*

$$L(R_1R_2) \triangleq \{ww'|w \in L(R_1)w' \in L(R_2)\}$$

Now we can this concatenation idea to any number of regular expressions:

**Definition 3.2.** *Let $R_1, R_2, \ldots, R_n$ be regular expressions:*

$$L(R_1, R_2, \ldots, R_n) \triangleq L(R_1, (R_2, \ldots, R_n))$$

- If L is a language, then $L^*$, the **Kleene star** or just "star," is the set of strings formed by concatenating zero or more strings from $L$, in any order.

$$L^* = \epsilon \cup L \cup LL \cup LLL \cup \ldots$$

  **Example:** $\{0, 10\}^* = \{\epsilon, 0, 10, 00, 010, 100, 1010, \ldots\}$

**Definition 3.3.** *For regular expression $R$, let $R^*$ define the language consist of words obtained by concatenating words from $L(R)$ some number of times.*

$$L(R^*) \triangleq \cup L(R, ...R_n)$$

.

## 3.1 Formal Definition

L is being defined recursively as:

- **Basic 1:** If $a$ is any symbol, then **a** is a RE, and $L(\mathbf{a}) = \{a\}$.
- **Basic 2:** $\epsilon$ is RE, then $L(\epsilon) = \{\epsilon\}$

- **Basic 3:** $\varnothing$ is RE, then $L(\varnothing) = \varnothing$
- **Induction 1:** If $E_1$ and $E_2$ are regular expressions, then $E_1 + E_2$ is a regular expression, and $L(E_1 + E_2) = L(E_1) \cup L(E_2)$.
- **Induction 2:** If $E_1$ and $E_2$ are regular expressions, then $E_1E_2$ is a regular expression, and $L(E_1E_2) = L(E_1)L(E_2)$
- **Induction 3:** If $E$ is a RE, then $E^*$ is a RE, and $L(E^*) = (L(E))^*$

**Representations of Regular Languages**

Representations can be formal or informal. Example

- (formal): represent a language by a RE or FA defining it.
- (informal): a logical or prose statement about its strings:
  - "The set of strings consisting of some number of 0's followed by the same number of 1's."

### 3.1.1 Precedence of Operators

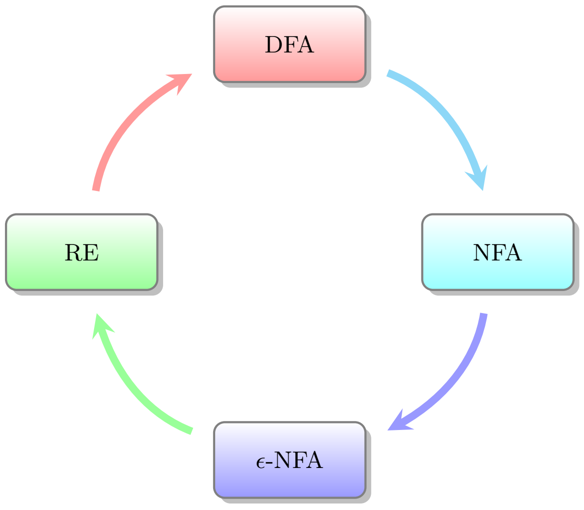
Parentheses may be used wherever needed to influence the grouping of operators. Order of precedence is $*$ (highest), then concatenation, then $+$ (lowest). **Example:**

$L((0 + 10) * (\epsilon + 1))$ = all strings of 0's and 1's without two consecutive 1's.

$*$ operator first create a language of strings ending all in 0's. Then using concatenation, we get two types of strings again, one ending with all in 0's i.e. due to $\epsilon$ and other ending all in 1's i.e. due to 1. Hence there will be no case when 11's will be present consecutively.

### 3.1.2 Equivalence of RE's and Finite Automaton

- For every Regular Expressions, there is a finite automaton that accepts the same language. To show this, we will have to pick the most powerful automaton type: the $\epsilon$-NFA for proving so.
- For every finite automaton, there is a RE defining its language. To show this, we will have to pick the most restrictive type of finite automaton: the DFA.



### 3.1.3 Identities and Annihilators

- $\varnothing$ is the identity for $+$.

- $\epsilon$ is the identity for concatenation.
- $\varnothing$ is annihilator for concatenation.

## 3.2 Applications of Regular Expressions

### 3.2.1 UNIX Regular Expressions

UNIX, from the beginning, used regular expressions in many places, including the "grep" command. **Grep** = "Global (search for a) Regular Expression and Print." Most UNIX commands use an extended RE notation that still defines only regular languages.

**UNIX RE Notation:**

- $[a_1, a_2, \ldots, a_n]$ is shorthand for $a_1 + a_2 + \ldots + a_n$
- **Ranges** indicated by first-dash-last and brackets. Example:
  - $[a - z]$ = "any lower-case letter," $[a - zA - Z]$ = "any letter."
- Dot = "any character."
- | is used for union instead of +.
- But + has a meaning: "one or more of." $E+ = EE*$. Example:
  - $[a - z]+$ = "one or more lowercase letters.
- ? = "zero or one of." $E? = E + \epsilon$ Example:
  - $[ab]?$ = "an optional a or b".

### 3.2.2 Text Processing

But the RE for such strings is easy: .∗ing where the dot is the UNIX "any". Even an NFA is easy an represented in the figure below with $q_4$ as final accepting state.

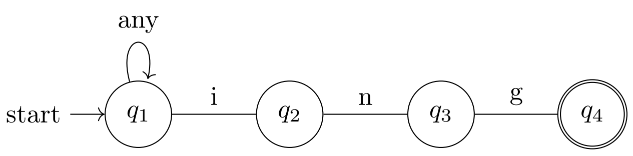


Figure 3.1: NFA for "Ends in ing"

### 3.2.3 Lexical Analysis

The first thing a compiler does is break a program into **tokens** = sub-strings that together represent a unit. Example:

- identifiers, reserved words like "if," meaningful single characters like ";" or "+", multi-character operators like "<=".

Using a tool like Lex or Flex, one can write a regular expression for each different kind of token. **Example:** In UNIX notation, identifiers are something like $[A - Za - z]$ $[A - Za - z0 - 9]*$. Each RE has an associated action. **Example:** return a code for the token found.
A language class is a set of regular languages. **Language classes** have two important kinds of properties: Decision properties and Closure properties.

## 3.3 Decision Properties of Regular Languages

A decision property for a class of languages is an algorithm that takes a formal description of a language (e.g., a DFA) and tells whether or not some property holds. **Example**: Is language L empty?

### 3.3.1 Emptiness Property

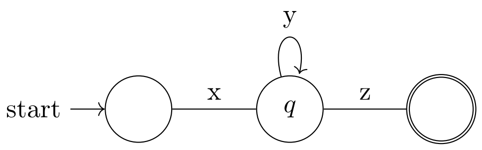
Given a regular language, does the language contain any string at all?
Assume representation is DFA. Compute the set of states reachable from the start state. If at least one final state is reachable, then the language would be non empty else empty.

### 3.3.2 Infiniteness Problem

Is a given regular language infinite?
Start with a DFA for the language. **Key idea**: if the DFA has n states, and the language contains any string of length n or more, then the language is infinite. Otherwise, the language is surely finite. Limited to strings of length n or less.

**Proof of infiniteness:**

If an n-state DFA accepts a string w of length n or more, then there must be a state that appears twice on the path labeled w from the start state to a final state. This is called the **Pigeon Hole Principle**. Let w = xyz: Then x$y^i$z is in the language for all $i > 0$. Since y is not $\epsilon$, we see an



infinite number of strings in L.
**Second Key Idea:** if there is a string of length > n (= number of states) in L, then there is a string of length between n and 2n-1. Test for membership all strings of length between n and 2n-1. If any are accepted, then infinite, else finite.

**The Pumping Lemma**

**Theorem 9.** *For every regular language L There is an integer n which is the number of states of DFA for L, such that For every string w in L of length $\geqslant n$. We can write w = xyz such that:*

1. $|xy| \geqslant n$
2. $|y| > 0$
3. *For all* $i \geqslant 0$, $xy^iz$ *is in L.*

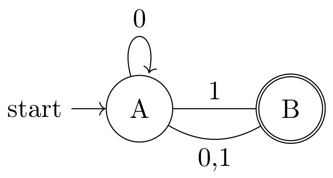
### 3.3.3 Equivalence Property

Equivalence problem is to see that if given two languages L and M, are they equivalent i.e. L=M? For this, lets introduce **Product DFA** which is constructed from DFA's of L and M.

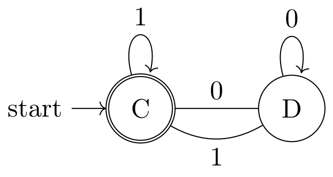
1. Let DFA's of L and M have set of states Q and R. Then their product DFA will have states Q $\times$ R i.e. pairs [q,r] with q in Q and r in R.
2. The transition state is represented as $\delta([q, r], a)$ as $[\delta_L(q, a), \delta_M(r, a)]$ with $\delta_L$ and $\delta_M$ are transition states of L and M respectively.

**Example:**

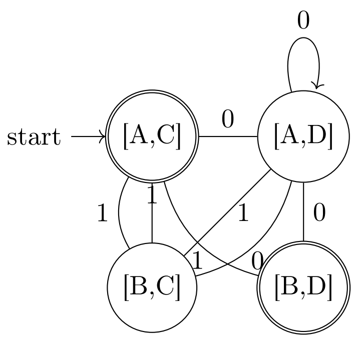
1. **First DFA**



2. **Second DFA**



3. **Final Product DFA:**



### Equivalence Algorithm

Make the final states of the product DFA be those states [q, r] such that exactly one of q and r is a final state of its own DFA. Thus, the product accepts w iff w is in exactly one of L and M. L = M if and only if the product automaton's language is empty.
In the example above the final states are [A,C] and [B,D]. State [A,C] is reachable since it is the start state but state [B,D] is not reachable because no arrow it is pointing inward. Hence, there is no string that is accepted by first and not by second since for any string to be accepted by first automaton it needs to reach B. But, since [A,C] is also a final state, an empty string will be accepted by second automaton and not by first i.e. it will distinguish between these two. And it will violate condition written in above paragraph. Hence, both the automaton are different.

### Containment Property

Given regular languages L and M, is M $\supseteq$ L? Algorithm also uses the product automaton. How do you define the final states [q, r] of the product so its language is empty iff M $\supseteq$ L i.e. the product automaton's language is empty. It will be when q is in the final state and r is not in the final state. Let's consider the previous example only, [B,D] is that state where q will be in final state and r will not be in final state. But as we have seen that [B,D] is not reachable. Hence the language of product automaton is **empty** and hence L is a subset of M.

### The Minimum-State DFA for a Regular Language

Construct a table with all pairs of states. If you find a string that distinguishes two states (takes exactly one to an accepting state), mark that pair. Algorithm is a recursion on the length of the shortest distinguishing string.
Whenever you find two states that are indistinguishable i.e. the go to same states on any input, we can replace all such indistinguishable states by a single representative state $q$. When all such indistinguishable states are combined together, we can remove states that are not reachable from the start state.
We can also verify that this DFA obtained is minimum of all the DFA's possible.

## 3.4 Closure Properties

A closure property of a language class says that given any language in the class, an operation (e.g., union) produces another language in the same class. **Example**: the regular languages are obviously closed under union, concatenation, and (Kleene) closure.

1. **Union Property:** If L and M are regular languages, so is L ∪ M.
2. **Concatenation Property:** If L and M are regular languages, so is LM.
3. **Kleen Star Property:** If L is a regular language, so is L*.
4. **Intersection Property:** If L and M are regular languages, so is L ∩ M. Proof is done using the product automaton corresponding to the regular languages L and M, such that final state is only one which consists of final states of A and B (automaton for L and M respectively).
5. **Difference Property:** If L and M are regular languages, so is L-M=strings in L but not in M. Again, this can be proved by constructing a product DFA with final states as final states in A (DFA for RL L) and not in B (DFA for RL M).
6. **Complementation:** The complement of a language L (with respect to an alphabet Σ such that Σ* contains L) is Σ* - L.
7. **Reversal Property:** Given language L, $L^R$ is the set of strings whose reversal is in L.
   - **Example:** If L=0,0011,0101, then $L^R$ =0,1100,1010.]
   - Let E be the regular expression for regular language L. And if E is:
     - $F + G$, then $E^R = F^R + G^R$,
     - $FG$, then $E^R = G^R\ F^R$,
     - $F^*$, then $E^R = (F^R)^*$

### 3.4.1 Homomorphism

A homomorphism on an alphabet is a function that gives a string for each symbol in that alphabet.
**Example:** Let h(0)=ab and h(1)=$\epsilon$, then we can extent it into strings as h(01010) = ababab.

#### Closure under Homomorphism

If L is a regular language, and h is a homomorphism on its alphabet, then h(L) = {h(w) | w is in L} is also a regular language. **Example:** Let h(0)=ab and h(1)=$\epsilon$, L is given as $01^* + 10^*$. Then we can say that h(L)= $ab\epsilon^* + \epsilon(ab)^*$. As we know that $\epsilon * = \epsilon$. h(L) will be simplified as $ab + (ab)^*$ which is equal to $(ab)^*$ as $ab$ is already contained in it, so its union will return $(ab)^*$ only.

### 3.4.2 Inverse Homomorphism

Let h be a homomorphism and L a language whose alphabet is the output language of h. $h^{-1}$(L)= {w | h(w) is in L}. **Example:** Let L ={abab,babab}. For first term the $h^{-1}$ is L(1*01*01*). Closure property also follows for inverse homomorphism.

# Chapter 4

# Context-Free Grammars

A context-free grammar is a notation for describing languages. **Example:** CFG for $\{0^n1^n|n \geqslant 1\}$ will be:

- S→01 i.e. basis is 01 in the language.
- S→0S1 i.e. the induction step; so if we say that w is the language then so is 0w1.

## 4.1 Formal Definitions

- **Terminals** = symbols of the alphabet of the language being defined.
- **Variables** = non terminals = a finite set of other symbols, each of which represents a language.
- **Start symbol** = the variable whose language is the one being defined.
- A **production** has the form variable (head) → string of variables and terminals (body).
  - A, B, C, ... and also S are variables.
  - a, b, c,... are terminals.
  - ..., X, Y, Z are either terminals or variables.
  - ..., w, x, y, z are strings of terminals only.
  - $\alpha$, $\beta$, $\gamma$, ... are strings of terminals and/or variables.
- **Derivations:** We derive strings in the language of a CFG by starting with the start symbol, and repeatedly replacing some variable A by the body of one of its productions. **Example:** S → 01 and S → 0S1. Then S => 0S1 => 00S11 => 000111. **Iterative derivation** is represented by $=>^*$ i.e. zero or more derivation steps.
- Any string of variables and/or terminals derived from the start symbol is called a **sentential form**.
- If G is a CFG, then L(G) is the **language** of G, is {w| S $=>^*$ w}. **Example:** If G has production, S → 01 and S → 0S1 then L(G) = $\{0^n1^n|n \geqslant 1\}$.
- **Context-Free Languages:** A language that is defined by some CFG is called a context-free language.

**Example:** Here is a formal CFG for $\{0^n1^n|n \geqslant 1\}$.

- Terminal = {0,1}
- Variables = {S}

- Start Symbol = S
- Production=
  - S→01 i.e. basis is 01 in the language.
  - S→0S1 i.e. the induction step; so if we say that w is the language then so is 0w1.

## 4.2 Backus-Naur Form

Grammar for programming languages is written in BNF form. **Variables** are expressed by words inside triangular brackets < ... > and **Terminals** are often multi character strings indicated by boldface or underline.

- Symbol ::= is used instead of → and | is used instead of "or". **Example:** S → 0S1 | Sis used instead of S → 01 and S → 0S1.
- Symbol ... is used to say "one or more".
  - **Example:** <digit> ::= 0|1|2|3|4|5|6|7|8|9 and hence <unsigned integer> ::= <digit> ....
  - **Translation:** Replace $\alpha\ldots$ with a new variable A and productions A → $A\alpha|\alpha$.
- Surround one or more symbols by [...] to make them **optional**.
  - **Example**: <statement> ::= if <condition> then <statement> [; else <statement>]
  - **Translation**: replace $[\alpha]$ by a new variable A with productions A → $\alpha|\epsilon$.
- **Grouping:** Use {...} to surround a sequence of symbols that need to be treated as a unit. **Example:** <statement list> ::= <statement> [{;<statement>} ...]

**Example:** L → S[{;S} ...]

- Let A = {;S} then we can say that L→ S [A ...].
- Let B = A ... $|\epsilon$; then L → SB.
- Finally, C → AC | A i.e. B→ C $|\epsilon$. C stands for A....

## 4.3 Leftmost and Rightmost Derivative

### Leftmost Derivation

The derivation done by removing the leftmost variable in the string to reach a string consisting of terminals only is **leftmost derivation**. Say wA $\alpha$ $=>_{lm}$ w$\alpha\beta$ if w is a string of terminals only and A → $\beta$ is a production.

### Rightmost Derivation

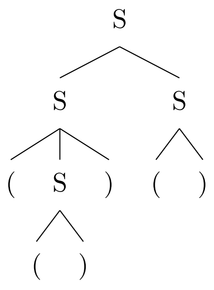
The derivation done by removing the rightmost variable in the string to reach a string consisting of terminals only is **rightmost derivation**. Say $\alpha$ Aw$=>_{rm}$ $\alpha\beta$w if w is a string of terminals only and A → $\beta$ is a production.

### Example: Balanced-parentheses grammar

S → SS | (S) | (). Here, S should be balanced as indicated by the balanced parentheses () and if we put a balanced string inside the parentheses then also it'll be balanced and finally we can also say that if we concatenate two balanced strings, they'll also be balanced. By following either derivation we'll end with (())().

## 4.4 Parse Trees

Parse trees are trees labeled by symbols of a particular CFG. **Leaves:** labeled by a terminal or $\epsilon$. **Interior nodes:** labeled by a variables. Lets draw a parse tree for: S → SS | (S) | ().



### 4.4.1 Yield of a Parse Tree

The concatenation of the labels of the leaves in left-to-right order i.e. in the order of a pre-order traversal is called the yield of the parse tree. **Example:** in the previous example, the yield is (())().

### 4.4.2 Generalisation

We sometimes talk about trees that are not exactly parse trees, but only because the root is labeled by some variable A that is not the start symbol. Basically, *parse trees with root A*.

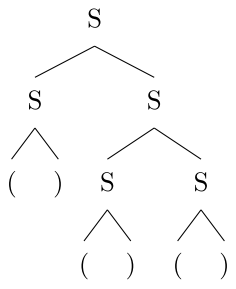
### 4.4.3 Parse tree and derivations

If a parse tree exists then we can show the existence of left and right derivations and for any derivation in general and similarly vice-versa.

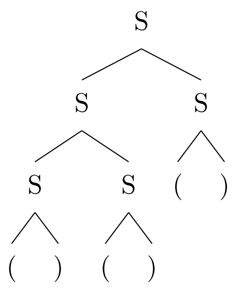
## 4.5 Ambiguous Grammar

A CFG is **ambiguous** if there is a string in the language that is the yield of two or more parse trees. **Example:** for S → SS | (S) | (), the two parse trees can be:

1. **First Parse Tree**



2. **Second Parse Tree**



If there are two different parse trees, they must produce two different leftmost derivations by the construction given in the proof. **Conversely**, two different leftmost derivations produce different parse trees by the other part of the proof. Likewise for rightmost derivations.

Thus, equivalent definitions of "**ambiguous grammar**" is: There is a string in the language that has two different leftmost derivations/ rightmost derivations.

**For the balanced-parentheses language, here is another CFG, which is unambiguous.**

1. B $\rightarrow$ (BR $|\epsilon$
2. R $\rightarrow$ ) | (RR.

with B as a start symbol which derives balanced strings and R which derives balanced strings with one more right parentheses. Construct a unique leftmost derivation for a given balanced string of parentheses by scanning the string from left to right. **Example:** Lets convert into unambiguous form for (())(). The following steps will be followed as a part of leftmost derivation:

- B
- (RB
- ((RRB
- (()RB
- (())B
- (())(RB
- (())()B
- (())()

The last step is done by considering the derivation of B into epsilon.

### 4.5.1 LL1 Grammar

A grammar such, where you can always figure out the production to use in a leftmost derivation by scanning the given string left-to-right and looking only at the next one symbol is called LL(1) which stands for **"Leftmost derivation, left-to-right scan, one symbol of look ahead.** Most programming languages have LL(1) grammar because they're never ambiguous.

### 4.5.2 Inherent Ambiguity

Unfortunately, certain CFL's are **inherently ambiguous**, meaning that every grammar for the language is ambiguous. **Example:** The language $\{0^i1^j2^k|$ i = j or j = k$\}$ is inherently ambiguous. **Intuitively**, at least some of the strings of the form $0^n1^n2^n$ must be generated by two different parse trees, one based on checking the 0's and 1's, the other based on checking the 1's and 2's.

## 4.6 Normal forms

### 4.6.1 Variables that derive nothing

**Example:** S $\rightarrow$ AB, A $\rightarrow$ aA | a, B $\rightarrow$ AB. Although A derives all strings of a's, B derives no terminal strings as any production of B will always include B. Thus, S derives nothing, and the language is empty as its production is AB and has a non terminating variable B.

**Algorithm for eliminating variables will be:**

1. Discover all variables that derive terminal strings.
2. For all other variables, remove all productions in which they appear in either the head or body.

**Example:** S → AB | C, A → aA | a, B → bB, C → c

1. **Basic:** A and C are discovered initially as a and c are terminals and A → a and C → c.
2. **Induction:** S will be discovered as it produces C.
3. Nothing else can be discovered.

Finally, S → C, A → aA | a, C → c

### 4.6.2 Unreachable Symbols

Another way a terminal or variable deserves to be eliminated is if it cannot appear in any derivation from the start symbol. Easy inductions in both directions show that when we can discover no more symbols, then we have all and only the symbols that appear in derivations from S.

**Algorithm:** Remove from the grammar all symbols not discovered reachable from S and all productions that involve these symbols.

### 4.6.3 Eliminating useless variables

A symbol is useful if it appears in some derivation of some terminal string from the start symbol. Otherwise, it is useless. Eliminate all useless symbols by:

1. Eliminate symbols that derive no terminal string.
2. Eliminate unreachable symbols.

### 4.6.4 Epsilon Production

We can almost avoid using productions of the form A → $\epsilon$ (called $\epsilon$-productions). The problem is that $\epsilon$ cannot be in the language of any grammar that has no $\epsilon$-productions.

**Theorem 10.** *If L is a CFL, then L-{$\epsilon$} has a CFG with no $\epsilon$-productions.*

To eliminate $\epsilon$-productions, we first need to discover the nullable symbols = variables A such that A =>* $\epsilon$. **Example:** S → AB, A → AB |$\epsilon$, B → A. A is nullable since it derives $\epsilon$ and B is also nullable as it derives A and finally S is also nullable as it derives AB.

#### Eliminating Epsilon Productions

**Example:** S → ABC, A → aA |$\epsilon$ , B → bB |$\epsilon$, C → $\epsilon$.
New Grammar will be: S → ABC | AB | BC | AC | A | B | C. And since C is a nullable variable we'll remove all those productions which include C. And we'll be left with S → AB | A | B, A → aA | a and B → bB | b.

### 4.6.5 Unit Productions

A unit production is one whose body consists of exactly one variable. These productions need to be eliminated.

### 4.6.6 Cleaning Up a Grammar

**Theorem 11.** *If L is a CFL, then L-{$\epsilon$} has a CFG with*

1. *No useless symbols.*
2. *No $\epsilon$-productions.*
3. *No unit productions.*

*i.e., every body is either a single terminal or has length >= 2.*

**Algorithm:** Perform the following steps in order:

1. Eliminate $\epsilon$-productions.
2. Eliminate unit productions.
3. Eliminate variables that derive no terminal string.
4. Eliminate variables not reached from the start symbol.

## 4.7 Chomsky Normal Form

A CFG is said to be in Chomsky Normal Form if every production is of one of these two forms:

1. A $\rightarrow$ BC i.e. body is two variables
2. A $\rightarrow$ a i.e. body is a terminal.

**Steps to Follow for achieving a CNF:**

1. **Step 1:** "Clean" the grammar, so every body is either a single terminal or of length at least 2.
2. **Step 2:** For each body $\neq$ a single terminal, make the right side all variables:
   - For each terminal a create new variable $A_a$ and production $A_a \rightarrow$ a.
   - Replace a by $A_a$ in the bodies of length greater than equal to 2.

   **Example:** Consider production A $\rightarrow$ BcDe. Here we will replace c by $A_c$ and e by $A_e$.
3. **Step 3:** Break right sides longer than 2 into a chain of productions with right sides of two variables. **Example:** Recall A $\rightarrow$ BCDE is replaced by A $\rightarrow$ BF, F $\rightarrow$ CG, and G $\rightarrow$ DE.

# Chapter 5

# Pushdown Automaton

The Pushdown Automaton is an automaton equivalent to the CFG in language-defining power. Only the non-deterministic PDA defines all the CFL's.

1. A finite set of states (Q).
2. An input alphabet ($\Sigma$).
3. A stack alphabet ($\Gamma$).
4. A transition function ($\delta$).
5. A start state ($q_0$ in Q).
6. A start symbol ($Z_0$ in $\Gamma$).
7. A set of finite states (F $\subseteq$ Q).

**Conventions**

- a, b, . . . $\epsilon$ are used as input symbols.
- . . . X, Y, Z are stack symbols.
- . . ., w, x, y, z are input symbols.
- $\alpha, \beta, \ldots$ are string of stack symbols.

## 5.1 Transition Function

Takes three arguments:

- A state, in Q
- An input which is either a symbol in $\Sigma$ or $\epsilon$.
- A stack symbol in $\Gamma$.

$\delta$(q, a, Z) is a set of zero or more actions of the form (p, $\alpha$), where p is a state and $\alpha$ is a string of stack symbols.

If $\delta$(q, a, Z) contains (p, $\alpha$) among its actions, then one thing the PDA can do in state q, with a in front of the input symbol and Z on top of the stack is: change its state to p; Remove a from top of the input symbol; and Replace Z on the top of stack by $\alpha$.

**Example:**

Design a PDA to accept $\{0^n1^n | n \geqslant 1\}$.
**The states:**

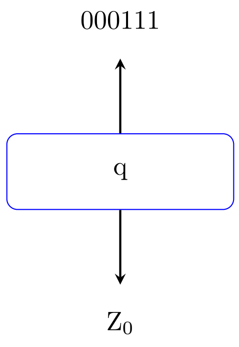
- q = start state. We are in state q if we have seen only 0's so far.
- p = we have seen at least one 1 and may now proceed only if the inputs are 1's.
- f = final state; accept.

**The stack symbols:**

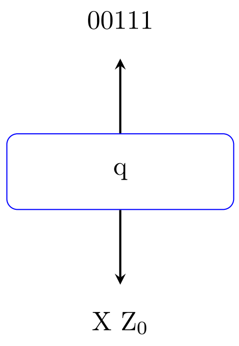
- $Z_0$ = start symbol. Also marks the bottom of the stack, so we know when we have counted the same number of 1's as 0's.
- X = marker, used to count the number of 0's seen on the input. Whenever we see a 1, we pop out one X.

Below are the steps to show the example of how a PDA works:

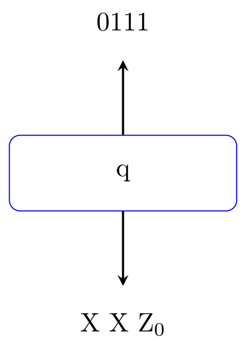
1. Initially, the input string is 000111 and the stack has the start symbol $Z_0$.



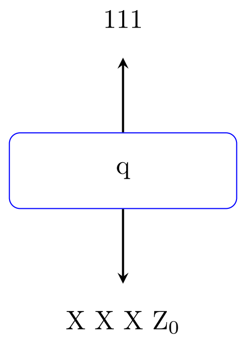
2. As, 0 was present at the front of the input string, we push X into the stack & it becomes the head of the stack inputs.



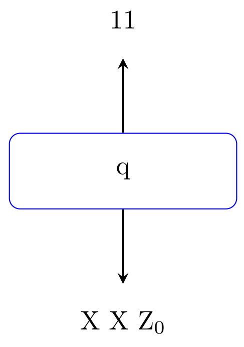
3. Again on encountering a 0, we push a X into the stack.



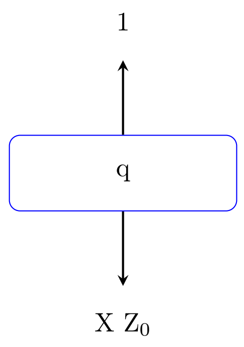
4. Now, we see a 1 as a front symbol of the input string, and we remove/ pop out X from the stack.



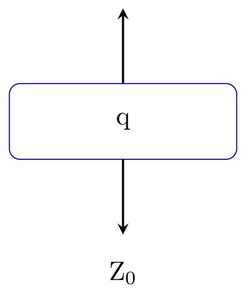
5. Again, we see 1 and pop out another X from the stack.



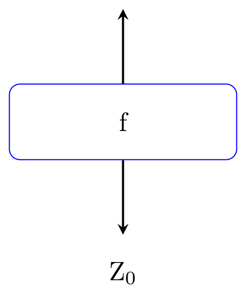
6. Finally, we're left with only with 1 as the input string and we pop a X again.



7. Now out input string finishes and even our stack is left only with the initial symbol.



8. Hence, the state finally reaches to the final accepting state f.



We can formalize the pictures just seen with an **instantaneous description (ID)**. An ID is a triplet $\delta$(q, w,$\alpha$) where q is the current state, w is the remaining input and $\alpha$ is the stack content, top at the left. To say that ID I can become ID J in one move of the PDA, we write I $\vdash$ J. We can extent $\vdash$ to $\vdash^*$ to represent zero or more moves.
**Example:** (q, 000111, $Z_0$) $\vdash$ (q, 00111, $XZ_0$) $\vdash$ (q, 0111, $XXZ_0$) $\vdash$ (q, 111, $XXXZ_0$) $\vdash$ (p, 11, $XXZ_0$) $\vdash$ (p, 1, $XZ_0$) $\vdash$ (p, $\epsilon$, $Z_0$) $\vdash$ (f, $\epsilon$, $Z_0$). Also, 0001111 is not accepted, because the input is not completely consumed.

## 5.2 Language of PDA

The common way to define the language of a PDA is by final state. If P is a PDA, then L(P) is the set of strings w such that ($q_0$, w, $Z_0$) $\vdash^*$ (f, $\epsilon$, $\alpha$) for final state f and any $\alpha$.
Another language defined by the same PDA is by empty stack. If P is a PDA, then N(P) is the

set of strings w such that ($q_0$, w, $Z_0$) $\vdash^*$ (q, $\epsilon$, $\epsilon$) for state q. **Both languages are equivalent ways of representing each other**.

## 5.3 Determinititic PDA

To be deterministic, there must be **at most one** choice of move for any state q, input symbol a, and stack symbol X. In addition, there must not be a choice between using input $\epsilon$ or real input.

## 5.4 Equivalence of PDA and CFG

CFG's and PDA's are both useful to deal with properties of the CFL's.

### Converting a CFG to PDA

- Let L = L(G).
- Construct PDA P such that N(P) = L.
- P has:
  - One state q.
  - Input symbols = terminals of G.
  - Stack symbols = all symbols of G.
  - Start symbol = start symbol of G.

At each step, P represents some **left-sentential form** (step of a leftmost derivation). If the stack of P is $\alpha$, and P has so far consumed x from its input, then P represents left-sentential form x $\alpha$. At empty stack, the input consumed is a string in L(G).

### Rules for Transition Function of P

1. **Type 1 rules:** $\delta$(q, a, a) = (q, $\epsilon$).
   This step does not change the LSF represented, but "moves" responsibility for a from the stack to the consumed input.
2. **Type 2 rules:** A $\vdash$ $\alpha$ is a production of G, then $\delta$(q, $\epsilon$, A) contains (q, $\alpha$).
   Guess a production for A, and represent the next LSF in the derivation.

w is in N(P) if and only if w is in L(G).

### From a PDA to a CFG

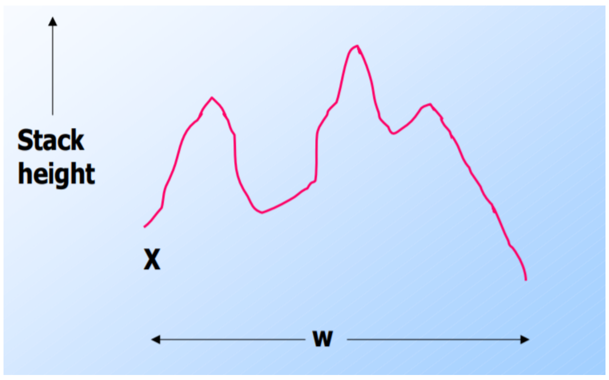


Figure 5.1: Plot showing popping of X

- Now, assume L = N(P).
- We'll construct a CFG G such that L = L(G).
- **Intuition:** G will have variables [pXq] generating exactly the inputs that cause P to have the net effect of popping stack symbol X while going from state p to state q.
- This variable generates all and only the strings w such that (p, w, X) $\vdash^*$ (q, $\epsilon$, $\epsilon$).
- P never gets below this X while doing so
- Each **production** for [pXq] comes from a move of P in state p with stack symbol X.

**Productions of G can in general case be concluded as:**

- Suppose $\delta$ (p, a, X) contains (r, $Y_1$,...$Y_k$) for some state r and k > 3.
- Generate family of productions [pXq] $\rightarrow$ a[r$Y_1s_1$] [$s_1Y_2s_2$] ... [$s_{k-2}Y_{k-1}s_{k-1}$][$s_{k-1}Y_k$q]

**Example:**

$\delta$ (p, a, X) contains (r, YZ) for some state r and symbols Y and Z. Now, P has replaced X by YZ. To have the net effect of erasing X, P must erase Y, going from state r to some state s, and then erase Z, going from s to q. Hence, [pXq] $\rightarrow$ a[rYs][sZq]

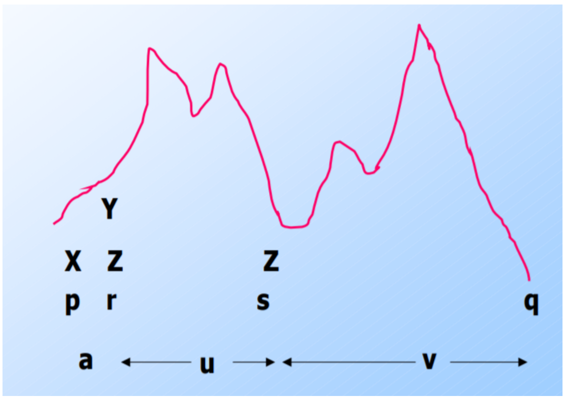


Figure 5.2: Plot showing popping of XYZ

## 5.5 The Pumping Lemma

Simple intuition would be that we can always find two pieces of any sufficiently long string to "pump" in tandem. **That is:** if we repeat each of the two pieces the same number of times, we get another string in the language.

**Theorem 12.** *For every context-free language L, there is an integer n, such that For every string z in L of length $\geqslant$ n. There exists z = uvwxy such that:*

1. $|vwx| \leqslant n$
2. $|vx| > 0$
3. *For all* $i \geqslant 0$, $uv^iwx^iy$ *is in L;*

### 5.5.1 Informal Proof:

We will proof this by considering a CNF grammar with m varibles and take n = $2^m$ and z be length greater or equal to n. Any parse tree which yield of z must have path of at least m+2 length or more (as in limiting case it'll resemble a binary tree which will have maximum of $2^m$ children) From the previous statement it is clear that the parse tree will have at least m+1 variables and since our grammar has only m variables as stated in the beginning, we can find two nodes with the same label, say A.

V and x are the portions of the yield of the yellow tree that precede and follow W respectively in the figure. Let u and y be the portions of z that precede v and follow x respectively. Let's look at the yellow. Since the path shown is as long as any other. And that path has at most m+1 variables. We know by the lemma 1 that we just proved, that the yield of the yellow plus purple is no longer than 2 to the power m, or n.

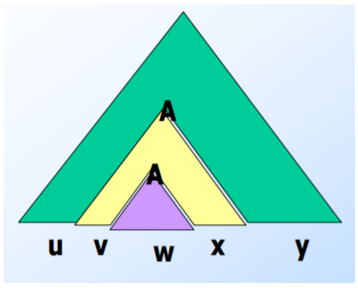


Figure 5.3: Parse Tree (only the bottom branches)

Moreover, once we have a variable automaton path there are no epsilon productions so we must generate from this variable at least one terminal. That is all we need to conclude that either V or X or both have length at least one.

**Example:** We can prove that z = $0^n10^n10^n$ is not a context-free language.

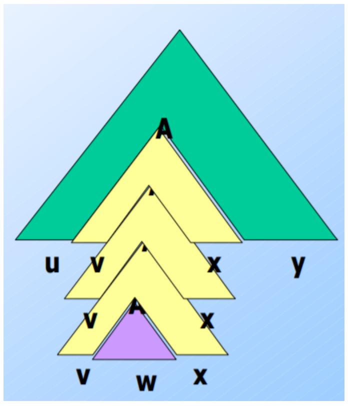
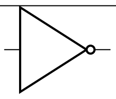
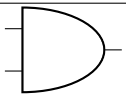
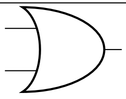
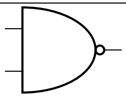
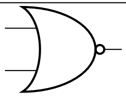
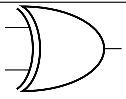


Figure 5.4: Plot showing pumming thrice

Chapter 6

# Digital Circuits

## 6.1 Logical Operations

| Operation | Gate | Truth Table | Notation |
|---|---|---|---|
| NOT |  | A: 0, 1; Y: 1, 0 | $Y = \overline{A}$ |
| AND |  | A B Y: 0 0 0; 1 0 0; 0 1 0; 1 1 1 | $Y = A.B$ |
| OR |  | A B Y: 0 0 0; 1 0 1; 0 1 1; 1 1 1 | $Y = A + B$ |
| NAND |  | A B Y: 0 0 1; 1 0 1; 0 1 1; 1 1 0 | $Y = \overline{A.B}$ |
| NOR |  | A B Y: 0 0 1; 1 0 0; 0 1 0; 1 1 0 | $Y = \overline{A + B}$ |
| XOR |  | A B Y: 0 0 0; 1 0 1; 0 1 1; 1 1 0 | $Y = A\overline{B} + \overline{A}B$ |

### 6.1.1 Sum of Products (SOP)

Sum of product form is a form of expression in Boolean algebra in which different product terms of inputs are being summed together. This product is not arithmetical multiply but it is Boolean logical AND and the Sum is Boolean logical OR.

**Example:**

X = $X_1 + X_2 + X_3 + X_4 = \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\overline{C} + AB\overline{C}$

| A | B | C | $X_1$ | $X_2$ | $X_3$ | $X_4$ | X |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

This form is called the standard sum-of-products form, and each individual term (consisting of all three variables) is called a **"minterm"**.

## 6.2 Product of sums (POS)

The short form of the product of the sum is POS, and it is one kind of Boolean algebra expression. In this, it is a form in which products of the dissimilar sum of inputs are taken, which are not arithmetic result & sum although they are logical Boolean AND & OR correspondingly.

**Example:**

X = $(A + B + C)(\overline{A} + B + C)(\overline{A} + \overline{B} + C)$.
This form is called the standard product-of-sums form, and each individual term (consisting of all three variables) is called a **"maxterm."**

### 6.2.1 The X condition

This is also called as **I don't care condition**. We'll understand this with the help of an example. I want to design a box (with inputs A, B, C, and output S) which will help in scheduling my appointments.

- A → I am in town, and the time slot being suggested for the appointment is free.
- B → My favourite player is scheduled to play a match (which I can watch on TV).
- C → The appointment is crucial for my business.
- S → Schedule the appointment.

The following truth table summarizes the expected functioning of the box.

| A | B | C | S |
|---|---|---|---|
| 0 | X | X | 0 |
| 1 | 0 | X | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Note that we have a new entity called X in the truth table. X can be 0 or 1 (it does not matter) and is therefore called the "don't care" condition. Don't care conditions can often be used to get a more efficient implementation of a logical function.

## 6.3 Karnaugh Maps

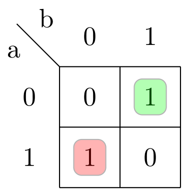
A Karnaugh map ("K-map") is a representation of the truth table of a logical function. A K-map can be used to obtain a **"minimal" expression** of a function in the sum-of-products form or in the product-of-sums form. A "minimal" expression has a minimum number of terms, each with a minimum number of variables. (For some functions, it is possible to have more than one minimal expressions, i.e., more than one expressions with the same complexity.) It can be implemented with fewer gates.

**The 1's can be enclosed by a rectangle in each case.**

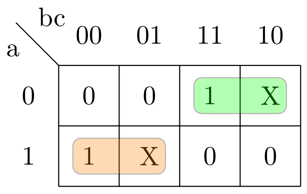
**Minimal:** smallest number of terms, smallest number of variables in each term.
$\rightarrow$ smallest number of rectangles containing 2k 1's, each as large as possible.

**Examples:**

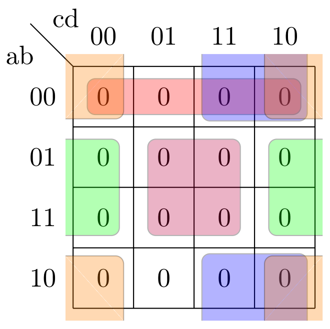
We begin by explaining the smallest representation of a K-map. We mark the 1s by red and green color in the figure. It is a way to represent the NOR-Gate i.e. $\overline{A+B}$.



Now the example below is the representation of $BC\overline{A}+A\overline{BC}$. We have two "I don't care" situations and we mark them by assuming to be 1.



In this final example we rather mark all the 0's which are all here and different colors represent different ways in which they're marked.



## 6.4 Adder circuit

**Adder circuit** is a combinational digital circuit that is used for adding two numbers. A typical adder circuit produces a sum bit (denoted by S) and a carry bit (denoted by C) as the output.

Typically adders are realized for adding binary numbers but they can be also realized for adding other formats like BCD (binary coded decimal, XS-3 etc. Besides addition, adder circuits can be used for a lot of other applications in digital electronics like address decoding, table index calculation etc.

### 6.4.1 Half Adder

**Half adder** is a combinational arithmetic circuit that adds two numbers and produces a sum bit (S) and carry bit (C) as the output. If A and B are the input bits, then sum bit (S) is the X-OR of A and B and the carry bit (C) will be the AND of A and B. From this it is clear that a half adder circuit can be easily constructed using one X-OR gate and one AND gate. Half adder is the simplest of all adder circuit, but it has a major disadvantage. The half adder can add only two input bits (A and B) and has nothing to do with the carry if there is any in the input. So if the input to a half adder have a carry, then it will be neglected it and adds only the A and B bits. That means the binary addition process is not complete and that's why it is called a half adder.

**Truth table of Half Adder**

| A | B | $C_0$ | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**Circuit Diagrams**

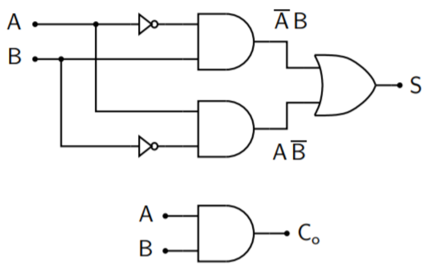


Figure 6.1: Implementation 1

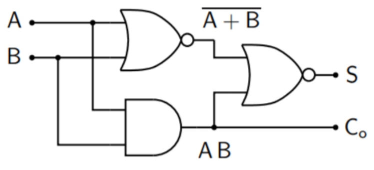


Figure 6.2: Implementation 2

### 6.4.2 Full Adder

Full Adder is the adder which adds three inputs and produces two outputs. The first two inputs are A and B and the third input is an input carry as $C_{in}$. The output carry is designated as $C_0$ and the normal output is designated as S which is SUM.

**Truth table**

| A | B | $C_{in}$ | $C_0$ | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

## 6.5 Multiplexer

The multiplexer, shortened to "MUX" or "MPX", is a combinational logic circuit designed to switch one of several input lines through to a single common output line by the application of a control signal. Multiplexers operate like very fast acting multiple position rotary switches connecting or controlling multiple input lines called "channels" one at a time to the output.

**Truth Table**

| $S_1$ | $S_0$ | Z |
|---|---|---|
| 0 | 0 | I0 |
| 0 | 1 | I1 |
| 1 | 0 | I2 |
| 1 | 1 | I3 |

**Board Diagram**

On any input one of the I0-I3 get matched with the output Z.

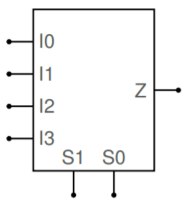


Figure 6.3: MUX Board Diagram

**8-to-1 type MUX**

The board described in above example is a 4-to-1 type it can even be 8-to-1 type as shown in figure below.

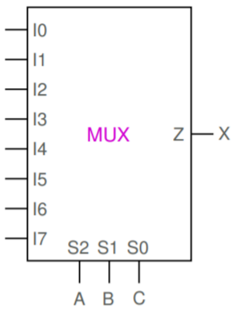


Figure 6.4: 8-to-1 MUX

**Circuit Diagram**

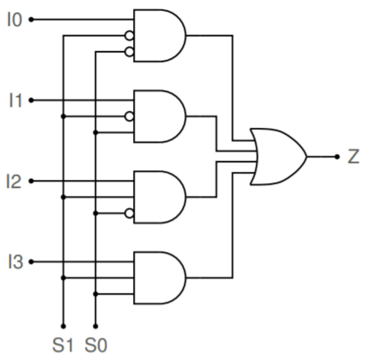


Figure 6.5: MUX Circuit Diagram

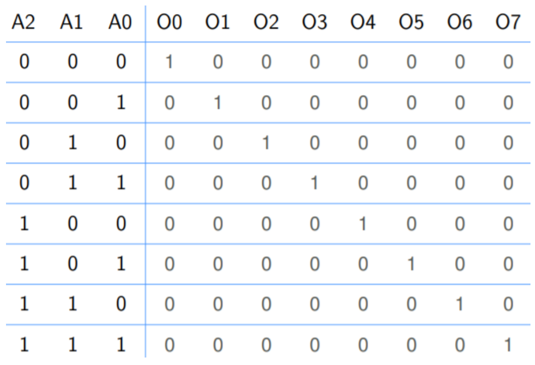
**The VHDL code for multiplexer is given in the next chapter.**

## 6.6 Decoders

Decoder is a combinational circuit that has ‘n’ input lines and maximum of $2^n$ output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled.

### Truth Table

Below is the truth table image for 3-to-8 type decoder (didn’t code because of more columns & complicated structure).



| A2 | A1 | A0 | O0 | O1 | O2 | O3 | O4 | O5 | O6 | O7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Figure 6.6: Truth table

### Board Diagram

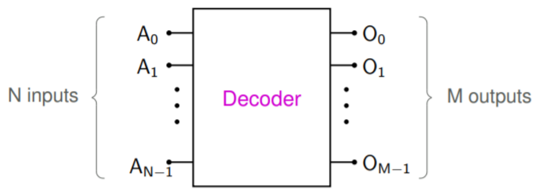


Figure 6.7: Decoder Board Diagram

# Chapter 7

# VHDL

VHDL is a structured hardware description language for writing logic circuits. The V is short for yet another acronym: VHSIC or **Very High-Speed Integrated Circuit** and the HDL stands for **Hardware Description Language**. The nice thing about VHDL is that the level of detail is unambiguous due to the rich syntax rules associated with it. In other words, VHDL provides everything that is necessary in order to describe any digital circuit. Likewise, a digital circuit/system is any circuit that processes or stores digital information. Second, having some type of circuit model allows for the subsequent simulation and/or testing of the circuit. The VHDL model can also be translated into a form that can be used to generate actual working circuits.

## 7.1 Terminology

#### Stimulation

*Stimulation* is the prediction of the behaviour of a design:

- VHDL provides many features suitable for the stimulation of digital design circuits.
- *Functional stimulation* approximates the behaviour of a hardware design by assuming that all outputs change at the same time.
- *Timing Stimulation* predicts the exact behaviour of a hardware design.

#### Synthesis

*Synthesis* is the transition of a design into a netlist file that describes the structure of a hardware description language.

- VHDL was not designed for the purpose of synthesis.
- Not all VHDL statements are synthesizable.

#### FPGA

A **Field Programmable Gate Array**, or FPGA, is a semiconductor device that comprises of logic blocks which are programmed to execute a specific set of functions. These programmable logic blocks are connected to each other with the help of an interconnect matrix.Hardware descriptive language key concepts, describing a hardware circuit implementation that your tool chain will eventually interpret and synthesize into FPGA logic cells. So whether it's schematic based or hardware descriptive language based, the tool has the ability to synthesize this description into logic cells.

**Invariants**

VHDL is not case sensitive and not even sensitive white spaces. Comments in VHDL begin with the symbol "−−".

## 7.2 Design Units

### 7.2.1 Entity

The VHDL entity construct provides a method to abstract the functionality of a circuit description to a higher level. It provides a simple wrapper for the lower-level circuitry. This wrapper effectively describes how the black box interfaces with the outside world. Since VHDL describes digital circuits, the entity simply lists the various inputs and outputs of the underlying circuitry. In VHDL terms, the black box is described by an entity declaration.

### 7.2.2 Standard Libraries

The VHDL language as many other computer languages, has gone through a long and intense evolution. Among the most important standardization steps we can mention are the release of the IEEE Standard 1164 package as well as some child standards that further extended the functionality of the language. In order to take advantage of the main implementable feature of VHDL you just need to import the two main library packages i.e. IEEE.std_logic_1164.all and IEEE.numeric_std.all.

### 7.2.3 Architecture

The VHDL entity declaration, introduced before, describes the interface or the external representation of the circuit. The architecture describes what the circuit actually does. In other words, the VHDL architecture describes the internal implementation of the associated entity. As you can probably imagine, describing the external interface to a circuit is generally much easier than describing how the circuit is intended to operate. This statement becomes even more important as the circuits you are describing become more complex.

### 7.2.4 Signals & Variables

In VHDL there are several object types. Among the most frequently used we will mention the **signal** object type, the **variable** object type and the **constant** object type. The signal type is the software representation of a wire. The variable type, like in C or Java, is used to store local information. The constant is like a variable object type, the value of which cannot be changed.

## 7.3 Programming Paradigm

**Signal Assignment Operator <=**

Algorithmic programming languages always have some type of assignment operator. In C or Java, this is the well-known "=" sign. In these languages, the assignment operator signifies a transfer of data from the right-hand side of the operator to the left-hand side. VHDL uses two consecutive characters to represent the assignment operator: "<=". This combination was chosen because it is different from the assignment operators in most other common algorithmic programming languages. The operator is officially known as a **signal assignment operator** to highlight its true purpose.

## 7.4 Logic Gates using VHDL

This and further sections will contain the code VHDL code for designing digital circuits. I will not include all the codes here as it is present in the GitHub Repository[1]that I've created for this project.

[1]Automata Theory- Digital Circuits

### 7.4.1 Or Gate

In the example shown below I've tried to code the **or gate** with the inputs as x and y and output as z.

```
library ieee;
use ieee.std_logic_1164.all;

entity or_gate is
port(x in std_logic;
     y in std_logic;
     z out std_logic
);
end or_gate;

architecture arch of or_gate is
begin
        process(x, y)
        begin
        if ((x='0') and (y='0')) then
                z <= '0';
        else
                z <= '1';
        end if;
        end process;

end arch;

architecture beh of or_gate is
begin
        z <= x or y
end beh
```

### 7.4.2 Xor Gate

Here, in the example shown below I've tried to code the **xor gate** with the inputs as x and y and output as z.

```
library ieee;
use ieee.std_logic_1164.all;

entity xor_gate is
port(x in std_logic;
     y in std_logic;
     z out std_logic
);
end xor_gate;

architecture arch of xor_gate is
begin
        process(x, y)
        begin
        if (x/=y) then
                z <='1';
        else
                z <='0';
        end if;
        end process;

end arch;

architecture beh of xor_gate is
begin
        z <= x xor y;
end beh
```

### Xor Gate Simulations

## 7.5 Multiplexer

The functioning of a multiplexer is explained in the digital circuits sections here I've only tried to show it's basic code in testbench and design formats. It is a 4-to-1 MUX with inputs as I0, I1, I2, I3 and ouput as S. On any given input S becomes either of the inputs given.

### Design

```
library ieee;
use ieee.std_logic_1164.all;

entity multiplexer is
port( I3:   in std_logic_vector(2 downto 0)
      I2:   in std_logic_vector(2 downto 0)
      I1:   in std_logic_vector(2 downto 0)
      I0:   in std_logic_vector(2 downto 0)
      S:    in std_logic_vector(1 downto 0)
      O:    out std_logic_vector(2 downto 0)
);
end multiplexer;

architecture beh1 of multiplexer is
begin
  process(I3,I2,I1,I0,S)
  begin
    case S is
      when "00" =>    O <= I0;
      when "10" =>    O <= I1;
      when "01" =>    O <= I2;
      when "11" =>    O <= I3;
      when others =>  O <= "ZZZ";
    end case;

  end process;
end beh1;

architecture beh2 of multiplexer is
begin
  O <=    I0 when S = "00" else
          I1 when S = "01" else
          I2 when S = "10" else
          I3 when S = "11" else
          "ZZZ";
end beh2;
```

### Testbench

A test bench is VHDL code that allows you to provide a documented, repeatable set of stimuli that is portable across different simulators. A test bench can be as simple as a file with clock and input data or a more complicated file that includes error checking, file input and output, and conditional testing.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity multiplexer_tb is
-- empty entity
end multiplexer_tb;

architecture testbench of multiplexer_tb is

    -- initialize the declared signals
    signal T_I3: std_logic_vector(2 downto 0):="000";
    signal T_I2: std_logic_vector(2 downto 0):="000";
    signal T_I1: std_logic_vector(2 downto 0):="000";
    signal T_I0: std_logic_vector(2 downto 0):="000";
    signal T_O:  std_logic_vector(2 downto 0);
    signal T_S:  std_logic_vector(1 downto 0);

    component multiplexer_tb
    port( I3:         in std_logic_vector(2 downto 0);
          I2:         in std_logic_vector(2 downto 0);
          I1:         in std_logic_vector(2 downto 0);
          I0:         in std_logic_vector(2 downto 0);
          S:          in std_logic_vector(1 downto 0);
          O:          out std_logic_vector(2 downto 0)
    );
    end component;

begin

    U_Mux: multiplexer port map (T_I3, T_I2, T_I1, T_I0, T_S, T_O);

    process

        variable err_cnt: integer :=0;

    begin

        T_I3 <= "001";                -- I0-I3 are different signals
        T_I2 <= "010";
        T_I1 <= "101";
        T_I0 <= "111";

        -- case select eqaul "00"
        wait for 10 ns;
        T_S <= "00";
        wait for 1 ns;
        assert (T_O="111") report "Error Case 0" severity error;
        if (T_O/="111") then
            err_cnt := err_cnt+1;
        end if;

        -- case select equal "01"
        wait for 10 ns;
        T_S <= "01";
        wait for 1 ns;
        assert (T_O="101") report "Error Case 1" severity error;
        if (T_O/="101") then
            err_cnt := err_cnt+1;
        end if;

        -- case select equal "10"
        wait for 10 ns;
        T_S <= "10";
        wait for 1 ns;
        assert (T_O="010") report "Error Case 2" severity error;
        if (T_O/="010") then
            err_cnt := err_cnt+1;
        end if;

        -- case select equal "11"
        wait for 10 ns;
        T_S <= "11";
        wait for 1 ns;
        assert (T_O="001") report "Error Case 3" severity error;
        if (T_O/="001") then
            err_cnt := err_cnt+1;
        end if;

        -- case equal "11"
        wait for 10 ns;
        T_S <= "UU";

        -- summary of all the tests
        if (err_cnt=0) then
            assert (false)
            report "Testbench of Mux completed sucessfully!"
            severity note;
        else
            assert (true)
            report "Something wrong, try again!"
            severity error;
        end if;

        wait;

    end process;

end testbench;

configuration config of multiplexer_tb is
        for testbench
        end for;
end config;
```

## 7.6 Finite State Machines

Finally, we code for sequential circuits and I'll include the code for its design and testbench both. A finite state machine (sometimes called a finite state automaton) is a computation model that can be implemented with hardware or software and can be used to simulate sequential logic and some computer programs. Finite state automata generate regular languages.

### Design

```
library ieee;
use ieee.std_logic_1164.all;

entity seq_design is
port( a:      in std_logic;
      clock:  in std_logic;
      reset:  in std_logic;
      x:      out std_logic
);
end seq_design;

architecture fsm of seq_design is
  --design states of FSM model

  type state_type is (S0, S1, S2, S3);
  signal next_state, current_state: state_type;

begin
  --concurrent process#1: state registers
  state_reg: process(clock, reset)
  begin
    if (reset='1') then
      current_state <= S0;
    elif (clock'event and clock='1') then
      current_state <= next_state;
    end if;

  end process;

  --concurrent process#2: combinational logic
  comb_logic: process(current_state, a)

  begin
    --use case statement to show
    --state transition

    case current_state is

      when S0 => x <= '0';
               if a = '0' then
                 next_state <= S0;
               elif a = '1' then
                 next_state <= S1;
               end if;

      when S1 => x<= '0';
               if a = '0' then
```

```
                 next_state <= S1;
               elif a = '1' then
                 next_state <= S2;
               end if;

      when S2 => x <= '0';
               if a = '0' then
                 next_state <= S2;
               elif a = '1' then
                 next_state <= S3;
               end if;

      when S3 => x<= '0';
               if a = '0' then
                 next_state <= S3;
               elif a = '1' then
                 next_state <= S4;
               end if;

      when others =>
               x <= '0'
               next_state <= S0;

    end case;

  end process;

end fsm;
```

### Testbench

A test bench is VHDL code that allows you to provide a documented, repeatable set of stimuli that is portable across different simulators. A test bench can be as simple as a file with clock and input data or a more complicated file that includes error checking, file input and output, and conditional testing.

```
library        ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity fsm_tb is
--entity declaration
end fsm_tb;

architecture tb of fsm_tb is

signal T_a: std_logic;
signal T_clock: std_logic;
signal T_reset: srd_logic;
signal T_x: std_logic;

component seq_design
port( a:    in std_logic;
     clock: in std_logic;
     reset: in std_logic;
     x:     out std_logic
);
end component;

begin
  u_fsm: seq_design port map(T_a, T_clock, T_reset, T_x);

  process
  begin
    T_clock <= '1';
    wait for 5 ns;
    T_clock <= '0';
    wait for 5 ns;
  end process;

  process
      variable err_cnt: integer := 0;
  begin

    --case 1
    T_reset <= '1'
    wait for 20 ns;
    assert (T_x = '0') report "Failed Case 1" severity error;
    if (T_x/='0') then
      err_cnt := err_cnt+1;
    end if;

    --case 2
    T_reset <= '0'
    wait for 20 ns;
    assert (T_x = '0') report "Failed Case 2" severity error;
    if (T_x/='0') then
      err_cnt := err_cnt+1;
    end if;

    --case 3
    wait for 30 ns;
    T_a <= '1'
    wait for 35 ns;
    assert (T_x = '1') report "Failed Case 3" severity error;
    if (T_x/='1') then
      err_cnt := err_cnt+1;
    end if;

    --case 2
    wait for 70 ns;
    T_reset <= '1'
    wait for 10 ns;
    assert (T_x = '0') report "Failed Case 4" severity error;
    if (T_x/='0') then
      err_cnt := err_cnt+1;
    end if;

    --summary of all testcases
    if (err_cnt = 0) then
      assert false
      report "Test bench executed succesfully!"
      severity note;

    else
      assert true
      report "Something is wrong!"
      severity note;
    end if;

    wait;

  end process;

end tb;

configuration cfg_tb of fsm_tb is
      for tb
      end for;
end cfg_tb;
```

## Conclusion

This report not just contains various topics of automata theory but also has other different topics that revolve around the theory and which are essential for having a better understanding of the automata theory. It includes digital circuits and the code for some of those logic circuits and the link for the GitHub Repository which contains code for all the logic circuits that I've studied as a part of the project.