Notes

Automata Theory – Complete Notes

Unit 1: Regular Languages and Finite Automata

1. Closure Properties of Regular Languages

Regular languages are closed under the following operations:

Union

- If L_1 and L_2 are regular, then $L_1 \cup L_2$ is also regular.

Intersection

- If L_1 and L_2 are regular, then $L_1 \cap L_2$ is also regular.
- Proof idea: Construct product automaton.

Complement

- If L is regular, then $\bar{L}$ (complement of L) is also regular.
- Proof idea: Take a DFA for L and swap accepting and non-accepting states.

Concatenation

- If L_1 and L_2 are regular, then $L_1 \cdot L_2$ is also regular.

Kleene Star

- If L is regular, then L^* is also regular.

2. Myhill–Nerode Theorem (Minimization of DFA)

Theorem: A language $L \subseteq \Sigma^*$ is regular if and only if the number of equivalence classes of the relation $\equiv_l$ is finite.

Where $x \equiv_l y \iff \forall z \in \Sigma^*, xz \in L \iff yz \in L$

Applications:

- Provides a method to prove non-regularity
- Foundation for DFA minimization

3. Regular Expressions and Finite Automata Equivalence

Theorem: Every language defined by a regular expression can be accepted by some finite automaton, and vice versa.

- **Regular Expression → NFA:** Use Thompson's Construction
- **NFA → DFA:** Use subset construction
- **DFA → Regular Expression:** Use state elimination method

4. Applications of Finite Automata

- Lexical analysis in compilers
- Pattern matching tools like grep, awk
- Text search algorithms
- String validation

5. Mealy and Moore Machines

Mealy Machine

- Output depends on current state and input symbol
- Output function: $\lambda: Q \times \Sigma \rightarrow \Gamma$

Moore Machine

- Output depends only on current state
- Output function: $\lambda: Q \rightarrow \Gamma$

Theorem: For every Mealy machine, there is an equivalent Moore machine and vice versa.

Example Problems

1. Construct DFA for the language of all strings over `{0,1}` ending in 01
2. Construct NFA and convert it into DFA
3. Convert regular expression to NFA and then to DFA
4. Minimize given DFA using equivalence partitioning

Unit 2: Non-determinism and Kleene's Theorem

1. Nondeterministic Finite Automaton (NFA)

An NFA is a 5-tuple: $M = (Q, \Sigma, \delta, q_0, F)$

Where:

- Q: Finite set of states
- Σ: Input alphabet

- $\delta: Q \times \Sigma \rightarrow 2^{\psi}$: Transition function (can go to multiple states)
- $q_0 \in Q$: Start state
- $F \subseteq Q$: Set of accepting states

Characteristics:

- Multiple transitions allowed for the same input
- Acceptance if any path reaches a final state

2. Epsilon-NFA (ε-NFA)

An extension of NFA that allows epsilon transitions (ε): $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^{\psi}$

- Input can be consumed without reading a symbol

3. DFA and NFA Equivalence Theorem

Theorem: For every NFA, there exists a DFA that accepts the same language.

Subset Construction Algorithm:

- Each state of DFA is a subset of NFA states
- Start state of DFA is ε-closure of NFA's start
- Use δ to generate transitions for each subset

4. Kleene's Theorem – Part I (with Proof)

Statement: A language is regular iff it is accepted by a finite automaton.

Direction 1: Regular Expression → FA

- Use Thompson's Construction to convert regex to ε-NFA

Direction 2: FA → Regular Expression

- Use state elimination method to extract regex from DFA/NFA

5. Kleene's Theorem – Part II (Intro Only)

Statement: A language is regular iff it can be described by a regular expression.

Used to confirm the equivalence between:

- Languages accepted by FA
- Languages generated by regular expressions

6. Minimal Finite Automata Theorem

Every regular language has a unique minimum state DFA (up to isomorphism).

DFA Minimization Steps:

1. Remove unreachable states
2. Merge equivalent states using partition refinement (Myhill-Nerode relation)

Example Problems

1. Convert ε-NFA to NFA
2. Convert NFA to DFA using subset construction
3. Minimize given DFA
4. Use Kleene's Theorem to convert FA to regex

Unit 3: Context-Free Grammar (CFG)

1. Context-Free Grammar (CFG) – Definition

A CFG is a 4-tuple: `G = (V, Σ, R, S)`

Where:

- `V`: Set of variables (non-terminals)
- `Σ`: Set of terminals
- `R`: Set of production rules of the form `A → α`, where `A ∈ V`, `α ∈ (V ∪ Σ)*`
- `S ∈ V`: Start symbol

2. Chomsky Hierarchy

- **Type 0:** Recursively enumerable languages (Turing Machines)
- **Type 1:** Context-sensitive languages
- **Type 2:** Context-free languages (PDA)
- **Type 3:** Regular languages (FA)

3. Derivation Trees and Ambiguity

- **Derivation Tree / Parse Tree:** Tree representation of derivations from CFG
- **Ambiguous Grammar:** A grammar is ambiguous if a string has more than one parse tree

Tip: Try leftmost and rightmost derivations to detect ambiguity.

4. Closure Properties of CFLs

Context-Free Languages (CFLs) are closed under:

- Union
- Concatenation

- Kleene Star

CFLs are NOT closed under:

- Intersection
- Complement

5. Grammar Construction

Union: For CFGs G_1 and G_2, construct new grammar: $S \to S_1 \mid S_2$ Where S_1 and S_2 are start symbols of G_1 and G_2.

Concatenation: $S \to S_1S_2$

Kleene Star: $S \to SS \mid \varepsilon$

6. Simplified Forms of CFG

- Removing Useless Symbols
- Removing ε-productions
- Removing Unit Productions: $A \to B$

7. Chomsky Normal Form (CNF)

All productions are of the form:

- $A \to BC$, where $B, C \in V$
- $A \to a$, where $a \in \Sigma$
- $S \to \varepsilon$ (only allowed if $\varepsilon \in L(G)$)

Used in parsing algorithms like CYK.

8. Greibach Normal Form (GNF)

All productions are of the form: $A \to a\alpha$, where $a \in \Sigma$, $\alpha \in V^*$

Used in Top-Down Parsing.

9. Backus-Naur Form (BNF)

A metalanguage for describing syntax: `<non-terminal> ::= <expression>`

Common in language design and compilers.

Example Problems

1. Construct CFG for palindromes over `{a,b}`
2. Convert given CFG to CNF

3. Check if a grammar is ambiguous
4. Simplify a CFG by removing useless, null, and unit productions

---

Unit 4: Pushdown Automata (PDA) and Parsing

1. Pushdown Automaton (PDA) – Definition

A PDA is a 7-tuple: $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Where:

- Q: Finite set of states
- Σ: Input alphabet
- Γ: Stack alphabet
- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{(Q \times \Gamma^*)}$: Transition function
- $q_0 \in Q$: Start state
- $Z_0 \in \Gamma$: Initial stack symbol
- $F \subseteq Q$: Set of accepting states

Acceptance Criteria:

- By Final State
- By Empty Stack

2. Deterministic and Non-deterministic PDA

DPDA (Deterministic PDA):

- At most one move per configuration
- No ambiguity in input + top stack symbol + state

NPDA (Non-deterministic PDA):

- May have multiple valid transitions for same input configuration

Note:

- All DPDA languages are CFLs
- But not all CFLs are accepted by DPDAs (strict subset)

3. CFG and PDA Equivalence Theorem

Theorem: For every CFG, there exists a PDA that accepts the same language, and vice versa.

CFG → PDA:

- Push variables and simulate derivations

PDA → CFG:

- Use transitions to generate corresponding productions

4. Applications of PDA

- Parsing in compilers
- Syntax checking
- Modeling recursive function calls

5. Top-Down Parsing (Predictive Parsing)

- Based on Leftmost derivation
- LL(k) parsers (Lookahead)
- Uses First and Follow sets
- Requires non-ambiguous, left-factored, non-left-recursive grammars

6. Bottom-Up Parsing (Shift-Reduce Parsing)

- Based on Rightmost derivation in reverse
- Builds parse tree from leaves to root
- LR parsers (LALR, SLR, Canonical LR)

Example Problems

1. Construct PDA for $\{a^n b^n \mid n \geq 0\}$
2. Design PDA that accepts palindromes over `{a,b}`
3. Convert CFG to PDA
4. Parse a given string using LL(1) or LR(0) method

Unit 5: Context-Free Languages (CFLs)

1. Definition

A language is Context-Free if it can be generated by a Context-Free Grammar (CFG): $G = (V, \Sigma, R, S)$

Where all productions are of the form $A \to \alpha$, with $A \in V$ and $\alpha \in (V \cup \Sigma)^*$

2. CFL vs Regular Language

- Every regular language is a CFL
- Not every CFL is regular (e.g., $\{a^n b^n \mid n \geq 0\}$)

3. Closure Properties of CFLs

CFLs are closed under:

- Union
- Concatenation
- Kleene Star

CFLs are NOT closed under:

- Intersection
- Complement

Tip: Intersection with regular languages is closed. `CFL ∩ REGULAR = CFL`

4. Pumping Lemma for CFLs

Used to prove a language is not a CFL.

Statement: If L is a CFL, then $\exists\ p \in \mathbb{N}$, such that for any $z \in L$ with $|z| \geq p$, z can be written as:

$z = uvwxy$

Such that:

- $|vwx| \leq p$
- $vx \neq \varepsilon$
- $\forall i \geq 0,\ uv^iwx^iy \in L$

Use this lemma to show contradiction for non-CFLs.

5. CFL Properties with Examples

Operation	Closure	Example
Union	Yes	$\{a^nb^n\} \cup \{a^nb^{2n}\}$
Concatenation	Yes	$\{a^nb^n\} \cdot \{b^nc^n\}$
Kleene Star	Yes	$(a^nb^n)^*$
Intersection	No	$\{a^nb^nc^n\} = L_1 \cap L_2$
Complement	No	By DeMorgan's Law: $A \cup B = \bar{A} \cap \bar{B}$

Example Problems

1. Prove $\{a^nb^nc^n \mid n \geq 0\}$ is not a CFL using pumping lemma
2. Show $\{ww^R \mid w \in \{a,b\}^*\}$ is CFL (palindrome)
3. Test closure under union for two given CFLs

Unit 6: Turing Machine (TM)

1. Turing Machine – Definition

A Turing Machine is a 7-tuple: $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Where:

- Q: Finite set of states
- Σ: Input alphabet (does not include blank symbol B)
- Γ: Tape alphabet (includes B)
- $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L,R\}$: Transition function
- $q_0 \in Q$: Start state
- $B \in \Gamma$: Blank symbol
- $F \subseteq Q$: Set of accepting (final) states

2. TM as Language Acceptor

- TM accepts a language by entering a final state
- Alternatively, by halting in a valid configuration

3. Computing a Partial Function with TM

TMs can compute functions by writing output on tape: $f: \Sigma^* \to \Sigma^*$ (partial)

If TM halts on input w, then output is the tape content.

4. Combining Turing Machines

Used to construct complex machines from simpler ones:

- Sequence of TMs: Output of one becomes input of next
- Conditional branching (based on symbol/state)
- Subroutines (modular design)

5. Variants of Turing Machines

All these variants are equivalent in power (can simulate each other):

Variant	Description
Multi-tape TM	Multiple tapes with individual heads
Multi-track TM	One tape, multiple tracks
Non-deterministic TM (NTM)	May have multiple transitions

Variant	Description
Off-line TMs	Read-only input tape, working tape
Semi-infinite TM	Infinite in only one direction
TM with stay option	Head can stay in place (move: L/R/S)

6. Universal Turing Machine (UTM)

- A UTM takes as input `⟨M,w⟩`, where `M` is encoded TM and `w` is input string
- Simulates TM `M` on input `w`
- **Key Idea:** TMs can encode other TMs → foundation for programmable computers

7. Applications of Turing Machines

- Defining computability and decidability
- Models for general-purpose computing
- Basis for modern computer architecture

Example Problems

1. Design TM to accept $\{a^n b^n \mid n \geq 1\}$
2. Create TM that computes binary addition
3. Construct UTM that simulates another TM

Quick Reference

Mathematical Symbols Used

- `∪` - Union
- `∩` - Intersection
- `⊆` - Subset
- `∈` - Element of
- `∀` - For all
- `∃` - There exists
- `ε` - Epsilon (empty string)
- `→` - Produces/Maps to
- `⟺` - If and only if
- `≡` - Equivalent to
- `*` - Kleene star
- `ℕ` - Natural numbers
- `|` - Such that / Cardinality

- $\langle\rangle$ - Angle brackets (encoding)
- Subscripts: $_{0}\,_{1}\,_{2}$ etc.
- Superscripts: $^{n}\,^{r}\,^{i}$ etc.

Common Notation

- L_1, L_2 - Languages
- Σ - Alphabet
- Q - States
- δ - Transition function
- λ - Output function
- Γ - Stack/Tape alphabet