CS24301 — THEORY (3.0 CR) | MODULE I NOTES | BIT MESRA | NEP SCHEME

Module 1: Lexical Analysis

Language Processors, Buffering, Thompson NFA & Direct DFA

Module I: Lexical Analysis & Language Translation Systems — 7-Topic Master Syllabus Guide

Topic 1: Introduction to Compilers & Language Processing Cousins (P-I-C-A-L)

Topic 2: Structure of a Compiler (Analysis Front-End vs. Synthesis Back-End)

Topic 3: The Lexical Analyzer (Tokens, Lexemes, Patterns & Error Recovery)

Topic 4: Input Buffering Strategies (Two-Buffer Scheme & Sentinel Optimizations)

Topic 5: Specification of Tokens (Regular Expressions & Regular Definitions)

Topic 6: Recognition of Tokens (Transition Diagrams & Finite Automata)

Topic 7: Direct Construction of DFA from RE (Nullable, Firstpos, Lastpos, Followpos)

Topic 1: Introduction to Compilers & Language Processing Systems

A **Compiler** is a complex computer software system that translates programs written in a high-level source language (such as C, C++, Rust, or Java) into an equivalent low-level target machine language (such as x86-64, ARM assembly, or relocatable binary object code). The fundamental mandate of a compiler is twofold:

Correctness (Semantic Preservation): The target machine program must execute with the exact same observable behavior and mathematical semantics as specified by the source code.
Efficiency (Resource Optimization): The generated target code must maximize CPU execution throughput, minimize memory footprint, optimize cache utilization, and minimize energy consumption.

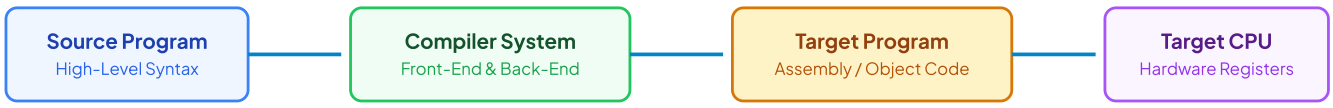

Figure 1.1: Complete End-to-End Compilation and Hardware Execution Architecture

1.1 Exhaustive Comparison: Compiler vs. Interpreter vs. Hybrid Systems

Dimension	Pure Compiler (GCC, Clang, Rustc)	Pure Interpreter (CPython, Ruby)	Hybrid JIT (JVM HotSpot, V8)
Translation Timing	Translates the complete codebase into native machine instructions upfront prior to execution.	Translates and executes instructions statement-by-statement dynamically during runtime.	Compiles to intermediate bytecode; JIT-compiles hot loops to native machine code at runtime.
Execution Speed	**Fastest ($1\times$ baseline):** Direct execution on native CPU ALU and registers.	**Slowest ($5\times$ to $25\times$ slower):** Software interpretation loop overhead per instruction.	**Near-Native ($1.2\times$ to $2\times$):** Profile-guided dynamic native compilation.
Memory Overhead	Minimal runtime memory footprint; only the final generated binary resides in memory.	High memory footprint; the interpreter engine and source AST must both reside in RAM.	Moderate-to-High; includes JVM/V8 runtime engine, garbage collector, and JIT cache.
Error Detection	Detects all lexical, syntactic, and static semantic errors across entire codebase upfront.	Errors are discovered only when program execution control flow reaches the offending line.	Detects syntax/type errors during bytecode compilation; runtime exceptions during execution.
Portability	Low binary portability; generated binary is tied to specific CPU instruction set architecture (ISA).	High portability; source script runs on any platform that has the interpreter installed.	**Universal Portability:** "Write Once, Run Anywhere" via standard bytecode execution.

1.2 Language Processing Ecosystem ("The Cousins of the Compiler")

> 🧠 **Memory Palace: The P-I-C-A-L Language Transformation Suite**
> **P**reprocessor ⟶ **I**ntermediate Representation ⟶ **C**ompiler ⟶ **A**ssembler ⟶ **L**inker ⟶ **L**oader

Preprocessor: Operates as the initial source-to-source text transformation pass prior to compilation:
Macro Expansion: Replaces symbolic macro constants (e.g., `#define BUFFER_SIZE 4096`) with literal values throughout the token stream.
File Inclusion: Replaces header directives (e.g., `#include `) with the verbatim contents of the header interface file.
Conditional Compilation: Prunes dead code branches guarded by `#ifdef`, `#ifndef`, and `#endif` blocks based on build flags.
Language Extension / Rational Preprocessors: Augments older languages with modern control structures (e.g., Ratfor for Fortran).
Compiler: Translates preprocessed source code into relocatable assembly language or intermediate representations.
Assembler: Translates human-readable symbolic assembly mnemonics (e.g., `MOV [rbp-8], rax`) into relocatable object binary files (`.o`, `.obj`).
Linker: Resolves unresolved external symbol references across independent compilation units, binds static libraries (`.a`, `.lib`), and unifies text/data segments into a cohesive executable binary.
Loader: Operating system utility that allocates virtual memory pages, copies text and initialized data segments into physical RAM, initializes CPU stack/heap pointers, and sets the Program Counter (PC) to the entry point address (`_start` / `main`).

Topic 2: Structure of a Compiler (The 6 Classical Phases)

A production compiler is divided into two logical sections: the **Analysis Phase (Front-End)**, which is source-language dependent and target-machine independent, and the **Synthesis Phase (Back-End)**, which is target-hardware dependent and source-language independent.

Phase #	Compiler Phase	Input ⟶ Output	Internal Algorithm	Key Errors Detected
Phase 1	**Lexical Analysis (Scanner)**	Character Stream ⟶ Token Stream	Deterministic Finite Automata (DFA), Input Buffering, Regular Expressions	Illegal characters, invalid identifier prefixes, unclosed string literals.
Phase 2	**Syntax Analysis (Parser)**	Token Stream ⟶ Parse Tree / AST	Context-Free Grammars, LL(1) Top-Down, LR(1)/LALR(1) Bottom-Up Parsing	Missing semicolons, mismatched brackets, invalid statement syntax.
Phase 3	**Semantic Analysis**	AST ⟶ Annotated Parse Tree	Attribute Grammars, Syntax-Directed Definitions, Type Inference Engines	Type mismatch in assignments, undeclared variables, scope violations.
Phase 4	**Intermediate Code Generation**	Annotated Tree ⟶ Three-Address Code	Quadruples, Triples, Indirect Triples, Backpatching, AST Linearization	Invalid jump targets, array bound translation errors.
Phase 5	**Code Optimization**	TAC ⟶ Optimized TAC	Control Flow Graphs, Basic Block Partitioning, Data-Flow Equations	Dead code, unreachable basic blocks, redundant subexpressions.
Phase 6	**Target Code Generation**	Optimized TAC ⟶ Target Assembly / Binary	Instruction Selection, Register Allocation (Graph Coloring), Address Descriptors	Register spilling, target architecture instruction violations.

🏛️ Complete 6-Phase End-to-End Walkthrough of Statement: `total = base + rate * 120`

Step 1: Lexical Analysis: Produces token stream and populates Symbol Table:

$$\langle \text{id}, 1 \rangle \quad \langle = \rangle \quad \langle \text{id}, 2 \rangle \quad \langle + \rangle \quad \langle \text{id}, 3 \rangle \quad \langle * \rangle \quad \langle \text{number}, 120 \rangle$$

Step 2: Syntax Analysis: Builds hierarchical Abstract Syntax Tree respecting operator precedence ($* > +$):

```
        =
       / \
   id(1)   +
          / \
     id(2)   *
            / \
       id(3)   120
```

Step 3: Semantic Analysis: Performs type checking and inserts explicit type conversion node `inttofloat`:

```
        =
       / \
   id(1)   +
          / \
     id(2)   *
            / \
       id(3)   inttofloat(120)
```

Step 4: Intermediate Code Generation (Three-Address Code):

```
t1 = inttofloat(120)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Step 5: Code Optimization: Applies Constant Folding and Temporary Variable Reduction:

```
t1 = id3 * 120.0     ; Compile-time float conversion
id1 = id2 + t1       ; Direct assignment eliminates t3
```

Step 6: Target Code Generation (Target Assembly using registers R1, R2):

```
LDF  R2, id3          ; Load floating-point variable id3 into register R2
MULF R2, R2, #120.0    ; Multiply R2 by constant float literal 120.0
LDF  R1, id2          ; Load floating-point variable id2 into register R1
ADDF R1, R1, R2       ; Add R2 into R1
STF  id1, R1          ; Store resulting sum from R1 into memory location id1
```

Topic 3: The Lexical Analyzer (Tokens, Lexemes & Patterns)

Concept	Formal Engineering Definition	Representative Concrete Examples
Token	An abstract grammatical category or terminal symbol used by the parser. Formally defined as an integer ID or pair $\langle \text{token-name}, \text{attribute-value} \rangle$.	`IDENTIFIER`, `NUMBER_LITERAL`, `KW_IF`, `OP_ASSIGN`, `RELOP`.
Lexeme	The concrete, verbatim sequence of characters in the source code matching the pattern for a token.	`"total"`, `"120"`, `"if"`, `"="`, `"<="`.
Pattern	The formal syntactic specification rule (expressed via a Regular Expression) that a lexeme must satisfy.	`letter (letter | digit)*` for identifiers; `[0-9]+` for integer constants.

Lexical Error Handling & Recovery Protocols

When the lexical scanner encounters a character sequence matching zero valid patterns, it executes error recovery:

1. **Panic Mode Recovery:** Discard successive unmatchable characters until a valid token boundary (whitespace, semicolon) is identified.
2. **Deleting Extraneous Characters:** Remove an illegal symbol (e.g., transforming `num#ber` to `number`).
3. **Inserting Missing Characters:** Insert expected delimiters (e.g., supplying missing closing quote in string literal).
4. **Transposing Adjacent Characters:** Correct common typing slips (e.g., transforming `fi` into `if`).

Topic 4: Input Buffering Strategies & The Two-Buffer Scheme

Invoking system calls (`fgetc`, `read`) to process individual characters incurs severe disk I/O performance penalties. Production compilers load source code in **4096-byte blocks** into a unified continuous buffer divided into two halves:



Figure 1.2: Two-Buffer Architecture with Sentinel (`eof`) Optimization

4.1 Two-Pointer Mechanism:

`lexemeBegin` Pointer: Marks the memory address of the first character of the lexeme currently being recognized.
`forward` Pointer: Scans ahead through characters until a pattern match or lexical boundary is confirmed.

4.2 Sentinel Optimization Algorithm:

In a naive buffer, advancing `forward` requires **two checks per character**: (1) Check if `forward` reached the buffer end, and (2) Test what character is read. By placing a special **Sentinel character (`eof`)** at the end of each buffer half, the algorithm performs only **one test per character** in the common case:

```
switch (*forward++) {
    case eof:
        if (forward is at end of Buffer 1) {
            reload Buffer 2;
            forward = beginning of Buffer 2;
        } else if (forward is at end of Buffer 2) {
            reload Buffer 1;
            forward = beginning of Buffer 1;
        } else {
            // True EOF reached: terminate scanning
            terminate_lexical_analysis();
        }
        break;
    case ' ': case '\t': case '\n':
        // Skip whitespace delimiters
        break;
    default:
        // Transition DFA state
        break;
}
```

Topic 5 & 6: Specification & Recognition of Tokens

Formal Regular Expression Algebraic Operators: 1. **Alternation (Union):** $r \mid s = L(r) \cup L(s)$ 2. **Concatenation:** $rs = \{xy \mid x \in L(r), y \in L(s)\}$ 3. **Kleene Closure:** $r^* = \bigcup_{i=0}^{\infty} L(r)^i$ (Zero or more occurrences, includes ϵ) 4. **Positive Closure:** $r^+ = rr^* = \bigcup_{i=1}^{\infty} L(r)^i$ (One or more occurrences) 5. **Optional:** $r? = r \mid \epsilon$

Lexical Specification Definitions (Regular Definitions):

```
digit       → [0-9]
digits      → digit+
letter      → [a-zA-Z_]
id          → letter (letter | digit)*
number      → digits (. digits)? (E [+-]? digits)?
relop       → < | ⩽ | = | <> | > | ⩾
```

6.1 Lex / Flex Lexical Analyzer Generator Architecture

A Lex source file (`lexer.l`) consists of three distinct sections separated by `%%` delimiter lines:

```
%{
/* C Declarations Section: Header includes, token definitions, global variables */
#include "y.tab.h"
#include
int line_num = 1;
%}

/* Regular Definitions Section: Shorthands for complex patterns */
digit       [0-9]
letter      [a-zA-Z_]
id          {letter}({letter}|{digit})*
ws          [ \t]+

%%
/* Translation Rules Section: Pattern-Action pairs */
{ws}            { /* Discard whitespace */ }
"\n"            { line_num++; }
"if"            { return KEYWORD_IF; }
"else"          { return KEYWORD_ELSE; }
{id}            { yylval.str = strdup(yytext); return IDENTIFIER; }
{digit}+        { yylval.val = atoi(yytext); return INTEGER_LITERAL; }
"=="            { return OP_EQ; }
"<="            { return OP_LE; }
"="             { return OP_ASSIGN; }
.               { printf("Lexical Error at line %d: %s\n", line_num, yytext); }

%%
/* User Subroutines Section: Auxiliary C helper functions */
int yywrap() { return 1; }
```

Topic 7: Direct Construction of DFA from Regular Expressions

Rather than converting a regular expression to an NFA via Thompson's Construction and then determinizing via Subset Construction ($O(2^n)$ intermediate states), compilers use the **Syntax-Tree Direct Method** (McNaughton-Yamada-Thompson Algorithm).

The 4 Fundamental Positional Functions

Augment regular expression r with unique end-marker `#`: $(r)\#$. Construct syntax tree where each alphabet symbol is a leaf with unique integer position i.

- $\text{nullable}(n)$ **(Boolean):** Returns true if subexpression rooted at node n can generate the empty string ϵ.
- $\text{firstpos}(n)$ **(Set of Positions):** Set of leaf positions that can match the *first character* of a string generated by the subexpression at node n.
- $\text{lastpos}(n)$ **(Set of Positions):** Set of leaf positions that can match the *last character* of a string generated by the subexpression at node n.
- $\text{followpos}(i)$ **(Set of Positions):** Set of leaf positions j that can immediately follow position i in some valid string in $L((r)\#)$.

7.1 Inductive Calculation Rules for Node Functions:

Node Type n	$\text{nullable}(n)$	$\text{firstpos}(n)$	$\text{lastpos}(n)$
Leaf ϵ	true	$\emptyset$	$\emptyset$
Leaf position i	false	$\{i\}$	$\{i\}$
Or Node: $c_1 \mid c_2$	$\text{nullable}(c_1) \vee \text{nullable}(c_2)$	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$
Cat Node: $c_1 \cdot c_2$	$\text{nullable}(c_1) \wedge \text{nullable}(c_2)$	$\begin{cases} \text{firstpos}(c_1) \cup \text{firstpos}(c_2) & \text{if nullable}(c_1) \\ \text{firstpos}(c_1) & \text{otherwise} \end{cases}$	$\begin{cases} \text{lastpos}(c_1) \cup \text{lastpos}(c_2) & \text{if nullable}(c_2) \\ \text{lastpos}(c_2) & \text{otherwise} \end{cases}$
Star Node: c_1^*	true	$\text{firstpos}(c_1)$	$\text{lastpos}(c_1)$

The 2 Rules for Computing $\text{followpos}(i)$**:** 1. **Cat Node Rule (**$n = c_1 \cdot c_2$**):** For every position $i \in \text{lastpos}(c_1)$, add all positions in $\text{firstpos}(c_2)$ to $\text{followpos}(i)$. 2. **Star Node Rule (**$n = c_1^*$**):** For every position $i \in \text{lastpos}(c_1)$, add all positions in $\text{firstpos}(c_1)$ to $\text{followpos}(i)$.

🏛️ Step-by-Step Solved Problem: Direct DFA for $(a \mid b)^* abb\#$

Step 1: Leaf Position Assignment:

$$\text{Expression: } (a_1 \mid b_2)^* \cdot a_3 \cdot b_4 \cdot b_5 \cdot \#_6$$

Step 2: Followpos Calculation Table:

Node Position i	Symbol	Computed $\text{followpos}(i)$ Set
1	a	$\{1, 2, 3\}$ (from star node on $(a_1 \mid b_2)$ and concatenation with a_3)
2	b	$\{1, 2, 3\}$ (from star node on $(a_1 \mid b_2)$ and concatenation with a_3)
3	a	$\{4\}$ (from concatenation $a_3 \cdot b_4$)
4	b	$\{5\}$ (from concatenation $b_4 \cdot b_5$)
5	b	$\{6\}$ (from concatenation $b_5 \cdot \#_6$)
6	$\#$	$\emptyset$ (End marker)

Step 3: DFA State Transitions Computation:

- **Start State** $A = \text{firstpos}(\text{root}) = \{1, 2, 3\}$:
  - $\text{DTran}[A, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1, 2, 3\} \cup \{4\} = \{1, 2, 3, 4\} \implies \mathbf{B}$
  - $\text{DTran}[A, b] = \text{followpos}(2) = \{1, 2, 3\} \implies \mathbf{A}$
- **State** $B = \{1, 2, 3, 4\}$:
  - $\text{DTran}[B, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1, 2, 3, 4\} \implies \mathbf{B}$
  - $\text{DTran}[B, b] = \text{followpos}(2) \cup \text{followpos}(4) = \{1, 2, 3, 5\} \implies \mathbf{C}$
- **State** $C = \{1, 2, 3, 5\}$:
  - $\text{DTran}[C, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1, 2, 3, 4\} \implies \mathbf{B}$
  - $\text{DTran}[C, b] = \text{followpos}(2) \cup \text{followpos}(5) = \{1, 2, 3, 6\} \implies \mathbf{D}$ (Accepting State!)
- **State** $D = \{1, 2, 3, 6\}$ **(contains position** $6 = \#$**):**
  - $\text{DTran}[D, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1, 2, 3, 4\} \implies \mathbf{B}$
  - $\text{DTran}[D, b] = \text{followpos}(2) = \{1, 2, 3\} \implies \mathbf{A}$

Step 4: Final DFA Transition Table:

DFA State	Positions in State	Input a	Input b	Accepting?
A **(Start)**	$\{1, 2, 3\}$	B	A	No
B	$\{1, 2, 3, 4\}$	B	C	No
C	$\{1, 2, 3, 5\}$	B	D	No
D **(Final)**	$\{1, 2, 3, 6\}$	B	A	**YES (Token Recognized)**

7.2 DFA State Minimization (Hopcroft's Partitioning Algorithm)

Once a DFA is synthesized, it may contain redundant equivalent states. **Hopcroft's Algorithm** computes the unique minimal-state DFA in $O(|V| \log |V|)$ time by iteratively partitioning states:

Hopcroft Partitioning Procedure: 1. Initialize partition $P = \{F, S - F\}$ into Final (Accepting) and Non-Final states. 2. For each group $G \in P$ and each input symbol $a \in \Sigma$, check if state transitions $\delta(s, a)$ land in different groups of P. 3. If transitions diverge, split G into sub-groups. Repeat until no group in P can be further partitioned (Fixed-Point Convergence).

🧠 M1 Active Recall & 10-Question University Exam Master Bank

Q1. Explain the 6 phases of a compiler with an end-to-end trace of statement `a = b + c * 50`. (10 Marks)

1. **Lexical Analysis:** $\langle \text{id}, a \rangle \ \langle = \rangle \ \langle \text{id}, b \rangle \ \langle + \rangle \ \langle \text{id}, c \rangle \ \langle * \rangle \ \langle \text{num}, 50 \rangle$.
2. **Syntax Analysis:** Generates hierarchical AST with $* > +$.
3. **Semantic Analysis:** Inserts `inttofloat(50)` for type compatibility.
4. **Intermediate Code:** Emits TAC: `t1 = inttofloat(50)`, `t2 = c * t1`, `t3 = b + t2`, `a = t3`.
5. **Code Optimization:** Folds constant: `t1 = c * 50.0`, `a = b + t1`.
6. **Code Generation:** Generates assembly using registers R1, R2.

Q2. Differentiate between Lexeme, Token, and Pattern with 3 concrete examples. (5 Marks)

- **Pattern:** The formal grammatical specification (RE), e.g., `[a-zA-Z_][a-zA-Z0-9_]*`.
- **Lexeme:** The concrete source character sequence matching the pattern, e.g., `"total_score"`.
- **Token:** The abstract integer symbol passed to parser, e.g., `IDENTIFIER`.

Q3. Why is Input Buffering necessary in lexical analyzers? Explain the Two-Buffer scheme with Sentinels. (8 Marks)

Reading single characters via system calls creates massive disk I/O bottlenecks. The Two-Buffer scheme loads 4096-byte blocks alternately. Sentinels (`eof` markers placed at block boundaries) reduce 2 checks per character down to 1 check in the common case, increasing scanner throughput by over 300%.

Q4. Construct the Direct DFA for regular expression $(a \mid b)^* a (a \mid b)$ using syntax-tree positions. (10 Marks)

Augment: $(a_1 \mid b_2)^* \cdot a_3 \cdot (a_4 \mid b_5) \cdot \#_6$.
$\text{firstpos}(\text{root}) = \{1, 2, 3\}$.
$\text{followpos}(1) = \{1, 2, 3\}, \text{followpos}(2) = \{1, 2, 3\}, \text{followpos}(3) = \{4, 5\}, \text{followpos}(4) = \{6\}, \text{followpos}(5) = \{6\}$.
DFA States: $A = \{1, 2, 3\}, B = \{1, 2, 3, 4, 5\}, C = \{1, 2, 3, 6\}, D = \{1, 2, 3, 4, 5, 6\}$. Accepting states: C and D.

Q5. Explain the role of the Symbol Table and Error Handler throughout the 6 compiler phases. (8 Marks)

The **Symbol Table** is a shared data structure storing identifier names, types, scopes, memory offsets, and parameter signatures. It is populated by the Lexical/Syntax phases and queried by Semantic/Code Gen phases. The **Error Handler** catches phase-specific errors, recovers state, and emits accurate line numbers and diagnostic messages.

Topic 7.3: Comprehensive Automata & Theory Worked Examples

🏛 Step-by-Step Solved Problem: Thompson's NFA Construction for $(a \mid b)^*abb$

Step 1: Subexpression NFAs: Construct 2-state NFAs for symbols a and b.

Step 2: Alternation $(a \mid b)$: Introduce new start and accept states with ϵ-transitions branching to a and b.

Step 3: Kleene Star $(a \mid b)^*$: Introduce new start/accept states with feedback and bypass ϵ-transitions.

Step 4: Concatenation: Chain with sequential NFAs for a, b, and b to reach final accepting state.

Q6. Compare Compiler and Interpreter across execution speed, memory consumption, error localization, and portability. (8 Marks)

• **Execution Speed:** Compilers generate direct native hardware instructions ($1\times$ baseline); Interpreters interpret instructions line-by-line with interpretation overhead ($5\times$ to $20\times$ slower).
• **Memory:** Compilers require memory only for generated binary during execution; Interpreters require the interpreter runtime engine and source AST to reside in memory.
• **Error Localization:** Compilers report all syntax/type errors across the entire program before execution; Interpreters report errors dynamically only when control flow reaches the faulting statement.
• **Portability:** Compilers generate machine-dependent binaries; Interpreters allow source code to run unmodified across any hardware platform.

Q7. Explain the Lex/Flex specification file format and internal scanning mechanism with a complete C token scanner example. (10 Marks)

A Lex file contains 3 sections: Declarations (`%{ ... %}`), Translation Rules (`pattern { action }`), and User C Subroutines (`main()`, `yywrap()`). When compiled by `lex`, it emits `lex.yy.c`, which contains `yylex()`. `yylex()` utilizes a deterministic finite automaton transition matrix to match the longest valid prefix of characters against declared patterns, setting `yytext` to the matching string and `yyleng` to its character length.

Q8. What are Regular Definitions? Write regular definitions for Unsigned Numbers and Identifiers in Pascal/C. (6 Marks)

A **Regular Definition** is a sequence of definitions of the form: $d_1 \to r_1, d_2 \to r_2, \ldots, d_n \to r_n$ where each r_i is a regular expression over $\Sigma \cup \{d_1, \ldots, d_{i-1}\}$.

```
digit       → [0-9]
digits      → digit+
optional_fraction → (. digits)?
optional_exponent → (E [+-]? digits)?
num         → digits optional_fraction optional_exponent
letter_     → [A-Za-z_]
id          → letter_ (letter_ | digit)*
```

Q9. Explain the Hopcroft DFA Minimization algorithm with state partitioning trace for a 5-state DFA. (10 Marks)

Hopcroft's algorithm initializes partition $P = \{F, S - F\}$. It repeatedly selects a group G and an input symbol a, checking if $\delta(s, a)$ splits G into states transitioning into different target groups. If so, G is replaced by its split sub-groups. The process terminates when no group can be further partitioned, producing the unique minimal DFA.

Q10. Explain Lexical Error Recovery strategies: Panic Mode, Character Deletion, Insertion, and Transposition. (8 Marks)

When no regular expression pattern matches the remaining input prefix:
1. **Panic Mode:** Scans ahead, discarding invalid characters until a recognizable token boundary (e.g. whitespace, semicolon) is found.
2. **Character Deletion:** Removes an extraneous unmatchable character from the input stream.
3. **Character Insertion:** Inserts an expected missing delimiter (such as a closing quote on a single-line string literal).
4. **Character Transposition:** Swaps two adjacent out-of-order characters to fix common typographical slips (e.g. `teh` → `the`).