1. Introduction of Compiler

1. Introduction :

A compiler is a system software that translates a program written in a high level programming language (HLL) into an Equivalent program in a low level language (LLL) or machine language.

The main objective of a compiler is to produce correct, efficient and optimized object code.

Example : Program written in C language is converted into machine code (0s and 1s) by a compiler.

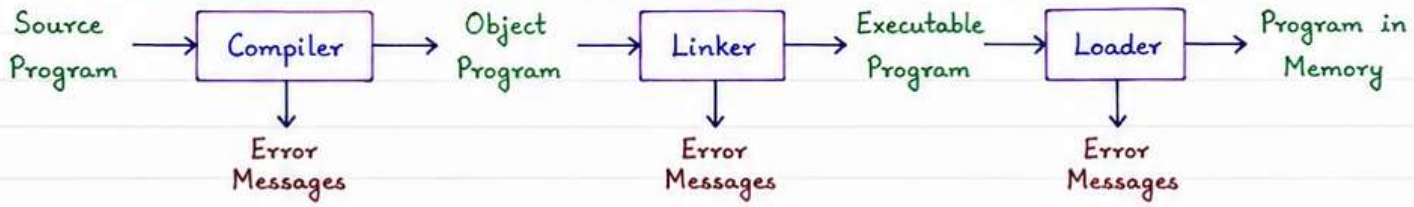
2. Definition :

"A compiler is a program that translates a source program written in a high level language into an equivalent target program in a low level language and reports the errors in the source program."

3. Role of Compiler :

- It acts as an interface between the programmer and the machine.
- It checks for errors in the source program.
- It translates the program into machine code.
- It optimizes the code to improve execution time and memory usage.
- It provides portability of programs from one machine to another.

4. Working of a Compiler :



Note : A compiler itself does not execute the program. It only translates the source program into object code.

5. Language Levels :

High Level Language (HLL)	:	Easy to understand by humans. (C, C++, Java, Python)
Low Level Language (LLL)	:	Closer to machine language. (Assembly Language)
Machine Language (ML)	:	Binary code (0s and 1s) understood by CPU.

6. Advantages of Compiler :

- Faster execution of programs.
- Detects errors at compile time.
- Produces efficient and optimized code.
- Portability across different machines.
- Supports structured programming.

7. Disadvantages of Compiler :

- Time consuming.
- Requires more memory.
- Complex design and implementation.
- Errors are reported after full compilation.

Important Points for Exams :

★ Compiler translates the entire program at once.

★ Output of compiler is object code.

★ Compiler has phases like Lexical Analysis, Syntax Analysis, Code Generation, Optimization.

★ Compiler is a one-time translator.

(Good Luck) ☺

2. Major Data Structures in Compiler

1. Introduction :

A compiler handles large amount of data during compilation.
To store, manage and access this data efficiently, different data structures are used.
These structures help the compiler to perform analysis, translation and code generation in an organized manner.

2. Major Data Structures :

(i) Lexeme Table
(ii) Symbol Table
(iii) Abstract Syntax Tree (AST)
(iv) Intermediate Code
(v) Literal Table
(vi) Name Table
(vii) Type Table

3. Description of Data Structures :

Data Structure	Description
(i) Lexeme Table	Stores all the lexemes (identifiers, keywords, constants etc.) appearing in the source program along with their attributes.
(ii) Symbol Table	Stores information about identifiers such as name, type, scope, value, memory location, etc.
(iii) Abstract Syntax Tree (AST)	Tree representation of the source program showing the hierarchical structure of operators and operands.
(iv) Intermediate Code	Represents the source program in an intermediate form (e.g., Three Address Code) used between front end and back end.
(v) Literal Table	Stores literal constants (numbers, strings, characters) used in the program.
(vi) Name Table	Stores names of labels, variables, procedures etc. used in the program.
(vii) Type Table	Stores type information of identifiers like integer, float, array, structure, pointer etc.

4. Example – Symbol Table :

Name	Type	Scope	Value	Address
a	int	Global	10	1000
b	float	Global	2.5	1004
sum()	function	Global	–	2000
x	int	Local	5	3000

5. Importance :

★ These structures organize data efficiently.
★ Helps in fast access and retrieval of information.
★ Improves error detection and reporting.
★ Essential for optimization and code generation.

Important Point :

+ Symbol Table is the most important data structure used in a compiler.

3. Types of Compiler

1. Introduction :

Compilers can be classified into different types based on the target machine, number of passes, input language, implementation and purpose.
Each type of compiler is designed for a specific environment and requirement.

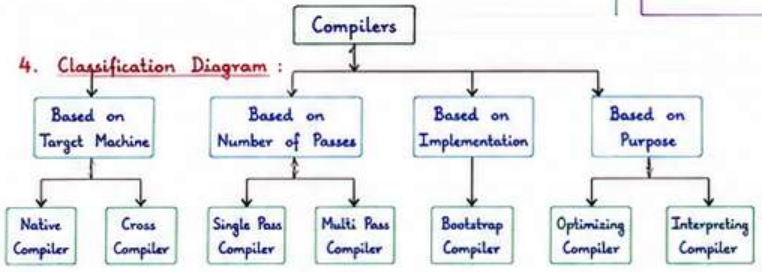
2. Types of Compiler :

(i) Native Compiler
(ii) Cross Compiler
(iii) Bootstrap Compiler
(iv) Single Pass Compiler
(v) Multi Pass Compiler
(vi) Optimizing Compiler
(vii) Interpreting Compiler

3. Description of Types :

Type of Compiler	Description
(i) Native Compiler	Compiles a program written in a high level language into machine code for the same machine.
(ii) Cross Compiler	Compiles a program written in a high level language into machine code for a different machine.
(iii) Bootstrap Compiler	A compiler written in a high level language and used to compile itself.
(iv) Single Pass Compiler	Scans the source program once from start to end and produces the target code.
(v) Multi Pass Compiler	Scans the source program multiple times. Each pass performs a specific task.
(vi) Optimizing Compiler	Produces efficient target code by applying optimization techniques.
(vii) Interpreting Compiler	Does not generate object code. It translates and executes the program statement by statement.

4. Classification Diagram :



5. Comparison (Native vs Cross Compiler) :

Feature	Native Compiler	Cross Compiler
Target Machine	Same as Host Machine	Different from Host Machine
Use	General Purpose	System/Embedded Development
Speed	Faster (No conversion issues)	Comparatively Slow
Example	C Compiler on Windows	ARM Compiler on Windows

Important Points for Exams :

★ Compilers are classified based on target machine, passes, implementation and purpose.
★ Cross compiler is used in cross-platform development.
★ Optimizing compiler improves code efficiency.
★ Bootstrap compiler is written in high level language and compiles itself.
★ Multi pass compiler gives better optimization.

4. Front-End of Compiler

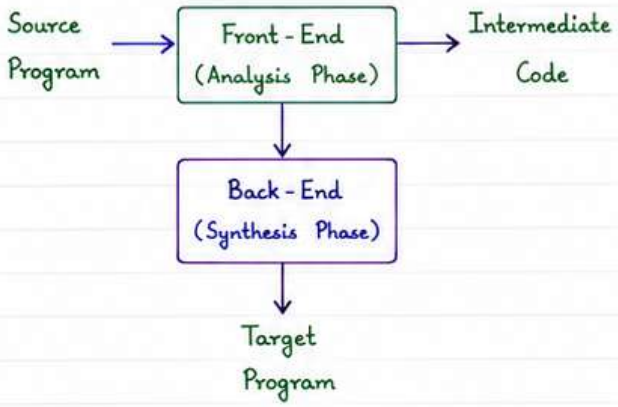
1. Introduction :

The front-end (analysis phase) of a compiler takes the source program as input and produces an intermediate representation of the program. It performs lexical, syntactic and semantic analysis and reports errors to the user.

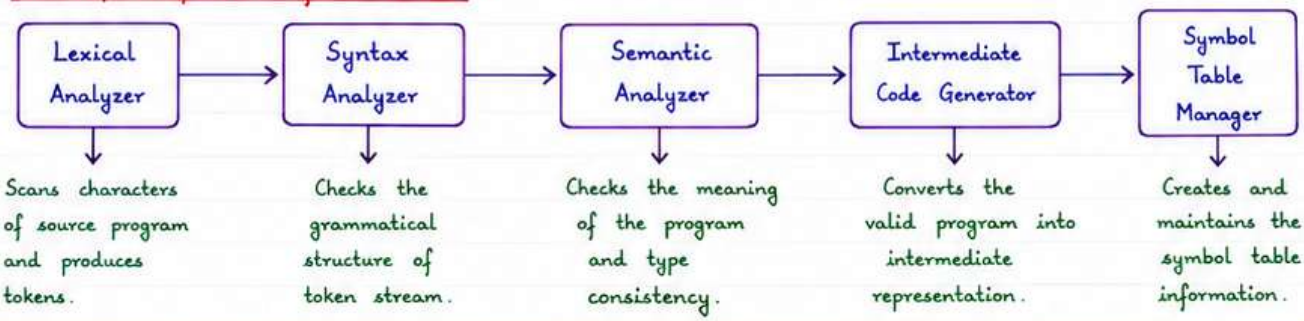
2. Functions of Front-End :

- Scans the source program and breaks it into tokens.
- Checks the grammatical structure of the program.
- Checks the meaning of the program.
- Collects information about identifiers and types.
- Generates intermediate code for the back-end.
- Reports errors in the source program.

3. Position of Front-End in Compiler :



4. Phases / Components of Front-End :



5. Description of Components :

(i) Lexical Analyzer :
Reads the source program as a sequence of characters and groups them into tokens such as keywords, identifiers, constants, operators, delimiters etc.

(ii) Syntax Analyzer :
Takes tokens as input and checks whether they are arranged in a valid grammatical structure according to the language syntax.

(iii) Semantic Analyzer :
Uses the parse tree and symbol table information to check semantic rules such as type checking, declaration checking, scope rules etc.

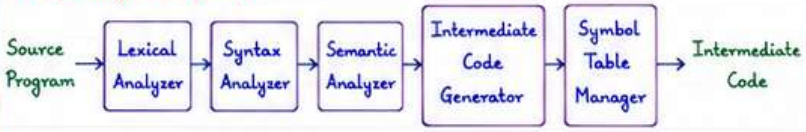
(iv) Intermediate Code Generator :
Generates an intermediate code (e.g., three address code, quadruples, triples) which is independent of any machine.

(v) Symbol Table Manager :
Creates, updates and manages the symbol table containing information about identifiers, types, scope, memory location etc.

6. Input and Output of Front-End :

Component	Input	Output
Lexical Analyzer	Source program (character stream)	Token stream, Symbol table entries
Syntax Analyzer	Token stream	Parse tree / Syntax tree
Semantic Analyzer	Parse tree, Symbol table	Annotated parse tree with semantic info
Intermediate Code Generator	Annotated parse tree	Intermediate code
Symbol Table Manager	Declarations in source program	Symbol table information

7. Example (Flow) :



Important Points for Exams :

★ Front-end is also called analysis phase.
★ It converts source program into intermediate representation.
★ Main components: Lexical, Syntax, Semantic analysis, Intermediate code generation and Symbol table management.
★ Front-end is machine independent.
★ It checks the correctness of the source program.
★ Errors detected in this phase are compile-time errors.

5. Back-End of Compiler

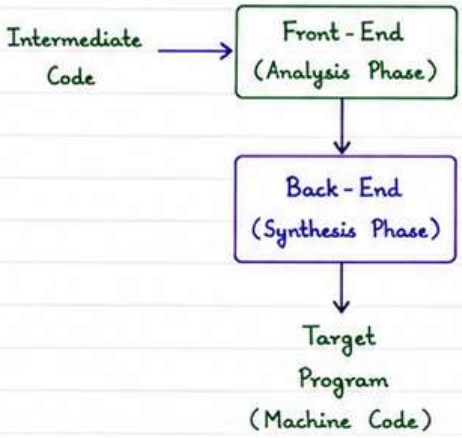
1. Introduction :

The back-end (synthesis phase) of a compiler takes the intermediate representation produced by the front-end and converts it into target program (machine code). It also performs code optimization to improve the quality of the target code.

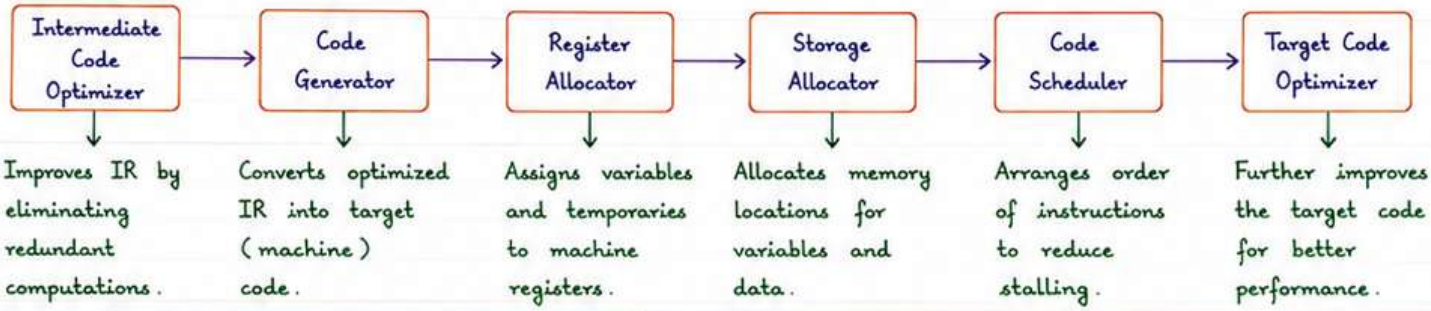
2. Functions of Back-End :

- Translates intermediate code into target code.
- Performs code optimization.
- Allocates storage for variables.
- Generates efficient and correct machine code.
- Maintains data structures like symbol table, register table, literal table etc.
- Provides portability of the compiler.

3. Position of Back-End in Compiler :



4. Phases / Components of Back-End :



5. Description of Components :

(i) Intermediate Code Optimizer :
Performs machine independent and dependent optimizations on the intermediate code to reduce execution time and memory usage.

(ii) Code Generator :
Translates the optimized intermediate code into target assembly / machine code.

(iii) Register Allocator :
Decides which variables / temporaries should be placed in registers for faster access.

(iv) Storage Allocator :
Assigns memory locations to variables, arrays, constants and other data.

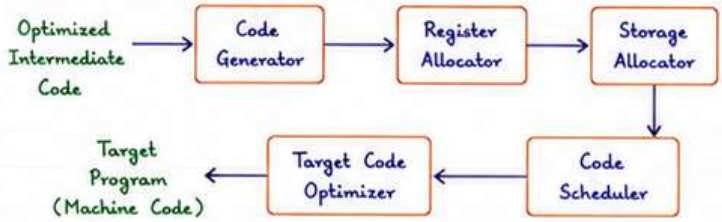
(v) Code Scheduler :
Reorders instructions to improve pipeline performance and reduce dependencies.

(vi) Target Code Optimizer :
Performs small improvements on the target code like peephole optimization to improve speed and size.

6. Input and Output of Back-End :

Component	Input	Output
Intermediate Code Optimizer	Intermediate Code	Optimized Intermediate Code
Code Generator	Optimized IR	Target Code
Register Allocator	IR, Symbol Table, Machine Info	Register Allocation Information
Storage Allocator	Symbol Table, Type Info	Memory Allocation Information
Code Scheduler	Target Code	Scheduled Target Code
Target Code Optimizer	Target Code	Optimized Target Code

7. Example (Flow of Back-End) :



Important Points for Exams :

★ Back-end is also called synthesis phase.
★ It converts intermediate code into machine code.
★ Main components : Code Optimization, Code Generation, Register Allocation, Storage Allocation, Code Scheduling.
★ Back-end is machine dependent.
★ Goal is to produce efficient, fast and correct target code.
★ Code optimization in back-end improves execution time and reduces memory usage.

6. Compiler Structure

1. Introduction :

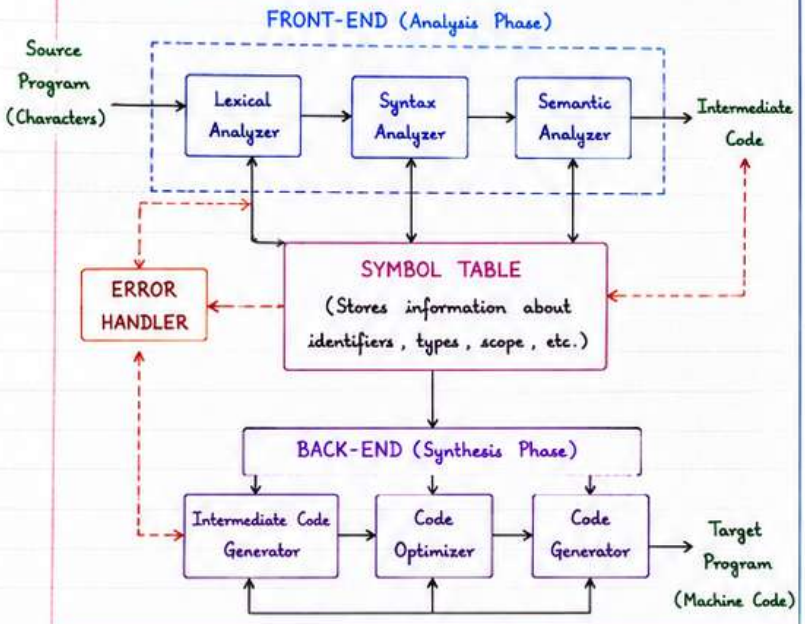
A compiler is organized into different components and their interactions form the compiler structure. It is broadly divided into two major parts :

→ Front-End (Analysis Phase)

→ Back-End (Synthesis Phase)

These work together with a symbol table and error handler to convert source program into target program.

2. Overall Structure of a Compiler :



3. Components of Compiler Structure :

(i) Lexical Analyzer
Reads the source program as a sequence of characters and groups them into tokens.

(ii) Syntax Analyzer (Parser)
Checks the grammatical structure of the program according to the grammar.

(iiii) Semantic Analyzer
Checks the meaning of the program, type checking and consistency of declarations.

(iv) Intermediate Code Generator
Converts the source program into an intermediate representation (e.g., three address code, quadruples).

(v) Code Optimizer
Improves the intermediate code to produce efficient target code.

(vi) Code Generator
Converts the optimized intermediate code into target program (machine code) for the machine.

(vii) Symbol Table
Stores information about identifiers, types, scope, memory locations, etc. It is used by many phases.

(viii) Error Handler
Detects, reports and recovers from errors. It works throughout the compilation process.

4. Working of Compiler Structure :

1. Source program is given to Lexical Analyzer.
2. Tokens are passed to Syntax Analyzer.
3. Syntax Analyzer checks grammar and sends structure to Semantic Analyzer.
4. Semantic Analyzer verifies meaning and updates information in Symbol Table.
5. Intermediate Code Generator produces intermediate code.
6. Code Optimizer improves intermediate code.
7. Code Generator produces target (machine) code.
8. Error Handler checks errors in all phases.

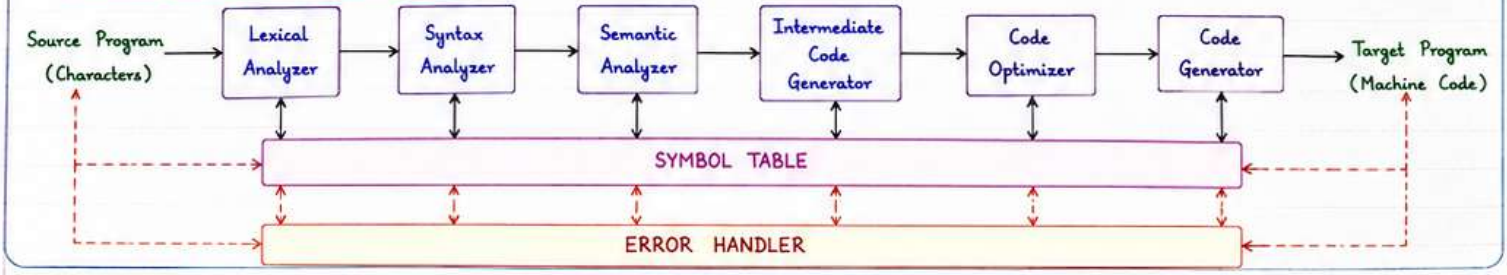
5. Importance of Compiler Structure :

- Makes the compiler modular and easy to design.
- Each phase can be developed and tested independently.
- Improves maintainability and portability.
- Efficient error detection and reporting.
- Produces optimized target code.

Key Points for Exams :

★ Compiler structure consists of front-end, back-end, symbol table and error handler.
★ Front-end performs analysis and produces intermediate code.
★ Back-end produces target code from intermediate code.
★ Symbol table is shared by almost all phases.
★ Error handler works in all phases of compilation.

6. Summary Diagram (Another View) :



7. Analysis - Synthesis Model

1. Introduction :

Analysis-Synthesis Model is a general model for designing compilers. It divides the compiler into two main parts :

⇒ Analysis (Front-End)

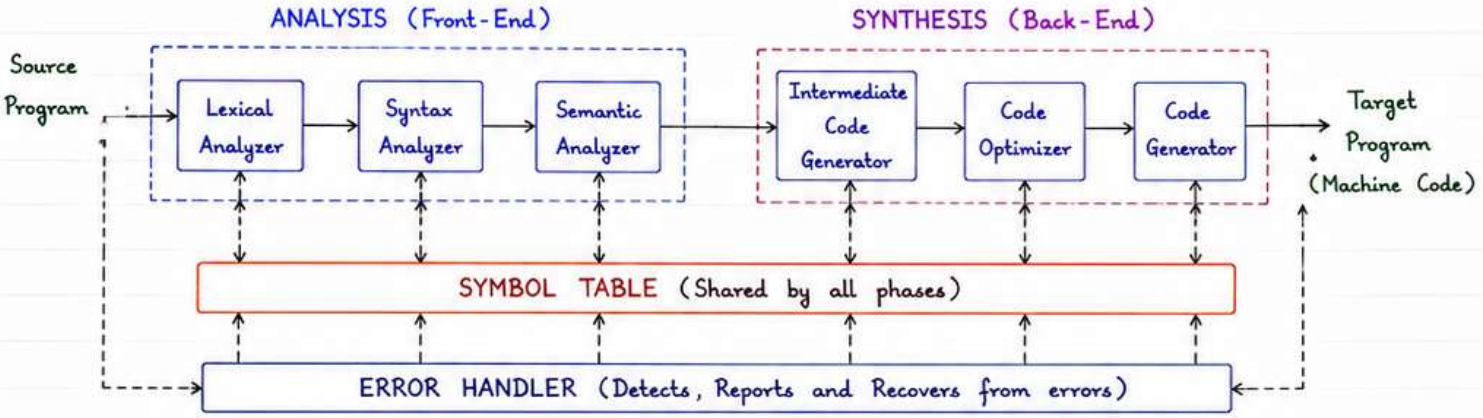
⇒ Synthesis (Back-End)

The analysis part gathers and checks information about the source program, and the synthesis part produces the target program.

2. Basic Idea :

- The source program is read and analyzed to collect useful information.
- The collected information is then used to generate the target (object) program.
- A symbol table is used by both parts to store and share information.
- An error handler is used to detect and report errors in all phases.

3. Analysis - Synthesis Model of a Compiler :



4. Role of Components :

(i) Lexical Analyzer : Scans the source program and produces tokens.

(ii) Syntax Analyzer : Checks the grammatical structure of tokens.

(iii) Semantic Analyzer : Checks the meaning, types and consistency of statements.

(iv) Intermediate Code Generator : Generates an intermediate representation (e.g., three address code).

(v) Code Optimizer : Improves the intermediate code to make it efficient.

(vi) Code Generator : Converts optimized intermediate code into target machine code.

5. Information Flow in the Model :

Flow	From	To	Information Passed
1	Source Program	Lexical Analyzer	Characters
2	Lexical Analyzer	Syntax Analyzer	Tokens
3	Syntax Analyzer	Semantic Analyzer	Parse tree / Syntax structure
4	Semantic Analyzer	Intermediate Code Generator	Annotated parse tree / semantic info
5	Intermediate Code Generator	Code Optimizer	Intermediate Code
6	Code Optimizer	Code Generator	Optimized Intermediate Code
7	Code Generator	Target Program	Machine Code

6. Characteristics / Advantages :

★ Clear separation of analysis and synthesis.
★ Each phase has a well defined input and output.
★ Easy to design, implement and maintain.
★ Phases can be developed and tested independently.
★ Promotes modularity and reusability.

7. Limitations :

★ Communication between phases through symbol table can become complex.
★ The model may not show all real world compiler details.

Important Points for Exams :

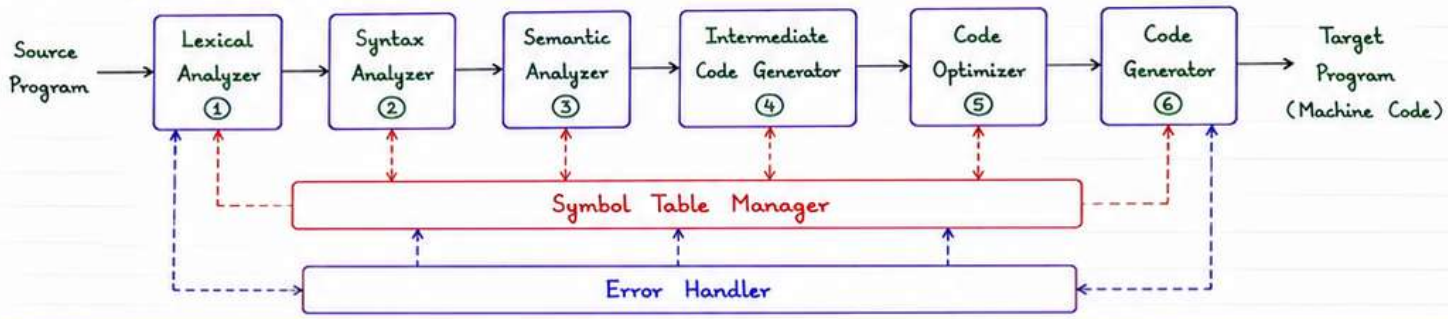
★ Analysis phase = Front-End of Compiler.
★ Synthesis phase = Back-End of Compiler.
★ Symbol Table is shared by all phases.
★ Error Handler works in both analysis and synthesis phases.
★ Intermediate code acts as a bridge between front-end and back-end.

8. Phases of Compiler

1. Introduction :

A compiler performs the translation of a source program written in a high level language into equivalent target program (machine code). This process is completed in a number of phases. Each phase has a specific task and contributes to the overall compilation process. The output of one phase becomes the input for the next phase.

2. List of Phases of Compiler :



3. Description of Phases :

(1) Lexical Analysis :
Reads the source program as a sequence of characters and groups them into tokens such as keywords, identifiers, constants, operators, delimiters etc.

(2) Syntax Analysis (Parsing) :
Checks the grammatical structure of the program. It verifies the tokens against the rules of the grammar and creates parse tree or syntax tree.

(3) Semantic Analysis :
Checks the meaning of the program. It verifies type compatibility, declaration, scope rules and other semantic constraints.

(4) Intermediate Code Generation :
Converts the source program into an intermediate representation (e.g., three address code, quadruples, triples). This code is machine independent and easy to optimize.

(5) Code Optimization :
Improves the intermediate code to produce efficient target code. It removes redundant code and improves time and space complexity.

(6) Code Generation :
Translates the optimized intermediate code into target program (machine code or assembly code) for the target machine.

4. Summary Table :

Phase No.	Phase Name	Input	Output	Main Task
1	Lexical Analyzer	Source Program (Characters)	Tokens	Scan and convert characters into tokens.
2	Syntax Analyzer	Tokens	Parse Tree / Syntax Tree	Check grammatical structure.
3	Semantic Analyzer	Parse Tree	Annotated Parse Tree	Check meaning and semantic rules.
4	Intermediate Code Generator	Annotated Parse Tree	Intermediate Code	Generate machine independent code.
5	Code Optimizer	Intermediate Code	Optimized Intermediate Code	Improve code for better performance.
6	Code Generator	Optimized Intermediate Code	Target Program (Machine Code)	Generate final machine code.

5. Role of Symbol Table Manager :

- ★ Stores information about identifiers, types, scope, memory location etc.
- ★ Used by almost all phases of the compiler.
- ★ Ensures consistency and quick retrieval of information.

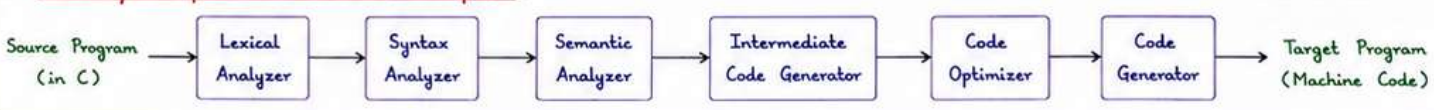
6. Role of Error Handler :

- ★ Detects, reports and recovers from errors.
- ★ Ensures that compilation process continues even after detecting errors.
- ★ Improves the reliability of the compiler.

7. Important Points for Exams :

- ★ Compiler works in a sequence of phases.
- ★ Each phase performs a specific task.
- ★ Symbol Table and Error Handler are used by all phases.
- ★ Output of one phase becomes input to the next phase.

8. Flow of Compilation Process (Example) :



9. Lexical Analysis

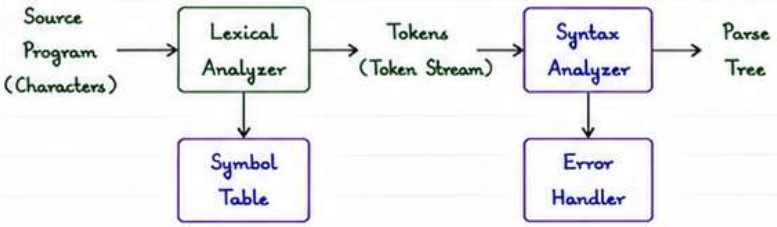
1. Introduction :

Lexical Analysis is the first phase of a compiler. It reads the source program as a sequence of characters and groups them into meaningful units called tokens. It removes blanks, tabls, comments and produces tokens for the next phase.

2. Purpose / Functions :

- Reads the input characters from the source code.
- Groups characters into tokens.
- Removes white spaces, comments and unnecessary characters.
- Records identifiers in the symbol table.
- Reports lexical errors.
- Passes tokens to Syntax Analyzer.

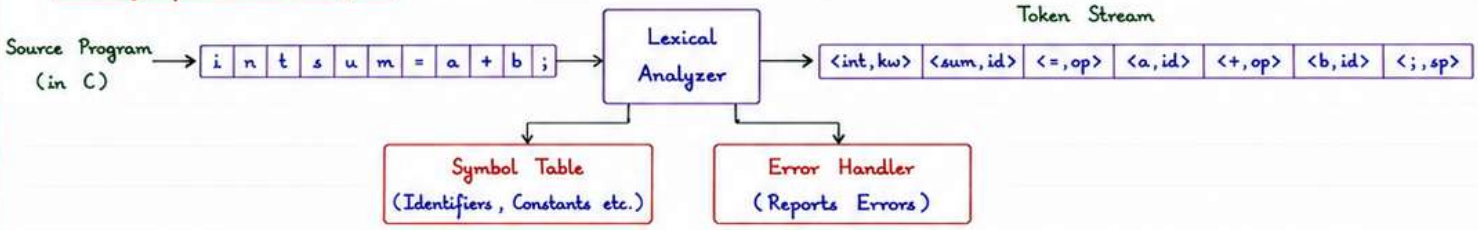
3. Position of Lexical Analysis in Compiler :



4. Input and Output of Lexical Analyzer :

Input	Output
Stream of characters (Source Program)	• Token stream (<token, attribute> pairs) • Symbol table entries • Errors (if any)

5. Working of Lexical Analyzer :



6. Example :

Source Program : `int total = a1 + 25 ;`

Tokens Generated :

Token	Lexeme	Token Name	Attribute
1	int	keyword	–
2	total	identifier	pointer to Symbol Table
3	=	operator	–
4	a1	identifier	pointer to Symbol Table
5	+	operator	+
6	25	constant	25
7	;	special symbol	;

7. Tasks Performed in Lexical Analysis :

(i) Scanning : Reads characters one by one.

(ii) Token Formation : Groups characters into tokens.

(iii) Removal of Blanks and Comments :
Ignores white spaces, tabs, new lines and comments.

(iv) Symbol Table Creation :
Stores information about identifiers, constants etc.

(v) Error Detection :
Detects illegal characters and reports lexical errors.

8. Lexical Errors :

- Invalid characters.
- Unclosed string or comment.
- Identifier too long.
- Malformed number.
- ★ etc.

9. Tools / Generators :

Lexical Analyzer can be written by hand or generated using tools like LEX (or FLEX).
It uses Regular Expressions to recognize tokens.

10. Importance :

- Takes raw input and converts it into meaningful tokens.
- Removes unnecessary characters and simplifies input.
- Prepares the input for Syntax Analysis.
- Plays a key role in detecting lexical errors early.

Important Points for Exams :

★ Lexical Analysis is the first phase of compiler.
★ It groups characters into tokens.
★ It removes spaces, comments and illegal characters.
★ Output is token stream and symbol table entries.
★ It uses Regular Expressions / Finite Automata.
★ Errors detected at this phase are lexical errors.
★ Tokens are passed to Syntax Analyzer.

10. Input Buffering

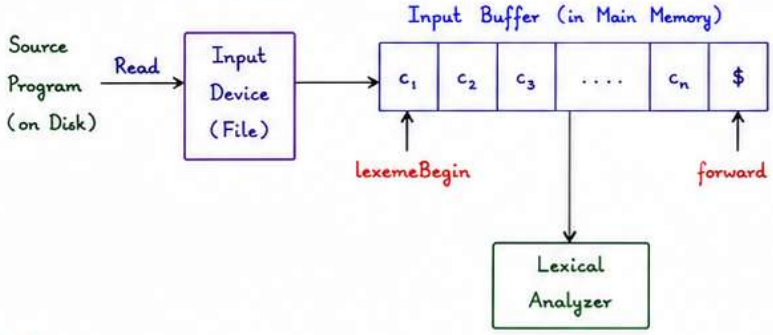
1. Introduction :

Input Buffering is a technique used in a compiler to read the source program efficiently from the input device (file) into main memory. It reduces the number of physical I/O operations by reading large blocks (buffers) of characters at a time instead of reading one character at a time.

2. Purpose / Advantages :

- Minimizes the number of I/O operations.
- Speeds up the lexical analysis.
- Provides a look-ahead facility (we can see next character without reading again).
- Efficiently handles large source programs.
- Works as an interface between the input device and the lexical analyzer.

3. Structure of Input Buffering :



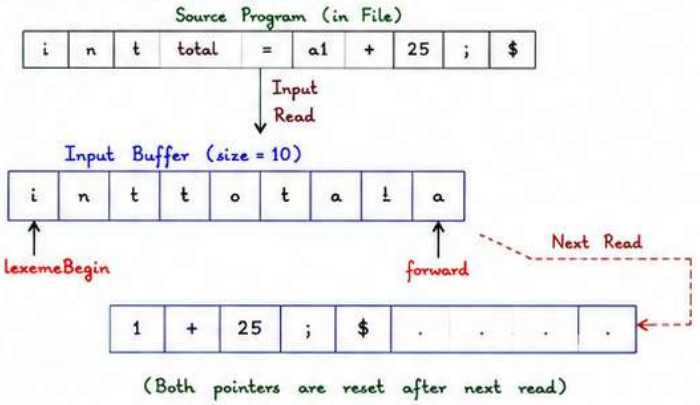
Where,

- Input Buffer is a fixed-size array of characters.
- $ is an end-of-file (EOF) marker appended at the end.
- lexemeBegin points to the beginning of the current lexeme.
- forward points to the next character to be read.

4. Working :

1. A block of characters is read from the input device into the input buffer.
2. Two pointers are maintained :
   - lexemeBegin - points to the first character of the current lexeme.
   - forward - points to the next character to be processed.
3. The lexical analyzer scans characters from lexemeBegin to forward.
4. When forward reaches the end of the buffer (at $), a new block of characters is read into the buffer and both pointers are reset.
5. This process continues until the entire source program is read.

5. Illustration / Example :



6. Key Points :

★ Buffer size is typically between 512 to 8192 bytes.
★ EOF marker ($) is inserted at the end of each buffer.
★ Input buffering is transparent to the lexical analyzer.
★ It improves the overall performance of the compiler.

7. Advantages :

★ Reduces I/O overhead.
★ Provides look-ahead without rereading.
★ Supports efficient scanning and token recognition.

Important Points for Exams :

★ Input buffering reads a block of characters into memory.
★ It uses lexemeBegin and forward pointers.
★ EOF marker ($) is placed at the end of the buffer.
★ When forward reaches $, a new block is read.
★ It acts as an interface between input device and lexical analyzer.
★ It improves the speed and efficiency of the compiler.

11. Tokens

1. Introduction :

A token is a group of characters that have a collective meaning. Tokens are produced by the lexical analyzer (scanner) from the source program. These tokens are then passed to the syntax analyzer for further processing.

2. What is a Token ?

A token is represented as pair (token-name , attribute-value) where ,

- token-name : Identifies the category of token.
- attribute-value : Additional information related to the token (if any).

Example :

Lexeme	Token (token-name , attribute-value)
int	(KEYWORD , int)
total	(ID , pointer to symbol table entry)
=	(ASSIGNOP , –)
25	(NUM , 25)
;	(SEMICOLON , –)

3. Types of Tokens :

Tokens are classified into the following categories :

1. Keywords
2. Identifiers
3. Constants (Literals)
4. Operators
5. Separators (Delimiters)

4. Examples in Source Program :

Source Program : int total = a1 + 25 ;

Tokens Produced :

(KEYWORD , int) ← Keyword
(ID , ptr_1) ← Identifier
(ASSIGNOP , –) ← Operator
(ID , ptr_2) ← Identifier
(PLUS , –) ← Operator
(NUM , 25) ← Constant
(SEMICOLON , –) ← Separator

ptr_1 and ptr_2 point to the entries of 'total' and 'a1' in the symbol table.

5. Token and Lexeme :

- Lexeme : The actual sequence of characters in the source program that matches a pattern.
- Token : The pair (token-name , attribute-value) returned by the lexical analyzer.

Example :

Lexeme → total
Token → (ID , pointer to symbol table entry)

6. Role of Tokens in Compiler :

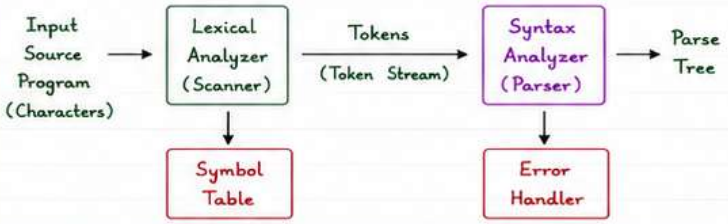
- Tokens serve as input for the syntax analyzer.
- They hide the details of individual characters.
- They simplify the parsing process.
- They help in detecting lexical errors.
- They provide a uniform interface between lexical analysis and syntax analysis.

7. Token Specification using Regular Expressions :

Each token is specified by a regular expression.

Token Category	Regular Expression (Example)
Identifier	[A-Za-z_] [A-Za-z0-9_]*
Integer	[0-9]+
Real Number	[0-9]+\.[0-9]+
Keyword (if, int, while)	if | int | while
Operator (+, -, *, /)	\+ | - | * | /
Relational Operator	== | != | < | > | <= | >=
Separator (; , ( ) { })	; | , | \(| \) | \{ | \}

8. Token Recognition Process :



9. Lexical Errors Related to Tokens :

- Invalid characters.
- Malformed numbers (e.g., 12.3.4)
- Unclosed strings or comments.
- Identifier too long.
- Use of illegal symbols.
- Reserved words used as identifiers.

10. Importance of Tokens :

- Tokens simplify the compiler design.
- They reduce the complexity of syntax analysis.
- They increase the efficiency of the compiler.
- They provide meaningful categories for the parser.
- They help in early detection of lexical errors.

Important Points for Exams :

★ Token = f (token-name , attribute-value).
★ Tokens are generated by lexical analyzer.
★ Tokens are the input to syntax analyzer.
★ Token categories : Keywords , Identifiers , Constants , Operators , Separators.
★ Lexeme is the actual string ; token is its representation.

12. Lexeme and Pattern

1. Introduction :

In lexical analysis, the source program is read as a sequence of characters and grouped into tokens.

A **lexeme** is the actual sequence of characters in the source program that matches the **pattern** of a token.

A **pattern** is a rule or description (usually in the form of regular expression) that describes the set of strings (lexemes) belonging to a token.

2. Lexeme :

- The actual sequence of characters in the source program that is matched by the pattern of a token.
- It is an instance (a string) of a token.
- Different lexemes may be recognized by the same pattern.

Examples :

Token	Pattern (Rule)	Lexemes (Examples)
ID	[A-Za-z_][A-Za-z0-9_]*	sum, total, _temp, value2
NUM	[0-9]+	12, 345, 0, 9876
REAL	[0-9]+\.[0-9]+	12.3, 0.456, 987.001
PLUS	\+	+
ASSIGNOP	=	=
SEMI	;	;

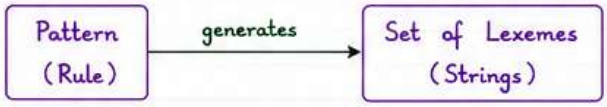
3. Pattern :

- A pattern is the abstract rule that describes the form of lexemes of a token.
- Patterns are specified using Regular Expressions (RE).
- The lexical analyzer uses these patterns to recognize lexemes.

Example :

Token	Pattern (Regular Expression)	Meaning
ID	[A-Za-z_][A-Za-z0-9_]*	Identifier starts with letter or underscore followed by letters, digits or underscore.
NUM	[0-9]+	One or more digits.
REAL	[0-9]+\.[0-9]+	Digits, dot, digits.
PLUS	\+	Plus symbol.
MUL	*	Multiplication symbol.
WS	[\t\n\r]+	One or more white spaces.

4. Relationship between Lexeme and Pattern :



Example :

Pattern for ID : [A-Za-z_][A-Za-z0-9_]*

↓

Lexemes (strings generated) :

x, sum, Total1, _temp, value_2, i123, abc, ...

5. Role in Lexical Analysis :

- The lexical analyzer reads input characters and forms the longest sequence that matches some pattern.
- This sequence is a lexeme of that token.
- The token name (and possibly attribute value) is returned to the parser.
- If no pattern matches the current lexeme, it is reported as a lexical error.

6. More Examples :

Token	Pattern (Regular Expression)	Lexemes (Examples)
IF	if	if
ELSE	else	else
WHILE	while	while
INT	int	int
NOT	!	!
LT	<	<
LE	<=	<=
GE	>=	>=
EQ	==	==
COMMENT	\/\/.*	// this is comment
	\/*[^*]**+([/*][^*]**+)*\/	/* comment */
CHARACTER	'(\\.][^\'\]'))'	'a', 'b', '1', '$'

7. Important Points :

★ Lexeme is an actual string from source program.
★ Pattern is a rule (regular expression).
★ Many lexemes can be generated by a single pattern.
★ Each lexeme must match exactly one pattern (the longest match is chosen).
★ White spaces and comments are usually removed (may produce WS token).

8. Summary :

- Pattern ⟶ Describes the form of token (using RE).
- Lexeme ⟶ Actual string that matches the pattern.

★ Lexical Analyzer uses patterns to recognize lexemes and returns tokens to the parser.
★ This process is the first phase of a compiler.

Important Points for Exams :

★ Define lexeme and pattern.
★ Explain relationship between lexeme and pattern with example.
★ List token patterns using regular expressions.
★ Give examples of tokens with their patterns and possible lexemes.
★ Explain the role of lexeme and pattern in lexical analysis.

13. Specification of Tokens

1. Introduction :

A token is represented as a pair (token-name, attribute-value). The token-name identifies the category of token and attribute-value contains additional information related to the token. Each token is specified by the lexical analyzer and passed to the syntax analyzer.

2. What is Token Specification ?

- It is the definition of all tokens of a programming language.
- It defines token names, their patterns and the information (attributes) associated with them.
- It is used by the lexical analyzer to recognize tokens.

3. Components of Token Specification :

Component	Description
Token Name	Name used to identify the category of token (e.g., ID, NUM, IF).
Pattern (Rule)	Regular expression describing the form of lexemes of that token.
Attribute	Additional information associated with the token (if any).
Example of Lexemes	Some strings that belong to the token.

4. General Form of Token Specification :

Token = < Token Name , Pattern (Rule) , Attribute (if any) >

Example :

< ID , Letter (Letter | Digit)* , Pointer to Symbol Table >

< NUM , Digit (Digit)* , Value >

< IF , if , - >

< PLUS , + , - >

5. Token Specification Table (Example) :

Token Name	Pattern (Regular Expression)	Attribute	Example of Lexemes
ID	[A-Za-z_][A-Za-z0-9_]*	Pointer to Symbol Table	sum, total, _temp, value_2, i123
NUM	[0-9]+	Numeric Value	123, 345, 0, 9876
REAL	[0-9]+\\.[0-9]+ ([eE][+|-]?[0-9]+)?	Real Value	12.3, 0.456, 98.7e-2
PLUS	\+	–	+
MUL	*	–	*
IF	if	–	if
WHILE	while	–	while
SEMI	;	–	;
COMMENT	\/*[^*]**+([^/*][^*]**+)*\/	–	/* this is comment */

('–' indicates that no attribute is needed for that token)

6. Attributes Associated with Tokens :

- Identifier (ID) : Pointer to symbol table entry.
- Numeric constant (NUM) : Actual integer value.
- Real constant (REAL) : Actual floating point value.
- String literal : Pointer to string table or actual string.
- Operators & Keywords : Usually no attribute.
- Comments & White spaces : Usually ignored.

7. Example – Source Program and Tokens Generated :

Source Program : int total = a1 + 25 ;

Token Name	Lexeme	Attribute Value
INT	int	–
ID	total	ptr_1 (to symbol table)
ASSIGNOP	=	–
ID	a1	ptr_2 (to symbol table)
PLUS	+	–
NUM	25	25
SEMI	;	–

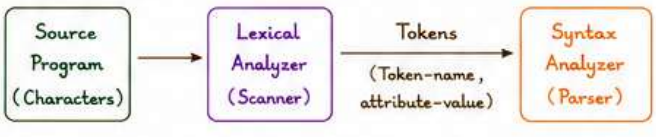
8. Rules for Token Specification :

- Patterns must be precise and unambiguous.
- No two tokens should have patterns that match the same lexeme (longest match rule is used).
- Keywords are usually specified before identifier token patterns.
- Order of specification is important in lexical analyzer.
- White spaces, comments and delimiters are either ignored or treated as separate tokens.

9. Importance of Token Specification :

- Provides a clear interface between lexical analyzer and syntax analyzer.
- Helps in efficient and accurate scanning of source program.
- Ensures correct recognition of tokens.
- Forms the first step in the compilation process.

10. Summary :



Important Points for Exams :

★ Token = (token-name , attribute-value)

★ Token specification defines all tokens using patterns.

★ Pattern is written using Regular Expressions.

★ Lexemes are strings matched by a pattern.

★ Lexical analyzer uses token specification to generate tokens.

14. Recognition of Tokens

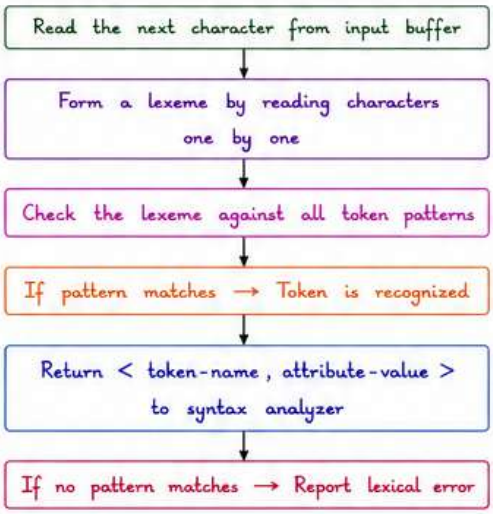
1. Introduction :

The lexical analyzer (scanner) reads the input character stream and groups the characters into lexemes. It then recognizes the lexeme as a valid token according to the token patterns (rules). Recognition of tokens is the main task of lexical analysis.

2. What is Token Recognition ?

- Token recognition is the process of identifying the next lexeme in the input and determining its token class.
- It converts a sequence of characters into a token using the patterns (regular expressions).
- It also returns the attribute value (if any).

3. Process of Recognition of Tokens :



4. Recognition Algorithm (General Steps) :

(i) Set lexeme := ε (empty string).
(ii) Read next character 'c' from input buffer.
(iii) Append 'c' to lexeme.
(iv) If lexeme matches some pattern (possibly with more characters), then go to step (ii).
(v) If lexeme does not match any pattern, then remove last character from lexeme.
(vi) Identify the token whose pattern matches the lexeme.
(viii) Return < token-name, attribute-value >.
(viii) Repeat from step (i) for next token.

5. Example :

Source Program : int total = a1 + 25 ;

Step	Input Stream	Current Lexeme	Action	Token Recognized
1	int total = a1 + 25 ;	i	match, continue	–
2	nt total = a1 + 25 ;	in	match, continue	–
3	t total = a1 + 25 ;	int	not a prefix	INT
4	total = a1 + 25 ;	(space)	ignored	–
5	total = a1 + 25 ;	t	match, continue	–
6	otal = a1 + 25 ;	to	match, continue	–
7	tal = a1 + 25 ;	tot	match, continue	–
8	al = a1 + 25 ;	tota	match, continue	–
9	l = a1 + 25 ;	total	not a prefix	ID
10	= a1 + 25 ;	=	operator found	ASSIGNOP
11	a1 + 25 ;	a1	identifier	ID
12	+ 25 ;	+	operator found	PLUS
13	25 ;	25	number	NUM
14	;	;	separator found	SEMI

Tokens generated : (INT, –) (ID, total) (ASSIGNOP, –) (ID, a1) (PLUS, –) (NUM, 25) (SEMI, –)

6. Methods for Recognition of Tokens :

(A) Finite Automata (DFA) Method
- The patterns are converted into a Deterministic Finite Automaton.
- The input characters are fed to DFA.
- The final state indicates a token.

(B) Regular Expression Method
- Patterns are described using Regular Expressions.
- The input is matched with RE to recognize tokens.
- Usually implemented using RE to NFA → DFA.

7. Tools Used for Token Recognition :

(i) Lex : A lexical analyzer generator tool.
(ii) Flex : Fast Lexical Analyzer generater.
(iii) DFA / NFA : Used to implement token recognition.
(iv) Regular Expressions : To specify token patterns.

8. Handling of Errors during Recognition :

- If no pattern matches the current lexeme → Lexical error.
- Illegal character is reported.
- Too long identifier or number → Error.
- Unclosed string or comment → Error.
- After reporting error, scanner discards characters until a valid token can be formed again.

9. Importance of Token Recognition :

- Converts raw input into meaningful tokens.
- Hides the details of input from parser.
- Simplifies the syntax analysis.
- Detects lexical errors early.
- Essential first phase of a compiler.

Important Points for Exams :

★ Token recognition converts characters into tokens.
★ It uses patterns (regular expressions) to match lexemes.
★ Lexeme is the actual string; token is its category.
★ If no pattern matches → lexical error.
★ DFA / RE based methods are used for recognition.
★ Token recognition is first step of compilation.
★ Returns < token-name, attribute-value > to parser.

15. Design of Lexical Analyzer

1. Introduction :

A lexical analyzer (scanner) is the first phase of a compiler. It reads the source program as a stream of characters and groups them into meaningful units called tokens. Designing a lexical analyzer involves deciding its structure, input buffering scheme, token specification, recognition technique and interface with the syntax analyzer.

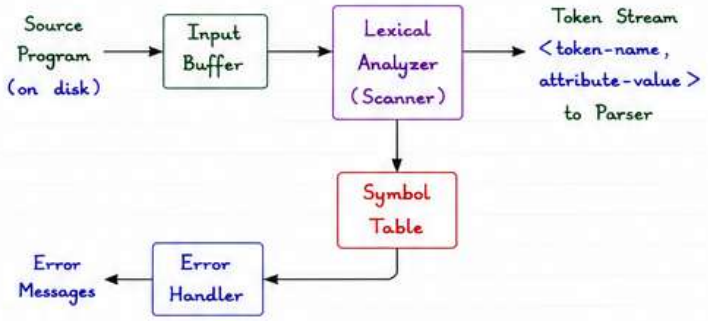
2. Role of Lexical Analyzer in Compiler :

- Reads input characters and forms lexemes.
- Recognizes tokens according to their patterns.
- Returns < token-name, attribute-value > to parser.
- Reports lexical errors.
- Eliminates comments and white spaces.
- Acts as an interface between source program and parser.

3. Design Issues / Considerations :

1. Specification of tokens (token name, pattern, attributes).
2. Recognition method (DFA, RE based, etc.).
3. Input buffering (efficient reading of input).
4. Error handling (illegal characters, overflows, etc.).
5. Eiterface with parser (token stream).
6. Efficiency (time and space).

4. Structure (Design) of Lexical Analyzer :



Components :

(i) Input Buffer : Reads characters from source program.
(ii) Lexical Analyzer (Scanner) : Recognizes lexemes and produces tokens.
(iii) Symbol Table : Stores identifiers and other information.
(iv) Error Handler : Handles and reports lexical errors.
(v) Interface : Sends tokens to syntax analyzer.

5. Specification of Tokens (used in Design) :

Token	Pattern (Rule)	Attribute	Example of Lexemes	Meaning
ID	Letter (Letter | Digit)*	Pointer to symbol table	sum, total, _temp, value2	Identifier
NUM	Digit+ (. Digit+)?	Numeric value	123, 45.67, 0.987	Number
REAL	Digit+ . Digit+ ([eE] [+-]? Digit+)?	Real value	12.3, 0.456, 98.7e-2	Real Number
IF	if	–	if	Keyword
PLUS	+	–	+	Operator
SEMI	;	–	;	Separator
COMMENT	/* (not * or /)* */	–	/* this is comment */	Comment

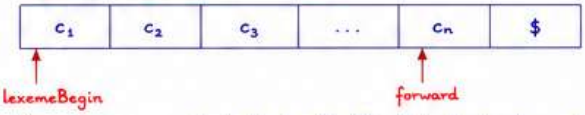
6. Recognition Methods (Design Choices) :

(A) Finite Automata (DFA) based :
- Regular expressions are converted into DFA.
- Input is processed by the DFA to recognize tokens.
- Fast and widely used.

(B) Regular Expression based :
- Patterns are written using Regular Expressions.
- Tools like Lex / Flex use this method.

(C) Hand-coded (Ad-hoc) Scanner :
- Scanner is written manually in a programming language.
- Used in small compilers.

7. Input Buffering in Design :



- Characters are read in blocks (buffers) instead of one by one.
- Two pointers are maintained :
  lexemeBegin → beginning of current lexeme
  forward → next character to be read
- $ represents end-of-file (EOF).

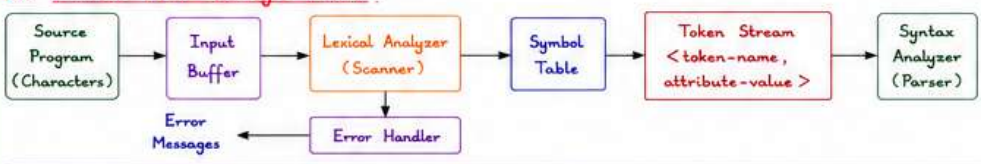
8. Interface with Parser :

- Scanner returns tokens one at a time to the parser.
- Each token is returned as a pair < token-name, attribute-value >.
- Parser requests next token when needed.
- Example : Parser calls `getNextToken()` to get next token.

9. Error Handling in Design :

- Illegal character → Report lexical error.
- Identifier too long → Report error or ignore extra part.
- Number out of range → Report overflow error.
- Unclosed comment → Report error.
- After reporting error, scanner recovers and continues.

10. Overall Flow (Design View) :



Advantages of Good Design :

★ Efficient memory and time usage.
★ Easy to maintain and extend.
★ Accurate token recognition.
★ Early detection of lexical errors.
★ Smooth interface with parser.

Important Points for Exams :

★ Lexical analyzer is the first phase of compiler.
★ It converts character stream into token stream.
★ Design includes specification, recognition, buffering, error handling and interface.
★ DFA and Regular Expression based methods are most popular.
★ Returns < token-name, attribute-value > to parser.
★ Eliminates white spaces and comments.

16. Lexical Analyzer Generator

1. Introduction :

A lexical analyzer generator is a tool that takes as input a set of token patterns (specified using regular expressions) and actions and automatically generates the source code of a lexical analyzer (scanner). It saves time and effort and produces efficient and reliable scanners.

2. Why do we need Lexical Analyzer Generator ?

- Manual writing of scanner is time consuming.
- Generators automatically build efficient scanners.
- They handle buffer management and error handling.
- Easy to modify token patterns and actions.
- Produces compact, fast and reliable code.

3. Inputs to Lexical Analyzer Generator :

(i) Regular Definitions (e.g., digit = [0-9])

(ii) Token Patterns (Rules) using regular expressions

(iii) Actions to be performed when a pattern is matched

(iv) Auxiliary definitions and options (if any)

Rules (Patterns) and Actions :

Rule No.	Pattern (Regular Expression)	Action (Token Returned)
1	{ws}	/* skip white spaces */
2	{id_start}{id_char}*	return ID
3	{digit}+	return NUM
4	"if"	return IF
5	"else"	return ELSE
6	[+*/=<>;()]	return that symbol
7	.	report error

4. Working of Lexical Analyzer Generator :

1. Read Specification (regular definitions, patterns, actions)
2. Convert Regular Expressions to Non-deterministic Finite Automata (NFA)
3. Combine all NFA's into a single NFA (using ε-transitions)
4. Convert NFA to Deterministic Finite Automaton (DFA) using subset construction
5. Minimize the DFA (remove equivalent states)
6. Produce tables and driver code for the scanner (Lexical Analyzer)

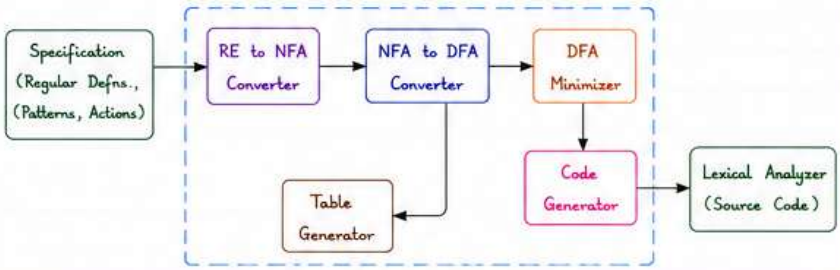
6. Output of Generator :

- Tables representing the DFA (transition table, accept state table).
- Code for input buffering and state transition.
- Code to execute actions when tokens are recognized.
- Error handling code.

7. Tools (Common Lexical Analyzer Generators) :

- Lex – Developed by Mike Lesk.
- Flex (Fast Lexical Analyzer) – Popular open source tool.
- JLex – Java based lexical analyzer generator.
- ANTLR – Powerful parser / lexer generator.

8. Architecture of Lexical Analyzer Generator :



9. Features of a Good Generator :

- Accepts regular expressions as input.
- Generates efficient and compact code.
- Handles large input files.
- Provides good error handling.
- Allows easy modification and portability.

10. Advantages and Disadvantages :

Advantages :

- Saves development time and effort.
- Automatically handles buffer management.
- Produces fast and optimized code.
- Easy to maintain and extend.
- Reduces chances of programming errors.

Disadvantages :

- Limited to tokens that can be described using regular expressions.
- Large DFAs may lead to large tables and more memory usage.
- Understanding the tool is necessary.

Summary :

A lexical analyzer generator takes token patterns and actions as input and automatically produces a lexical analyzer (scanner) that converts the input character stream into a stream of tokens.

Important Points for Exams :

★ Generator takes regular expressions and actions as input.
★ It converts RE → NFA → DFA → Min DFA.
★ It generates tables and source code of scanner.
★ Output is stream of tokens in the form <token-name, attribute-value>.
★ Lex and Flex are popular lexical analyzer generators.
★ First phase of a compiler is lexical analysis.
★ Lexical analyzer removes white spaces and comments.

17. LEX Tool

1. Introduction :

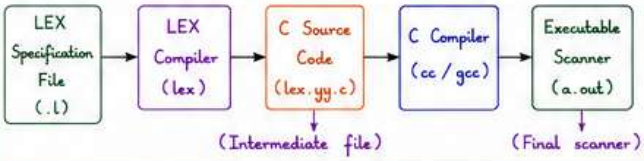
LEX is a tool (lexical analyzer generator) used to generate lexical analyzers (scanners) from high level specification. It is developed by Mike Lesk and is a standard UNIX tool.

It reads an input file describing patterns and actions and produces C source code of a lexical analyzer named lex.yy.c.

2. Purpose of LEX :

- To generate fast and efficient lexical analyzers.
- To reduce manual coding of scanners.
- To handle input buffering and pattern matching.
- To easily modify tokens and actions.
- To provide portability (works on UNIX systems).

3. How LEX Works :



4. Structure of LEX Specification File :

Definitions Section (Optional)	Declarations, macros, definitions.
Rules Section (Mandatory)	Patterns and actions.
User Subroutines Section (Optional)	C functions and main() function.

Format :

```
%{
  C declarations
%}
Definitions
%%
Rules (pattern { action })
%%
User subroutines (main() etc.)
```

5. Example of LEX Program :

```
%{
#include <stdio.h>
%}
digit       [0-9]
letter      [a-zA-Z]
id          {letter}({letter}|{digit})*
%%
{digit}+        { printf("NUM\t%s\n", yytext); }
{id}            { printf("ID\t%s\n", yytext); }
\n              { /* ignore newline */ }
[\t]+           { /* ignore blanks */ }
.               { printf("Unknown\t%s\n", yytext); }
%%
int main(){ yylex(); return 0; }
```

```
Input  :  total = a1 + 25 ;
Output :  ID total
          ASSIGN =
          ID a1
          PLUS +
          NUM 25
          SEMI ;
```

6. LEX Commands :

Command	Description
lex filename.l	Generates lex.yy.c from filename.l
cc lex.yy.c -lfl -o a.out	Compiles and links to produce executable
./a.out	Runs the lexical analyzer
lex -t filename.l	Prints the generated C code on terminal
lex -v filename.l	Prints version information
lex -h	Displays help

7. Components Used by LEX :

- Patterns (Regular Expressions) : Describe tokens.
- Actions : C code executed when pattern matches.
- yytext : String containing matched lexeme.
- yyleng : Length of matched lexeme.
- yylex() : Function called to perform scanning.
- Input Buffer : Buffers input characters for efficiency.
- DFA Engine : Matches patterns using DFA.

8. Regular Expression Symbols in LEX :

Symbol	Meaning
.	Matches any single character (except newline)
*	Matches zero or more occurrences
+	Matches one or more occurrences
?	Matches zero or one occurrence
[ ]	Character class (e.g., [a-z])
|	Alternative (OR)
( )	Grouping
^	Start of line
$	End of line

9. Advantages of LEX Tool :

- Automatically generates efficient scanner.
- Handles large inputs and buffering.
- Easy to specify tokens using regular expressions.
- Portable across UNIX systems.
- Reduces development time and manual errors.

10. Limitations of LEX Tool :

- Works only on UNIX/Linux systems.
- Limited look-ahead (uses longest match rule).
- Not suitable for context-sensitive tokens.
- Complex actions may reduce efficiency.

11. Applications :

- Used in compilers for lexical analysis.
- Used in interpreters and language processors.
- Used in text processing tools.
- Used in log analyzers, data validators, etc.

Important Points for Exams :

- ★ LEX is a lexical analyzer generator.
- ★ It takes patterns and actions as input and generates C code.
- ★ Output of LEX is lex.yy.c (C source code).
- ★ It uses Regular Expressions to describe tokens.
- ★ yytext contains the matched lexeme.
- ★ The longest match rule is applied.
- ★ First phase of a compiler is lexical analysis.
- ★ LEX removes white spaces and comments.