# Unit 1
# Chapter 1
# Introduction

## Structure of the Unit

1. Compilers
2. Analysis of Source Program
3. The Phases of a Compiler
4. Cousins of the Compiler
5. The grouping of phases
6. Compiler- Construction tools.
7. The Role of Lexical Analyzer
8. Input Buffering
9. Specifications of Tokens
10. Recognition of Tokens.

## 1.1 Learning Outcomes

After the successful completion of this unit, the student will be able to:

1. Recall the basic concept of a compiler and its role in software development.
2. Explain the main components and phases involved in the compilation process.
3. Describe the purpose of a lexical analyzer and its relationship with other compiler phases.
4. Apply the knowledge of lexical analysis to identify and perform tasks performed by the lexical analyzer, such as token recognition and input buffering.

## 1.2 Compilers

Compilers play a vital role in software development by translating high-level programming language programs into machine code that computers can execute. They are essential tools for transforming human-readable code into executable programs. Understanding the basics of compilers can enhance your programming skills and enable you to build efficient and robust software. An ***algorithm*** is defined as a finite set of instructions that accomplish a particular task. All algorithms must have the following features.

**Definition:**

A compiler is a program that takes source code written in a high-level programming language as input and converts it into machine code(Fig 1.1). Its main purpose is to bridge the gap between human-readable code and the computer's binary language, making it possible to execute the program on a specific hardware platform.

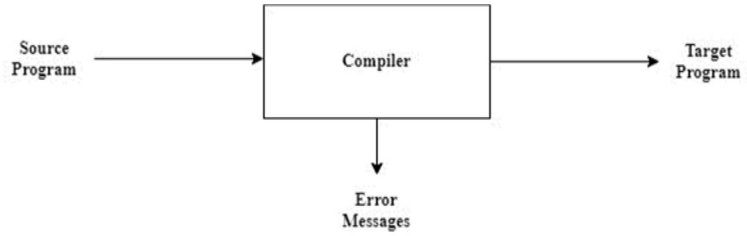


**Fig 1.1** A compiler

## 1.3 Analysis of Source Program

Analysis of source programs consists of three phases:

1. **Linear(Lexical) Analysis:** Read the source program character by character, group the characters into *tokens* . Token is a sequence of characters having a collective meaning.
2. **Hierarchical Analysis(Syntax Analysis pr Parsing):** In this phase the tokens are grouped hierarchically grouped into statements with collective meaning.
3. **Semantic Analysis:** This phase performs type checking so that components of a program fit together meaningfully.

**Linear analysis:**

This phase is also called as **lexical analysis** or **scanning**. Consider the statement

$$energy = mass * speed \wedge 2$$

The above statement is grouped into following tokens:

1. The identifier $energy$.
2. The assignment operator $=$.
3. The identifier $mass$.
4. The operator for multiplication $*$.
5. The identifier $speed$.
6. The raise to operator $\wedge$.
7. The number 2.

The blanks separating the tokens are eliminated during **preprocessing** or **lexical analysis**.

**Hierarchical Analysis:**

The phase is alternatively known as the **syntax analysis** phase or **parsing**. It organizes the tokens into statements according to their syntactic categories. These categories are depicted by parse trees. Figure 1.2 illustrates the parse tree for an assignment statement. The construction of this tree relies on operator priorities, with lower priority operators closer to the root and higher priority operators further down the tree. The hierarchical structure is usually expressed by recursive rules.

For example, the following rules define an *expression*.

1. An *identifier* is an *expression*.
2. A *number* is an *expression*.
3. If *expression1* and *expression2* are expressions then
    a. $expression1 + expression2$ is an expression.
    b. $expression1 * expression2$ is an expression.
    c. $(expression)$ is an expression.

Rules 1 and 2 are basis rules whereas rule3 is recursive.

The parse tree in Fig 1.2 is constructed using these rules. The parse tree in Fig 1.2 describes the syntactic structure of the input **energy = mass * speed ∧ 2**.

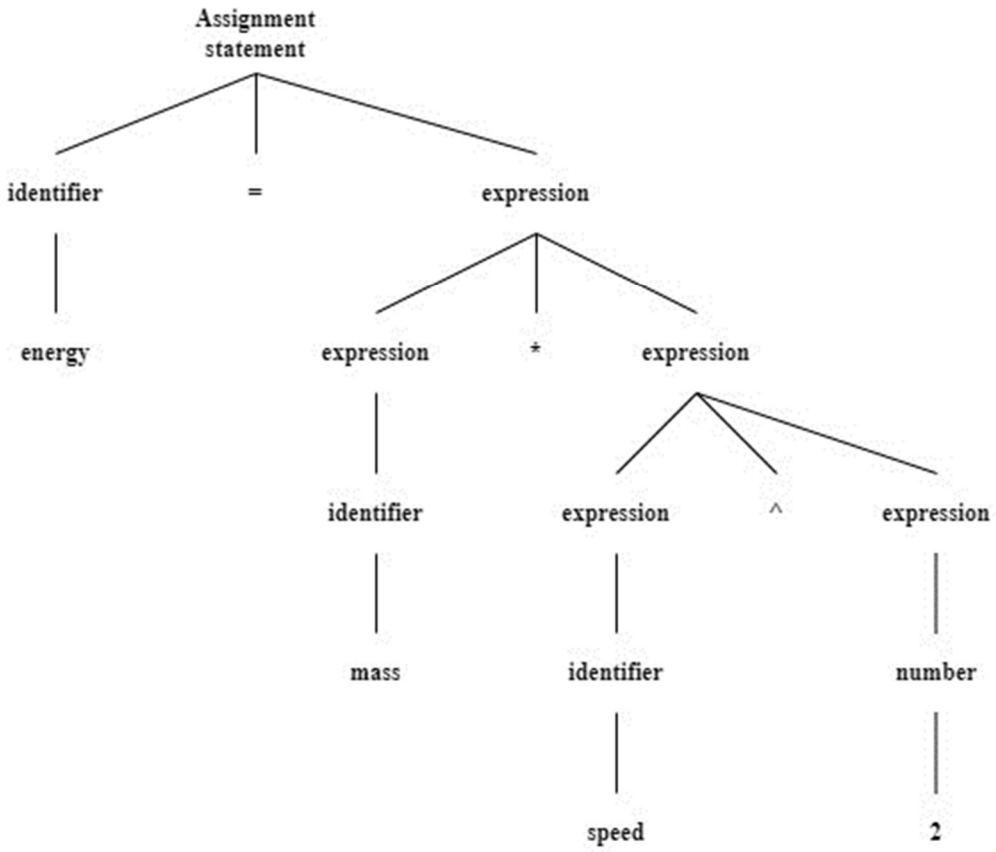


**Fig 1.2** Parse tree for the expression $energy = mass * speed \wedge 2$

**Semantic Analysis:**

This phase performs type checking and, if necessary, type conversion. During type checking, the compiler verifies whether the operators have the appropriate types of operands. For instance, when dealing with array indices, it ensures that they are integers; otherwise, the compiler detects and reports an error. However, certain languages may permit type conversion for specific operators. For instance, in the expression $a + b * 10$, both "a" and "b" could be real numbers, while "10" is an integer. In such cases, the compiler converts the integer value of "10" to a real number.

## 1.4 The Phases of a Compiler

The compilation process can generally be divided into several phases. Each phase transforms the source program from one representation to another. The typical phases of a compiler are shown in Fig 1.3.

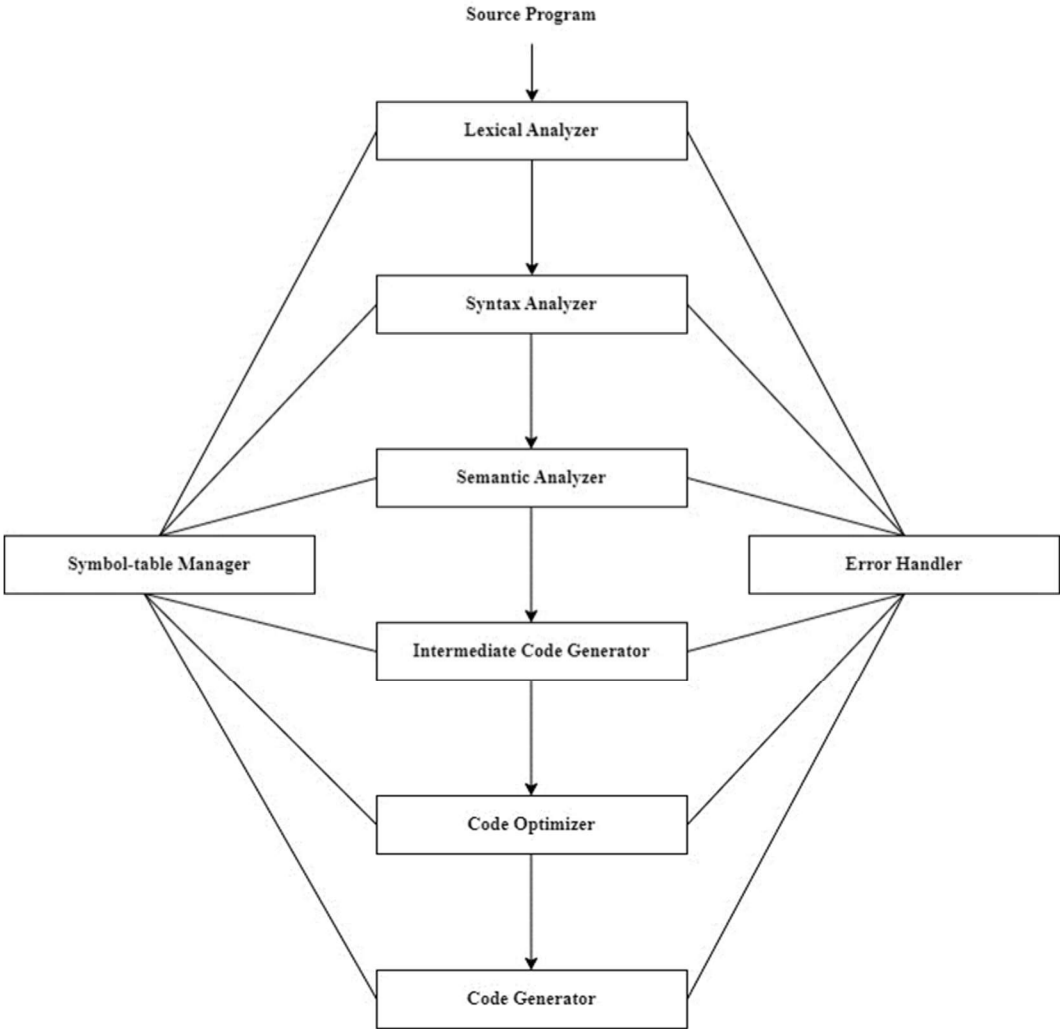


**Fig 1.3** Phases of the compiler

As discussed in section 1.3 the first three phases are analysis phases.

**Lexical Analysis:** Also known as scanning, this phase reads the source code character by character and groups them into meaningful units called tokens. A token represents a logically cohesive sequence of characters, such as identifiers, keywords, numbers, punctation symbols etc. It removes unnecessary whitespaces and comments.

**Syntax Analysis:** This phase, also called parsing, analyzes the structure of the source code based on a grammar defined by a formal language. It creates a parse tree or an abstract syntax tree (AST) that represents the syntactic structure of the program.

**Semantic Analysis:** This phase checks the semantics or meaning of the program. It verifies that the source code follows the rules of the programming language and performs tasks like type checking, variable scoping, and symbol table construction.

**Intermediate Code Generation:** The compiler may generate an intermediate representation of the source code. This intermediate code is usually more abstract and closer to the target machine code than the high-level source code. This phase separates the machine independent and machine dependent phases i.e., the analysis phases(First three) and synthesis phases(the remaining phases). The intermediate code should have two properties; it should be easy to produce and easy to translate to the target code.

**Optimization:** This phase optimizes the intermediate code to improve the efficiency of the generated code. It applies various techniques like constant folding, loop optimization, and dead code elimination to produce optimized code.

**Code Generation:** In this phase, the compiler translates the intermediate code or the AST into the target machine code or bytecode. It may involve mapping the high-level language constructs to low-level instructions or translating to a virtual machine code.

**Code Optimization:** After the code generation, additional optimizations are applied to the target machine code. These optimizations focus on improving performance, reducing code size, and enhancing other aspects of the generated code.

It's worth noting that the specific phases and their order may vary depending on the compiler and the target language. Different compilers may have additional or combined phases to suit their requirements.

## 1.5 Cousins of the Compiler

In this section, we will explore the issues surrounding the input to a compiler, which can come from one or more preprocessors. Additionally, we will delve into the processing required for the machine code produced by the compiler before it can be executed on the target machine.

**Preprocessors:**

Preprocessors produce input to the compilers.

1. *Macro processing:* Allow the users to define macros(short hands) for longer code segments.
2. *File Inclusion:* Facilitate inclusion of header files into the source program, especially while using the library functions.
3. *Rational Preprocessors:* Augment older languages with modern flow of control and data structuring facilities.
4. *Language Extensions:* Language extensions refer to additional features or capabilities added to a programming language to enhance its functionality or address specific requirements. These extensions typically build upon the existing syntax and semantics of the language.

**Assemblers:**

Some compilers produce relocatable machine code that can be directly passed to the loader/linker. Other compilers generate the assembly code as output which is further passed to the assembler. Assemblers take the assembly code as input and generate relocatable object code as output. Assembly code is the mnemonic version of the machine code, i.e., instead of binary codes for the operations mnemonics are used. Also, names are given to the memory locations and registers.

**Loaders and Link-Editors:**

In the context of software development, loaders are components or programs responsible for loading and preparing executable code, data, or resources into memory for execution by a computer system. Loaders take the relocatable machine code as input, alter the relocatable addresses and place the altered instructions and data in the memory at proper locations.

Link editors, also known as linkers or link loaders, are software tools that combine multiple object files or modules generated by a compiler into a single executable program or library. Their primary purpose is to resolve dependencies, perform address relocation, and generate a final executable or library that can be executed or linked to by other programs. Link editors are an essential part of the software development process and play a crucial role in creating executable code from source code.

## 1.6 Grouping of Phases

The compilation process typically consists of several phases, which can be grouped into three main categories: front-end, middle-end, and back-end. These phases are organized based on the order in which they are executed during the compilation process and the type of tasks they perform. Here's a brief introduction to each group:

1. **Front-End:**

- **Lexical Analysis:** This phase, also known as scanning, reads the source code character by character and groups them into meaningful units called tokens. It removes whitespace, comments, and other irrelevant information.
- **Syntax Analysis:** Also known as parsing, this phase analyzes the structure of the program based on the grammar of the programming language. It builds an abstract syntax tree (AST) that represents the hierarchical structure of the program.
- **Semantic Analysis:** In this phase, the compiler checks the program for semantic correctness. It verifies that the program follows the rules and constraints of the programming language. This includes type checking, name resolution, and other semantic checks.

2. **Middle-End:**

- **Intermediate Code Generation:** The middle-end phase involves the generation of an intermediate representation (IR) of the program. The IR is a lower-level and platform-independent representation that simplifies the subsequent optimization and code generation stages.
- **Code Optimization:** This phase focuses on improving the efficiency and performance of the program. It applies various transformations to the intermediate code, such as constant folding, loop optimization, dead code elimination, and more. The goal is to produce optimized code that preserves the program's semantics but executes more efficiently.
- **Intermediate Code Generation (continued):** Depending on the compiler's design, intermediate code generation may occur in multiple stages. After optimization, another pass of intermediate code generation may be performed to refine the IR or prepare it for further transformations.

3. **Back-End:**

- **Code Generation:** The back-end phase generates the target machine code or assembly language that can be executed directly by the target hardware. It maps the optimized intermediate code to the specific instructions and memory layout of the target architecture.
- **Register Allocation:** This phase assigns variables and intermediate results to the limited number of available processor registers. It aims to minimize the use of memory accesses, which can be slower, by efficiently utilizing the registers for storing frequently used values.
- **Instruction Selection and Scheduling:** In this phase, the compiler selects appropriate machine instructions to implement each operation from the intermediate code. It considers the target architecture's instruction set and scheduling constraints to generate efficient code.
- These phases illustrate a typical organization of compiler tasks into front-end, middle-end, and back-end categories. However, the actual compilation process can vary across different compilers and programming languages, and additional phases or variations of the mentioned phases may exist.

**Passes:**

One pass is one scan over the input. Generally, many phases of a compiler are grouped in a single pass. Since there is lot of variation in the way the phases are grouped in to passes by different compilers, we prefer to discuss compiling around phases rather than passes.

## 1.7 Compiler Construction Tools

There are several popular tools available for compiler construction that help developers build compilers and related language processing tools more efficiently. These tools provide various functionalities, such as parsing, lexical analysis, code generation, optimization, and more. Here are some widely used compiler construction tools:

1. **Parser Generators:** These produce syntax analyzers using the context free grammar. Writing syntax analyzer requires a large fraction of the intellectual effort and also a large amount of running time of the compiler. YACC and Bison, JavaCC, Coco/R and Gold parser are few parser generators used.
2. **Scanner Generators:** Scanner generators, also known as lexical analyzer generators, are tools that automate the process of generating lexical analyzers or scanners. These tools take regular expressions or patterns as input and generate code that can tokenize input streams based on those patterns. Scanner generators are an essential component of compiler construction and language processing. Some popular scanner generators are LEX/FLEX, RE2C, Ragel etc.

3. **Syntax Directed Translation Engines:** These produce routines that traverse the parse tree and generate the intermediate code. Some popular syntax directed engines are ANTLR, YACC and Bison, JavaCC, Tree-Sitter etc.

4. **Automatic Code Generators:** Translation of each operation of intermediate code is defined by certain rules. Automatic Code Generators take these rules and the intermediate code as input and generate the target machine code.

5. **Data Flow Engines:** Good code optimization depends on the data flow analysis. Data flow analysis gathers the information about how values are transmitted from one part of the program to each other part. Data flow engines gather the information by performing the data flow analysis.

## 1.8 Lexical Analysis

### 1.8.1 The Role of The Lexical Analyzer

Lexical analysis, also known as scanning, is the initial phase of the compilation process in which the source code of a program is divided into a sequence of tokens. Tokens are the smallest meaningful units of a programming language, such as keywords, identifiers, operators, constants, and punctuation symbols. These Tokens are identified by a lexical analyzer, also called a lexer or scanner, and passed on to the parser for further processing. The interaction between the lexer and parser is shown in Fig 1.4.

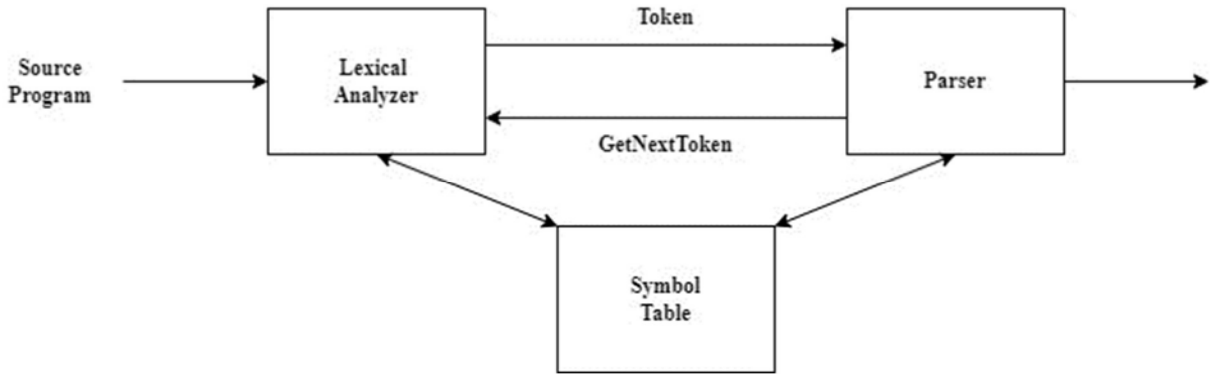


**Fig 1.4** Lexer parser interaction

The parser communicates with the lexer by issuing a "GetNextToken" command, prompting the lexer to read the input characters until it recognizes the next token. Once identified, the lexer sends the identified token back to the parser.

Since the lexical analyzer reads the source program one character at a time, it may also perform certain secondary tasks. They are

- Removal of white spaces and comments,
- Preprocessing,
- Keep track of the number of newline characters and associate line numbers with error messages.

### 1.8.2 Issues in Lexical Analysis

The lexical analysis phase and parser are kept as separate phases for the following reasons:

1. **Simpler Design:** The conventions used to specify the tokens and syntactic constructs(statements in programming languages) are different. While designing a new language, separate conventions for lexer and parser lead to a cleaner design.
2. **Compiler efficiency is improved:** Since the lexer reads the source program one character at a time specialized buffering techniques can be used to significantly improve the performance of the compiler
3. **Compiler portability is enhanced:** The source language peculiarities and anomalies can be restricted to the lexical analyzer.

### 1.8.3 Tokens, Patterns, Lexemes

In lexical analysis and few other phases, we often come across the terms token, lexeme and patterns.

**Token:** A token is a terminal symbol in the grammar or the source language, i.e., tokens are the building blocks of the programming language constructs. Tokens represent the smallest meaningful units of code.

**Pattern:** It is the rule describing the tokens. Usually, regular expressions are used to represent the rules.

**Lexeme:** A sequence of characters in the source program that is matched by a pattern for a token.

| Token | Sample Lexemes | Informal Description of the pattern |
|---|---|---|
| ID(identifier) | sum, prod12, d2x, marks1, marks2 | Letter followed by letter or digits |
| NUM(number) | 103, -2345, 34.67, 54.3E21 | Any numeric constant |
| if | if | if |
| for | for | for |
| RELOP(relational operator) | <, <=, >, >=, =, <> | <, <=, >, >=, =, <> |
| Literal | “compiler design’ | Any sequence of characters between “ and “ except “ |

**Table 1.1** Examples of tokens, patterns and lexemes

### 1.8.4 Attributes for Tokens

When two or more lexemes match the pattern for a token, the lexical analyzer must provide additional information about the lexemes to the remaining phases of the compiler. The tokens are important in making the parsing decisions, whereas the attributes help the translation of tokens. The lexer returns the token and its attribute to the parser, whenever parser request for a new token, i.e., lexer returns a pair of the type $< TOKEN, Attribute >$ to the parser. The attribute depends on the token type.

- For identifiers the lexer returns the pair $< ID, attribute >$ to the parser. $attribute$ is a pointer to the matched lexeme in the symbol table.
- For numbers the return value of lexer is the pair $< NUM, attribute >$, where the attribute is the number itself.
- For keywords the lexer returns the token and the attribute is the lexeme itself.
- For operators and special characters, the token and the attribute are the lexeme itself.

For example, when the statement $energy = mass * speed\text{^}2$ is processed by the lexical analyzer it returns the following sequence of tokens.

a. $< \boldsymbol{ID}, pointer\ to\ the\ symbol\ table\ entry\ for\ energy >$ for the identifier $energy$.

b. $< \boldsymbol{Assign - op}, => $ for the $=$ sign.

c. $< \boldsymbol{ID}, pointer\ to\ the\ symbol\ table\ entry\ for\ mass >$ for the identifier $mass$.

d. $< \boldsymbol{mult - op}, * >$ for the * sign.

e. $< \boldsymbol{ID}, pointer\ to\ the\ symbol\ table\ entry\ for\ speed >$ for the identifier$speed$.

f. $< \boldsymbol{raiseto - op}, \text{^} >$ for the ^ sign.

g. $< \boldsymbol{NUM}, 2 >$ for the number 2.

### 1.8.5 Lexical Errors

**Lexical errors:** Also known as lexical or lexical analysis errors, occur when the lexical analyzer (lexer) encounters tokens or characters that do not conform to the defined syntax of a programming language. These errors typically occur during the initial phase of the compilation process. Here are some common types of lexical errors:

1. **Illegal Characters:** The presence of characters that are not recognized or allowed by the programming language. For example, using special characters like "$", "#" or "&" in a variable name in languages that do not permit them.

2. **Invalid Identifiers:** Identifiers are names used to represent variables, functions, or other entities in a program. Lexical errors can occur if an identifier does not follow the rules defined by the language. For example, starting an identifier with a digit, using reserved keywords as identifiers, or containing invalid characters.

3. **Misspelled Keywords:** Using misspelled keywords or reserved words can lead to lexical errors. For example, writing "whille" instead of "while" or "retun" instead of "return".

4. **Numeric Format Errors:** Incorrectly formatted numbers can result in lexical errors. This includes using invalid characters in a number, such as a letter in place of a digit, or using incorrect number formats, like including multiple decimal points in a floating-point number.

5. **String Literal Errors:** Issues related to string literals can lead to lexical errors. For instance, forgetting to close a string literal with a matching quotation mark or using an incorrect escape sequence.

Sometimes the lexical analyzer is unable to proceed as none of the patterns for tokens matches a prefix of the remaining input string. To recover from this, the lexical analyzer skips successive characters of the remaining input string till it finds a well-formed token. This error recovery strategy is called panic mode error recovery.

Other possible error-recovery actions are:

1. Deleting an extra character.
2. Inserting a missing character.
3. Replacing an incorrect character by a correct character.
4. Transposing two adjacent characters.

## 1.9 Input Buffering

Input buffering is a technique used in lexical analysis to efficiently process the input source code by buffering a portion of it in memory. Instead of reading and processing the source code character by character, input buffering allows the lexer to read a chunk of characters into a buffer and operate on that buffer, which improves performance by reducing the number of I/O operations.

The use of input buffering provides several benefits:

**Reduces I/O Operations:** By reading and processing a chunk of characters at a time, the lexer reduces the number of I/O operations required to fetch characters from the source code. This improves efficiency and reduces the overall time required for lexical analysis.

**Improves Performance:** Reading and processing larger chunks of characters at once reduces the overhead associated with reading individual characters, leading to improved performance.

**Facilitates Lookahead:** Input buffering allows the lexer to perform lookahead operations more easily. By having a portion of the input code available in the buffer, the lexer can examine subsequent characters to determine the appropriate tokenization based on the context.

**Simplifies Lexical Analysis Implementation:** Input buffering provides a simpler implementation for the lexer by abstracting away low-level I/O operations. The lexer can focus on token extraction and processing without the need for frequent I/O interactions.

It's important to note that the size of the buffer should be chosen carefully to balance performance and memory usage. Too small of a buffer may result in frequent refilling operations, while an excessively large buffer may consume excessive memory. The buffer size is often determined based on factors such as the available memory resources, expected input size, and performance considerations.
Many buffering schemes can be used, but in this section, we will discuss one scheme. In this scheme, the buffer is divided in to two halves each of $N$ characters, where $N$ is typically the size of the disk block. Lexical analyzer uses the buffering scheme as follows.

- A system read command is used to read $N$ characters in one half.
- If the number of characters is less than $N$, then the special character ***eof*** is placed at the end of the input. The character ***eof*** marks the end of the input. A special character which is not part of any source program may be used to mark the end of input. This character is called the ***sentinel*** character.
- The ***sentinel*** character ***eof*** is also placed at the end of each buffer.
- Two pointers are used while scanning the input in the buffer. They are ***lexeme-beginning*** and ***forward pointer***.
- The pointer ***lexeme-beginning*** is pointing to the first character of the lexeme.
- The ***forward pointer*** is initially at the beginning of the lexeme, and is advanced to the next position each time a character is read from the buffer. The scanning terminates when the lexer reads a break character. The string between these two pointers is the lexeme.
- After processing the lexeme both the pointers are placed at the beginning of the next lexeme and the process continues till the end of the input.

The following code snippet illustrates the how the lexer manages the input buffers.

```
forward = forward + 1;
if forward = eof then begin
    if forward at the end of the first half then begin
        reload the second half
        forward = forward + 1;
    end
    else if forward at the end of the second half then begin
                reload the second half
                move forward to the beginning of the first half
         end
         else /* end of input */
              terminate the lexical analyzer
end
```

## 1.10 Specification of Tokens

Regular expressions play an important role in defining patterns effectively. Below is a concise overview of the terminology necessary to study regular expressions.

1. **Alphabet:** An alphabet is a finite nonempty set of symbols. It is denoted by Σ.

   **Examples:**

   a. Binary alphabet: Σ = {0, 1}

   b. English alphabet: Σ = {a...z, A...Z}

   c. Alphabet for the programming language 'C': Σ = {a...z, A...Z, 0...9, +,-,*, /, %,<,>, &, | …etc.}

2. **Strings:** A string is a sequence of zero or more symbols chosen from a particular alphabet.

   E.g., 001100 is a binary string.

3. **Empty string:** String with zero occurrences of symbols. Empty string is denoted by ε.

4. **Length of a string:** Number of positions in the string. If 'w' is any string its length is denoted by |w|. e.g., w= 01011101 |w| = 8.

   (It is not the number of symbols in the string, for example in the above string there are two symbols '0' and '1', but the length is 8)

5. **Powers of an alphabet:** If Σ is an alphabet then Σk is the set of all strings of length 'k'.

   **Example:**

   a. If $\Sigma = \{0, 1\}$ then $\Sigma^0 = \{\varepsilon\}$,

   b. $\Sigma^1 = \{0, 1\}$

   c. $\Sigma^2 = \{00, 01, 10, 11\}$

   d. $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

6. Set of all the strings over an alphabet is denoted Σ*

   $$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cdots$$

7. If empty string ε is excluded then the set of all nonempty strings is denoted by Σ+

   $$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cdots$$

   $$\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$$

8. **Concatenation of strings:** If 'x' and 'y' are two strings then 'xy' denotes the concatenation of 'x' and 'y'.

   **Example:** Let x=11011 and y=00110 then xy=1101100110 and yx=0011011011.

9. **Language:** Set of strings all of which are chosen from some Σ* is called a language.

   If L is a language on some alphabet Σ then $L \subseteq \Sigma *$.

### 1.10.1 Operations on Languages

1. **Union:** Union of two languages L and M, denoted L U M, is the set of strings in L or M, or both. For example, if L = {0, 11, 001, 101} and M = {ε, 1, 10, 111, 1101}, then L U M = {ε, 0, 1, 10, 11, 001, 101, 111, 1101}.

2. **Concatenation:** Concatenation of two languages L and M, denoted LM, is the set of strings formed by taking any string in L and concatenating any string in M. If L = {0, 01, 110} and M = {1, 10, 100} then LM = {01, 010, 0100, 011, 0110, 01100, 1101, 11010, 110100}

3. **The Closure (or star or Kleene closure):** The closure of a language L, denoted L*, represents the set of those strings formed by taking any number of strings from L, possibly with repetitions and concatenating all of them. For example, if L = {0, 1} then L* is the set of all possible strings of 0's and 1's including ε.

**Example:**

Let $L = \{A, B, \cdots, Z, a, b, \cdots, z\}$, set of upper- and lower-case letters and $D = \{0,1, \cdots, 9\}$, set of digits. Here are few new languages defined by performing the operations specified in the above section.

$L \cup D$ is the set of letters and digits.

$LD$ is the set of strings consisting of a letter followed by a digit.

$D^3$ is the set of three-digit numbers.

$L^5$ is the set of five-letter strings.

### 1.10.2 Building Regular Expressions

- Expressions start with some elementary expressions.
- More expressions are constructed by applying set of operators on elementary and previously constructed expressions.
- Parentheses are used to group operators with their operands.
- Operators for the operations union (U or +), concatenation (dot) and closure (*) are used.

### 1.10.3 Recursive Definition of Regular Expressions

If $E$ is a regular expression, then $L(E)$ denotes the language of $E$.

**Basis:** The basis consists of three parts:

a. The constant $\varepsilon$ is a regular expression, denoting the language $\{\varepsilon\}$ i.e., $L(\varepsilon) = \{\varepsilon\}$

b. The constant ø is a regular expression, denoting the language ø i.e., $L(ø) = ø$.

c. If $a \in \Sigma$, then a is a regular expression. The language of $a$, i.e., $L(a) = \{a\}$.

**Induction:** If E and F are regular expressions then

a. $E + F$ is a regular expression and $L(E + F) = L(E)\ U\ L(F)$.

b. $EF$ is a regular expression and $L(EF) = L(E)\ L(F)$.

c. $E^*$ is a regular expression and $L(E^*) = ((LE))^*$

d. $(E)$ is a regular expression and $L((E)) = L(E)$.

### 1.10.4 Precedence of Regular Expressions

The following is the order of precedence for the operators:

1. The star operator is of highest precedence. It applies only to the smallest sequence of symbols to its left which is a well-formed regular expression.
2. Next precedence is for the concatenation or dot operator.
3. Finally, all unions (+) are grouped with their operands.

**Examples:**

- An identifier in a programming language is a letter followed by a sequence of zero or more letters/digits. It is represented by the regular expression $\boldsymbol{letter}\ (\boldsymbol{letter} \mid \boldsymbol{digit})*$ .
- An unsigned integer constant in a programming language is the sequence of one or more digits and is represented by the regular expression $\boldsymbol{digit}+$ .

### 1.10.5 Regular Definitions

A regular definition is the name given to a regular expression.

**Examples:**

1. $letter \rightarrow A|B|\cdots|Z|a|b|\cdots|z|$
2. $digit \rightarrow 0|1|\cdots|9$
3. $id \rightarrow letter(letter|digit)*$

## 1.11 Recognition of Tokens

In the following section, our focus will be on the process of token recognition carried out by the lexical analyzer. Specifically, we will delve into the recognition of various tokens such as identifiers, numbers and relational operators. To aid in understanding, we will utilize transition diagrams that portray the actions executed by the lexical analyzer when identifying each token.

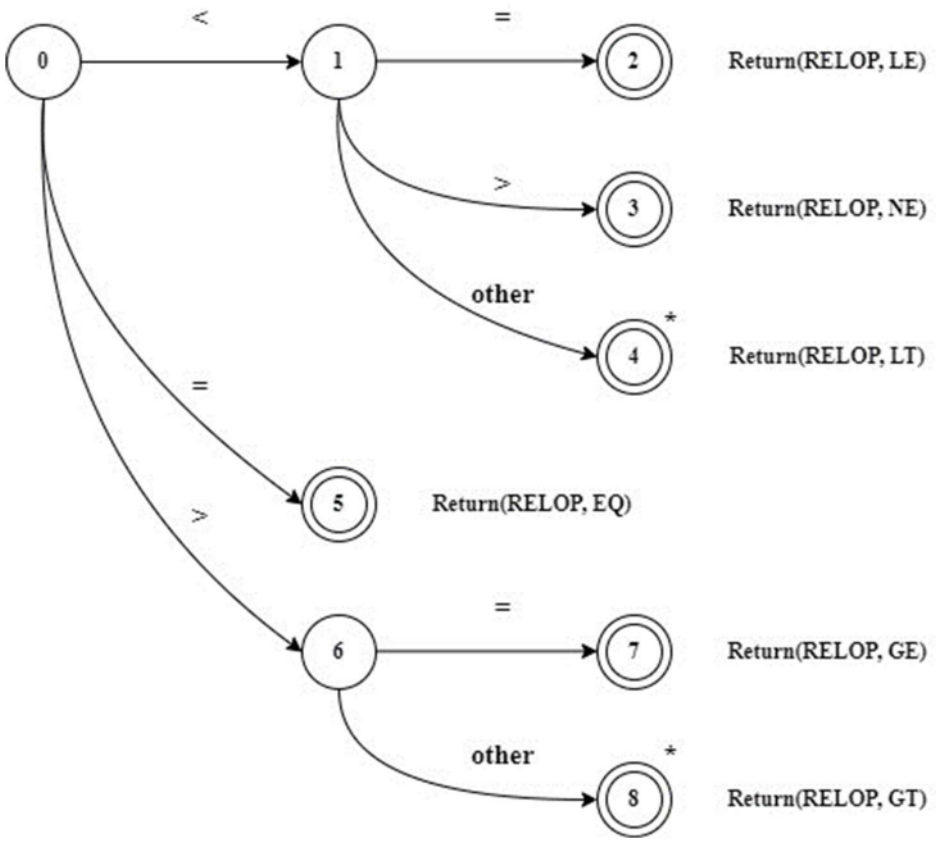


**Fig 1.5** Transition diaram for relational operators

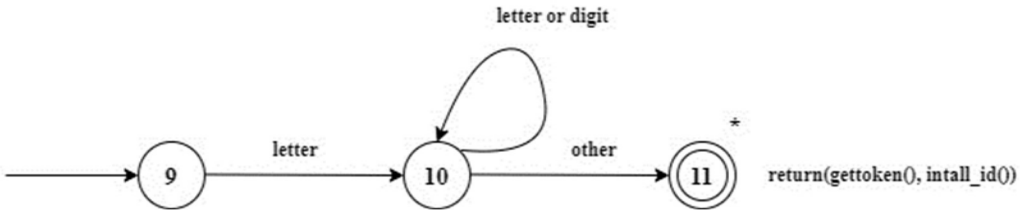


**Fig 1.6** Transition diadgram for identifiers and keywords

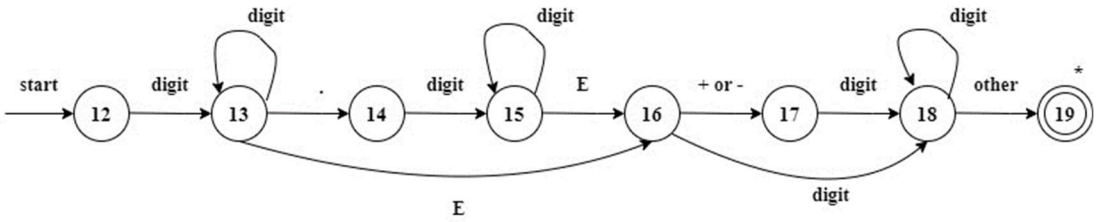


**Fig 1.7** Transition diadgram for numbers

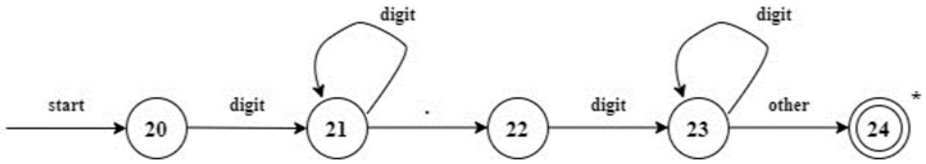


**Fig 1.8** Transition diadgram for real numbers

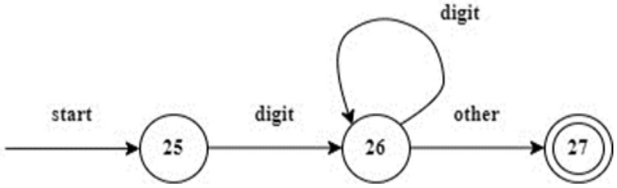


**Fig 1.9** Transition diadgram for integers

## 1.12 Multiple Choice Questions

1. Which of the following best describes the role of a lexical analyzer in the compilation process?
    a. Parsing the source code
    b. Generating machine code
    c. Identifying and tokenizing lexemes
    d. Optimizing the program
2. Input buffering in lexical analysis refers to:
    a. Reading the entire source code into memory
    b. Storing the output of the lexical analyzer
    c. Buffering the input from the keyboard
    d. Reading and storing a portion of the source code for processing
3. Which phase of the compiler is responsible for specifying the structure and characteristics of different tokens?
    a. Lexical analysis
    b. Parsing
    c. Semantic analysis
    d. Code generation
4. Which of the following is NOT a typical step in the process of lexical analysis?
    a. Input buffering
    b. Lexeme matching
    c. Syntax tree construction
    d. Token recognition

5. The primary purpose of the syntax analysis phase in a compiler is to:
    a. Identify and tokenize lexemes
    b. Verify the program's adherence to the language's syntax rules
    c. Optimize the code for better performance
    d. Generate the target machine code
6. What is the purpose of input buffering in the lexical analysis phase of a compiler?
    a. To store intermediate results during parsing
    b. To optimize the execution speed of the compiler
    c. To efficiently read and store a portion of the source code for processing
    d. To identify syntax errors in the program
7. In input buffering, the source code is typically read and stored in:
    a. Registers
    b. Cache memory
    c. Main memory (RAM)
    d. Secondary storage (e.g., hard disk)
8. Which of the following best describes the advantage of input buffering in lexical analysis?
    a. Reduces memory requirements
    b. Accelerates the lexical analysis phase
    c. Enhances error detection and recovery
    d. Improves code optimization

## 1.13 Summary

This unit introduces the topic of lexical analysis in the field of compilers. It touches upon key aspects such as the analysis of the source program, the phases involved in a compiler, and related concepts. Additionally, it highlights the role of the lexical analyzer and its functions, including input buffering and the specification and recognition of tokens.

## 1.14 Keywords

- Source program
- Target program
- Lexical analysis
- Syntax analysis
- Semantic analysis
- Preprocessor
- Assembler
- Loader
- Link editor
- Token
- Pattern
- Lexeme
- Lexical errors
- Panic mode error recovery
- Input buffering
- Sentinel character

## 1.16 Recommended Learning Resources


1. Ellis Horowitz, Sartraj Sahni, S Rajasekaran ., 1998, Fundamentals of Computer Algorithms, 2ndEd., Galgotia Publication.
2. Anany Levitin., 2011, Introduction to The Design and analysis of algorithms, 3rd Ed., Pearson India Publishers.
3. Time and Space, Tab No. 3, Video Lecture URL: https://nptel.ac.in/courses/106106131