# Computer Science & Information Technology

# Compiler Design

Comprehensive Theory

*with* Solved Examples and Practice Questions

MADE EASY Publications

**MADE EASY Publications Pvt. Ltd.**

Corporate Office: 44-A/4, Kalu Sarai (Near Hauz Khas Metro Station), New Delhi-110016
E-mail: infomep@madeeasy.in
Contact: 011-45124660, 8860378007

Visit us at: www.madeeasypublications.org

**Compiler Design**



First Edition: 2015
Second Edition : 2016
Third Edition : 2017
Fourth Edition : 2018
Fifth Edition : 2019
Sixth Edition : 2020
Seventh Edition : 2021
**Eighth Edition : 2022**

# Contents

## Compiler Design



■■■■

# Compiler Design

## GOAL OF THE SUBJECT

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space. In order to understand or construct the compiler one must be aware of its design principles. Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

## INTRODUCTION

In this book we tried to keep the syllabus of Compiler Design around the GATE syllabus. Each topic required for GATE is crisply covered with illustrative examples and each chapter is provided with Student Assignment at the end of each chapter so that the students get the thorough revision of the topics that he/she had studied. This subject is carefully divided into six chapters as described below.

1. Introduction to Compiler: In this chapter we will introduce you to the Analysis-Synthesis model of compilation, various stages of Compilation (namely: lexical, syntax, semantic, intermediate code generation, code optimization and code optimization), grouping of various stages into analysis phase and synthesis phase, passes in compiler and finally we discuss the bootstrapping.
2. Lexical Analysis: In this chapter we will study Tokens, lexemes and their patterns and finally we discuss various ways of specifying tokens.
3. Syntax Analysis (Parser): In this chapter we introduce you the types of parsers, Top down parser: LL (1) parsing, Bottom up parsers: LR parser, SLR parser, CLR parser, LALR parser and Operator precedence parsing.
4. Syntax Directed Translation: In this chapter we will study about the Syntax directed definition, attributes (synthesized and inherited), Construction of Syntax trees, of S-Attributed Definitions, L-Attributed Definitions, Bottom up evaluation of inherited Attributes, dependency graph using SDT, SDT for intermediate code generation.
5. Intermediate Code Generation: In this chapter, we will study about the code generation for program and their various representations. We will also discuss basic blocks and flow graphs.
6. Run Time Environment: In this chapter we will study about the activation trees, control stacks, Storage organization, storage allocation strategies, scope of variables, Symbol table representations, parameter passing.

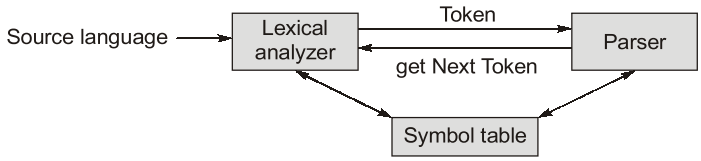
■■■■

CHAPTER 02

# Lexical Analysis

## 2.1 Introduction

- As the first phase of compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of token for each lexeme in the source program. **The stream of tokens is sent to the parser for syntax analysis**.
- Lexical analyzer also interacts with symbol table. As when the lexical analyzer discover a lexeme constituting an identifier, it needs to enter that lexemes into the symbol table. This interaction is shown below:




- Above interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the get Net Token command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

**NOTE**

- Since, lexical analyzer directly deals with the source text, it performs stripping out comments and white space (blank, new line, tab, and perhaps other characters that are used to separate tokens in the input). It also keeps track of the number of new line characters seen, so it can associate a line number with each error. If the source program uses macros, the expansion of macros may also be performed by the lexical analyzer.

## 2.2 Tokens, Patterns and Lexemes

- A **token** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a king of lexical unit e.g. a particular keyword, or a sequence of input characters denoting an identifier.
- A **pattern** is a description of the form that the lexemes of a token may take. In case of a key word as a token, the pattern is just the sequence of characters that form the key word.

- A **lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

*Example:*

| Token | Sample Lexemes | Informal Description of Pattern |
|---|---|---|
| const | const | const |
| if | if | if |
| relation | <, <=, =, <>, >, >= | < or <= or = or <> or > or >= |
| id | pi, count | letter followed by letters and digits |
| num | 3. 14, 286 | any numeric constant |
| literal | "Made Easy" | any characters between " and " except " |

**Example-2.1** Identify lexeme and the respective token in the following C-statement.

printf("Total = %d\n", score);

***Solution:***

printf is lexeme corresponding to identifier score is lexeme corresponding to identifier "Total = %d\n" is a lexeme corresponding to literal.

## 2.3 Recognition of Tokens

In order to understand the following example consider a grammar shown below:

stmt → **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| ∈

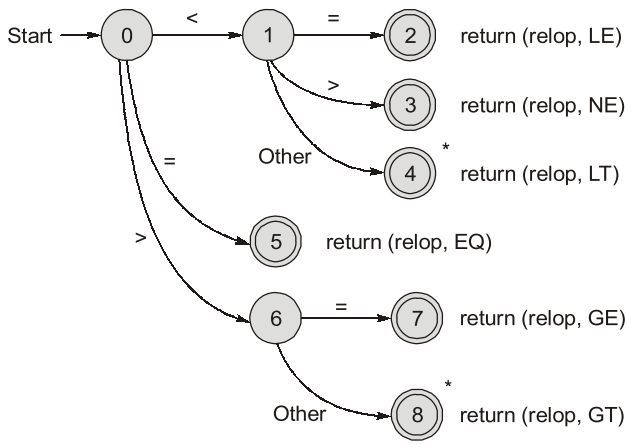
expr → term **relop** term
| term

term → **id**
| **number**

Above grammar represents branching statements.

The terminals of the grammar, which are if, then else, relop, id, and number, are the names of tokens as far as the lexical analyser is concerned.

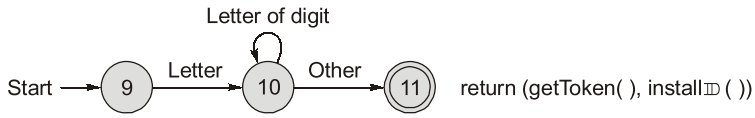
### 2.3.1 Transition Diagram for Relop



Analysis:

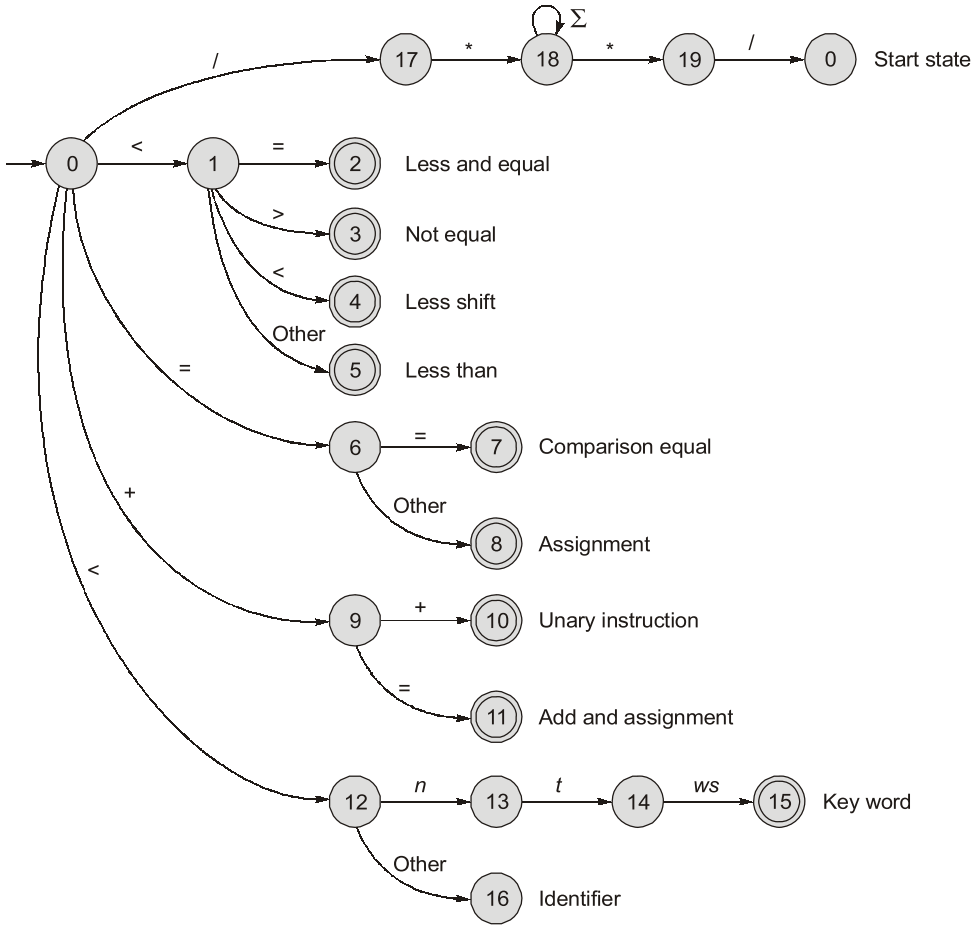
1. We begin in state 0, the start state.
2. If we see < as the first input symbol, then among the lexemes that match the pattern for **relop** we can only be looking at <, < >, or < = . We therefore goto state 1, and look at the next character.
3. If it is =, then we recognize lexeme < =, enter state 2, and return the token **relop** with attribute LE (less and equal).
4. If it is >, the we recognize lexeme < >, enter state 3, and return the token **relop** with attribute NE (not equal).
5. On any other character, the lexeme < and we enter state 4 to return that information.

### 2.3.2 Transition Diagram for Keywords and Identifiers




**Analysis:** Recognizing keywords and identifiers presents a problem usually, keywords like if or then are reserved so they are not identifiers even though they look like identifiers. Thus, although we typically uses a transition diagram like above to search for identifier lexemes, this diagram also the keywords if, then and else of our running example.

### 2.3.3 Transition Diagram for Almost Every Operation



**Example-2.2** A lexical analyzer uses the following patterns to recognize three tokens $T_1$, $T_2$ and $T_3$ over the alphabet {a, b, c}.

$T_1$ : a? (b|c)*a
$T_2$ : b? (a|c)*b
$T_3$ : c? (b|a)*c

Note that '$x$?' means 0 or 1 occurrence of the symbol $x$. Note also that the analyzer outputs the token that matches the longest possible prefix.

If the string bbaacabc is processed by the analyzer, which one of the following is the sequence of tokens it outputs?

(a) $T_1T_2T_3$ (b) $T_1T_1T_3$
(c) $T_2T_1T_3$ (d) $T_3T_3$

***Solution: (d)***

**Note:** Conflicts resolution in lexical is decided based on longer prefix to a shorter prefix.

Input string: bbaacabc
$T_1$ matches bba
$T_2$ matches bb
$T_3$ matches bbaac
Thus, keep $T_3$ token.
Again for abc
$T_1$ does not match
$T_2$ matches ab
$T_3$ matches abc
Thus, keep $T_3$ token again.
Hence, option (d) is true.

## 2.4 Attributes for Tokens

When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler. For example, the pattern num matches both the strings 0 and 1. The lexical analyzer collects information about tokens into their associated attributes. Attributes are used to distinguish different lexemes in a token. Tokens affect syntax analysis and attributes affect semantic analysis.

***Example:*** The tokens and associated attribute-values for the statement A = B * C + 5 are written below as a sequence of pairs:

<id, pointer to symbol-table entry for A>
<assign_op, >
<id, pointer to symbol-table entry for B>
<mult_op, >
<id, pointer to symbol-table entry for C>
<add_op, >
<num, integer value 5 >

## 2.5 Lexical Errors

If none of patterns for tokens matches a prefix of the remaining input, in that case lexical analyzer gets struck and it has to recover from this state to analyze the remaining input.

### 2.5.1 Recovery Methods

The simplest recovery strategy is "Panic mode error recovery". We delete successive characters from the remaining input until the lexical analyzer can find a well-formed token. Other error recovery methods are:

- Delete an extraneous character
- Insert a missing character
- Replace an incorrect character by a correct character
- Transpose two adjacent characters

**Example - 2.3** The C statement if (++$x$ == 5) foo(3);

Find the number of tokens?

***Solution:***

| if | ( | ++ | ID | == | 5 | ) | ID | ( | int | ) | ; |
|---|---|---|---|---|---|---|---|---|---|---|---|

There are 12 tokens.

**Example - 2.4** Consider the regular expression-token mapping given below:

| Regex | Token |
|---|---|
| ca*b | 1 |
| (a\|b)*b | 2 |
| c* | 3 |

Choose the correct output when lexical analyzer scans the following input: "aaabccabbb"

(a) 232 (b) 132

(c) 231 (d) 123

***Solution: (a)***

aaab → 2

cc → 3

abbb → 2

**Example - 2.5** In a compiler the module that checks the token arrangement against the source code grammar is called _____.

(a) Lexical analyzer (b) Syntax analyzer

(c) Semantic analyzer (d) Code optimizer

***Solution: (b)***

- **Lexical analyzer** scan the source code as a stream of characters and counts it into meaning full lexemes.
- **Syntax analyzer** checks the token arrangement against the source code grammar.
- **Semantic analyzer** check whether the parse tree constructed follows the rules of language.
- **Code optimizer** do code optimization of the intermediate code.

**Summary**

- Lexical analyser also called Scanner.
- When syntax analyser request for token then only lexical analyser generate token.
- For an invalid token lexical analyser generate token error.
- Sequence of character are called lexemes.
- Set of character are called token.
- Lexical analyser uses finite state automaton to identify different tokens.
- Strings of letters and digits which is started with letter only are called identifier.
- Lexical analyser scan the lexemes always left to right.

## Student's Assignments

**Q.1** The number of tokens in the following C statement
printf ("$i$ = %d, and $i$ = %$x$", $i$, &$i$);
(a) 3 (b) 26
(c) 10 (d) 21

**Q.2** Consider the following statements:
$S_1$ : The set of string described by a rule is called pattern associated with token.
$S_2$ : A lexeme is a sequence of character in the source program that is matched by pattern for a token.
(a) Both $S_1$ and $S_2$ are true
(b) $S_1$ is true but $S_2$ is false
(c) $S_2$ is true but $S_1$ is false
(d) Both $S_1$ and $S_2$ are false

**Q.3** Match the following errors corresponding to their phase:

Group A
1. Unbalanced parenthesis
2. Appearance of illegal characters
3. Undeclared variables

Group B
A. Syntactic error
B. Semantic error
C. Lexical error

(a) 1 → A, 2 → C, 3 → B
(b) 1 → B, 2 → C, 3 → A
(c) 1 → A, 2 → B, 3 → C
(d) 1 → B, 2 → C, 3 → A

**Q.4** Find the number of tokens in the following C code using lexical analyzer of compiler.

```
main ( )
{
    /* int a = 10; */
    int * u, * v, s;
    u = &s;
    v = &s;
    printf ("%d%d", s, * u);
    // code ended
}
```

**Q.5** Consider the following program:

```
1. main ( )
2. {    int x = 10;
3.      it (x < 20;
4.      else
5.      y = 20;
6. }
```

When lexical analyzer scanning the above program, how many lexical errors can be produced?

**Q.6** In some programming language, an identifier is permitted to be a letter following by any number of letters or digits. If L and D denotes the sets of letters and digits respectively, which of the following expressions defines an identifier?
(a) $(L \cup D)^*$ (b) $L(L \cup D)^*$
(c) $(L \cdot D)^*$ (d) $L \cdot (LD)^*$

**Q.7** Consider the following program segment:

```
main ( )
{
    int a, b;
        a = 5 + 8 +;
        printf("%d", a);
        /*&b = 5; */
}
```

The number of token present in the above program segment _____.

**Q.8** Which of the following is not a functionality of C compiler?
(a) Identifying syntax error
(b) Identifying tokens
(c) Linking
(d) None of these

**Q.9** Match the following groups:

**List-I**
**A.** Lexical analyzer
**B.** Syntax analyzer
**C.** Type checking
**D.** Intermediate code generation

**List-II**
**1.** Checks the structure of the program.
**2.** Analysis of entire program by reading each character.
**3.** High level language is translated to simple machine independent language.
**4.** Checks the consistency requirements in a context of the program.

**Codes:**

| | A | B | C | D |
|---|---|---|---|---|
| (a) | 1 | 2 | 4 | 3 |
| (b) | 2 | 1 | 4 | 3 |
| (c) | 2 | 4 | 3 | 1 |
| (d) | 1 | 4 | 3 | 2 |

**Q.10** In C programming, which of the following is not used as a token separator during lexical analysis?
(a) White space (b) Comment
(c) Semicolon (d) None of these

**Q.11** Consider the C program:

```
main( )
{
    int x = 10;
    x = x + y + z;
}
```

How many tokens are identified by lexical analyzer?

**Q.12** How many tokens are generated by the lexical analyzer, if the following program has no lexical error?

```
main( )
{
    int x, y;
    fl/*gate oat z;
    x =/*exam*/10;
    y = 20;
}
```

**Q.13** Consider the following C program:

```
1.  #include <stdio.h>
2.  main( )
3.  {
4.      int a = 2, b = 3;
5.      char *x;
6.      x = &a = &b;
7.      a = 1xab;
8.      printf("%d%d", a, *x);
9.  }
```

If scanner reads an entire program then find the line number in which lexical error is produced.

**Q.14** Find the regular expression that correctly identifies the variable name in C program.
(a) [a – z][a – zA – Z_0 – 9]*
(b) [a – zA – Z_][a – zA–Z_0 – 9]*
(c) [a – zA – Z_][a – zA – Z0 – 9]*
(d) [a – zA – Z][a – zA – Z_0 – 9]*

**Q.15** Which of the following is not a token in C language?
(a) Semicolon (b) Identifier
(c) Keyword (d) White space

**Q.16** Lexical error is _____.
(a) An error produced by lexical analyzer when an illegal character appears.
(b) An error produced by lexical analyzer when a missing left parenthesis in an expression.
(c) An error produced by scanner when both operator and parenthesis appeared consecutively.
(d) All of the above

**Q.17** Match **Group-I** and **Group-II** and select the correct answer using the codes given below the lists:

Group-I

A. Token
B. Pattern
C. Lexeme

Group-II

1. Sequence of characters in the source program that matches the pattern of a token.
2. A pair consisting of a token name and an optional attribute value.
3. Description of the form that can be accepted.

Codes:

| | A | B | C |
|---|---|---|---|
| (a) | 1 | 2 | 3 |
| (b) | 2 | 3 | 1 |
| (c) | 3 | 1 | 2 |
| (d) | 1 | 3 | 2 |

**Q.18** Consider the following expression of C program:

abcd + (2 – 5 + × 6/2 – +;

How many tokens are generated by the above expression during lexical analysis?

**Q.19** Consider the following program segment:

```
main ( )
{
    int a, b, c;
        a = 50;
        b = &a;
    printf("%d", b);
}
```

The number of tokens in the above C code are _____.

**Q.20** Consider the following statements:

I. The module that checks every character of the source text is called symbol table in a compiler.
II. Keywords of a language are recognized during lexical analysis.
III. Temporary variables are one of the contents of an activation record.

Which of the above statements is/are correct?

(a) I and III only (b) I only
(c) II and III only (d) I and II only

**Q.21** Consider the following C program:

```
# include <stdio.h>
main ( )
{
    int x = 10, y = 12;
    char * a;
    a = &x;
    x = 1xab;
    printf ("%d %d", x, * a);
}
```

Which of the following type of error (earliest phase) is identified during compilation of the program?

(a) Lexical error (b) Syntax error
(c) Semantic error (d) None of these

***Answer Key:***

| | | | | |
|---|---|---|---|---|
| **1.** (c) | **2.** (a) | **3.** (a) | **4.** (34) | **5.** (0) |
| **6.** (b) | **7.** (24) | **8.** (c) | **9.** (b) | **10.** (d) |
| **11.** (18) | **12.** (17) | **13.** (7) | **14.** (b) | **15.** (d) |
| **16.** (a) | **17.** (b) | **18.** (14) | **19.** (28) | **20.** (c) |
| **21.** (a) | | | | |

## Student's Assignments Explanations

**1. (c)**

We have,

printf ( "$i$ = %d, and $i$ = %$x$" , $i$ , & $i$ ) ;
① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Hence there are 10 tokens.

**Note:** & a ⇒ 2 tokens, address operator and identifier.

& & a ⇒ 2 tokens, logical AND and identifier.

**2. (a)**

$S_1$ : The set of string described by a rule is called pattern associated with token.

$S_2$ : A lexeme is a sequence of character in the source program that is matched by pattern for a token.

Both statements are true.

3. (a)
   (i) Errors like appearance of illegal characters, unmatched string comes under lexical phase errors.
   (ii) Misspelled keywords, unbalanced parenthesis appear during syntax analysis phase of compiler.
   (iii) Incompatible type of operands, undeclared variables detected during semantic analysis phase.

4. (34)

```
main ( )
{                                   {Lexical analyzer
    /* int a = 10; */               ignores comment lines}
    int * u, * v, s ;
    u = &s;
    v = &s;
    printf ("%d%d", s, * u);
    // code ended
}
```

5. (0)
   At line 3: No lexical error but produces syntax error by syntax analyzer.
   At line 5: No lexical error but produces semantic error by semantic analyzer (declaration error).
   ∴ There is no lexical error in the above program.

6. (b)
   Identifier is permitted to be a letter followed by any combination of letter and digits. Thus,
   L (L ∪ D)* resembles identifier.

7. (24)

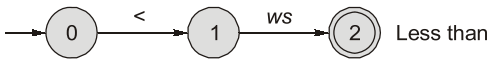
```
   1     2  3
main   (   )
4
{
      5   6  7  8  9
     int  a  ,  b  ;
     10 11 12 13 14 15 16
     a  =  5  +  8  +  ;
       17    18    19  20  21 22 23
     printf  (" % d"  ,  a  )  ;
     /* & b = 5; */
24
}
```

   Total 24 tokens.

8. (c)
   Linking is done by a linker after compilation process.
   Compiler can identify token, generates compilation error which can be lexical, syntax or semantic.

9. (b)
   A. Lexical analyzer checks the entire source code by reading each character.
   B. Syntax analyzer receives stream of tokens and generates parse tree with respect to grammar so that structure is valid or not.
   C. Type checking check the consistency whether context is meaningful or not.
   D. Intermediate code converts high level language to machine language.
   Hence, A → 2, B → 1, C → 4, D → 3

10. (d)
    White space is a token separator as:




    Comment is also a token separator.
    Semicolon is also a token separator trivially.

11. (18)
    Tokens are:
    1. main
    2. (
    3. )
    4. {
    5. int
    6. *x*
    7. =
    8. 10
    9. ;
    10. *x*
    11. =
    12. *x*
    13. +
    14. *y*
    15. +
    16. *z*
    17. ;
    18. }

    Hence there are 18 tokens in total.

**12. (17)**

Tokens are:

1. main
2. (
3. )
4. {
5. int
6. *x*
7. ,
8. *y*
9. ;
10. fl
11. 10
12. ;
13. *y*
14. =
15. 20
16. ;
17. }

Hence there are 17 tokens totally.

**Note:** /* ..... */ is not a token since comments.

**13. (7)**

Line 7 i.e. a = 1*xab*;

This will generate lexical error because identifiers cannot be start with digits.

**14. (b)**

Variable name in C program must starts with either alphabet o underscore and followed by any combination of alphabets, digits and underscore.

Hence, [a – zA – Z_][a – zA–Z_0 – 9]*.

Option (b) is true.

**15. (d)**

White space is not a taken in C language.

**16. (a)**

When an illegal character appears, lexical analyzer generates error.

**17. (b)**

- A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit e.g. a particular keyword or a sequence of input characters denoting an identifier.
- A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
- A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

**18. (14)**

abcd + ( 2 – 5 + × 6 / 2 – + ; ⇒ 14 tokens

**19. (28)**

```
   1   2  3
main (   )
4
{
     5   6  7  8  9  10 11
    int  a  ,  b  ,  c  ;
      12 13 14 15
       a  =  50 ;
      16 17 18 19 20
       b  =  &  a  ;
    21    22   23   24  25 26 27
   printf (" %d"  ,   b   )  ;
28
}
```

**20. (c)**

The module that checks every character of the source text is called lexical analysis, keywords recognized during lexical analysis.

Temporary variables are part of activation record.

**21. (a)**

Statement '7' of the 'C' program which states,

$$x = 1xab;$$

If represents lexical error.

Since, the predefined rule in 'C' language for an integer is "An identifier can only have alphanumeric characters (*a-z*, *A-Z*, 0-9) and underscore (_)".

The first character of an identifier can only contain alphabet (*a-z*, *A-Z*) or underscore (_).