

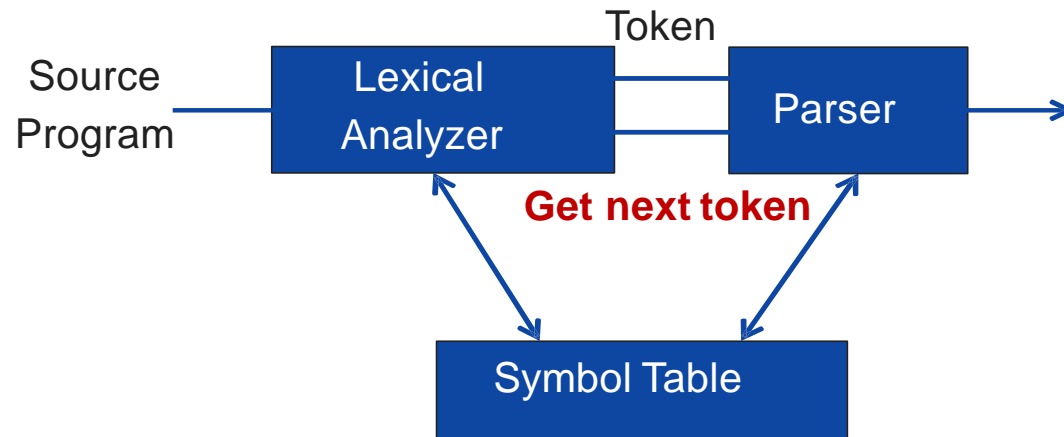
# Compiler Design

## Unit-2 Lexical Analysis

The Role of the Lexical Analyzer, Specification of Lexemes, Tokens and Patterns, Recognition of Tokens.

# Role of Lexical Analyzer

- Upon receiving a **“Get next token”** command from parser, the lexical analyzer reads the input character until it can identify the next token.



# Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

# Lexical Analyzer

1. Remove all whitespace and comments.
2. Identify tokens within a string.
3. Return the lexeme of a found token, as well as the line number it was found on.

# Token

- Sequence of character having a collective meaning is known as **token**.
- The output of our lexical analysis phase is a streams of tokens.
- A token is a syntactic category.
- In English this would be types of words or punctuation, such as a “noun”, “verb”, “adjective” or “end-mark”.
- In a program, this could be an “identifier”, a “floating-point number”, a “math symbol”, a “keyword”, etc...
- Categories of Tokens:
  1. Identifier
  2. Keyword
  3. Operator
  4. Special symbol
  5. Constant

# Pattern

- The **set of rules** called **pattern** associated with a token.
- The pattern is said to match each string in the set.
- For **example**, the pattern to describe an identifier (a variable) is a string of letters, numbers, or underscores, beginning with a non-number.
- The pattern for identifier token, ID, is
$$\text{ID} \rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$$
- Patterns are typically described using **regular expressions.(RE)**

# Lexemes

- The **sequence of character** in a source program **matched with a pattern** for a **token** is called lexeme.
- **Example:** the pattern for the **relation\_op** token contains six lexemes ( =, < >, <, < =, >, >=) so the lexical analyzer should return a **relation\_op** token to parser whenever it sees any one of the six.

# Example

- `count=count+temp;`
- Token: id, operator, punctuation (;)
- Lexeme: count, temp, +



# Attributes of Tokens

- Whenever a lexeme is encountered in a source program, it is necessary to keep a track of other occurrences of the same lexeme i.e. if this lexeme has been seen before or not.
- Symbol Table -> Lexeme are stored in Symbol Table
- The pointer to this symbol table entry becomes an attributes of that particular token.

**Example 3.2:** The token names and associated attribute values for the Fortran statement

`E = M * C ** 2`


are written below as a sequence of pairs.

`<id, pointer to symbol-table entry for E>`  
`<assign_op>`  
`<id, pointer to symbol-table entry for M>`  
`<mult_op>`  
`<id, pointer to symbol-table entry for C>`  
`<exp_op>`  
`<number, integer value 2>`

Note that in certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value. In this example, the token **number** has been given an integer-valued attribute. In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for **number** a pointer to that string. □

```
int count;
```

```
char x[] = " Ganpat University ";
```

Name	Type	Size	Dimension	Line of Declaration	Line of Usage	Address
count	int	2	0	--	--	--
x	char	12	1	--	--	-- 

Name	Type	Size	Dimension	Line of Declaration	Line of Usage	Address

```

main()
{
char a[5];
int x;
.
.
.

```

Name	Type	Size	Dimension	Line of Declaration	Line of Usage	Address
a	char	5 bytes	1	3	6 → 10	
x	int	4 bytes	0	4		

```

1 main()
2 {
3 char a[5];
4 int x;
  .
  .
  .

```

# Process of Lexical Analyzer

- Take source program as input
- Scan program character by character
- Group these characters into lexemes
- Pass the tokens & attributes to the parser

# Lookahead

- Lookahead will typically be important to a lexical analyzer.
- Tokens are typically read in from left-to-right, recognized one at a time from the input string.
- It is not always possible to instantly decide if a token is finished without looking ahead at the next character. For example...
- Is “i” a variable, or the first character of “if”?
- Is “=” an assignment or the beginning of “==”?

# Example

Consider the code:

```
if (i==j) ;  
    z=1;  
else;  
    z=0;  
endif;
```

# Example

Consider the code:

```
if (i==j) ;  
    z=1;  
else;  
    z=0;  
endif;
```

```
if_(i==j); \n\tz=1; \nelse; \n\tz=0; \nendif;
```

This is really nothing more than a string of characters:  
During our lexical analysis phase we must divide this string into  
meaningful sub-strings.



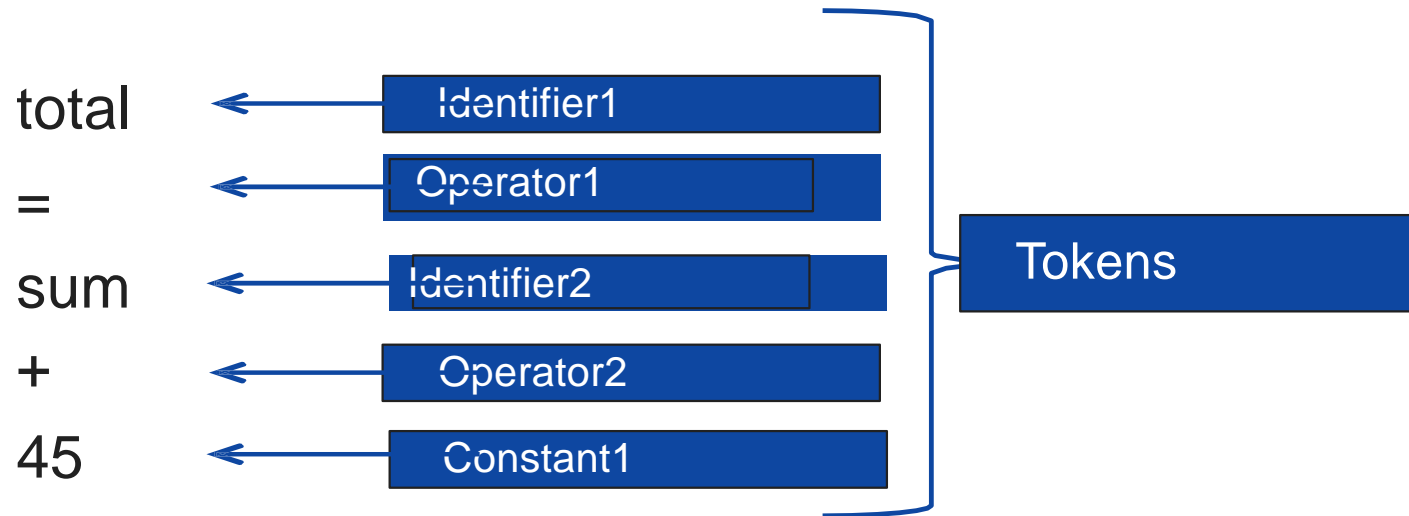
# Example

Token	Informal description	Sample lexemes
<b>if</b>	Characters i, f	if
<b>else</b>	Characters e, l, s, e	else
<b>comparison</b>	< or > or <= or >= or == or !=	<=, !=
<b>id</b>	Letter followed by letter and digits	pi, score, D2
<b>number</b>	Any numeric constant	3.14159, 0, 6.02e23
<b>literal</b>	anything but ", surrounded by "'s	"core dumped"

- `printf("total = %d\n", score);`
- both `printf` and `score` are lexemes matching the pattern for token **id**, and `"Total = %d\n"` is a lexeme matching **literal**.

**Example: total = sum + 45**

**Tokens:**



**Lexemes**

Lexemes of identifier: total, sum

Lexemes of operator: =, + Lexemes of constant:

45

# Recognition of tokens

- The next step is to formalize the patterns:

*digit* -> [0-9]

*Digits* -> digit+

*number* -> digit(.digits)? (E[+-]? Digit)?

*letter* -> [A-Za-z\_]

*id* -> letter (letter|digit)\*

*If* -> if

*Then* -> then

*Else* -> else

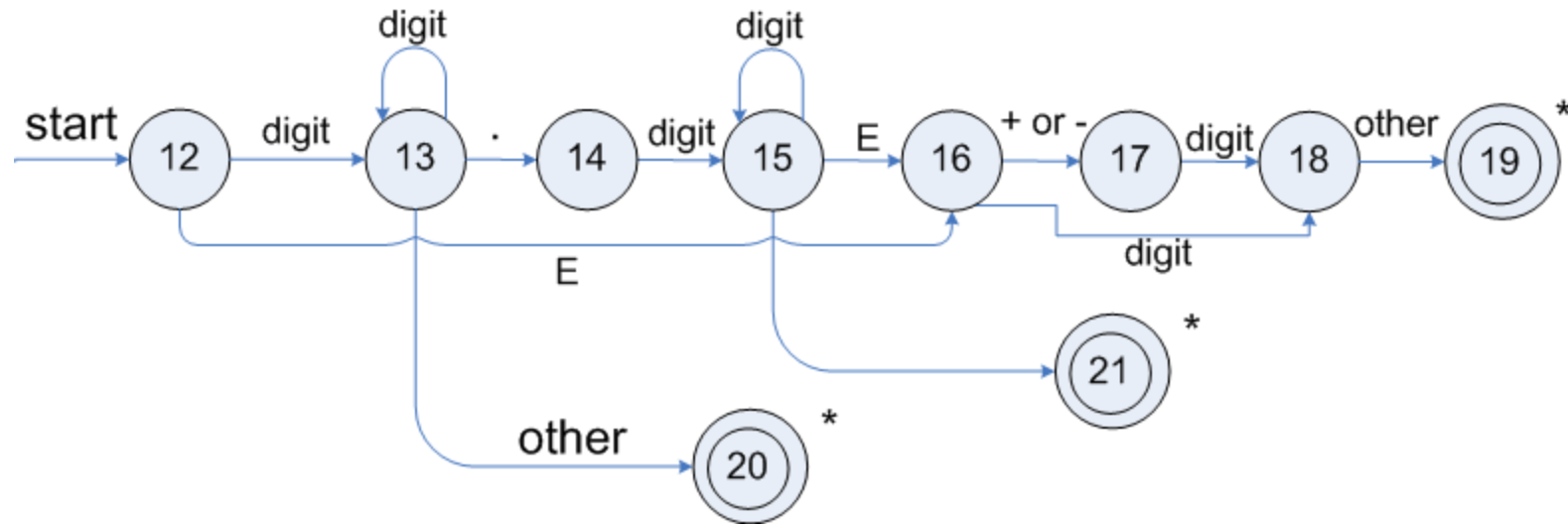
*Relop* -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

*ws* -> (blank | tab | newline)+

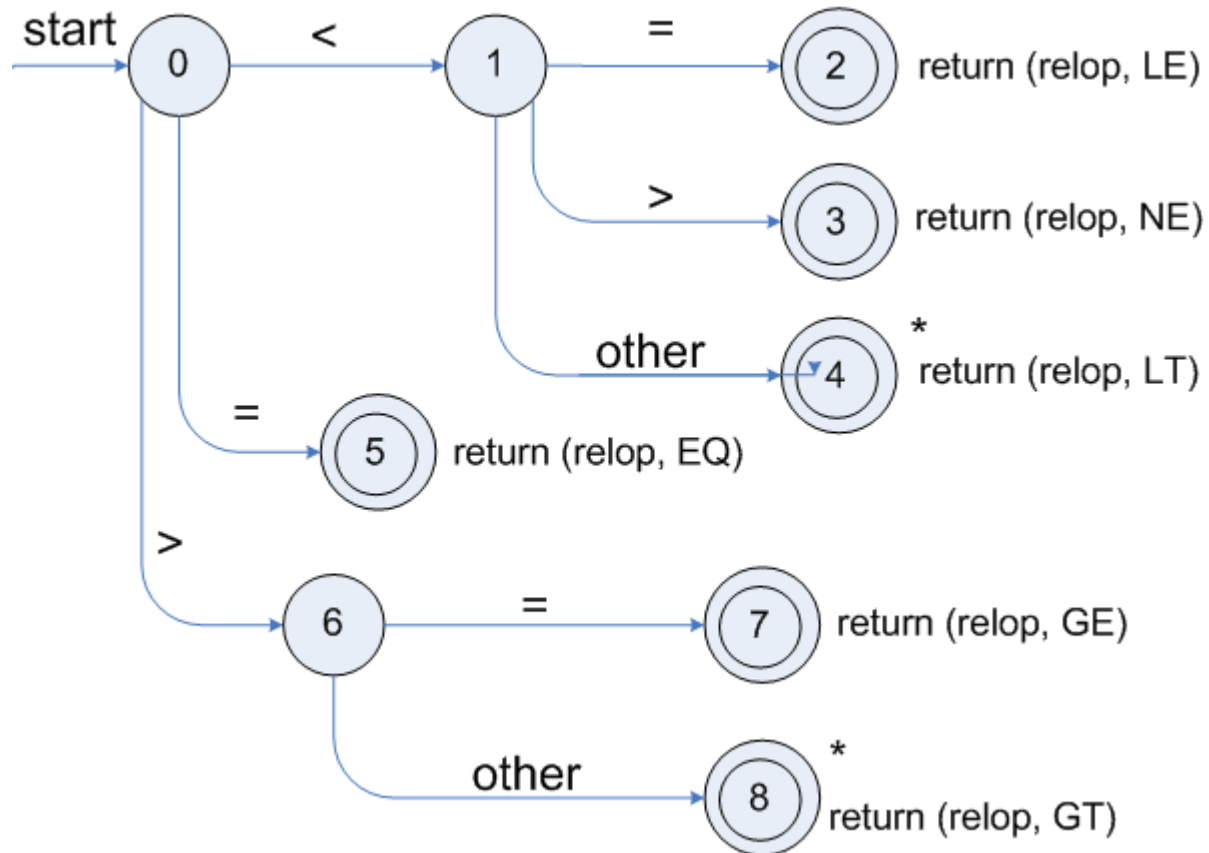
# Transition diagrams

- Transition diagram for unsigned numbers



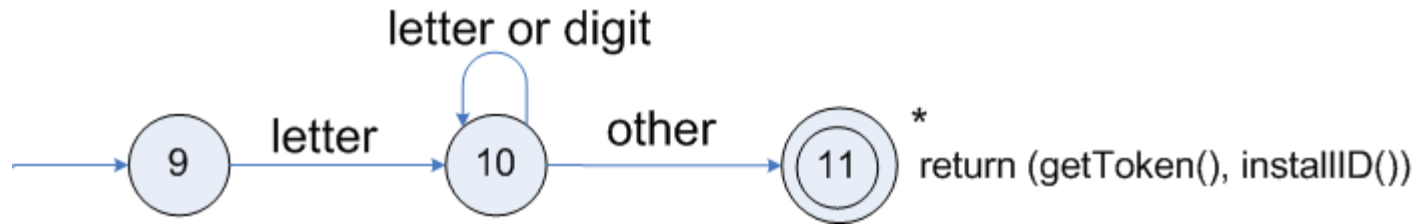
# Transition diagrams

- Transition diagram for relop



# Transition diagrams

- Transition diagram for reserved words and identifiers



# Lexical errors

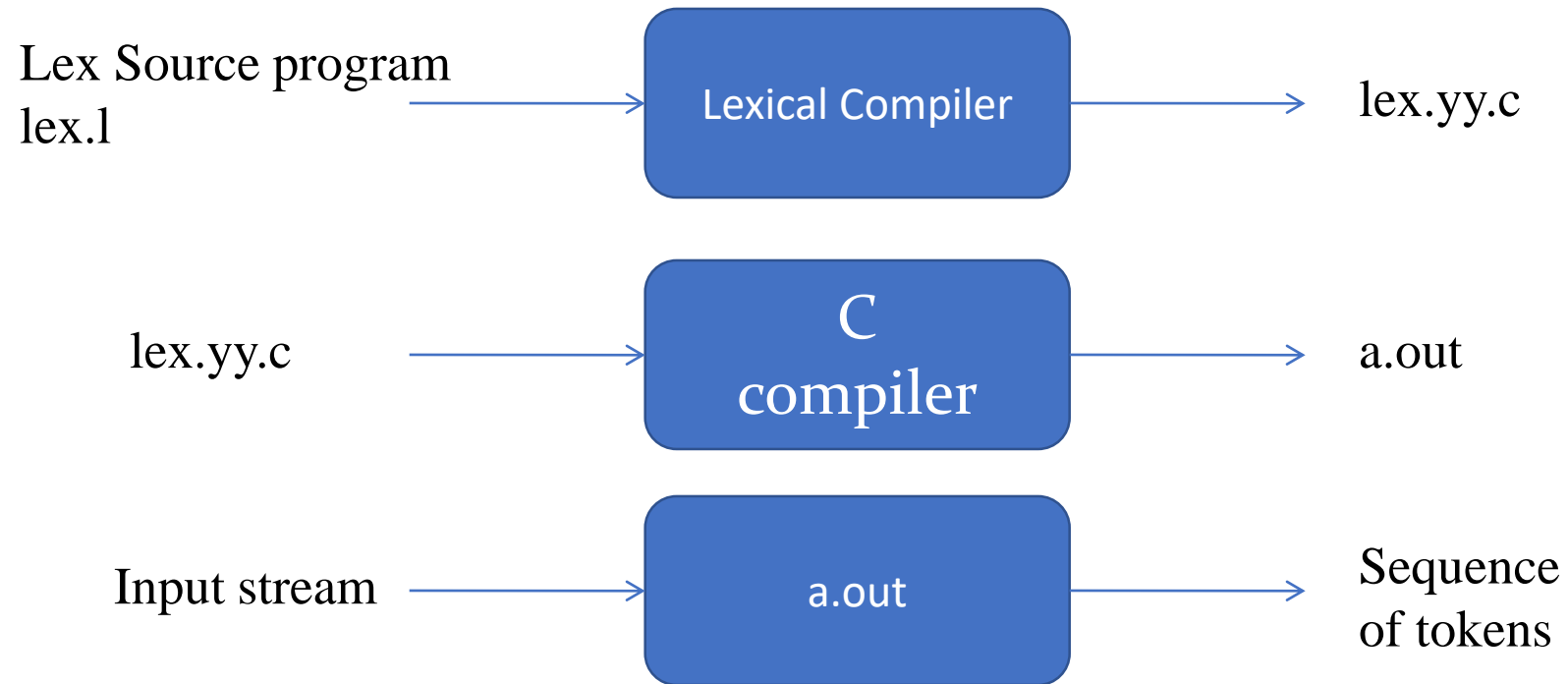
- A Lexical Analyzer may not proceed if no rule/pattern (for tokens) matches the prefix of the remaining input.
- Some errors are out of power of lexical analyzer to recognize:
- `fi (a == f(x)) ...`
- However it may be able to recognize errors like: `d = 2r`
- Such errors are recognized when no pattern for tokens matches a character sequence

# Error recovery

- **Panic mode:** successive characters are ignored until we reach to a well formed token
- **Delete** one character from the remaining input
- **Insert** a missing character into the remaining input
- **Replace** a character by another character
- **Transpose** two adjacent characters



# Lexical Analyzer Generator - Lex



# GATE 2000

- The number of tokens in the given C statement is \_\_\_\_\_.
- `printf("pt = %d, &pt = %x", pt, &pt);`
- A. 10
- B. 21
- C. 20
- D. 11

# GATE 2000

- The number of tokens in the given C statement is \_\_\_\_\_.

- `printf("pt = %d, &pt = %x", pt, &pt);`

- A. 10

- B. 21

- C. 20

- D. 11

- Answer: 10

- We have six C tokens: identifiers, constants, keywords, operators, string literals, and other separators. We have ten tokens within the above `printf` statement.

```
printf
(
"pt = %d, &pt =
%x"
,
pt
,
&
pt
)
```

# GATE 2011

- In any compiler, we can recognize keywords of a language during \_\_\_\_.
- A. Parsing of the program.
  - B. The code generation.
  - C. Lexical analysis of the program.
  - D. Dataflow analysis.

# GATE 2011

- **In any compiler, we can recognize keywords of a language during \_\_\_\_.**

- A. Parsing of the program.
- B. The code generation.
- C. Lexical analysis of the program.
- D. Dataflow analysis.

**Answer: C**

# GATE 2011

- **Consider the following statements:**
  - **(1) The output of the lexical analyzer is groups of characters.**
  - **(2) Total tokens count in the `printf("pt=%d, &pt=%x", pt, &pt);` are 11.**
  - **(3) Symbol table can be implemented with an array and hash table but not a tree.**
  - **Which of the given statement(s) is/are correct?**
- A. Only (1).  
B. Only (2) and (3).  
C. Only (1), (2), and (3).  
D. None of the above.

# GATE 2011

- **Consider the following statements:**
- **(1) The output of the lexical analyzer is groups of characters.**
- **(2) Total tokens count in the `printf("pt=%d, &pt=%x", pt, &pt);` are 11.**
- **(3) Symbol table can be implemented with an array and hash table but not a tree.**
- **Which of the given statement(s) is/are correct?**

- A. Only (1).
- B. Only (2) and (3).
- C. Only (1), (2), and (3).
- D. None of the above.

Answer: D

The correct statement is as follows

- (1) The output of the lexical analyzer is tokens.
  - (2) Total tokens count in `printf("pt=%d, &pt=%x", pt, &pt);` are 10.
  - (3) We can implement a symbol table with an array, hash table, tree, and linked lists.
- So, option (D) is correct.

- The lexical Analysis for any modern programming language such as Java needs the power of which one of the below machine models in a sufficient and necessary sense?
  - A. Finite state [Automata](#)
  - B. Non-deterministic automata
  - C. Deterministic pushdown automata
  - D. Turing machine



- The lexical Analysis for any modern programming language such as Java needs the power of which one of the below machine models in a sufficient and necessary sense?
  - A. Finite state Automata
  - B. Non-deterministic automata
  - C. Deterministic pushdown automata
  - D. Turing machine

**Answer: A**

- In the lexical analysis, finite automata are used to produce tokens in the form of keywords, identifiers, and constants from the input program. Pattern recognition is used to search keywords with string-matching algorithms.

# GATE 2018

- The lexical analyzer uses the given patterns for recognizing three tokens, T1, T2, and T3, over the alphabets a,b,c.
- T1:  $a?(b|c)^*a$  & T2:  $b?(a|c)^*b$  & T3:  $c?(b|a)^*c$ .
- Note: 'x?' means 1 or 0 occurrences of the symbol x. Also, the analyzer outputs the token matching the longest possible prefix. If the analyzer processes the string "bbaacabc", which of the below is the sequence of tokens it outputs?

- A. T1T2T3
- B. T1T1T3
- C. T2T1T3
- D. T3T3

# GATE 2018

- The lexical analyzer uses the given patterns for recognizing three tokens, T1, T2, and T3, over the alphabets a,b,c.
  - T1:  $a?(b|c)^*a$  & T2:  $b?(a|c)^*b$  & T3:  $c?(b|a)^*c$ .
  - Note: 'x?' means 1 or 0 occurrences of the symbol x. Also, the analyzer outputs the token matching the longest possible prefix. If the analyzer processes the string "bbaacabc", which of the below is the sequence of tokens it outputs?
- A. T1T2T3  
B. T1T1T3  
C. T2T1T3  
D. T3T3
- **Answer: D**

• **The output of any lexical analyzer is?**

- A. A parse tree.
- B. Machine code.
- C. Intermediate code.
- D. A stream of tokens.

• **The output of any lexical analyzer is?**

- A. A parse tree.
- B. Machine code.
- C. Intermediate code.
- D. A stream of tokens.

**Answer: D**

- In a compiler the module that checks every character of the source text is called:
  - A. The code generator.
  - B. The code optimiser.
  - C. The lexical analyser.
  - D. The syntax analyser.

- In a compiler the module that checks every character of the source text is called:

- A. The code generator.

- B. The code optimiser.

- C. The lexical analyser.

- D. The syntax analyser.

- **Answer: C**

# GATE 1987

- Using longer identifiers in a program will necessarily lead to:
  - A. Somewhat slower compilation
  - B. A program that is easier to understand
  - C. An incorrect program
  - D. None of the above



- Using longer identifiers in a program will necessarily lead to:
  - A. Somewhat slower compilation
  - B. A program that is easier to understand
  - C. An incorrect program
  - D. None of the above
- **Answer: A**

- Consider the following ANSI C program:

```
int main()  
{  
Integer x;  
return 0;  
}
```

- Which one of the following phases in a seven-phase C compiler will throw an error?
  - A. Syntax analyzer
  - B. Semantic analyzer
  - C. Machine dependent optimizer
  - D. Lexical analyzer

- Consider the following ANSI C program:

```
int main()  
{  
Integer x;  
return 0;  
}
```

- Which one of the following phases in a seven-phase C compiler will throw an error?
  - A. Syntax analyzer
  - B. Semantic analyzer
  - C. Machine dependent optimizer
  - D. Lexical analyzer

**Answer: B**

- **What elements are included in non-tokens components?**
- Macro, extra white spaces, preprocessor directives, and comments are included in non-tokens elements.