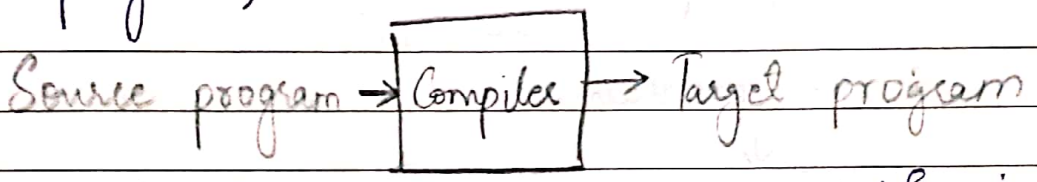
Q. Define Interpreter & Compiler. List Differences between them.

Date
Page

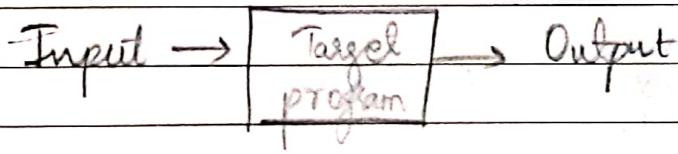
UNIT - I

* Language Processors :- Compiler
  - Interpreter
  - Assemblers
  - Loaders & Linkers
  - Preprocessors
  - Device Drivers.

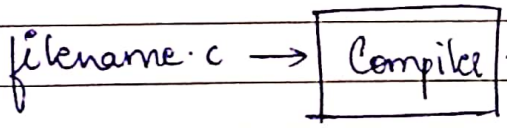
* Compiler :- Compiler is a system software / language processor which converts high-level language (source program) into its equivalent machine level program (target program).

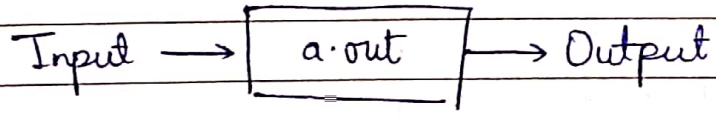


When the target program is an executable code

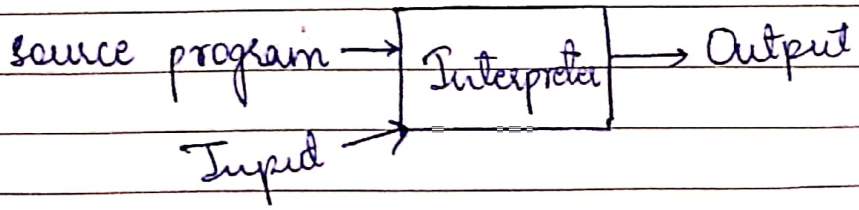


for eg:-

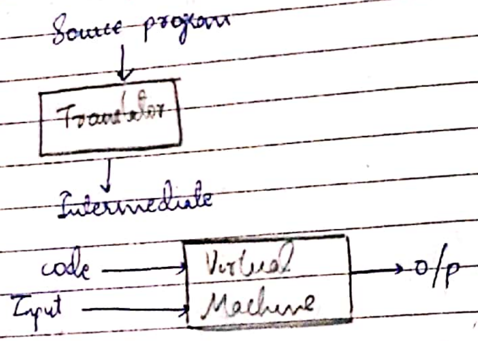




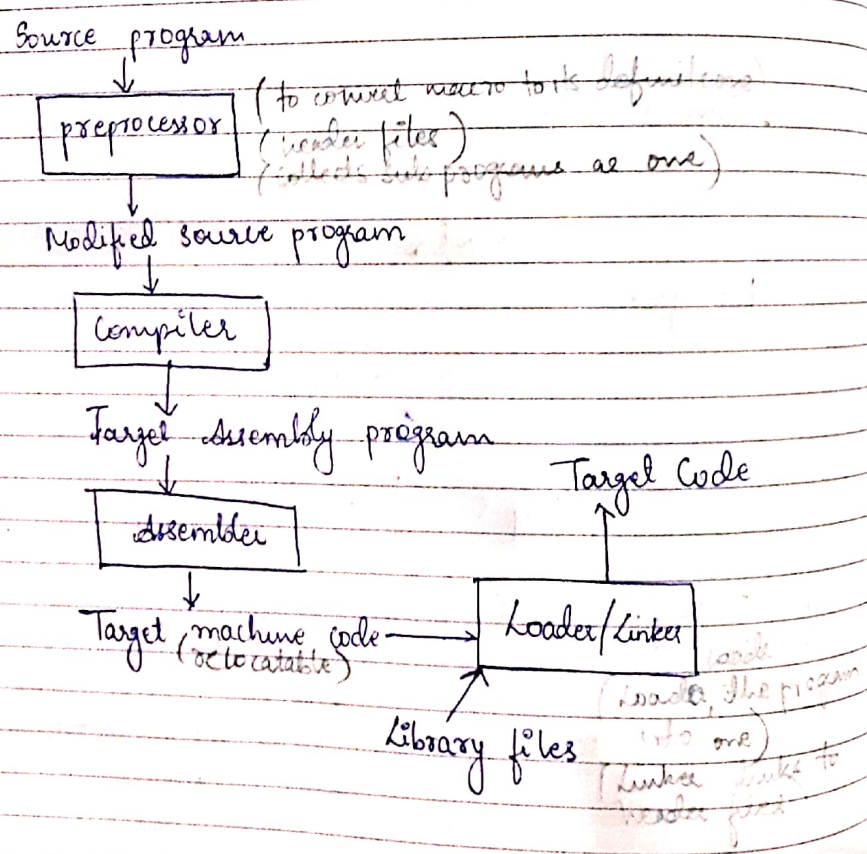
* Interpreter :- Compiles source program line by line
→ Does not generate any target program



★ For Java: Both the Compiler & Interpreter are clubbed.



★ Language processing System:



★ ⇒ Structure of Compiler (Diagram + Phases Explanation + Symbol Table Management).

★ ⇒ Symbol Table Management

★ ⇒ Grouping of phases into passes

1. Lexical Analyser
2. Syntax Analyser
3. Semantic Analyser
4. Intermediate code Generated
5. M/C independent Code ~~Code~~ Optimizer
6. Code generator
7. M/C dependent code optimizer

(1–4: grouped into one pass)

★ ⇒ Compiler construction tools: For lexical Analyser, we use lex tool, Syntax analyser - yacc tool. (slower compared to hard coding)

i) Parser Generator: Generates parse tree: yacc tool
ii) Scanner: Generators lexical analyser: lex tool
iii) Syntax directed translation engines: Used for intermediate code generators.
iv) Code generator tool.
v) Dataflow analysis engine: Used for optimization.
vi) Compiler construction toolkit: Contains tools for all phases

* Lexical Analysis

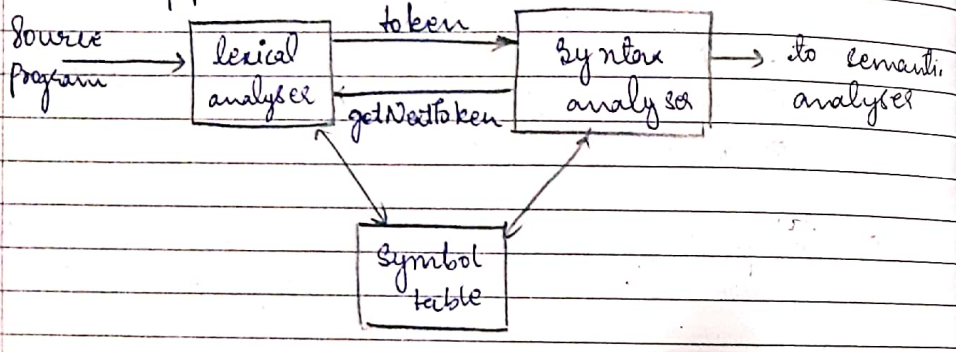
⇒ The role of lexical analyser:

i) – Removal of white spaces, tab or new line character.

ii) – Keeps track of the no of new line character encountered in the program.

iii) – Generates sequence of tokens.

iv) – Lexer & parser communication



⇒ Lexical Analysis vs. parsing.

i) Simplicity of design
ii) Compiler efficiency is improved.
iii) Compiler portability is enhanced.

} On seperation of lexical analysis & parser we achieve these 3 pts.

order of processing: ③ ① ②

⇒ Tokens, patterns & Lexemes

- A pattern is a description of the form that the lexemes of a token may take
- Lexeme is a sequence of characters in the source program that matches the pattern for a token & is identified by the lexical analyser as an instance of that token.

- A Token is a pair consisting of a token name and an optional attribute value

Q: Identify the tokens, patterns and lexemes for the sentence, printf(" Total = %d \n", total);

Lexemes ⇒ printf, total are identifiers.
⇒ ;, () → punctuations
⇒ " " → string literal

patterns ⇒ identifiers : [a-z][a-z 0-9 _]*
punctuations : [, ; ( )]

Tokens ⇒ printf: <id, entry 1 (or) to the symbol table for printf>
< (>
< literal >
< , >
< id, 2 >
<) >
< ; >

* Attributes for Tokens

Q: Identify tokens, patterns, lexemes for the statement
E = M * C ** 2
The above statement is in fortran language.

Soln:

Lexeme	Pattern / RE	Token
E	[a-z A-Z _] [_ a-z A-Z 0-9]*	< id, pointer to the symbol table entry for E >
=		< = >

M	[a-z A-Z] [a-z A-Z 0-9]*	<id, pointer to the symbol table entry for M>
*		<*>
C	[a-z A-Z][a-z A-Z 0-9]*	<id, pointer to the symbol table entry for C>
**		<ex_op><*>
2		<2> or <number, 2>

Q:
```
if (a == b)
{
    printf("Hello \n");
}
else
{
    printf("Bye \n");
}
```

- 23 lexemes
- if else printf → keywords
- == operators

Sln:-

Token	Informal Description (Pattern)	Sample Lexeme
if	i, f	if
	characters [i][f] or RE	
else	e, l, s, e	else
Comparison	<, >, >=, <=, !=, ==	<, >

one character as a token

id — letter followed by printf, a letters or digits

number — any numeric constant [0-9]+ 1, 8

literal — anything but surrounded by "" "hello"

* Lexical Error

```
fi(a == b)
{
}
```

* Error recovery strategies -

1. Panic mode: We delete successive characters from the remaining input until the remaining input matches with the pattern
2. We delete one character from the remaining input
3. Insert a missing character into the remaining input.
4. Replace a character by another character
5. Transpose to adjacent character

$S \rightarrow eSe \mid \varepsilon$ is a recursive grammar.

* Regular definitions are not recursive.

* Regular Definitions (Names given to regular expressions)

→ For notational convinience, we give names to certain regular expressions and use those names in subsequent expressions.

→ If summation is an alphabet then a regular definition is a sequence of definitions of the form

$d_1 \rightarrow r_1,\ d_2 \rightarrow r_2,\ d_3 \rightarrow r_3 \cdots d_n \rightarrow r_n$

→ where, i) each d_i is a new symbol, not in summation and not the same as any other d's.

ii) each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2 \cdots d_{i-1}\}$

Ex:1) Write the regular expression for an identifier.

→ Letter → [a-z A-Z] (r_1)
Digit → [0-9] (r_2)
Identifier → Letter (Letter | Digit)* (r_3)

2) Write regular definition for unsigned integer.

→ Digit → [0-9]
Digits → Digit+
Number → Digits (. Digits)? (E[+-]? digits)?

* Input Buffering (fig 115 2nd edition)

(to differentiate between two lexemes)

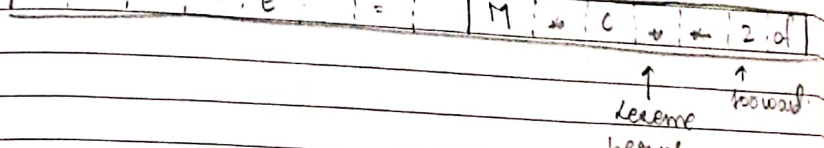
→ To process the large number of input characters during the compilation specialized buffering techniques have been developed to increase the speed of reading the input.

→ An important scheme involves 2 buffers that are alternatively reloaded as shown in the figure

→ Two pointers to the input maintained are.

i) Pointer lexeme begin marks the beginning of the current lexeme.

ii) Pointer forward scans ahead until a pattern is found.



* Recognition of Tokens

1 Grammar for branching stmt and conditional stmt.

stmt → if expr then stmt
| if expr then stmt else stmt
| ε

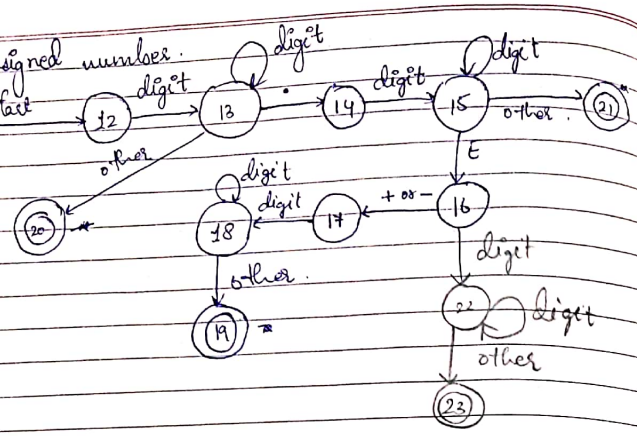
expr → term relop term | term
term → id | number

Soln:- The six tokens (or terminals) are if, then, else, relop (comparison), id, number
- Variables / Non-terminals stmt, expr, term

→ Patterns :- if → if
then → then
else → else
relop → > | < = | > | > = | = | < >
letter → [a-z]
digit → [0-9]
id → letter (letter | digit)*

Different types of tokens :- keyword, identifier, operators, punctuation

iv) for unsigned numbers:



* Architecture of a Transition diagram based lexical analyser.

```
TOKEN getRelop()
{
  TOKEN retToken = new(RELOP);
  while(1) {
    switch (state) {
    case 0: c = nextChar();
            if (c == '<') state = 1;
            else if (c == '=') state = 5;
            else if (c == '>') state = 6;
            else fail();
            break;
    Case 1 : c = nextChar();
             if (c == '=') state = 2;
             if (c == '>') state = 3;
             else fail();
             break;
```

```
case 2: retract();
        retToken.attribute = LE;
        return(retToken);

case 3: retract();
        retToken.attribute = NE;
        return(retToken);

case 4: retract();
        retToken.attribute = LT;
        return(retToken);
```