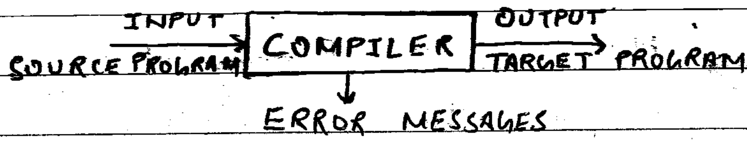
COMPILER DESIGN

Date ___/___/___
Page ________

UNIT - 1 INTRODUCTION TO COMPILING & LEXICAL ANALYSIS

① Introduction to compiler -

Compiler is a program which takes one language (source program) as input and translates it into an equivalent another language (target program)



② Major Data Structures in a compiler -

Tokens - It represent basic program entities such as identifiers, literals, reserved words, operators, delimiters etc.

Syntax Tree - It is generated by the parser. It is usually constructed as a standard pointer-based structure that is dynamically allocated as parsing proceeds.

Symbol Table - It keeps information associated with all kind of identifiers. Eg - constants, variables, functions, parameters, types, fields etc.

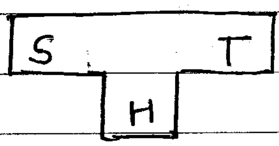
Literal Table - It stores constant and strings used in a program. Quick insertion and lookup are essential. Deletion is not necessary.

Temporary files - Used historically by old compilers due to memory constraints. Hold the data of various stages.

③ Bootstrapping and Porting -

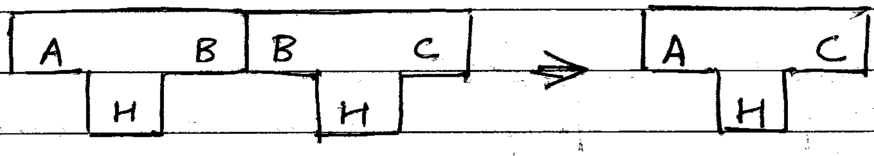
Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle far more complicated program. This complicated can further handle even more complicated program and so on.

T-diagram -

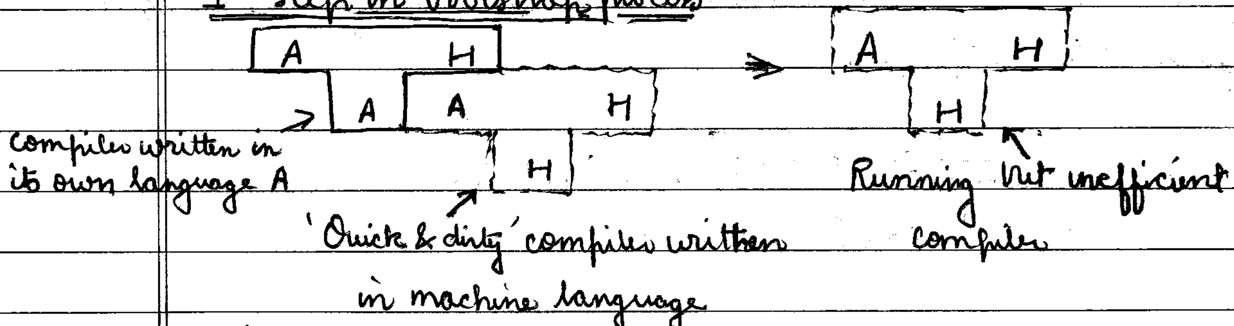


⇒ A compiler written in language H (for host language) that translates language S (for source language) into language T (for target language).

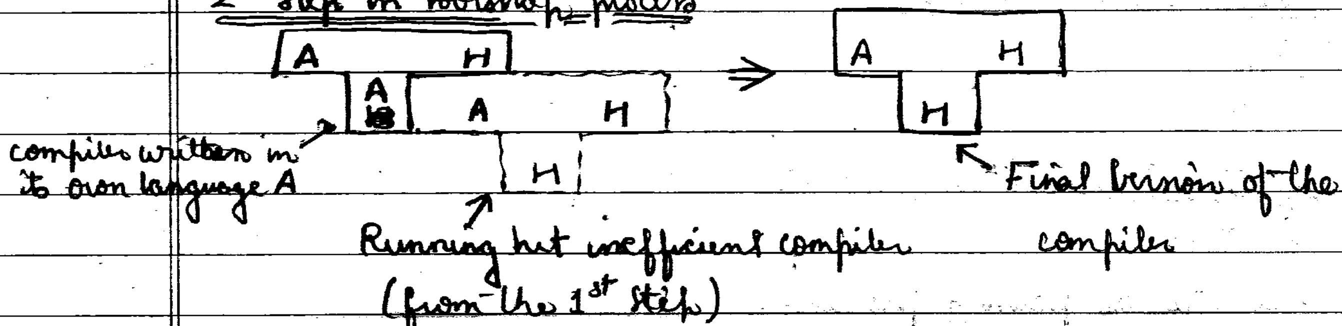
Eg –



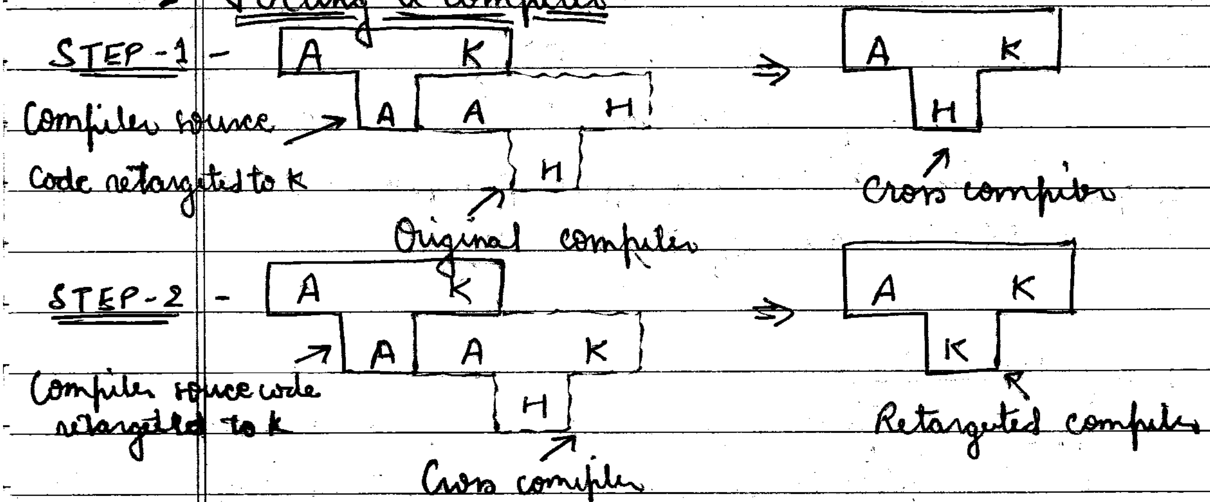
1st Step in bootstrap process –



2nd Step in bootstrap process –

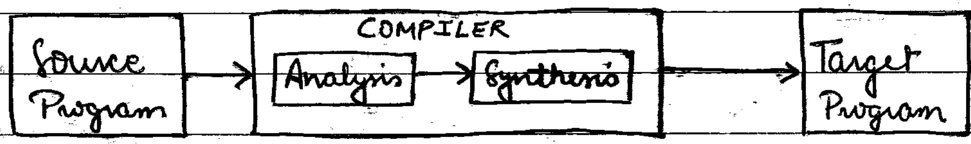


→ Porting a compiler –



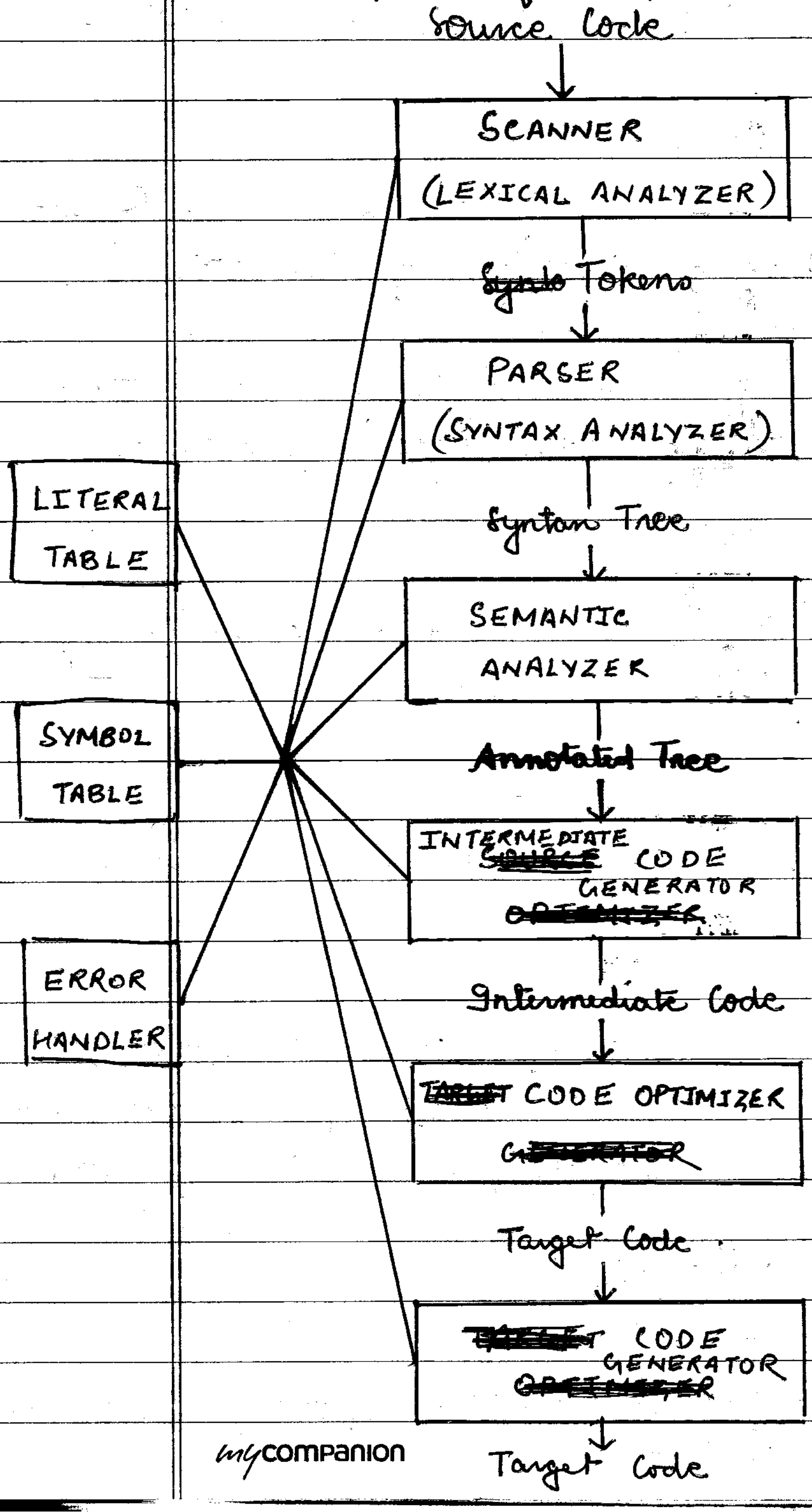
COMPILER STRUCTURE –

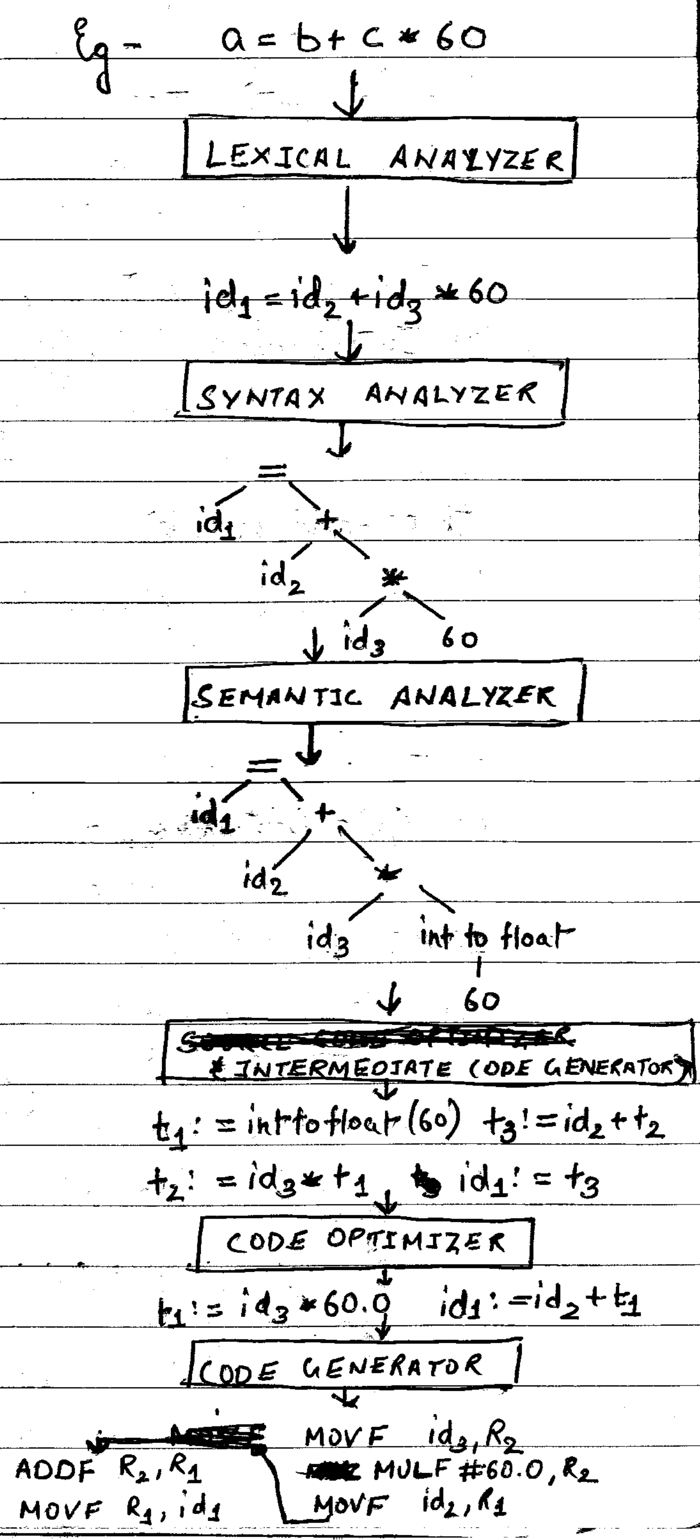
① Analysis-Synthesis model of compilation



Analysis of the source program contains three steps that is lexical analysis, syntax analysis and semantic analysis. ~~Intermediate code is generated from the input source program.~~
In synthesis part, three steps are involved that is intermediate code generation, code generation and code optimization.

② Various phases of a compiler -





(1) <u>Lexical Analysis (Scanning)</u> -
It is a phase of compilation in which the complete source code is scanned and source program is broken up into group of strings called <u>token</u>. A token is a sequence of characters having a collective meaning.

(2) <u>Syntax Analysis (Parsing)</u> -
In this phase, tokens generated by the lexical analyzer are grouped together to form a hierarchical structure & called parse tree or syntax tree.

(3) ~~Sem~~ <u>Semantic Analysis</u> -
It determines the meaning of the source string ~~eg~~ like matching of parenthesis, checking the scope of operation etc.

(4) <u>Intermediate Code Generation</u> -
It is a kind of code which is easy to generate and this code can be easily converted to target code. It can represented as three-address code.

(5) <u>Code optimization</u> -
It improves the intermediate code which reduces memory consumption and have a faster executing code which improve the runtime of the target program.

(6) <u>Code Generation</u> -
Target code is generated as assembly code or machine code

LEXICAL ANALYSIS -

① <u>Input Buffering</u> -
The lexical analyzer scans the input string from left to right one character at a time. It uses two pointers begin pointer (bptr) and forward pointer (fptr) to keep track of the portion of the input scanned.

bptr ↓

| i | n | t | | i | , | j | ; | i | = | i | + | 1 | ; | j | = | j | + | 1 | ; |

↑ fptr

<u>Initial Configuration</u>

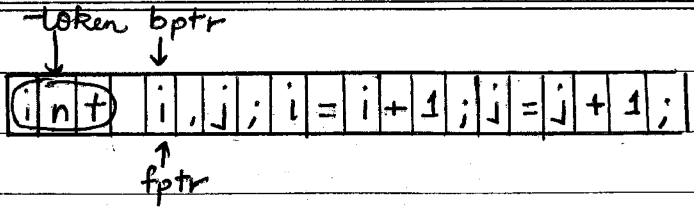
As soon as the blank space is encountered by fptr, it incidat
(or comma, or operators)

Date ___/___/___

Page ______

<u>Lexemes</u> - Sequence of characters in the source program that are matched with the pattern of the token.
Eg - int, i, num, ans, choice;

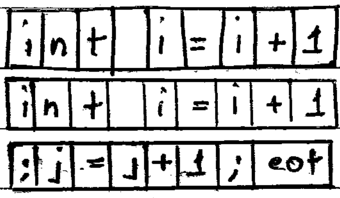
end of lexeme.



→ <u>Two methods are used</u> -

→ One buffer scheme →

→ Two buffer scheme →



buffer 1

buffer 2

If lexeme is short we ~~use on~~ can use one buffer scheme but if lexeme is long we have to use two buffer scheme in which the 1st buffer and 2nd buffer are scanned alternatively and when the end of current buffer is reached the other buffer is filled.

If the length of lexeme is longer than the length of the buffer then the input can not be scanned completely.

eof is introduced at the end of both the buffers is called <u>sentinel</u> which is used to identify the end of buffer.

→ <u>Code for input buffering</u> -

```
if (fptr == eof(buff1))  /* encounter end of first buffer */
  {
    /* Refill buffer2 */
    fptr++;
  }
else if (fptr == eof(buff2))  /* encounter end of second buffer */
  {
    /* Refill buffer 1 */
    fptr++;
  }
else if (fptr == eof(input))
    return;                   /* terminate scanning */
else
    fptr++  /* still remaining input has to be scanned */
```

(2) Specification of tokens –

To specify tokens regular expressions are used. When a pattern is matched by some regular expressions ~~then token can be recognized~~.

→ **Strings and language** –

String is a collection of finite number of alphabets or letters. The strings are synonymously called as words.

Collection of string is called the language.

|S| → length of a string ε → empty string ϕ → empty set of strings

→ **Operation on language** –

Union ($L1 \cup L2$), Concatenation ($L1.L2$), Kleen Closure (L^*), Positive Closure (L^+)

→ A language denoted by regular expressions is said to be a regular set or a regular language.

Eg – Regular Expression for identifier ⇒ letter (letter + digit)*

*** Unspecific information cannot be represented by regular expressions

(3) Recognition of Tokens –

Token Representation → | Token Type | Token value |

Token Type → category of token Token value → Information regarding token.

The lexical analyzer reads the input program and generates a symbol table for tokens.

Eg –

TOKEN	CODE	VALUE
if	1	–
else	2	–
while	3	–
for	4	–
identifier	5	Ptr to symbol table
constant	6	Ptr to symbol table
<	7	1
<=	7	2
>	7	3
>=	7	4

TOKEN	CODE	VALUE
!=	7	5
(	8	1
)	8	2
+	9	1
–	9	2
=	10	–

```
if (a<10)
    i=i+2;
else
    i=i-2;
```

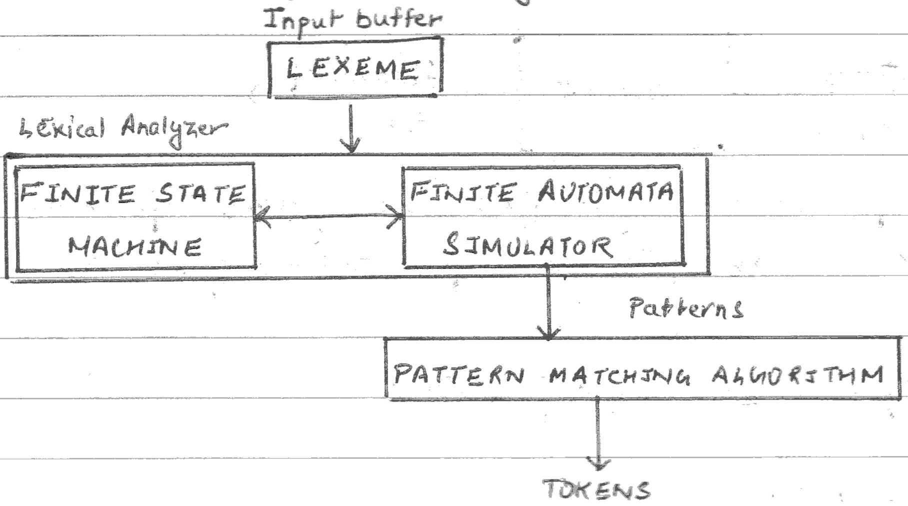
Our lexical analyzer will generate following token stream.

1, (8,1), (5,100), (7,1), (6,105), (8,2), (5,107), 10, (5,107), (9,1), (6,110), 2, (5,107), 10, (5,107), (9,2), (6,110).

Corresponding symbol table for identifiers and constants will be -

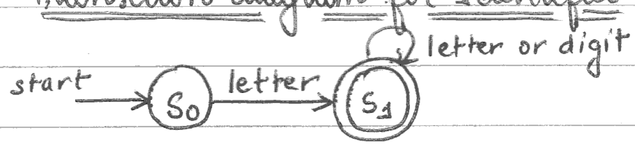
LOCATION COUNTER	TYPE	VALUE
100	IDENTIFIER	a
105	CONSTANT	10
107	IDENTIFIER	i
110	CONSTANT	2

(4) Block schematic of lexical analyzer -



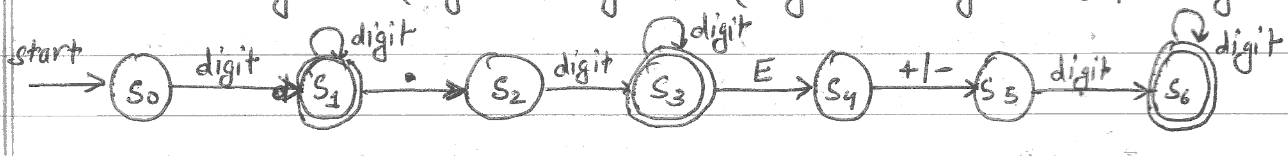
~~Transition~~

(5) Transition diagram for Identifier - R.E. = letter (letter + digit)*



Transition diagram for constant -

$R.E = digit^+ + (digit^+ (.) \; digit^+) + (digit^+ (.) \; digit^+ \; E(+/-) \; digit^+)$

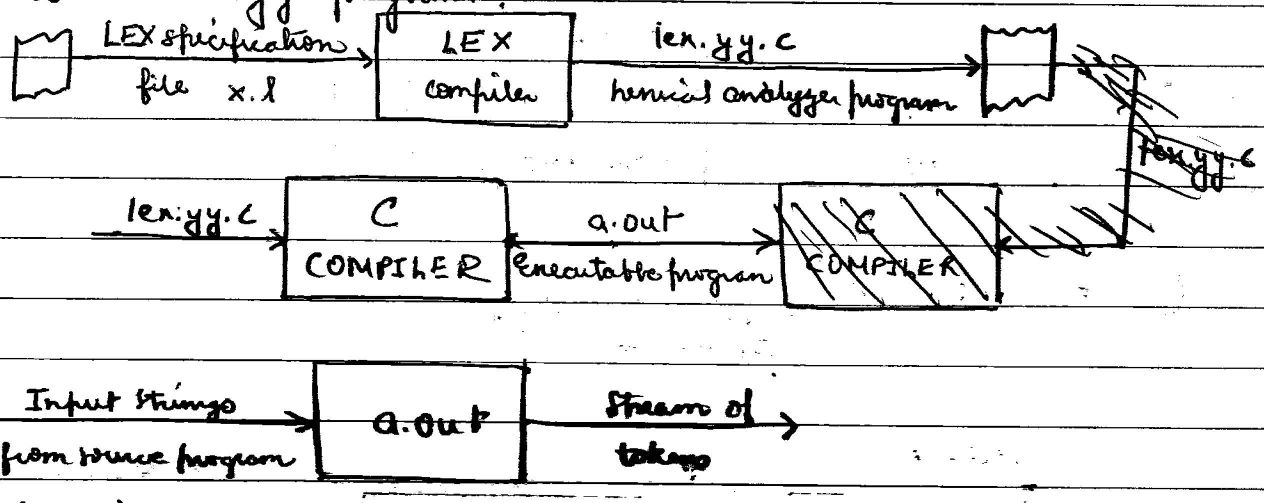


(6) <u>LEX</u> (Lexical Analyzer Generator) -

It is a UNIX utility. In LEX tool, designing the regular expressions for corresponding tokens is a logical task.

LEX is used to write specification file with .l (dot L) extension.

→ LEX converted .l to .c where .c is a C program which is actually a lexical analyzer program.



→ (x.l) Specification file stores the regular expressions for the tokens

→ lex.yy.c consists of tabular representation of the transition diagrams

→ LEX program consists of three parts -

```
%{
    DECLARATION SECTION
%}
%%
    RULE SECTION
%%
AUXILIARY PROCEDURE SECTION
```

- <u>Declaration section</u> → declaration of variable constant

<u>Rule section</u> → Consists of regular expression with associated action.

Eg - R_1 {action$_1$}

<u>Auxiliary procedure section</u> → Required procedures are defined

Eg :-

```
%{
%}
%%
"Rama" |
"Seeta" |
"Geeta" |
"Neeta"     printf ("\n Noun");
"Sings" |
"dances" |
"eats"      printf ("\n Verb");
%%
main ()
  {
    yylex(); ⟶ Routine, defined in lex.yy.c program.
  }           Scanner starts scanning the source program
int yywrap() → called when scanner encounters end of file.
  {
    return 1;
  }
```

Rule 1 (for "Rama", "Seeta", "Geeta", "Neeta")

Rule 2 (for "Sings", "dances", "eats")

Following commands are used to run the lex program x.l in UNIX.

```
$ lex x.l
$ cc lex.yy.c
$ ./a.out
```

After entering this commands a blank space for entering input gets available. There we can give some valid input.

Roma eats

Noun

verb

Then press either control + c or control + d to come out of the output

→ **LEX actions** –

(1) BEGIN – It indicates the start state.

(2) ECHO – It emits the input as it is.

(3) yytext – When lexer matches or recognizes the token from input token then the lexeme is stored in a null terminated string called yytext

(4) yyin – Standard input file that stores input source program

(5) yyleng – Stores the length of the input string.

(6) yylex()

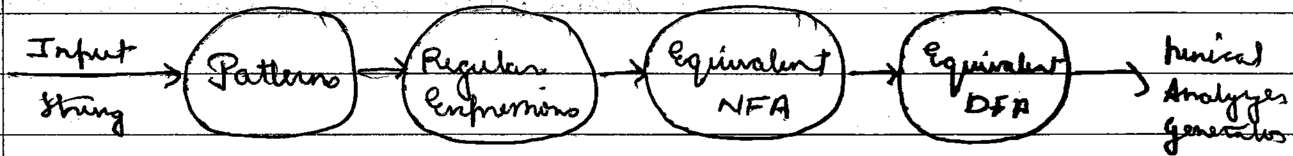
(7) yywrap()

→ **Design of Lexical Analyzer Generator** –

Two approaches –

- Pattern matching using NFA
- Using DFA for lexical analyzer.

→ **Pattern matching using NFA** –



(7) **~~Issues~~ Advantages of lexical analyzer** –

(1) Lexical analysis and syntax analysis are separated out which reduces the burden on parsing phase.

(2) Compiler efficiency get increased due to separation.

(8) **Issues of lexical analyzer** –

(1) Ambiguity of words (same word can have different meaning)

(2) Lookahead (identifying limits for sentence boundaries, Eg - Dr., Mr.)