Test for S_4 :- $[y_{in_1}, y_{in_2}] = S_4 \times w$

$$= [0\ 0\ 1\ 0] \begin{bmatrix} 1 & 0 \\ 2 & 0 \\ 0 & 2 \\ 0 & 1 \end{bmatrix} = [0 \quad 2]$$

apply the activation function

$\therefore [y_1 \quad y_2] = [0 \quad 1]$

Hence the prediction is ~~not~~ same as t_4.

So it recognise the S_4.

PLC - Lecture -1

o The art of- Language Design :-

→ Today there are thousands of the programming languages, and new ones continue to emerge.

→ Human being use Assembly language only for special purpose purpose applications.

→ Why there are so many Languages?

→ There are several possible answers to it →

① Evolution :- The late 1960's & early 1970's → saw a revolution in "structured programming" [extensively used → goto based control flow languages like Fortran, Cobol, & Basics.] → Gave a way (To use while loops & switch cases.)

In late 1980's the nested block structure of langage like Algol, Pascal, & Ada began to give way to the object oriented structures of C++, Eiffel, etc.

② Special purpose :- Lisp → used for manipulating symbolic data and complex data structures.

③ Personal Preference :-
- Prefrence of programming is a matter of taste.
- Some people love the terseness of C, Some hate it.
- So people uses recursion to solve problems So use iterations.

- Some people likes to work with pointer, whereas some people hate it and prefers Lisp & Smalltalk.

∴ The strength & variety of personal preference makes it impossible that anyone will ever develop a universally acceptable programming Language.

∴ ofcourse some programming L. are more successful than others. like. [Python is more successful than BASIC]

- Expressive Power :-

We can hear the arguments that one PL is more "powerful" than other,

but what is the meaning of powerful ?

∴ Powerfull means → Language features.

∴ Language features clearly have huge impact on the programmer's ability to write clear, concise, and maintainable code. especially for large systems.

Expressive power → abstraction facility

[use CNN & python as prog. example]

- Ease of use for the Novice :- o in Elementry school → Logo PL is used → even 5 yr old can understand it. (that is the simplicity of logo PL).

o c++ was used for object oriented prog. now Java has emerged to play the same role and Java is more like english L.

- **Ease of Implimentation:-**

∴ The PL should be easily implimentab on tiny machines with limited resources.

for Ex:- Basic prog. can run on all most any system with any configuration.

↓ However

If you want to run MatLab 2020Ra
↓ its mini requirment is 8 GB RAM
4 GB RAM

So, The Language can be such that, it can be implimented on any machine.

- **Standardization:-** of both the Language and a broad set of Libraries.

- **open source :-** Most programming Languages today have at least one open-source compiler or interpreter.

→ But some language — C in particular — are much more closly associated than others with freely distributed, peer-reviewed, community-supported computing.

Linux → written on 'C'.

- **Excellent Compilers :-**

- **Economics, Patronage, and Inertia :-**

PL/I → depends or owes its life to IBM.
Cobol / Ada → U.S. Department of defence.

Q/A Time

① What is the difference between ML & AL?

② In what way(s) are

- Von Neumann Architecture :- It is based on the stored-program computer concept, where instruction data and program data are stored in the same memory.

- Simula 67 :-
  → Simula is two simulation programming Language.
  Simula I & Simula 67. → 1960's
  ↓
  Norwegian Computing Center
  ↓
  Oslo

  Simula 67
  ↓
  introduced object, classes, inheritance, subclasses, etc.

Translator :- one Language program → Another Language Program.

→ Error Detection
→ Rule violation.

(i) Compiler
(ii) Interpreter
(iii) Assembler

:- Question / Answers :-

① What is the difference between machine and Assembly L ?

Ans :-

ML	AL
① ML is only understand by humans.	① AL is only understand by Humans.
② In ML data is represent using binary format - 1/0.	② In AL data is represented using mnemonics such as. MOV, Add, Sub, End, etc.
③ ML is very difficult to understand by Human beighs.	③ its easy to understand as compare to ML.
④ Modification & Error fixing is nearly impossible to can't be done ML.	④ Modification & Error fixing can be done in AL.
⑤ ML is very difficult to memorize, so its very difficult to learn ML.	⑤ Easy to memorize & learn as compare to ML.
⑥ Execution is fast in ML as all are in binary.	⑥ Execution is slow.
⑦ There is no need of translator for Execution.	⑦ Assembler is used for conversion of AL → ML.
⑧ ML is hardware dependent	⑧ AL is machine dependent & Not portable.

② In what ways the HLL an improvement on ALL? In what circumstances does it still make sense to program in assembler?

③ Why are there so many programming Languages?

④ What makes a programming L. successful?

⑤ What distinguishes declearative languages from imperative languages

⑥ What organization spearheaded the development of Ada?

⑦ What is generally considered the first high-level prog. L?
↳ Fortran → 1958/1960

⑧ What was the first functional L?
↳ LISP → Late 1950's for IBM 700/7000 series of scientific computers by John McCrathy. at MIT.

define circarea(r) (3.1415 * (r) * (r))

circlearea (Argument)

↑ replace

(3.1415 * (r) * (r))

r=5

3.1415 * 5 * 5

DATE
TIME

define PI 3.1415

```
#include <stdio.h>
#define PI 3.1415
#define circlearea(r) (PI * r * r)
int main()
{
    float radius, area;
    printf("Enter the radius: ");
    scanf("%f", &radius);
    area = circlearea(radius);
    printf("Area = %.2f", area);

    return 0;
}
```

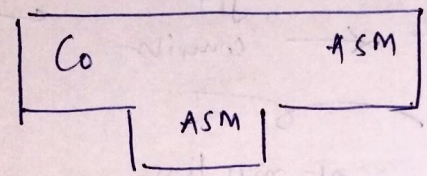
→ Bootstrapping is a process in which simple L. is used to translate more complex language. complicated prog. →

↓ that can be used to translate

↓ even more complicated programs

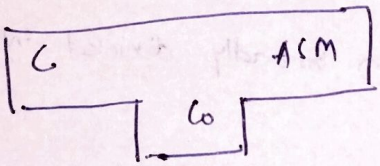
↓ and so on.

Step ① → First we write a compiler for small C in AL.



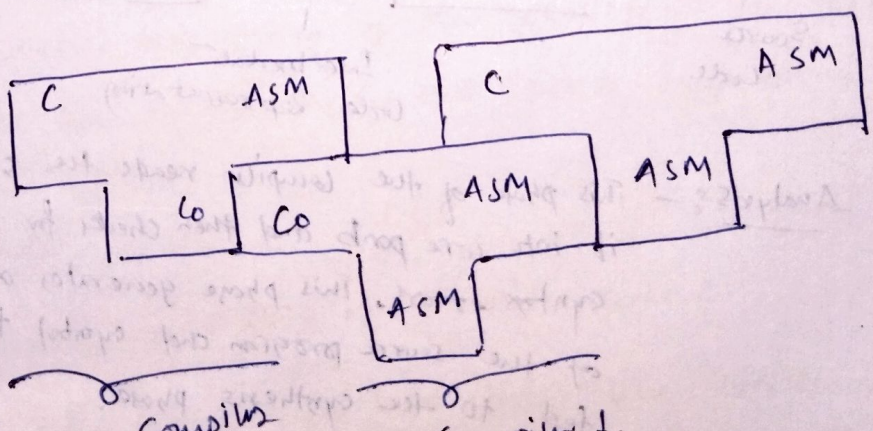
→ Compiler 1

Step ② → Then using with small subset of C i.e. Co, for the Source Language C the compiler is written.



→ Compiler 2.

Step ③ → Compile the second compiler using compiler 1, the compiler 2 is ~~compiled~~ compiled.

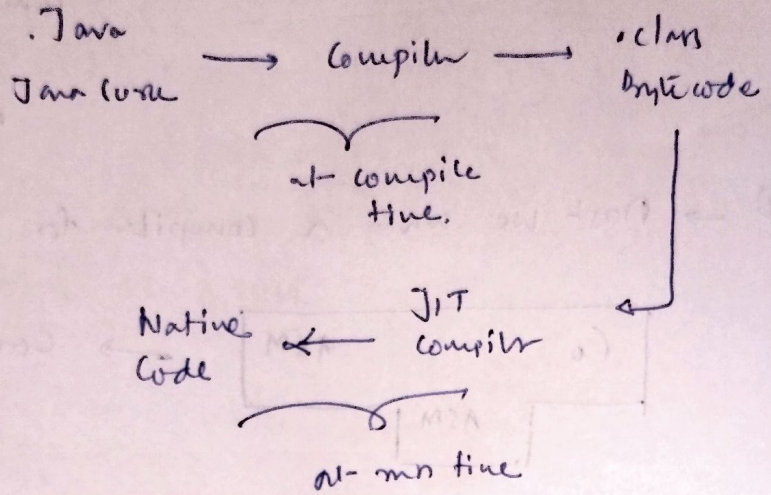


Step ④ → We get a compiler written in ASM, which compiles C and generates code in ASM.

~~JustinT~~

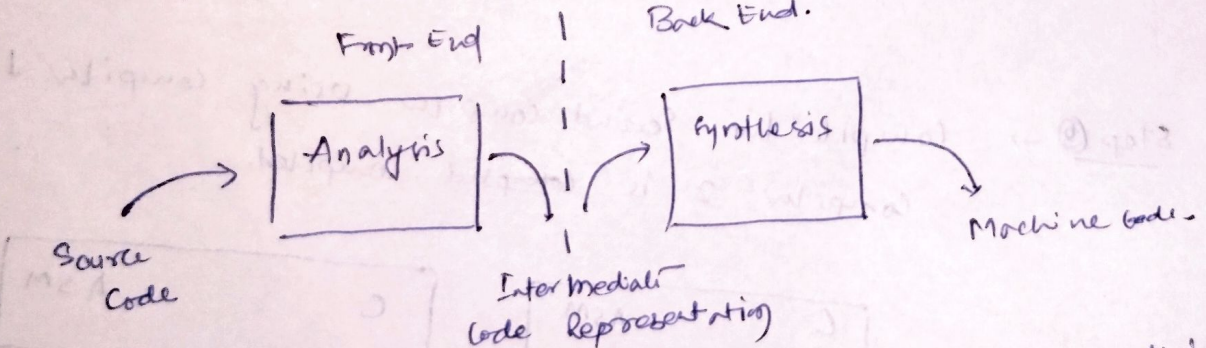
Just in Time Compiler

At compile Time



Structure of Compiler

→ A compiler can broadly divided into two phases, on the way they compile.



Analysis:- This phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammer, & syntax errors. This phase generates an intermediate representation of the source program and symbol table, which should be fed to the synthesis phase.

Synthesis:- It generates the target program with the help of intermediate code & symbol table.

7067
Tokens.

[Lexical unit, Lexical Element, or Tokens are the smallest unit of program i.e. commands, variables, constants, operators, and many more symbols called operators & punctuators.

Phases of Compiler

Compiler operates in various phases, each phase transforms the source program from one representation to another. Every phase takes input from its previous stage and feeds its output to the next phase of the compiler.

* There are six phases in compiler. Each phase help in converting the high-level language to the machine code. The phases of compiler are as follows:-

① Lexical Analysis :- (scanner)

→ its first phase of the compiler where compiler scans the code.

→ This scanning process can be left to right.

→ Character by character, and group these characters into tokens.

What happens :-

① there the character stream from the source program is grouped into meaningful sequences by identifying tokens.

② It makes the entry of the corresponding tokens into the symbol table and passes the token to next phase.

Primary functions :-

① Identify lexical units in a source code.

② classify lexical units into classes like. constants, reserve words, and enter them in different tables.

⑤ Ignore and remove the comments.

④ Identify the tokens ~~with~~ which are not part of the language.

Ex:- $X = Y + 10$.

Tokens	
X	Identifier 1.
=	Assignment operator
Y	Identifier 2
+	Addition operator
10	Number (constant)

② Syntax Analysis :- Syntax Analysis is all about discovering structure in code.

→ It determines whether or not a text follows the expected/standard format or not.

→ The main aim of this phase is to make sure that the source code was written by the programmer is correct or not.

- Syntax Analysis is based on the specific programming language by constructing parse tree with the help of tokens.

- Main tasks :-

① Obtain tokens from the lexical Analyzer.

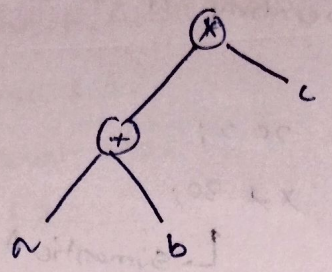
② Check if the expression is ~~syntati~~ syntactically correct or not.

③ Report all syntax errors.

④ Construct a hierarchical structure which is known as parse tree.

Ex:- (a+b)*c

(a+b)*c



In parse tree:-

① Interior nodes :-

② Leaf node.

③ Semantic Analysis :-

→ Semantic Analysis checks the semantic consistency of the code.

→ It uses the syntax tree of the previous phase along with the symbol table to verify that the source code is semantically consistant.

→ It also checks whether the code is conveying an appropriate meaning or not.

- It will check for → ① Type mismatch.
  ② incompatible operands.
  ③ a function called with improper arguments.
  ④ undeclared variables. etc.

- Main Tasks :-

① Helps you to store type information gathered and save it in symbol table or syntax tree.

② Allows you to perform type checking.

③ collects type information and checks for type compatibility.

④ checks if the source language permits the operand or not.

⑤ In the case of type mismatch, where there are no exact type corrections rules which satisfy the desired operation a semantic error is shown.

Ex:-

```
float x = 20.2;
float y = x * 30;
```

↳ Semantic Analyzer will type cast integer 30 to float 30.0.

⑭ Intermediate Code Generation:-

Once the semantic analysis phase is over, the compiler generates intermediate code for the target machine.

→ Intermediate code is between the high-level and machine level language.

→ This intermediate code needs to be generated in such a manner that makes it easy to translate it into the target machine code.

Main Tasks:-

① It should be generated from the semantic representation of the ~~track~~ source program.

② Holds the value computed during the process of translation.

③ Helps you to translate the intermediate code into target machine code.

④ Allows you to maintain precedence ordering of the source language.

⑤ It holds the correct number of operands of the instruction.

Ex:- total = count + rate * 5;

Intermediate code with the help of address code method.

$t_1 := int\_to\_float(5);$

$t_2 := rate * 5;$

$t_3 := count + t_2;$

$total = t_3;$

④ Code optimization :-

→ This phase removes unnecessary code lines and arrange the sequence of statements to speed up the execution of program without wasting resources.

→ The main goal of this phase is to improve on the intermediate code to generate a code that runs faster and occupies less space.

Primary functions :-

① It helps you to establish a trade-off between execution and compilation speed.

② Improves the running time of the program.

③ Generates fast running code with same representation.

④ Remove unreachable code and unused variables.

⑤ Remove the statement which are not altered from a loop.

Ex:-

∴

```
a = int_to_float(10);          b = c*10.0;
b = c * a;          ⇒          f = e + b;
d = e * b;
f = d;
```

⑥ Code Generation :- its a final phase of compiler.

→ its get input from the code optimization phase and produces the page code or object code as a result.

→ The objective of this phase is to allocate storage and generate relocatable machine code.

→ It also allocates memory locations for the variable.

→ The instructions in the intermediate code are converted into machine instructions.

→ This pase converts the optimized or intermediate code to the target language.

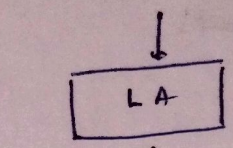
→ In this phase, all the memory locations and registers are also selected and allotted during this phase.

Ex :- a = b * 60.0

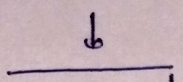
would be possibly translated to registers.

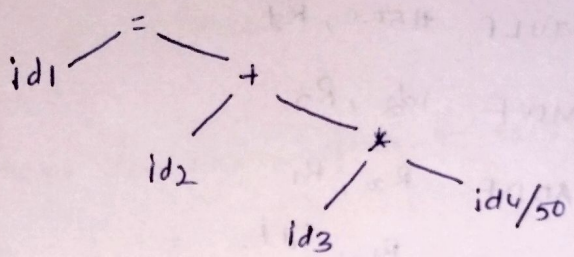
```
MOVF b, R1
MULF #60.0, R1
MOVF R1, a.
```

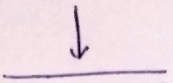
Sum := old sum + Rate * 50

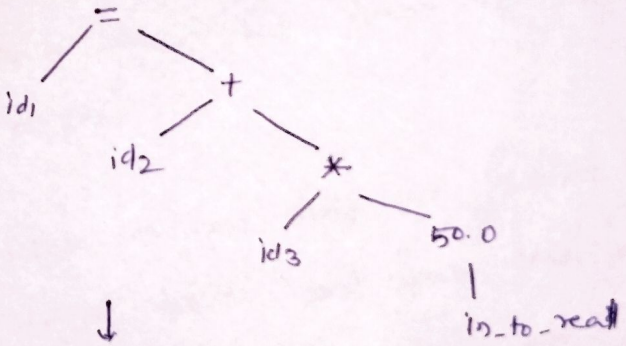


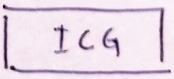
id1 := id2 + id3 * id4 / 50









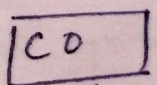


temp1 := into_real (50)

temp2 := id3 * temp1

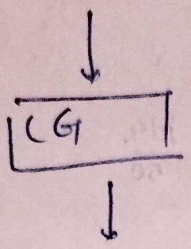
temp3 := id2 + temp2

id1 := temp3



temp 1 := id3 * 50.0

id 1 := id2 + temp1




```
MOVF   id3, R1
MULF   #50.0, R1
MOVF   id2, R2
ADDF   R2, R1
MOVF   R1, id1.
```

Lexical Analysis

→ It very first- phase in CD.

→ A Lexical analyzer (Lexer) takes the modified source code which is written in the form of- sentences.

o In other words →

"Lexer converts a sequence of- characters into a sequence of tokens."

o It also removes the extra space & comments written in the source code.

→ So, Programs that- performs lexical analysis are called "Lexical analyzer or Lexer."

→ ∴ If Lexer detects that the token is invalid, it generates an error.

→ It reads character streams from the source code, checks for the legal tokens, and pass the data to the syntax analyzer when it demands.

→ Example:- "How Pleasent- Is The weather?"

∴ Here we can easily recognize that there are five words.

→ Humans can recognize the separators, blanks, and the punctuation symbols.

Let try again →

How presents the else other?

→ now is bit difficult to read but still we can manage to read it.

→ Lexer will check for all the valid tokens.

○ Basic Terminologies :-

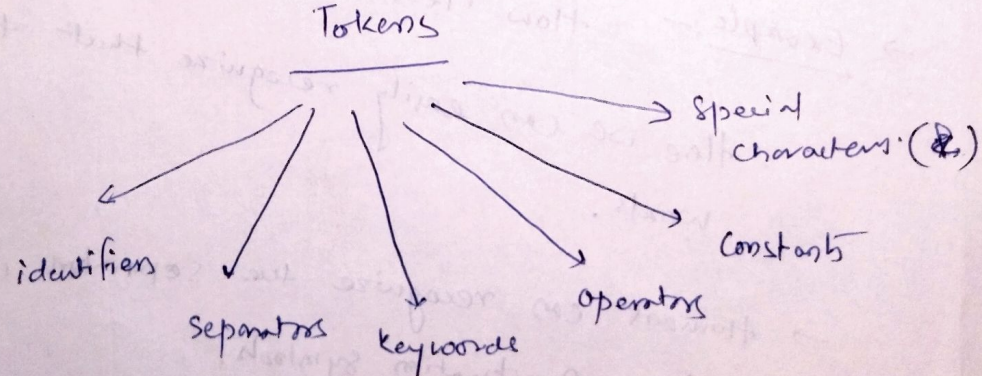
① **Lexeme** :- A lexeme is a sequence of characters that are included in the source program according to the matching pattern of a token.

- It nothing but an instance of token.

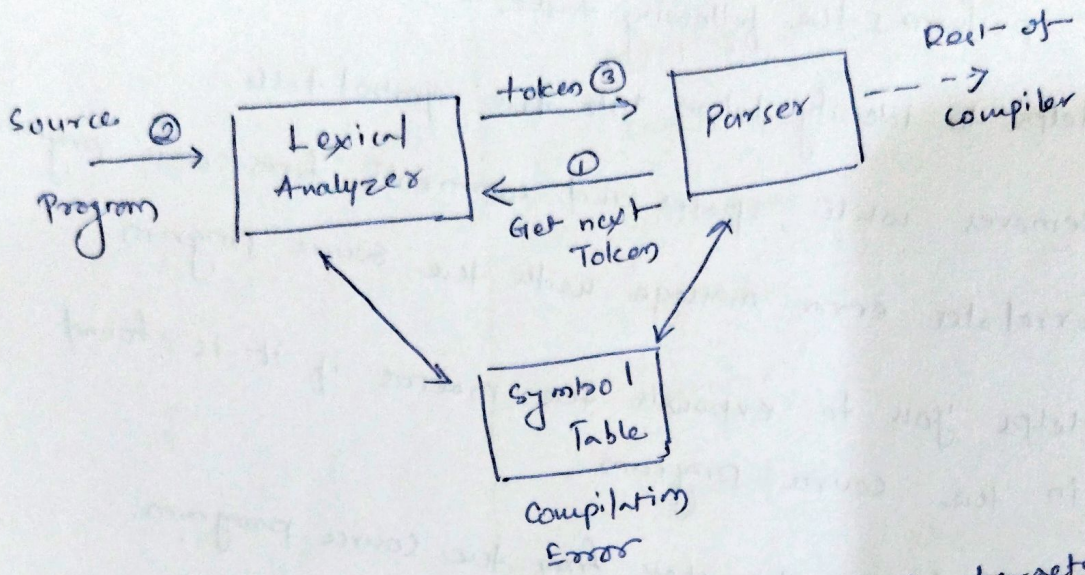
② **Token** :- The token is a sequence of characters which represents a unit of information in the source program.

③ **Pattern** :- A pattern is a description which is used by the token.

→ In case of keyword which uses as a token, the pattern is a sequence of characters.



Lexical Analyzer Architecture :-



→ The main task of lexer is to read input characters in the code and produce tokens.

→ Lexer scans the entire source code of the program.

→ It identifies each token one by one.

→ scanners are usually implemented to produce tokens only when requested by a parser.

Working :-

1. "Get the next token" command which is sent from the parser to the lexical analyzer.
2. On receiving this command, the lexer scans the input until it finds the next token.
3. It return the token to the parser.

→ Lexer will skips whitespaces and comments while creating tokens.

→ If any error is present, then lexer will correlate that error with the source file and line number.

o Roles of- Lexical analyzer :-

it performs the following tasks:-

① Helps to identify token into the symbol table.

② Removes white spaces and comments from source program.

③ Correlates error message with the source program.

④ Helps you to expands the macros if it is found in the source program.

⑤ Read input- character from the source program.

Example :-

```
# include <stdio.h>
  int maximum (int x, int y)
  {
   // This will compare 2 numbers.
    if (x > y)
      return x;
    else {
      return y;
    }
  }
```

Ans:-

Lexeme	Token
int	keyword
maximum	identifier
(	operator

int	keyword
x	Identifer
,	operator
int	keyword
y	Identifier
)	Operator
{	operator
if	keyword
(	operator
x	Identifier
>	operator
y	Identifier

Nontokens :—

Type	Examples
comment	// This will compare 2 number.
Pre-processor directive	# include <stdio.h>
Pre-processor directive	# define NUMS 8,9
Macro	NUMS
Whitespace	/n, /b, /t

⊙ Lexical Errors :-

→ A character sequence which is not possible to scan into any valid token is a lexical error.

- Lexical errors are not very common but it should be managed by a scanner.
- Misspelling of identifies, operators, keywords are considered as lexical errors.
- Generally, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Error Messages → exceeding length
→ Unmatched string
→ illegal characters
→ Misspelling

⊙ Error Recovery in Lexer :-

So few most common error recovery techniques :-

① Removes one character from the remaining input.

② In the panic mode, the successive characters are always ignored until we reach a well-formed token.

③ By inserting the missing character into the remaining input.

④ Replace a character with another character.

Ex:- `printf("i = %d, & i = %x", i, &i);`

① Lexical Analyzer & Parser :-

LA	Parser
① Scans Input program.	① Performs syntax analysis
② Identify Tokens.	② Create an abstract representation of the code.
③ Insert tokens into symbol Table.	③ Update symbol Table entries.
④ It generate Lexical errors.	④ It generates syntax errors.

② Why separate Lexical & Parser :-

① The simplicity of the design : It eases the process of lexical analysis and the syntax analysis by eliminating unwanted tokens.

② To improve compiler efficiency : Helps you to improve compiler efficiency.

③ Specialization : specialized techniques can be applied to improve the lexical analysis process.

④ Portability : only the scanner requires to communicate with the outside world.

⑤ Higher Portability :-

• Adv. of Lexer :-

① Lexer can be used by the programs like compilers which helps to generate compiled binary executable code.

② It is used by the web browsers to format and display a web page with the help of parsed data from Javascript, HTML, CSS.

③ A separate lexical analyzer helps you to construct a specialized and potentially more efficient processor for the task.

• Dis. Adv. of Lexer :-

① Time consuming :- You need significant time to read source code & form tokens.

② Some regular expressions are quite difficult to understand.

③ More effor is needed to develop and debug the lexer and its token descriptions.

④ Additional runtime overhead is required to generate the lexer tables and construct the tokens.

Ex:- Find the lexemes and tokens from the following 'C' Code ?

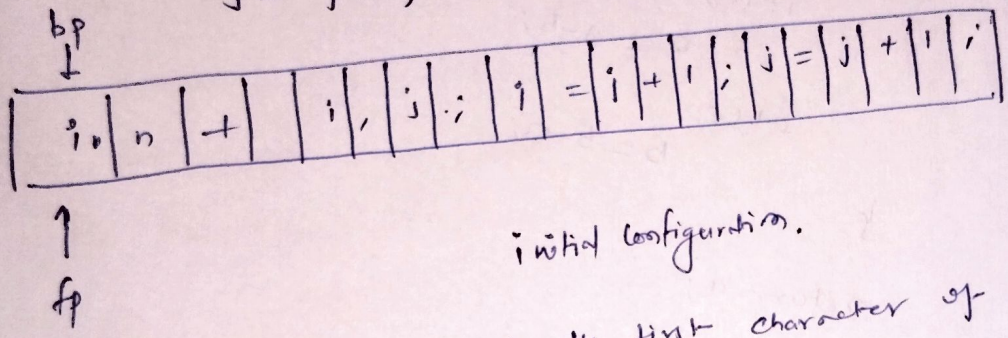
```
int get (int a, int b)
{
    while(a != b)
    {
        if(a > b)
            a = a-b;
        else
            b = b-a;
    }
    return a;
}
```

Input Buffering in Compiler Design

→ The lexer scans the input from L to R & char by char. at a time.

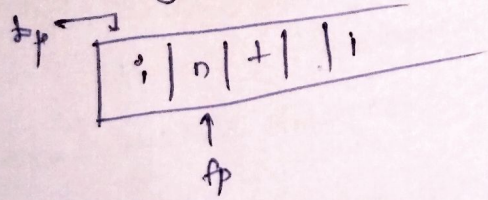
→ It uses two pointers begin ptr (bp) and forward ptr (fp) to keep track of the pointer of input scanned.

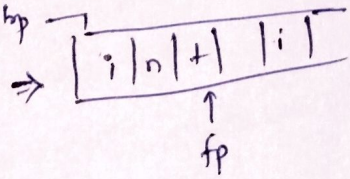
```
∴ int i, j;
  i = j+1;
  j = j+1;
```



initial configuration.

→ initially both pointers point to the first character of the input string as shown below.



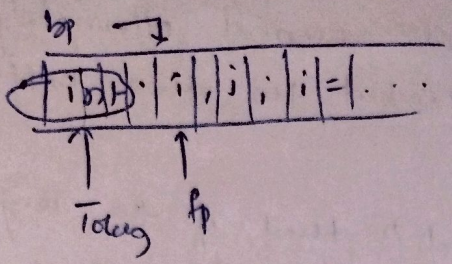


→ The fp will moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicate the end of lexeme.

↳ space occurs .ie. → (int) lexeme is identified.

→ The fp will moved ahead at white space, when fp encounters white space, it ignore and move ahead.

→ Then both pointer [bp+fp] one set- at next token.



→ The input character is read from secondary storage. but reading in this way from secondary storage is costly.

→ ∴ Buffering technique is required / used.

→ A block of data is first read into a buffer, and then scanned by lexical analyzer.

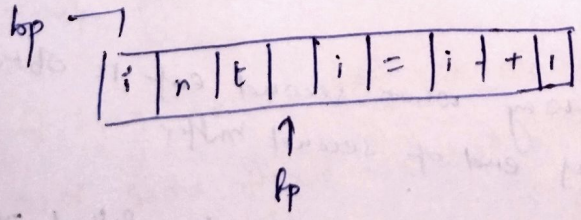
→ There are two methods used in this context:

1) One Buffer Scheme
2) Two Buffer Scheme.

① 1 Buffer Scheme:- In this scheme, only one buffer is used to store the input string.

→ Problem with this scheme → Lexem Lengths.

o/o if lexeme is very long then it crosses the buffer boundary, ∴ to scan rest of the lexeme the buffer has to be refilled, that makes overwriting of first lexeme.



② 2 Buffer Scheme:-

→ To overcome the problem of one buffer scheme, two buffer scheme is used to store the input string.

→ The first buffer and second buffer are scanned alternately.

→ When end of current buffer is reached the other buffer is filled.

→ The only problem with this method is that if the length of the lexeme is longer than the length of buffer then scanning input cannot be scanned completely.

→ Initially both bp and fp will point to the first char of the first buffer.

→ Then fp moves forward to search for the end of lexeme

→ As soon as the blank character is recognized, the string between bp and fp is identified as Token.

→ To identify the ~~boundary~~ boundary of first buffer, end of buffer char is placed at the end of first buffer.

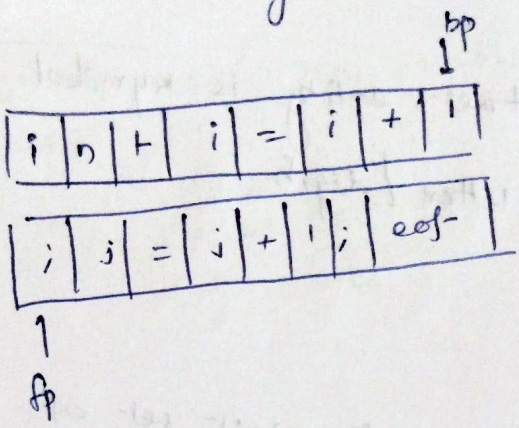
→ Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer.

- When fp encounters first eof, then one can recognize end of first buffer and hence filling up second buffer is started.

→ In the same way when second eof is obtained then it indicates of end of second buffer.

- Alternatively both the buffers can be filled up until end of input program and stream of token is identified.

→ This eof character introduced at the end is called "Sentinel". which is used to identify the end of buffer.



FSA

Specifications of Tokens

There are three specifications of Tokens:

1) Strings
2) Language
3) Regular Expressions.

Continued

* Strings and Languages:—

→ An alphabet or character class is a finite set of symbols.

→ A string over an alphabet is a finite sequence of symbols drawn from that alphabet.

→ A Language is any countable set of strings over some fixed alphabet.

∴ The length of string is generally written as $|s|$.

Strings and Language:-

→ Symbol :- An abstract entity is symbol.

Ex:- letter / digits

~~→ String over A finite~~

→ Char Class / Alphabet Class :- Its finite set of symbols.
{0,1} {A,B,C}

→ String :- A string over an alphabet is a finite set of symbols drawn from that alphabet.

Ex:- ① abcb
cabc → are string over a, b, & c.

② |w| is the length of string

③ ε is an empty string of length 0.

→ Language :- is any countable set of strings over some fixed alphabet.

① ϕ & {ε} are language.

② The set palindromes over {0,1} is infinite L.
↳ The set of palindromes of 5 char one {0,1} is a f.L.

③ The set of strings {01, 10, 11, 00} over {0,1} is f.L.

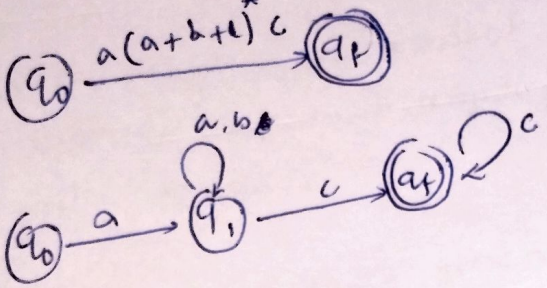
∴ If Σ is an alphabet
↳
Σ^* is the set of all strings over Σ.

→ Each subset of Σ^* is a language.

→ RE (T3) or RL, CFG (T2 or CFL), CSL (T1 or CSL), or To L. or REL.

→ RE ⊂ CFL ⊂ CSL ⊂ REL (To)

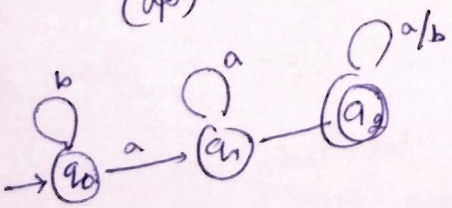
→ $a(a+b+c)^*c$



Ex → string ending with

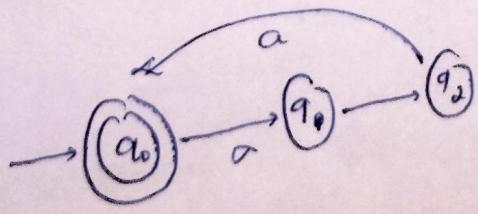
Ex:- all string with sub-string 'ab'

$(a/b)^*ab(a/b)^*$

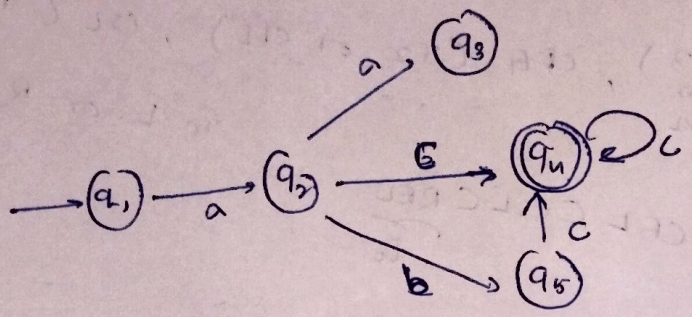


Ex:- string in which no of a is divisible by 3. $\Sigma = \{a\}$

$L = \{ a^{3n} \mid n \geq 0 \}$



→ $a(a+b+c)^* c$



—: Specification of Tokens :-

- ∴ Regular definitions, a mechanism based on regular Expressions are very popular for specification of Tokens.
  1. It has been implimented in the lexical analyzer generator tool → LEX
  2. We study regular expression first, and then specification of Tokens using LEX.

- Transition diagrams, a variant of finite automata, are used to implement regular definitions and to recognize tokens.
  1. LEX automatically generate FSA from regular definations.
  2. So we will study FSA and their generation from regular expressions in order to understand transition diagrams in LEX.

But first we need to understand → The following terms.

1. Symbols → Its an abstract entity, not defined.
   Ex:— digits, letters, special symbols.

2. char class | Alphabet class → A finite set of symbols.
   Ex:- {0,1}, Σ = {A, B, C}, etc.

3. String :- String over an set of alphabet Σ is the finite sequence of symbol drawn from Σ.
   Ex. 101010...
   Σ = {0,1} with alternate 1's and 0's

3. Prefix of string :- String obtained by removing zero or more symbols from end of string.

Ex:- ban, bana, banana, ∈, are prefix of "banana".

4. Suffix of string :- String obtained by removing zero or more symbols from beginning of string.

Ex:- nana, na, banana, ∈, are suffix of "banana".

5. Substring :- String obtained by deleting any prefix and any suffix from string.

Ex:- banana, na, ana, ∈ are substring of "banana"

6. Proper Prefix :- Prefix which is not equal to ∈ or string it self.

Ex:- ban, banan

7. Proper Suffix :- Suffix which is not equal to ∈ or string it self.

Ex:- na, anana, etc.

8. Sub-sequence :- String formed by deleting zero or more not necessarily consicutive positions of string.

Ex:- baan, anaa, . . .

Operations on Language :-

operations	definition & notation
① Union of L & M.	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$ $= \{s \mid s \in L \text{ or } s \in M\}$
② Concatination of L & M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$ $= \{st \mid s \in L \text{ and } t \in M\}$
③ Kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$
④ Positive closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Regular Definition :- (used for Token specifications).

→ for some alphabet set Σ, sequence of regular definitions :

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\vdots$$
$$d_n \rightarrow r_n \quad \text{where,}$$

(1) each d_i is new symbol (not in Σ or other d_i)

(2) r_i is regual expression over $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Regular Expression :-

① The Language accepted by finite automata can be easily described by simple expressions are called Regular Expressions.

→ It is most effective way to represent any language.

② " A regular expression is an algebraic formula whose value is a pattern consisting of a set of strings, called the language of expression."

③ Regular Expressions can also be described as a sequence of pattern that defines a string.

∴ For Instance →

→ In RE, x^* means zero or more occurrence of x.
{ i.e. ε, x, xx, xxx, xxxx, ... }

→ In RE, x^+ means one or more occurrence of x
{ i.e. {x, xx, xxx, xxxx ... }

* R.E. can be recursively defined as follows :-

- ε is RE indicates the language containing an empty string $L(\varepsilon) = \{\varepsilon\}$
- φ is a RE denoting an empty language $L(\phi) = \{\}$.
- X is a RE where $L = \{x\}$
- If X is RE denoting the Language L(X) and Y is a RE denoting the language L(Y), then
  (1) X+Y is a R.E. corresponding to the language $L(X) \cup L(Y)$ ∴ $L(X+Y) = L(X) \cup L(Y)$

(ii) $X.Y$ is a R.E. corresponding to the language $L(X).L(Y)$

$\therefore L(X.Y) = L(X).L(Y)$

(iii) r^* is an R.E. then $L(r^*) = R^* = \bigcup_{i=0}^{\infty} R^i$

$\therefore R = L(r)$

$\therefore R^* = (L(r))^*$

$\therefore$ If we apply any of the above rule, they are RE.

o Some RE examples :-

RE	Regular set-
① $(0+10^*)^*$	$L = \{0, 1, 10, 100, 1000, 10000, \ldots\}$
② (0^*10^*)	$L = \{1, 01, 10, 010, 001, 100, 0010, 00100, \ldots\}$
③ $(0+\varepsilon)(1+\varepsilon)$	$L = \{\varepsilon, 0, 1, 01\}$
④ $(a+b)^*$	$L \Rightarrow$ set of strings of a's and b's of any length including null string. $L = \{a, b, ab, aa, bb, \ldots\}$
⑤ $(a+b)^*ab$	$L \Rightarrow$ set of strings of a's and b's ending with ab. $L = \{ab, aab, bab, abab, \ldots\}$
⑥ $(11)^*$	$L \Rightarrow$ set consisting even number of 1's. $L = \{11, 1111, 111111, \ldots\}$
⑦ $(aba)^*(bb)^*b$	$L \Rightarrow$ set of string consisting even no of a's followed by odd no's of b's $L = \{b, aab, aabbb, \ldots\}$

⑧. $(aa + ab + ba + bb)^*$

$L \Rightarrow$ string of a's and b's with even length obtained by concatinating any combination of 'aa, bb, ab, ba.

$L = \{aa, ab, bb, ba, aabb, abba, \ldots\}$

○ Regular Set → Any set that represents the value of the Regular Expression is called Regular Set.

∴ Properties of Regular Set →

① The union of two regular set is regular.

② The intersection of two regular set is regular.

Proof →

$RE_1 = a(aa)^*$

$RE_2 = (aa)^*$

$\therefore L_1 = \{a, aaa, aaaaa, \ldots\}$ → odd length of a

$L_2 = \{\epsilon, aa, aaaa, \ldots\}$ → even length of a

$\therefore L_1 \cup L_2 = \{\epsilon, a, aa, aaa, \ldots\}$ → string of all possible length of a

$RE(L_1 \cup L_2) = a^*$

Proof → $RE_1 = a^+$ $RE_2 = (aa)^*$

$\therefore L_1 = \{a, aa, aaa, \ldots\}$ $L_2 = \{\epsilon, aa, aaaa, \ldots\}$

$L_1 \cap L_2 = \{aa, aaaa, aaaaaa, \ldots\}$

$RE(L_1 \cap L_2) = aa(aa)^*$

③ The complement of regular set is regular.

$\therefore RE = (aa)^*$

$L = \{\epsilon, aa, aaaa, \ldots\}$

$\therefore L' = \{a, aaa, aaaaa, \ldots\}$

$RE(L') = a(aa)^*$

④ The difference of two regular set is regular.

$\therefore RE_1 = a(a^*)$

$RE_2 = (aa)^*$

$\therefore L_1 = \{a, aa, aaa, aaaa, \ldots\}$

$L_2 = \{\epsilon, aa, aaaa, \ldots\}$

$L_1 - L_2 = \{a, aaa, aaaaa, \ldots\}$

$RE(L_1 - L_2) = a(aa)^*$

⑤ The reversal of regular set is regular.

$\therefore L = \{10, 10, 00, 11\}$

$RE(L) = 01 + 10 + 00 + 11$

$L^R = \{10, 01, 00, 11\}$

$RE(L^R) = 10 + 01 + 00 + 11$ i.e. RE.

∴ Q.A. →

① Write a RE for a Language accepting all combinations of a's, over $\Sigma = \{a\}$.

Ans. → RE = a^*

② Except null string

Ans:- RE = $a(a)^* = a^+$

③ Any no of a's and b's

Ans:- $(a+b)^*$

④ RE for the language accepting all string which are starting with 1 and ending with 0. over

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

Ans:- RE = $1(0+1)^*0$

⑤ RE for the language string and ending a and having any combinations of b's in between over

$\Sigma = \{a,b\}$

Ans:- RE = ab^*a.

⑥ RE for → starting with a but not having consicutive b's

Ans:- $(a+ab)^*$

(7) Write a RE for the language having a string which should have at least one 0 and at least one 1.

Ans:- $RE = [(0+1)^* 0 (0+1)^* | (0+1)^*] + [(0+0)^* | (0+1)^* 0 (0+1)^*]$

(8) $\Sigma = \{0, 1\}$ " RE for string do not contain substring 01".

$L = \{\epsilon, 0, 1, 100, 1000, 1100, 1110, \ldots\}$

Ans:- $RE = (1^* 0^*) = (1^* 0^*)$.

(9) RE for L containing the string in which every 0 is immediately followed by 11.

Ans:- $RE = (011 + 1)^*$

---

* Regular defination :-

A regular defination is a sequence of definitions in the form:-

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$d_3 \rightarrow r_3$$
$$\vdots$$
$$d_n \rightarrow r_n$$

where:-

(i) each d_i is a distinct symbol

(ii) each r_i is a regular expression over the symbols $\Sigma \cup \{d_1, d_2, d_3, \ldots, d_{i-1}\}$.

Ex:~① C identifiers are strings of letters, digits, and underscores, Here is a regular definition of the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

letter_ → A|B|...|Z|a|b|...|z|_
digit → 0|1|...|9
id → letter_(letter_|digit)*

Ex:~② Unsigned numbers (integer or floating point) are string such as 5280, 0.01234, 63.36E4, or 1.89E-4. The regular definition is.

digit → 0|1|...|9
digits → digit(digit)*
optional fraction → .digits|ϵ
optional Exponent → (E(+|-|ϵ)digits)|ϵ
number → digits.optional fraction.optional Exponent.

✱ Extensions of Regular Expressions →

① One or More Instances → The unary, postfix '+' operator represents the positive closure of a regular expression and its language. That is if r is a RE, then $(r)^+$ denotes the language $(L(r))^+$.

(11) Zero or one instance → The unary postfix operator '?' means 'zero or one occurrence'. That is r? is equivalent to r|ε or we can say $L(r?) = L(r) | \{\varepsilon\}$?

(12) Character class → A regular expression $a_1|a_2|\cdots|a_n$ where a_i are each symbols of the alphabets can be replaced by → $[a_1, a_2, \cdots, a_n]$. More importantly when $a_1, a_2, \cdots, a_n$ form a logical sequence e.g. (Consicutive upper letters, lowercase letters, digits) we can replace them by $a_1 - a_n$

Thus. $|a|b|c|\cdots|z \rightarrow [a,b,c,\cdots,z] \rightarrow [a-z]$

Lets revisit Ex ① & ② again →

Ex:-① →

letter_ → [A-Z a-z _]
digit → [0-9]
id → letter_(letter|digit)*

Ex:-② →

digit → [0-9]
digits → digit+
number → digits(.digits)? (E[+-]? digits)?

o Rego Recognition of Tokens. :-

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

expr → term relop term
     | term

term → id
     | number.
```

↳ "A grammer for branching statements"

↳ Patter of Tokens (Regular Definations)

number

```
digit → [0 - 9]
digits → digit+
number → digits (.digits)? (E[+|-]? digits)?
```

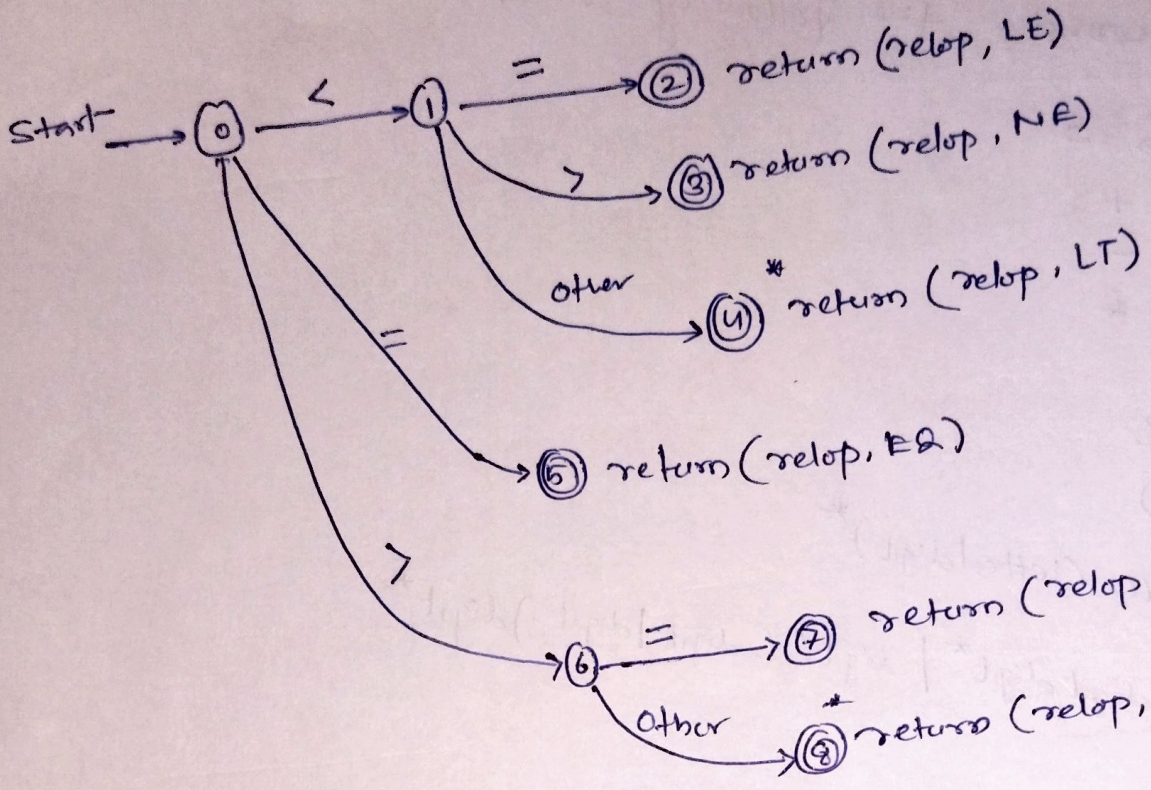
id

```
letter_ → [A - Z a - z _]
id → letter_ (letter_ | digits)*
```

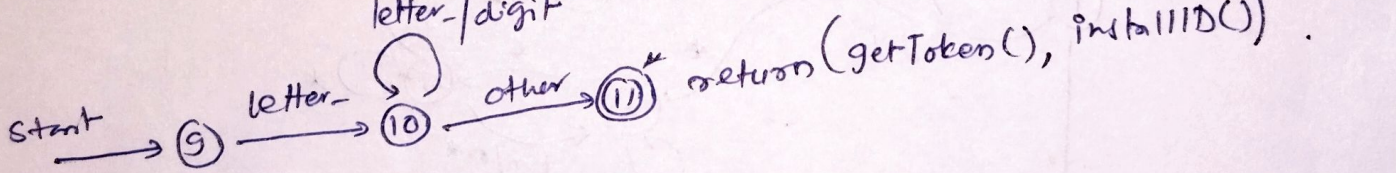
if

```
if → if
then → then
else → else
relop → < | > | <= | >= | = | <>
ws → (blank | tab | new | line)+
```

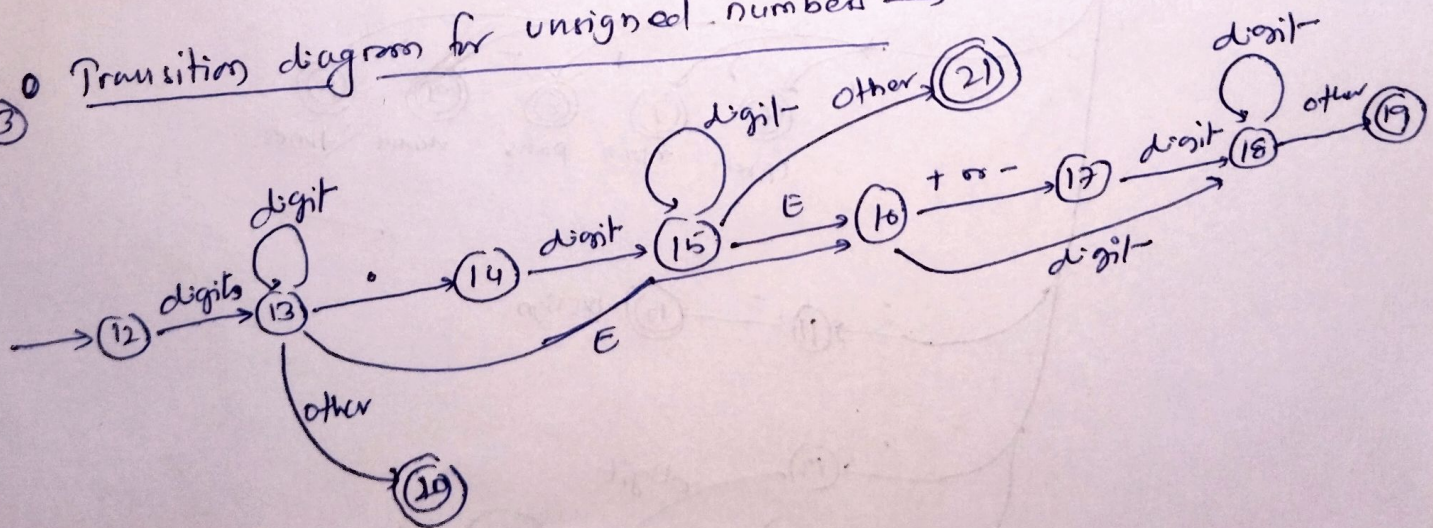
(1) • Transition diagram from relop →



(2) • Transition diagram for id's & keyword →



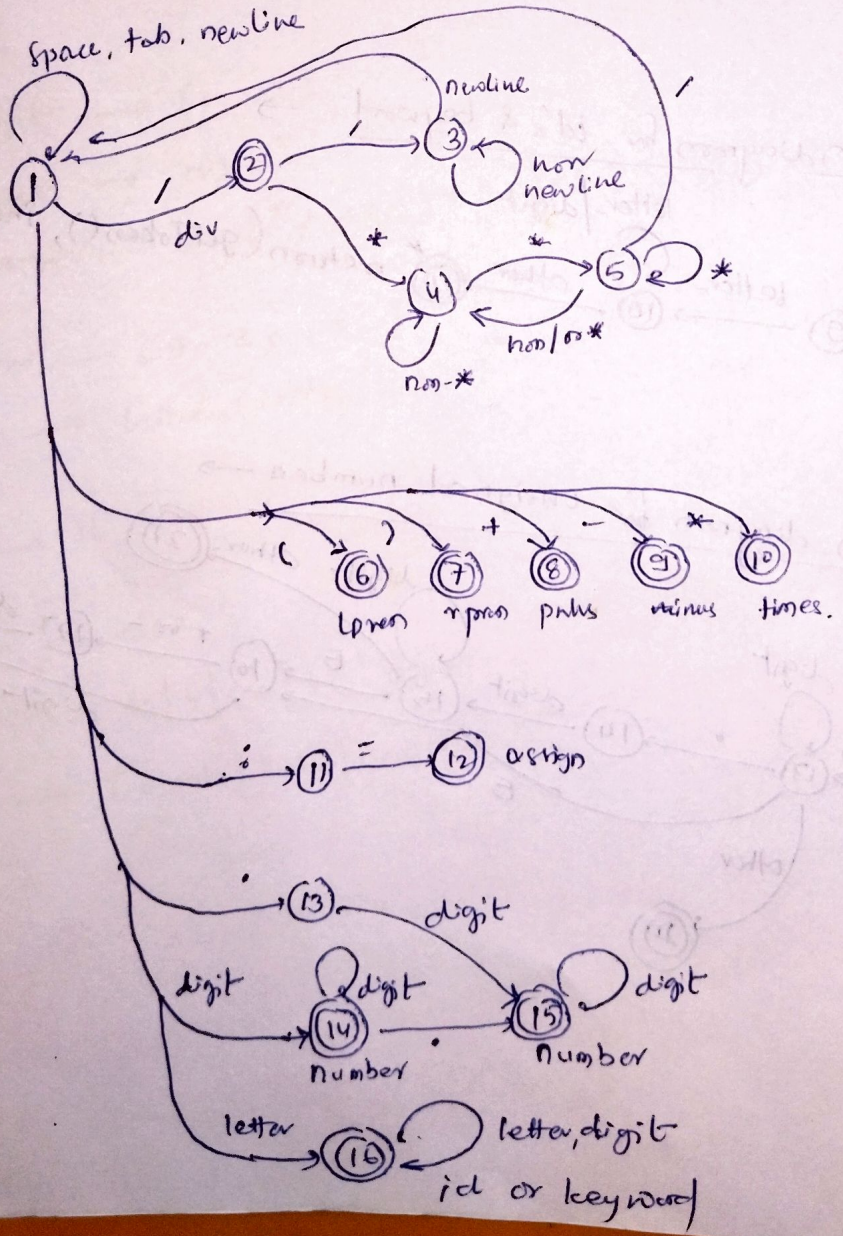
(3) • Transition diagram for unsigned numbers →



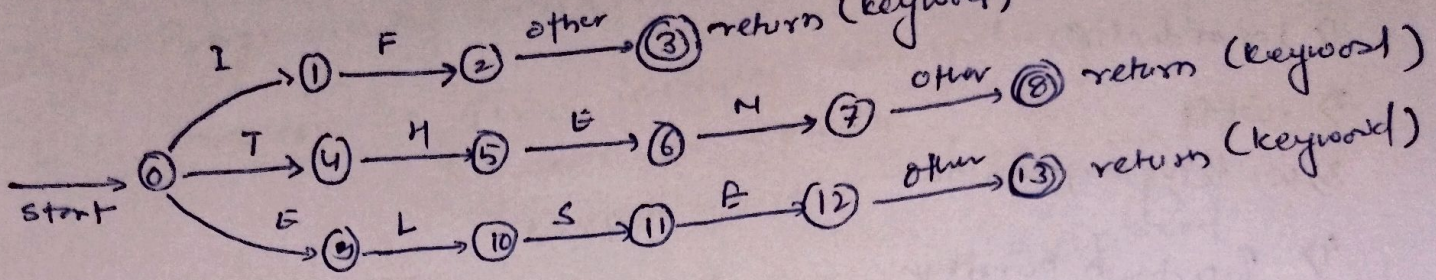
Tokens for a calculator language →

Let us consider the following set of tokens:-

assign → :=
plus → +
minus → −
times → *
div → /
lparen → (
rparen →)
id → letter (letter|digit)*
number → digit digit* | digit* (. digit | digit .) digit*



④ keyword → IF | THEN | ELSE



⑤ Delimiter / whitespace.

delim → space | tab | newline.