⊛ Recognition of Tokens :- → variable/identifiers/constant/Keywords

- Recognition of tokens means how to tokens are identified in a given language.
- Tokens can be recognised with the help of transition diagrams.

Examples:- ① Recognition of tokens (identifiers)

② Recognition of delimiters

③ Recognition of Relational Operators

④ Recognition of keywords such as: if, else, for, etc.

⑤ Recognition of Numbers (int, floating point) ~~or~~ Constant

⑥ ~~Recognition of [illegible]~~

⇒ ① Recognition of Identifiers (token):-

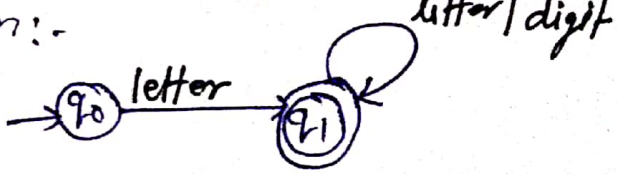
RE for recognition of identifiers

letter → a|b|....|z| A|B|C|....|Z|

digit → 0|1|....|9|

id → letter (letters | digit)* → Rule for identifiers.

∴ Transition diagram:-



Eg:- a, abc12, seven, etc. these are identifiers

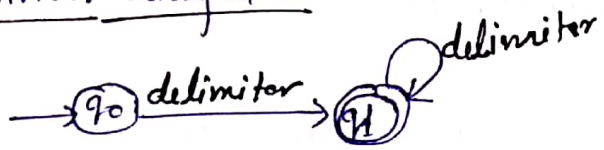
~~if~~ ...

(2) Recognition of Delimiters:-

A delimiter may be a blankspace, tabspace, newlinecharacter etc. ...

ws → delimiter (delimiter)*

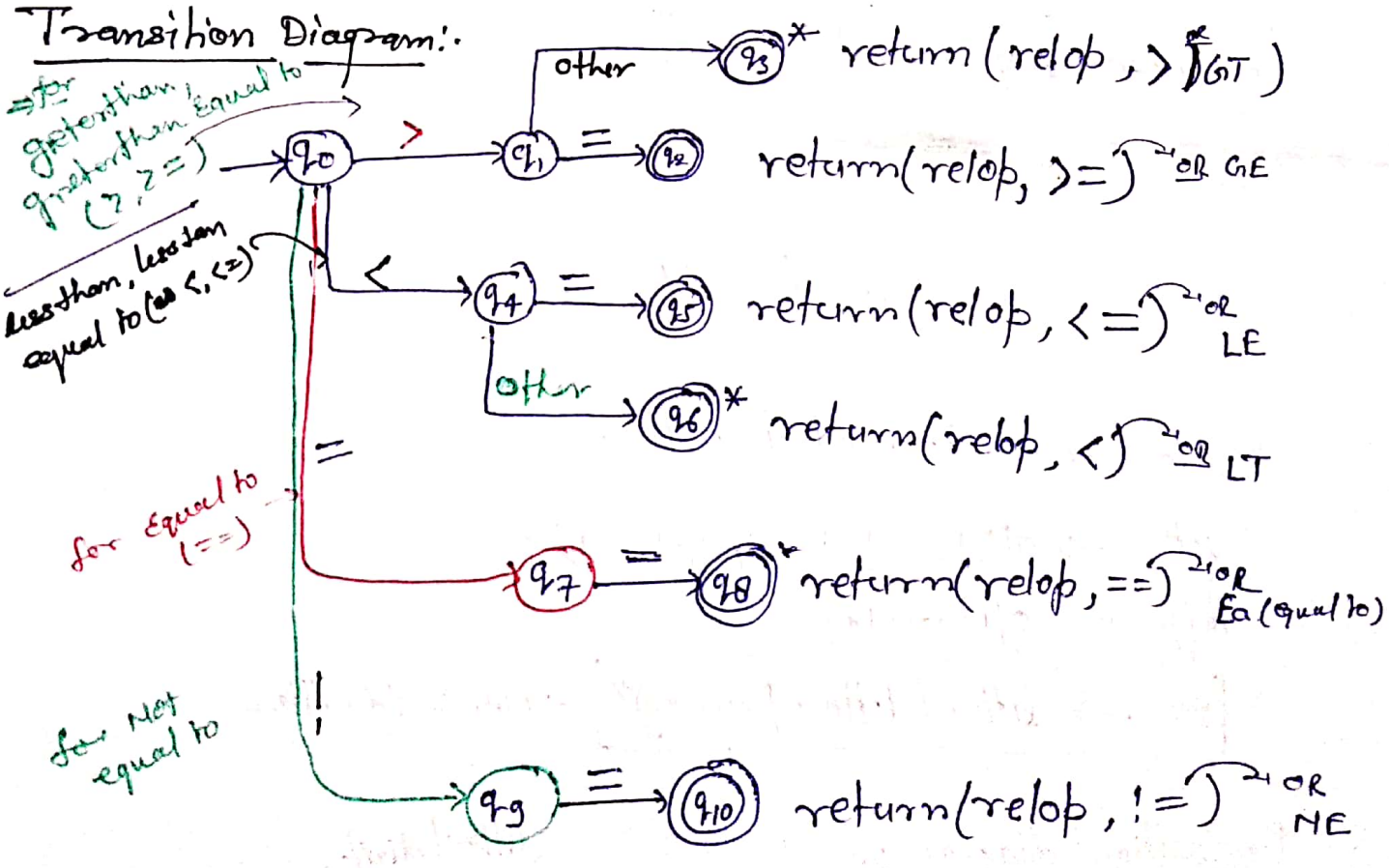
Transition diagram:-



(3) Recognition of Relational operators :-

Relational operators are: >, >=, <, <=, ==, !=/<>

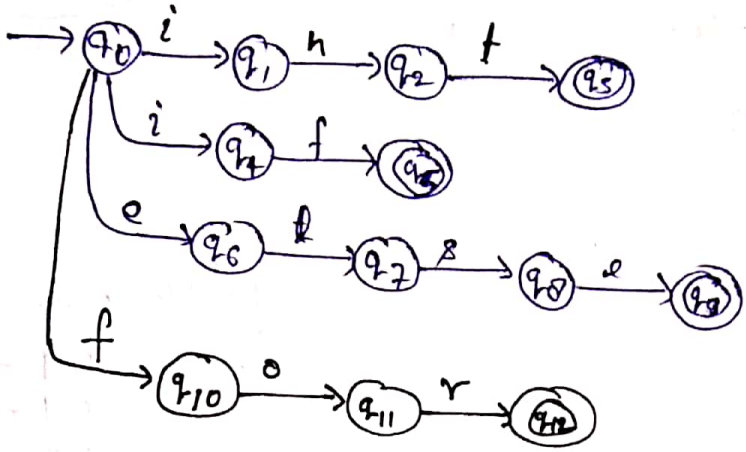
Transition Diagram:-



④ Recognition of Keywords :-

Keywords : int, float, if, for, etc. ... else,

Transition Diagram:-



⑤ Recognition of Numbers ~~/digit~~ :-

Eg:- Reg. expr. for digits (int no., floating point no.)

~~Ref~~ ~~RE~~

digit → 0|1|...|9|

digit → digit (digit)*

number → digit (·digit)?

123, 456, 987, ... → int no.

123.45?? ... → floating pt. no.

123.45 E23, ... → Exponential Notation

123.45 E-23, ...

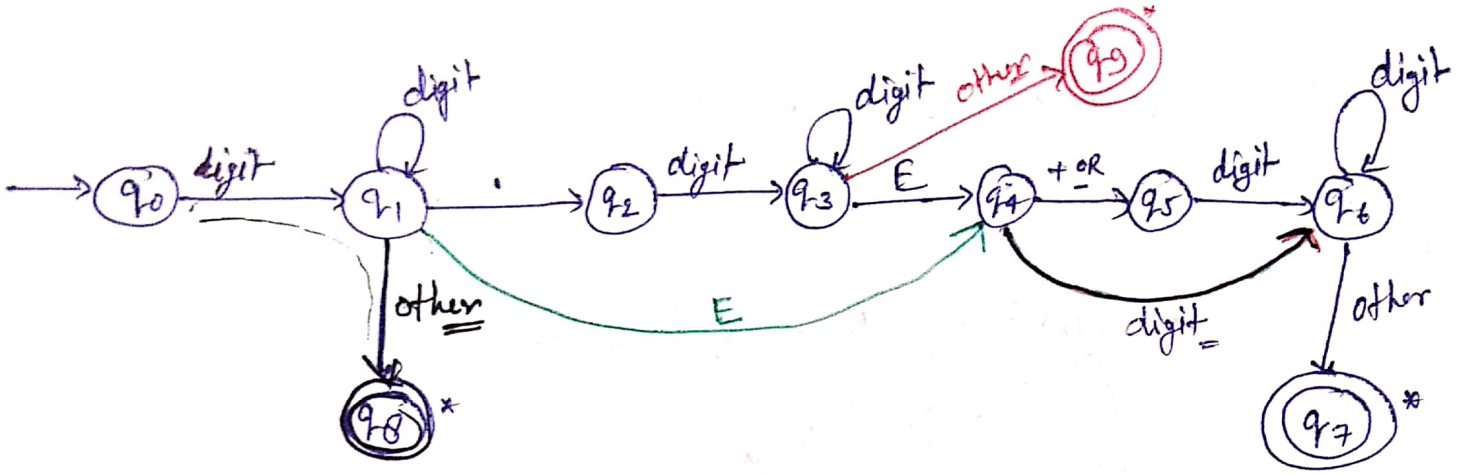
123E+23 (0 exponent)

number → digit (·digit)? (E[+-]? digit)?

⇓ R.E. for Recognition of No. ⇓

RE: number → digit⁺ (·digit⁺)? (E[+,-]? digit⁺)?

Transition diagram:-



possible combinations:-

- D → —
- D → —
- D → —
- ✓ →

⊛ Recognition of Tokens :-

eg:- if :⇒ int a[4][5]; find the no. of tokens?

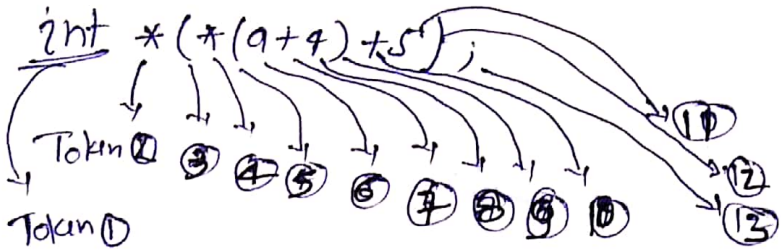
Soln:-

int a[4][5]; //* 2D Array

it will convert in

int *(*(a+4)+5);

Then calculate token / identify token



Total no. of token is = 13

So we can write 1D Array as.

$a[i] = *(a+i) = *(i+a) = i(a)$

2D Array

a[i][j] ⇒ a[i][j]

first solve this ①

then solve this ②

eg:- a[i][j]

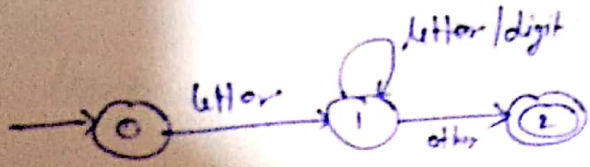
① *(a+i)[j]

consider as term

② *(*(a+i)+j)

⊛ LEX :-

- Lex is a program (generator) that generates lexical analyzer (widly used on UNIX).
- it is mostly used with Yacc parser generator.
- it is written by Eric Schmidt and mike Lick.
- it reads the input stream (specifying the lexical analyzer) and outputs source code implementing the lexical analyzer in the C programming language.
- Lex will read patterns (regular expressions), then produces C code for lexical analyzer that scans for identifiers.
- Regular expressions are translated by Lex to a computer program that mimics an FSA.
- A simple pattern is letter(letter | digit)*.



Start: goto state0

State 0: read c
if c = letter goto state1
goto state 0

State 1: read c
if c = letter goto state 1
if c = digit goto state 1
goto state 2

State 2: accept string.

Note: Some limitations, Lex cannot used to recognize nested structure such as parenthesis, since it only has states and transitions b/w states. So Lex is good at pattern matching, while Yacc is for more challenging tasks.

⊛ Lex file format: -OR Structure of Lex Program :-

Lex source is separated into three sections by %.% delimiters.
The general format of lex source is:

```
{definitions}
%%
{transition Rules}
%%
{user subroutines}
```

{definitions}, {transition Rules} → Required section
{user subroutines} → optional section

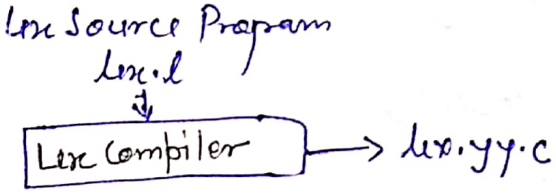
*The absolute minimum lex program is thus:

```
%%
```

---

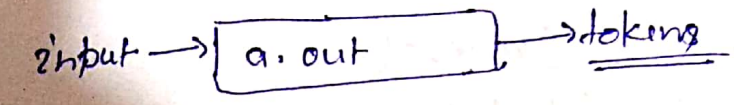
*The function of Lex is as follows:-

Step ① :-



Step ② :-

Step ③ :-



Eg:- Write a lex prog:- Let us see the Lex Program for the regular expression is letter (Letter | digit)*

Pg:-

```
digit [0-9]
letter [A-Z a-z]
%{
    int count;
%}
%%
    /* match identifier */
{letter}({letter}|{digit})*    count++
%%
int main(void){
    yylex();
    printf("No. of identifiers = %d\n", count);
    return 0;
}
```

⊛ Lex Predefined Variables :-

Name	function
int yylex(void)	call to invoke lexer, returns token
Char* yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
file *yyout	output file
FILE* yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start condition
ECHO	write matched string.

⇒ Syntax Analyzer :-

grammar $G: (V, T, P, S)$

Suppose we write type-2 Grammar (type-2 grammar / CFG)

Production Rule form:

$$V \to (VUT)^*$$

G:

$P: \{ S \to \alpha A \beta \}$
$\{ A \to a Ab / \lambda \}$ P

$V = \{S, A\}$, $T = \{\alpha, \beta, a, b\}$

Derivation: a string $w \in L(G)$ iff there exist some derivations.

Let us consider: $w_1 = \alpha\beta$

find drivation: $S \to \alpha A \beta \xrightarrow{A \to \lambda} (\alpha\beta)$

$w_2 = \alpha ab\beta$

drive string $S \to \alpha A \beta \xrightarrow{A \to aAb} \alpha a A b \beta \downarrow A \to \lambda$ $(\alpha ab\beta)$.

Parsing :- finding a sequence of production by which a string $w \in L(G)$ is drived.

Eg: $w_2 = \alpha ab\beta$

$S \to \alpha A \beta \longrightarrow \alpha a A b \beta \longrightarrow (\alpha ab\beta)$ drived ⇒ These sequence is called parsing.

Eg:- Let $G: (V, T, P, S)$, a is CFG.

$S \to ABC$
$A \to a$
$B \to b$
$C \to c$

$L(G) = \{abc\}$

if $w = abc$

drive w: $S \to \underline{A}BC \to aBC \to abC \to (abc)$ r

LMD

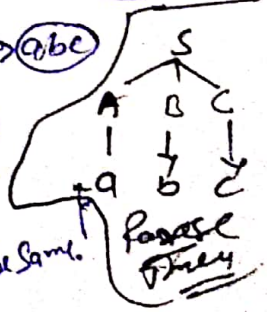
RMD $S \to AB\underline{C} \to ABc \to Abc \to (abc)$

$S \to A\underline{B}c \to AbC \to abc \to (abc)$

$S \to \underline{A}BC \to aBC \to abC \to (abc)$.

$S \to A\underline{B}C \to AbC \to abC \to (abc)$

Here for All drivations the parse tree will be Same.



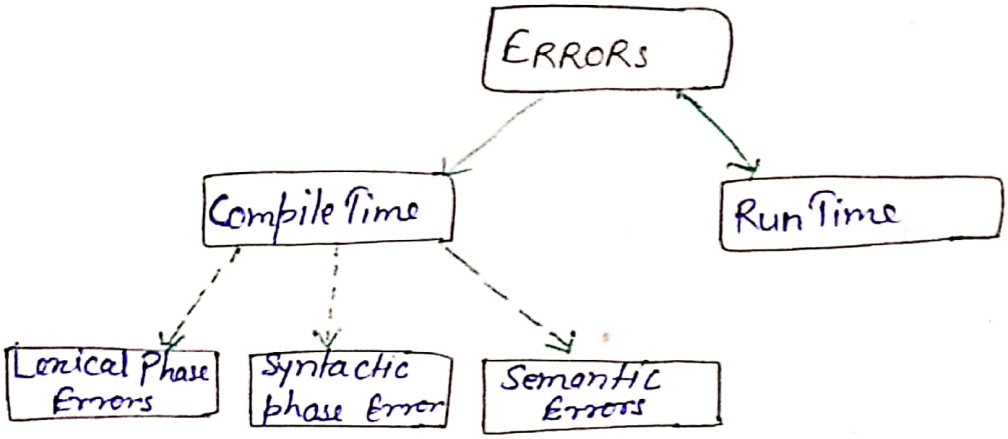
Parse tree

Here total No of derivation exist:-

$S \to ABC$ — ③ × ② × ① = $3 \times 2 \times 1 = 6$

⊛ ERROR HANDLER:-

Error handler is detecting each phases' error, reporting it to the user and then making recovery strategy for handling error. An error is a blank entry in the symbol table.



⊛ Classification of Compile-Time Error -

Compile-time errors rises at compile time, before execution of program. Syntax error or missing file reference that prevents the program from successfully compiling is the example of this.

1. Lexical :- This includes misspelling of identifiers, keywords or operators.
2. Syntactical :- missing semicolon or unbalanced parenthesis
3. Semantical :- incompatible value assignment or type mismatches between operator and operand.
4. Logical :- Code not reachable, infinite loop.

⊛ Lexical Error :-

→ During lexical phase, lexical ~~error~~ can be detected
- Lexical error is a sequence of characters that does not match the pattern of any token.
- Lexical phase error is found during the execution of the program.

Lexical Error Can be:-

- Spelling error
- ~~Exceed~~ Exceeding length of identifier or numeric constants.
- Appearance of illegal characters
- To remove the character that should be present.
- To replace a character with an incorrect character.
- Transposition of two characters.

Eg:-

```
Void main()
{
int x = 10, y = 20;
Char *a;
a = &x
x = 4 x ab;
}
```

In this code 4 x ab is neither a number ~~nor~~ or nor an identifier.
So, this code will show the lexical error.

⑧ Syntax Error:-

- Syntax error is appears during Syntax analysis phase.
- it is found during the execution of the program.
- Some Syntax error can be:-
  - Error in Structure
  - Missing operators
  - Unbalanced parenthesis

eg: if we using only "=" equal to, when "==" is needed.

```
if (number = 200)        → Here when checking the equation we need to write (number == 200) → This is correct syntax.
    Cout << "number is equal to 200";
else
    cout << "no. is not equal;
```

Here Syntax error.

in this example Syntax warning will come.
In this code, if expression used the equal sign which is actually an assignment operator not the relational operator which test for equality.

eg:- float x = 1.2 // semicolon is missing.

Errors in expressions.

x = (3+5 ; // missing closing parenthesis ')'.

⇒ Semantic Error :-

- Semantic error occurs during semantic phase. it is detected at Compile time.

Semantic Error can be :-

- Incompatible type of operators.
- Undeclared variable
- Not matching of actual argument formal argument.

eg:- Use of non-initialized variable:

```
int i;
Void f(int m)
 {
  m=t;     // it is undeclared, it shows semantic error.
```

eg:- Type incompatibility:

```
int a = "hello";   // the type String & int are not compatible
no:        char
```

eg:- Errors in Expressions :-

```
string s= "....";
int a = 8.8; // the '_' does not support arguments of type String.
```

eg:- ~~int fa~~ int a[10], b;

```
{
 a=b;     // Somatic error bez. here we equalize a normal variable & array.
}
```

eg:-

```
if (K=1)
{
 return 1;
 printf("%d", K);   // here semantic error bez. we written return stmt before printf.
}
```