* LANGUAGE PROCESSORS.

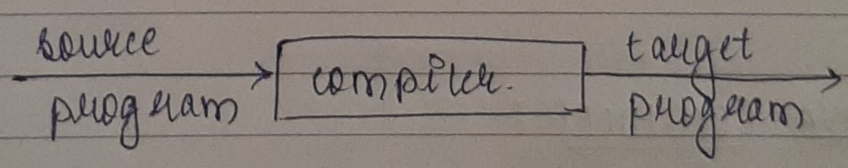
Language processors.

compiler, assemblers, loaders and linkers, preprocessors

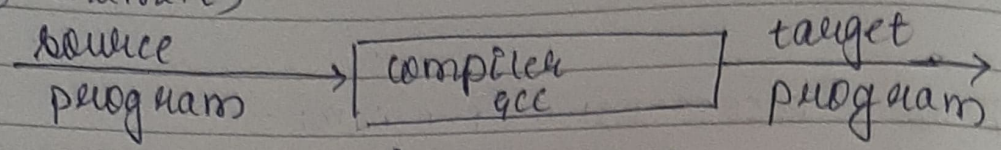
* Applications:

COMPILER : It is a software which converts HLL into LLL.

compiler is a system software / language processor which converts high level language (source program) into its equivalent machine level program (target program)

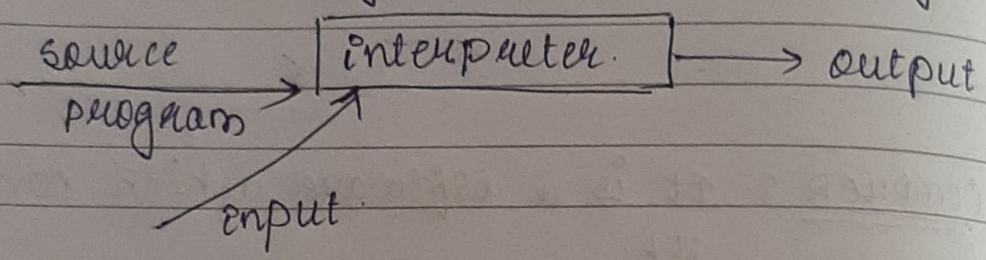


(filename.c) a.out



ADVANTAGES: It displays error.

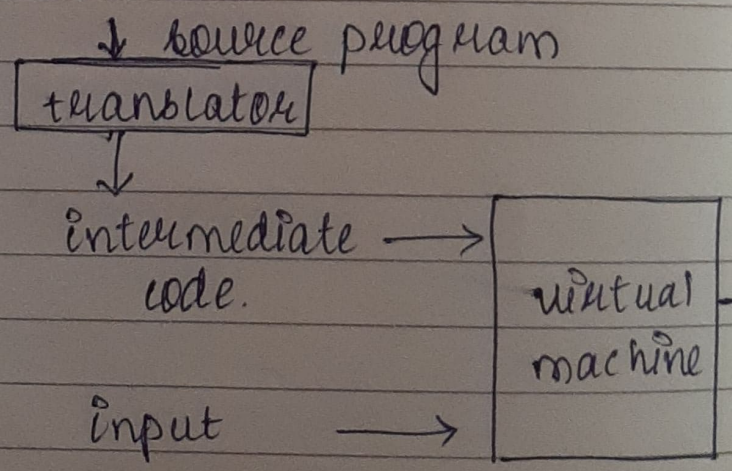
* INTERPRETER: Interpreter compiles line by line
- Interpreter handles error better way because it executes line by line.
- Interpreter will not generate the target program.



5m Q

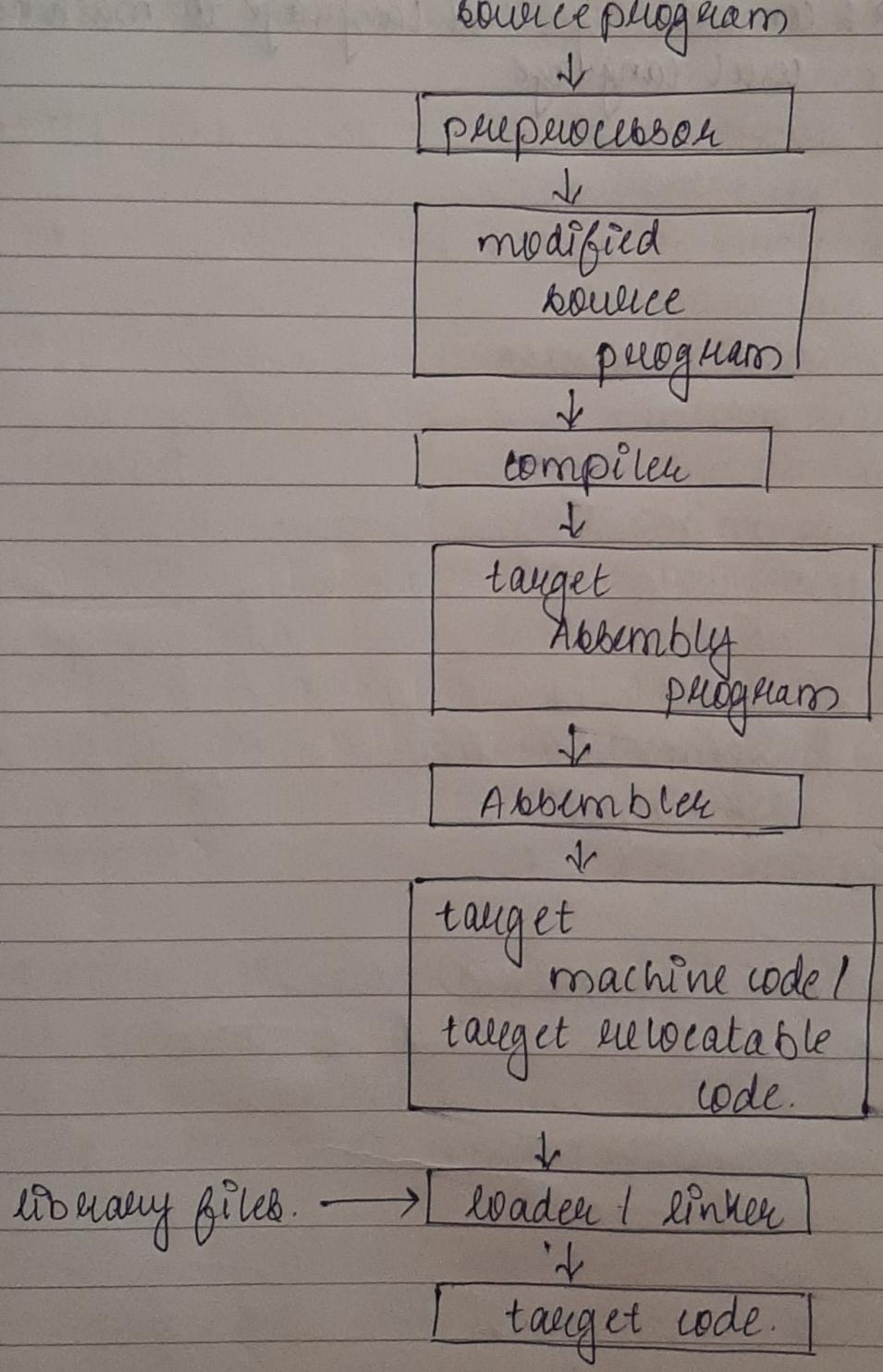
* JAVA COMPILER:

(in general java compiler)



6m

* **LANGUAGE PROCESSING SYSTEM:**



Preprocessor: Adds

STRUCTURE OF COMPILER.

character stream
↓
[Lexical Analyzer / lexer]
↓ token stream
[System Analyzer]
↓ syntax tree
[Symentaic Analysis]
↓ syntax tree
[Intermediate code Generator]
↓ Intermediate representation
[Machine Independent code optimizer.]
↓ Intermediate representation
[code Generator.]
↓ Target machine code
[Machine Dependent code optimizer.]
↓ Target machine code.

[Symbol Table]

- machine Independent / Dependent code optimizer can be optional.
- If we dont consider these phases then the synthesis part will contain only 1 phase.

→ compiler is divided into 2 parts
① Analysis part this is called as front end
② synthesis part this is called as Back end

Analysis part consists of 4 phases.
synthesis part contains of 3 phases or 1.

① lexical analyzer: takes input as character and produces O/P as sequence of characters called as ~~lexins~~. lexeme.

eg: int a, b, c;
there are 7 lexeme in the above eg. i.e (int a, b
1 2 3 4
Every lexeme will have a token
< token-name, attribute-value >

→ It will read character one at a time.
→ Spaces & comments will be deleted by the lexical analyzer.
→ ~~lexins~~ lexemes are the meaningful sequence.

⊙ lexical analysis is the 1st phase of which takes as input and groups them into meaning ful sentence called lexeme.
For each lexeme the lexical analyzer produces a token as a output.

token: → abstract symbol
< token-name, attribute-value >
is the syntax which is used for token.

From above eg:
< key-word, int >
< id, a >
< punctuation, , >
↓ abstract symbol
→ token name
↓ attribute value.

• Each token need not have 2 attributes.
eg: < 10 >
< num, 10 > } valid.

Eg: int a = 10.

3 address code → maximum we have 3 operands assumate
RHS have max 1 operator (assignment statement)

Symbol Table

1.	position	...
2.	initial	...
3	rate	...

position = initial + rate * 60

↓

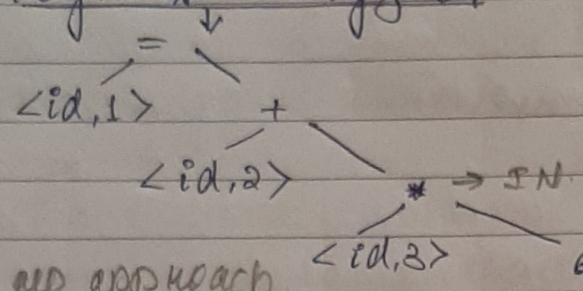
[Lexical analyzer]

↓

<id,1> <=> <id,2> <+> <id,3> <*> <60>

↓

[Syntax Analyzer]



internal node → operator
leaf node → operand

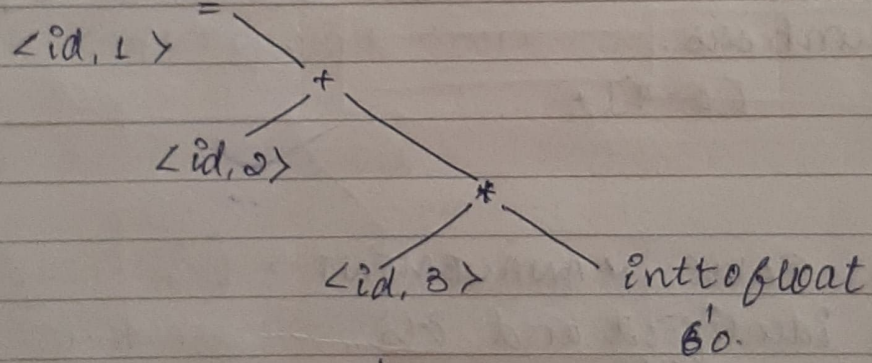
higher priority
first multiplication then addition and then result.

bottom up approach

↓

[Semantic Analyzer] it does type checking & type conversion (int to float) n al

↓



↓

[Intermediate code generator] output is in 3 address code.

↓

t1 = inttofloat (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

→ upto here it is frontend of compiler.

↓

[code optimizer]

can reduce instruction set, fast in generate less memo storage

↓

t1 = id3 * 60.0
id1 = id2 + t1

MUL → multiplication # → immediate ;
F → float
R → register
LD → load
ST → store

↓

[code generator] generate a code
↓ assembly code easy to read

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

add

value is stored here.

eg $u = 2 + 2$.

$R_1 = R_1 + R_2$

add

store

* Regular expression for identifier:

[a-z] [a-z 0-9] * | [_a-zA-z] [a-zA-Z

for numbers.

[0-9]*.

→ SYMBOL TABLE MANAGEMENT → this should be written when structure of compiler is asked.

- contains identifier and its attributes → type, location
- functions & its attributes.

GROUPING OF PHASES INTO PASSES.

1. Lexical Analyzer.
2. Syntax
3. Symantic
4. Intermediate code generator
5. m/c independent code optimizer
6. code generator
7. m/c dependent code optimizer.

5th one is optional.
6 & 7 can be combined together.
when first 4 is grouped together we get

Imp
** COMPILER CONSTRUCTION TOOLS

- readily available tools
  eg: lex, yacc, debugger
  lex → generates 1st phase
  yacc → generates 2nd phase.

(1) parser generators : generates ~~y~~ parser
eg yacc
(2) scanner generator : generates lexical analyzer
eg lex tool
(3) syntax directed translation engines : used for intermediate code generator.
(4) ~~Port~~ code generators : to generate code generator.
(5) Data flow analysis engines: used for optimization

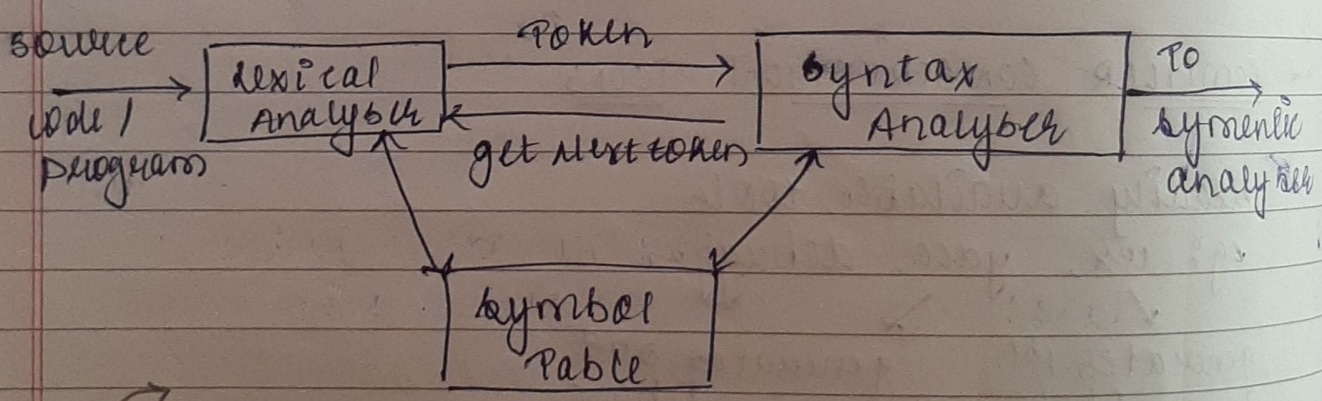
lex → regular expression (to check if they are valid)
yacc → grammar (to check if it is valid)
↓
shows syntax error call.

(8) compiler construction tool kits

→ LEXICAL ANALYSIS.

(•) The role of lexical analyser.
→ deleting comments and ~~byte~~ white spaces.
white space → tab, space
→ keeps tracks of new line character. '\n'.
→ generates tokens
→ It indicates the error on the specific line and hence it allocates line numbers.

→ LEXICAL ANALYSER AND PARSER COMMUNICATION



(•) Explaining role of lexical above points and the above diagram along with the explaination.

→ LEXICAL ANALYSIS V/S PARSING.

- simplicity of design
- compiler efficiency is improved
- compiler portability is enhanced.
when lexical analyser is separated from parser

ullman.

classmate
Date
Page

we observe the above points.

Imp 6m
*** → TOKENS, PATTERNS AND LEXEME defination with eg.

1st 2nd 3rd
Pattern, lexeme, tokens the order

- Pattern → Patterns are regular expressions. these are used to identify lexeme.

give defi of RE
A pattern is a description of the form that the lexemes of a token may take.

- Lexeme → It is a sequence of chara in the source program that matches the pattern for a token and is identified by

A token is pair consisting of token name and an optional attribute value.

eg: printf (" Total = %d \n", total);

Identify patterns token and lexeme from above eg.

eg: printf (" Total = %d \n", total);

→ first find the lexeme.

(a) printf
(b) (
(c) " Total = %d \n"
(d) ,
(e) total
(f))
(g) ;

anything between " " is considered 1 lexeme.

∴ 7 lexeme.

- " " anything between double quotes is literal.

printf → identifier
(→ punctuation
" Total = %d \n" → literal
, → punctuation
total → identifier
) → punctuation
; → punctuation

→ Patterns.
identifier [a-z] [a-z 0-9-]*
punctuation [,] [;] [(] [)]
literal.

→ Tokens.
<id, 1>
OR
<id, entry to the symbol table for printf>
<(>
<literal>
<,>
<id, 2> OR <id, entry to the symbol table for total>.
<)>
<;>

symbol table.

1.	printf	....
2	total	....

if we dont want to draw symbol table.

ATTRIBUTES FOR TOKENS

① 5-6m ** Identify tokens, patterns and lexems for the following
E = M * C * * 2

Lexeme	Pattern or Regular expression	token
E	[a-z A-Z] [a-z A-Z 0-9] *	<id, E> or <id, 1> or <id, pointer to symbol table entry for E>
=	[=]	<assign_op> or <=>
~~symbol table~~ M	[a-z A-Z] [a-z A-Z 0-9] *	<id, M> or <id, 2> or <id, pointer to symbol table for entry M>
*	"*"	<*> or <multi_op>
C	[a-z A-Z] [a-z A-Z 0-9] *	<id, C> or <id, 3>
**		<exp_op>
~~<2>~~ 2.	[2]	<2> or <num_2>

Symbol table:

1	E	...
2	M	...
3	C	

②
```
1 if 2(a 3 = 4 = 5 b 6)
7 {
    8 printf 9 (" 10 Hello \n" 11 ); 12
13 }
else 14
15 {
    16 printf 17 (" 18 bye \n" 19 ); 20
21 }
```

⇒ There are 21 lexem

	TOKEN	INFORMAL DESCRIPTION / PATTERN	SAMPLE LEXEME
1	if	[i][f] → formal [i,f] → informal	if
2	else	[e, l, s, e] "else" → formal.	else.
3	comparison	>, <, >=, <=, !=, ==	<=. >=
4	id	[letter] [letters & digits]	printf a, b.
5	number	any numeric constant → informal [0-9]*	0, 2, 5.
6	literal	anything but " surrounded by " (" ")	"help"
7	Operators.	[+] [-]	+, -
8	Punctuation	[;]	;

LEXICAL ERRORS.

```
fi( a= = b)
{

}.
```

ERROR RECOVERY STRATEGIES.

① Panic mode : we delete successive characters from the remaing input until the remaining input matches with the pattern.

② Delete one character from the remaining input

③ insert a missing character into remaining input

④ Replace a character by another character

⑤ Transpose to adjacent character.

eg : int a, $, b;
delete one character from the input here $ will be deleted and next will be considered, we can also have add in, swap and else instead of delete.

SPECIFICATION OF TOKENS.

① Strings & Languages:

② Operations on languages : union concatination & ~~clos~~ closure.

③ Regular expressions : used for pattern matching having meta character.

PREFIX & SUFFIX

① India → I, In, Ind, Indi, India → prefix
→ a, ia, dia, ndia, India → sufix

② India → I, In, Ind, Indi → proper prefix
→ a, ia, dia, ndia → proper suffix

③ IDA is subsequence of india.
here n and i are missing but it is in sequence.

imp
5-6m
** REGULAR DEFINATION :

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
⋮ ⋮
$d_n \quad r_n$

For notational convinience we give names to certain regular expressions and use those names in subsequent expression.
$d_1, d_2 \ldots d_n$ are names of the regular expression
$r_1, r_2 \ldots r_n$ are regular expression.
if Σ is an alphabet, then a regular defination is a sequence of definations of the form
$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
⋮
$d_n \rightarrow r_n$

- where each "d_i" is a new symbol, not in the summation and not same as any other "d's"
- each "r_i" is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2 \ldots d_{i-1}\}$.

- Write the regular expression for an identifier.

→ letter → [a-z] [a-z]* d_1
digit → [0-9] d_2
identifier → [a-z] [A-Z 0-9]*
letter (letter / digit)* d_3.

eg: S → eSe | e (Recursive grammar)
as LHS and RHS has S is above
eg LHS ≠ RHS.

- Write a regular defination for unsigned number.

→ digit → [0-9]
digits → (digit)$^+$ or [0-9]* or digit (digit)*
number → digits (. digits)?
Exponential → digits (.digits)? (E [+-]? digits)?

imp

INPUT BUFFERING

eg: int a, b;
Here int is a lexeme. the cursor pointer stops after t and at the start of a

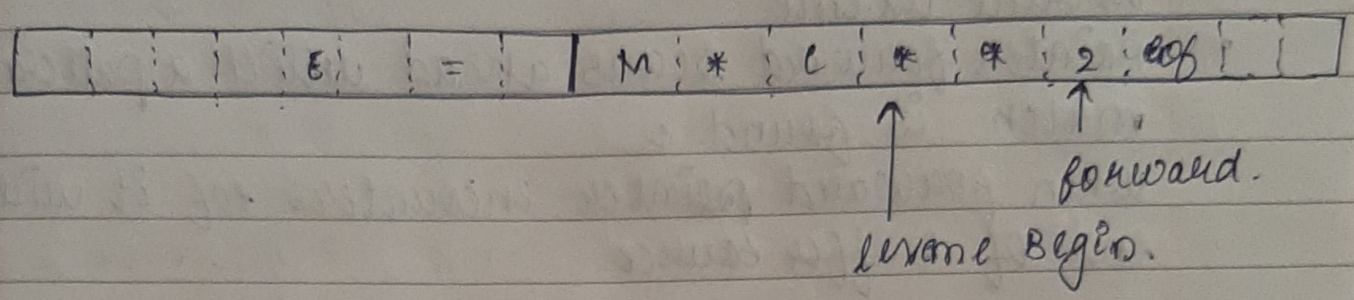
FARTRAN

eg: DO 5 I = 1.25
Here DO5I is considered as 1 lexeme.

instead of 1.25 if it is 1,05 then the 1st lexeme is only 00.

Imp 5-6 m

INPUT BUFFERING



lexeme Begin pointer indicates it has started identifying the lexeme.
forward pointer
Input buffering is used to speed up the process of processing read & write.
It is 4098 bytes.

Eg: int a, b;
lexeme begin pointer will be pointing at "i" initially.
int will be 1st lexeme.
forward pointer will be pointing a initally after scanning int.
Now both lexeme as forward pointer will be at a.
forward will now go to,

Page 115 (second edition) buffer pairs.

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

An important scheme involves 2 buffer that are alternatively reloaded as shown in the figure
Each buffer is of same size N.
2 pointers to the input are maintained

- Pointer lexeme begine marks the beggining of current lexeme
- Pointer forward scans ahead until a pattern match is found

when forward pointer encounters eof it will be end of ~~buffer~~ source.

Imp **SENTINALS** : It is a special type of key which indicates the end of buffer.

imp

RECOGNISATION OF TOKENS.

Pattern → Transition dia → code

Grammar for branching statement and conditional expression.

stmt (statement) → if expr then stmt
| if expr then stmt else stmt
| ε

expr → term relop term
| term

term → id
| number.

six terminals are: if, then, else, relop, id, number

else, then, if is keyword

relop → operation.

number, id → identifier

<u>token</u> <u>pattern</u>.

if → if

then → then

else → else

relop → < | <= | > | >= | = | <>

letter → [a-z]

digit → [0-9]

id → ~~[a-z 0-9]~~* letter (letter | digit)*

digit → [0-9]

digits → digit+

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

constant → number.

- keywords, identifiers, operators, punctuations, identifiers are some of keywords.

- ws → (tab | newline | bank)+ <u>pattern</u> <u>for</u> <u>white</u> <u>space</u>.

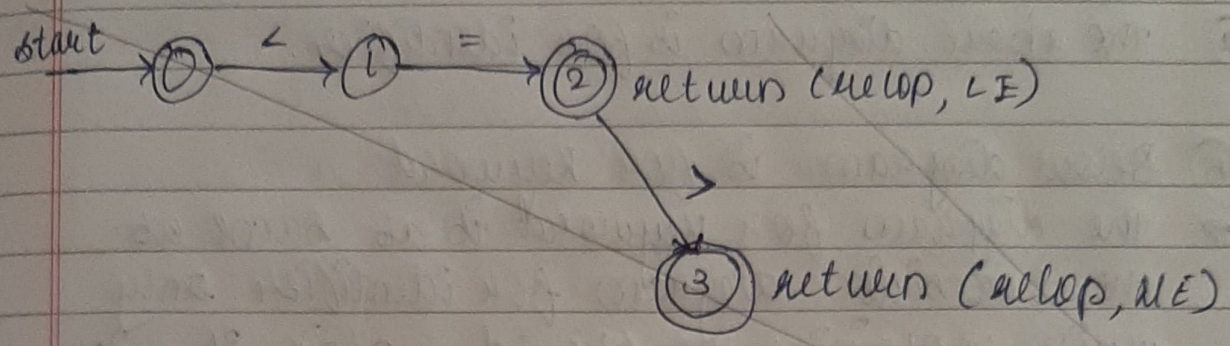
- ws will not be forwarded to parser it will be handled by the lexeme by ignoring it or deleting it.

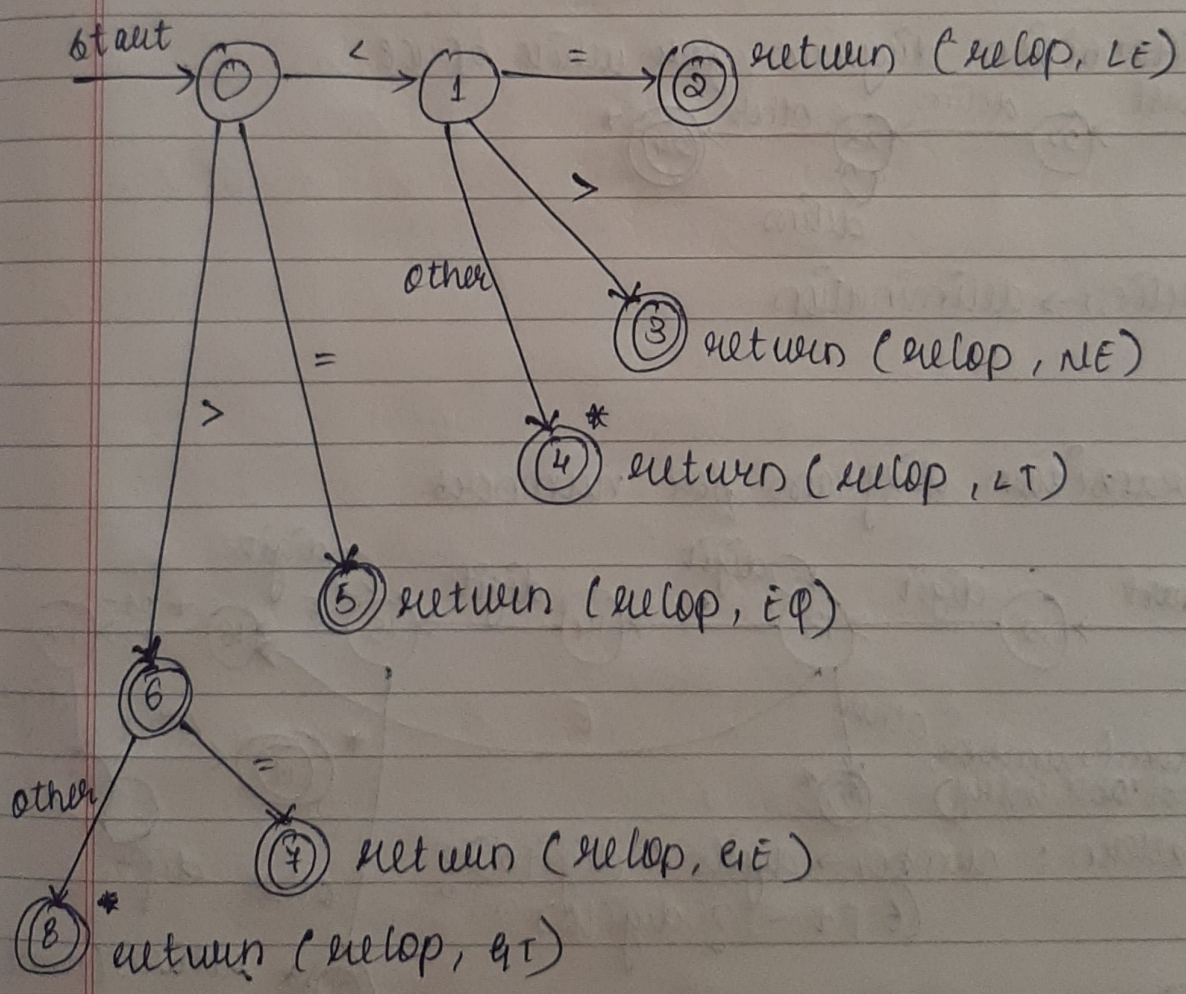
relop → relational

	lexemes	Token name	attribute value
1	Any WS	-	-
2	if	if	-
3	then	then	-
4	else	else	-
5	Any id	id	pointer to table entry.
6	Any number	number eg. 4 <4> or <number, 4>	Pointer to symbol table entry.
7	<	relop	LT (less than)
8	<=	relop	LE (less than or equal to)
9	>	relop	GT
10	>=	relop	GE
11	=	relop	EQ
12	<>	relop	NE (Not equal to)

① Imp Transition diagram for relational operator

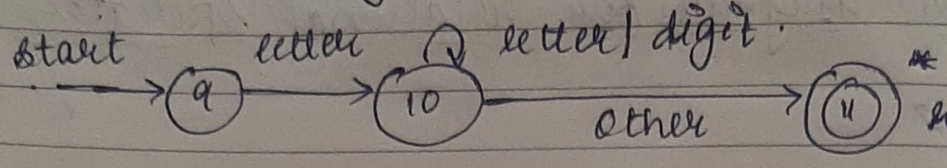
~~Fig~~





- install id will install id to the symbol table
- get token will retrive all info related to keyword / identifier

② Transition diagram for keywords and identi



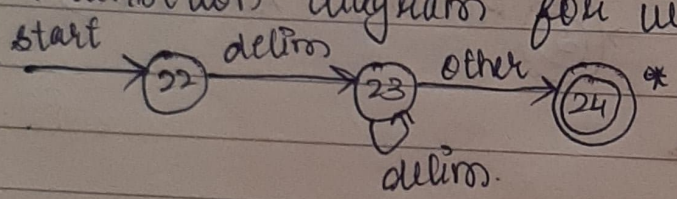
return (getTok
install ID()

ⓐ The above diagram is for identifier.

ⓑ Below diagram is for keyword.

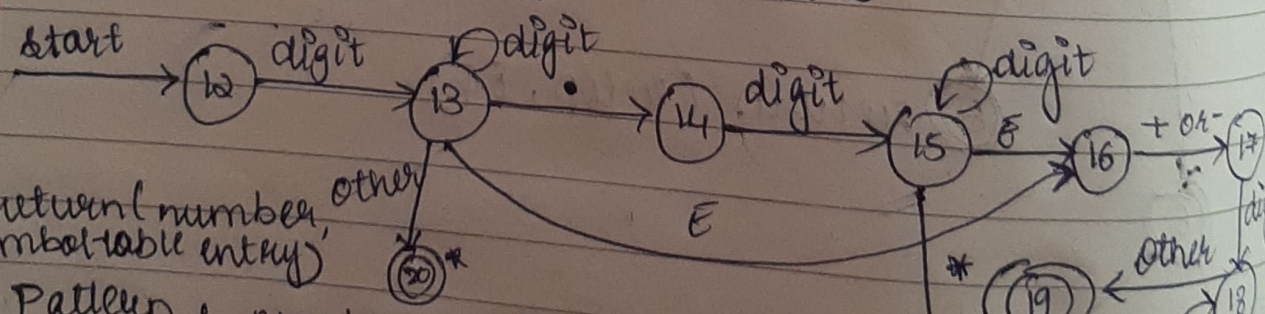
→ The diagram for keyword is as same as the transition diagram for identifier only while explaining we should explain it with respective transition diagram.

③ Transition diagram for white space.



delim → delimination

④ Transition diagram for numbers.



pointer return (number, symbol table entry)

Pattern : number → digits (. digits)? (E [+ -]? digits)?

(19)* return (number, pointer to symbol table entry)

other digit

(21)* return (num
pointer to s
table entry

A architecture of a transition diagram based lexical analyzer.

```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP);
    while (1)   // repeat character processing until a
    {                return or a failure occurs
        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:  ...
                   :

          case 8:  retract();
                   retToken.attribute = GT;
                   return (retToken);
        }
    }
}
```

cases to case 8 we should write.

```
          case 1:  c = next char();
                   if (c == '=') state = 2;
                   else if (c == '>') state = 3;
                   else fail();
                   break;
          case 2:  retract();
                   retToken.attribute = LE;
                   return (retToken);
```

ARCHITECTURE OF A TRANSITION DIAGRAM BASED LEXICAL ANALYZER.

IMPLEMENTATION OF RELOP TRANSITION DIAGRAM

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) // repeat character processing until
             a return or failure occurs //
    {
      switch (state)
      {
      case 0: c = nextchar();
              if (c == '<') state = 1;
              else if (c == "=") state = 5;
              else if (c == ">") state = 6;
              else fail(); // lexeme is not a relop
              break;

      case 1: c = nextchar();
              if (c == '=') state = 2;
              else if (c == '>') state = 3;
              else if (c == other) state = 4;
              else fail();
              break;

      case 2: retract();
              retToken.attribute = LE;
              return(retToken);
              break;
      case 3: retToken.attribute = NE;
              return(retToken);
              break;
```

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

case 5 : retToken.attribute = EQ;
         return (retToken) ~~LT~~;
         break;

case 6 : c = nextchar();
         if (c == "=") state = 7;
         else if (c == other) state = 8;
         else fail;
         break;

case 7 : retToken.attribute = GT;
         return (retToken);
         break;
    }
  }
}
```