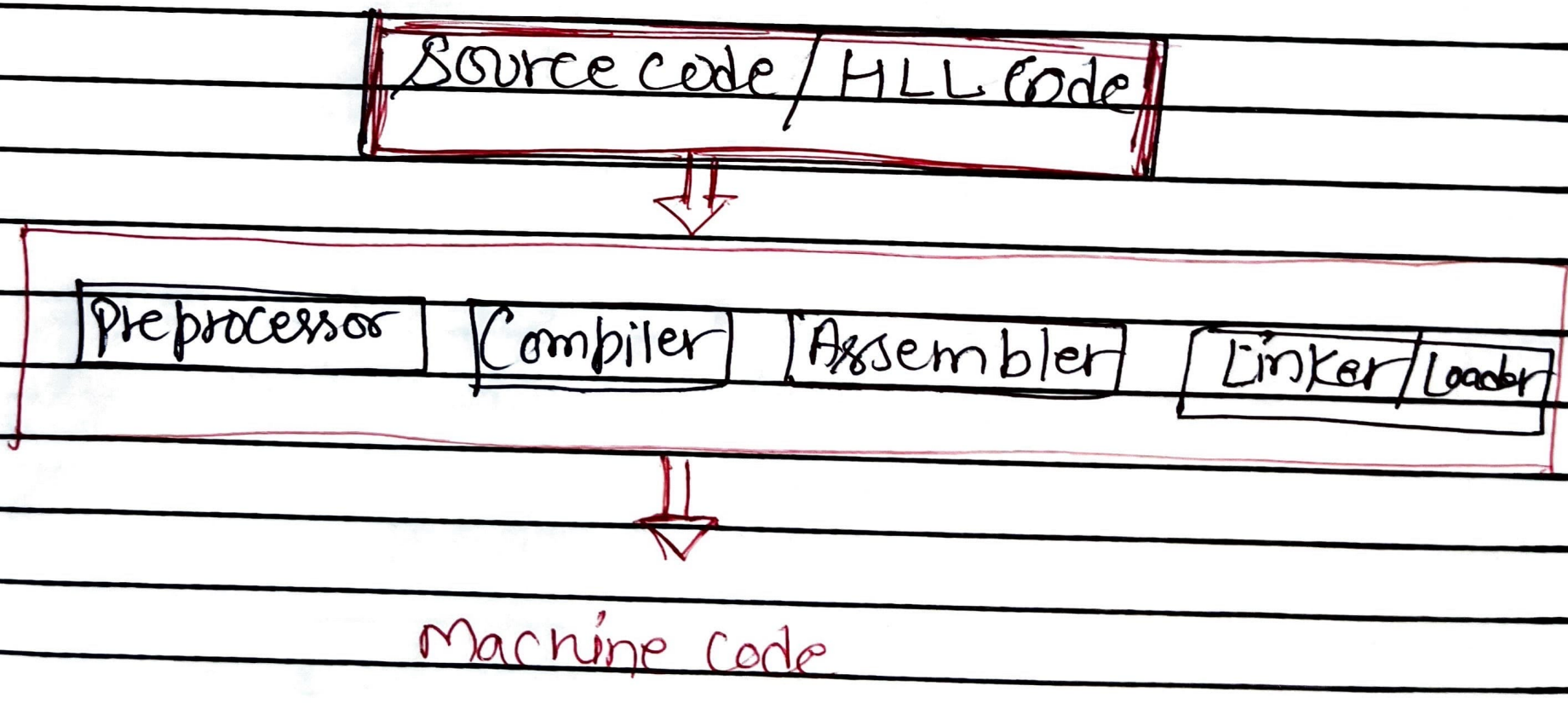
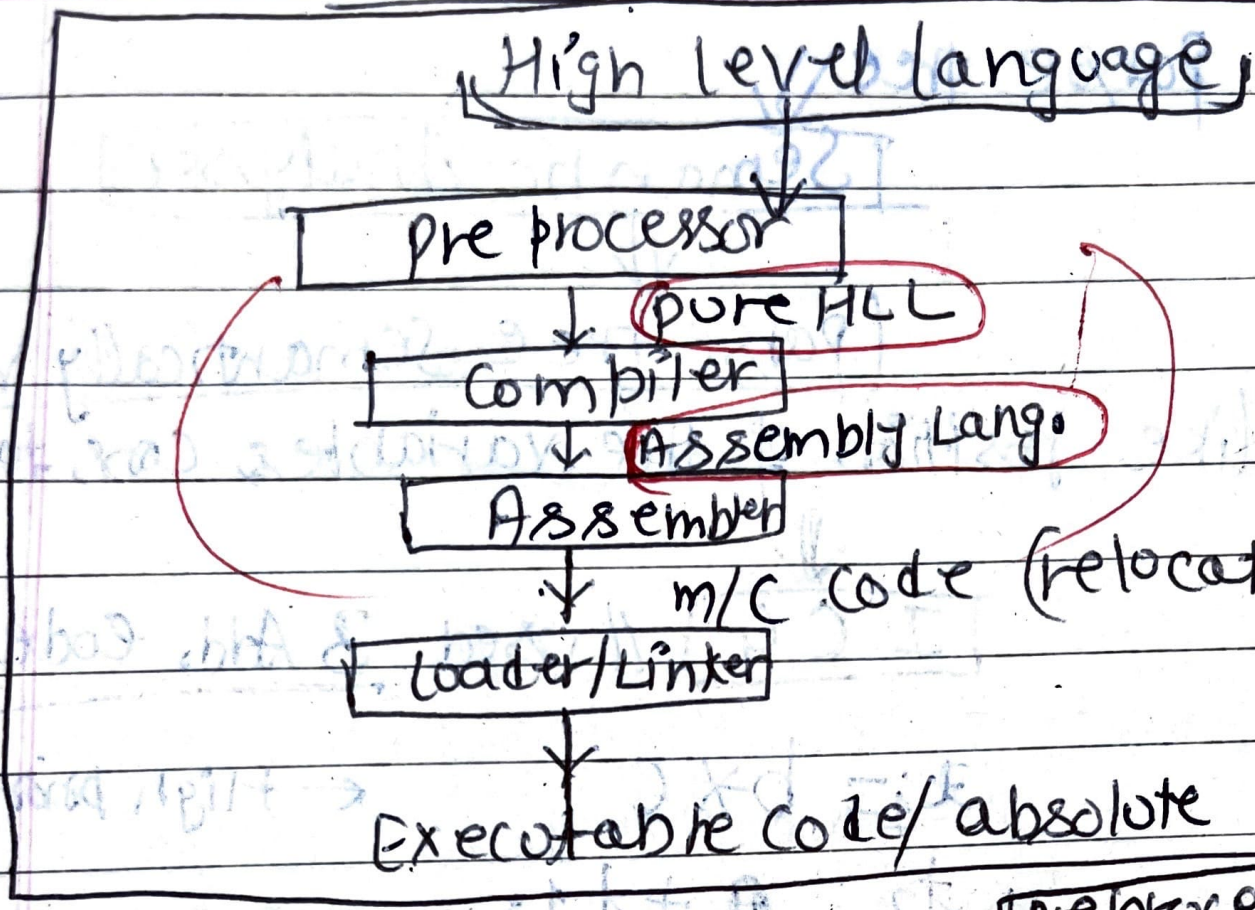
Language Translator

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



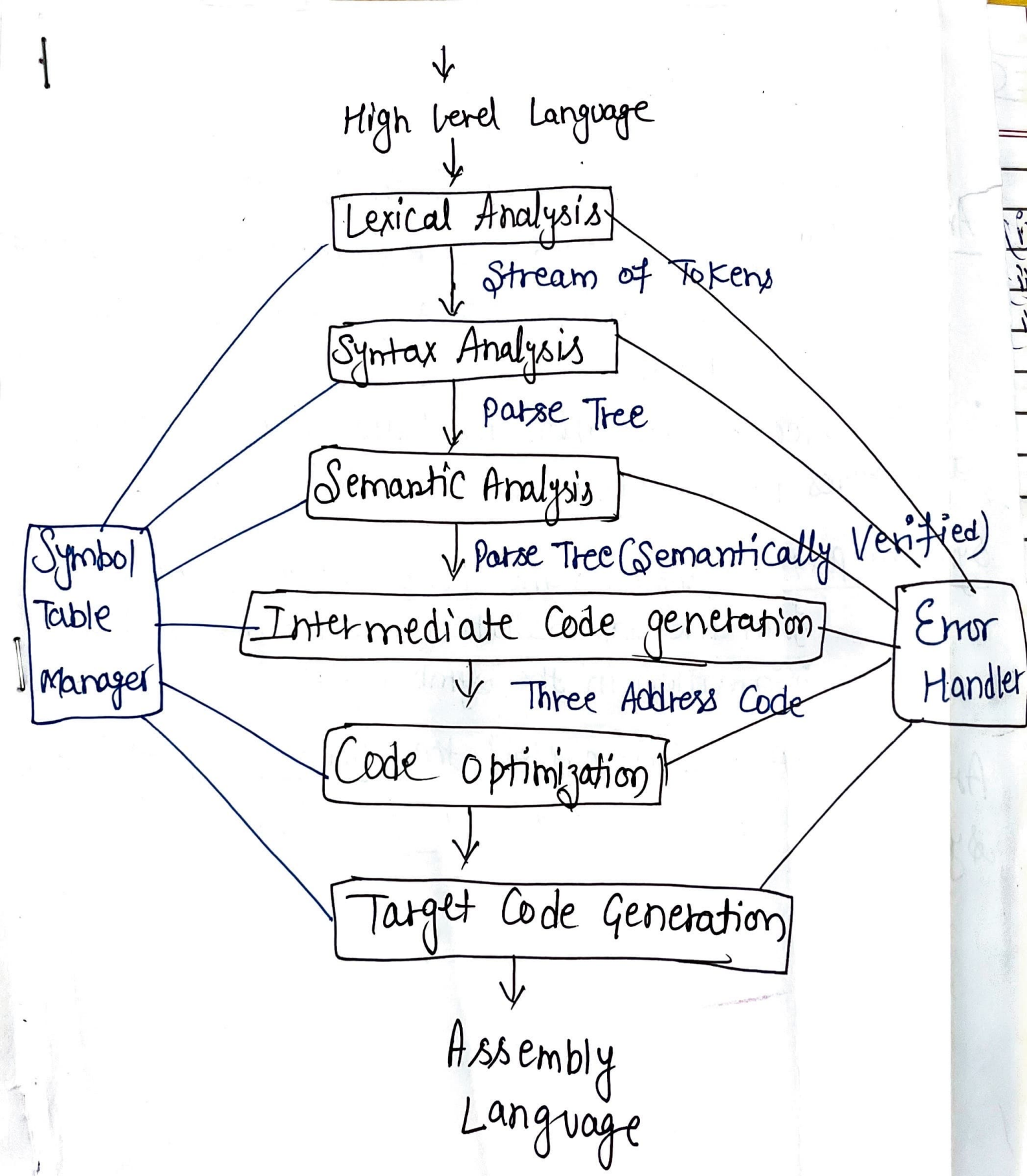


HLL :- C, C++, Java etc with Header files

preprocessor directives

<#include ~>

preprocessor :- ~~converts~~ Include these header files into the main program and remove these header file inclusion. and convert it into a pure HLL


High Level Language
Lexical Analysis
Stream of Tokens
Syntax Analysis
Parse Tree
Semantic Analysis
Parse Tree (Semantically Verified)
Intermediate Code generation
Three Address Code
Code optimization
Target Code Generation
Assembly Language
Symbol Table Manager
Error Handler

The Structure of a compiler

Two parts :- Analysis and Synthesis Part

Analysis Part:- • Breaks up the source program into constit pieces and imposes a grammatical structure on them.

• Also collects information about the source program and stores it in a data structure called a Symbol Tabl

Synthesis Part:- • Constructs the desired target program from the intermediate representation and the information in the symbol table.

Analysis part is often called the front end of the co
Synthesis " " " " Back end " " "

① Lexical Analysis :-

- First phase of a Compiler, also called Scanning
- Reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

token form :-

<token-name, attribute value>

- for each lexeme, the lexical analyzer produces an output a token.
- Example :- position = initial + rate * 60

Characters of this statement, grouped into lexemes :-

	lexeme	Tokens
①	position	token ⇒ <id, 1>
②	=	<=>
③	initial	token <id, 2>
④	"+"	mapped into the <+> token.
⑤	rate	<id, 3>
⑥	*	<*>
⑦	60	<60>

<id, 1> : 1 points to the symbol table entry.

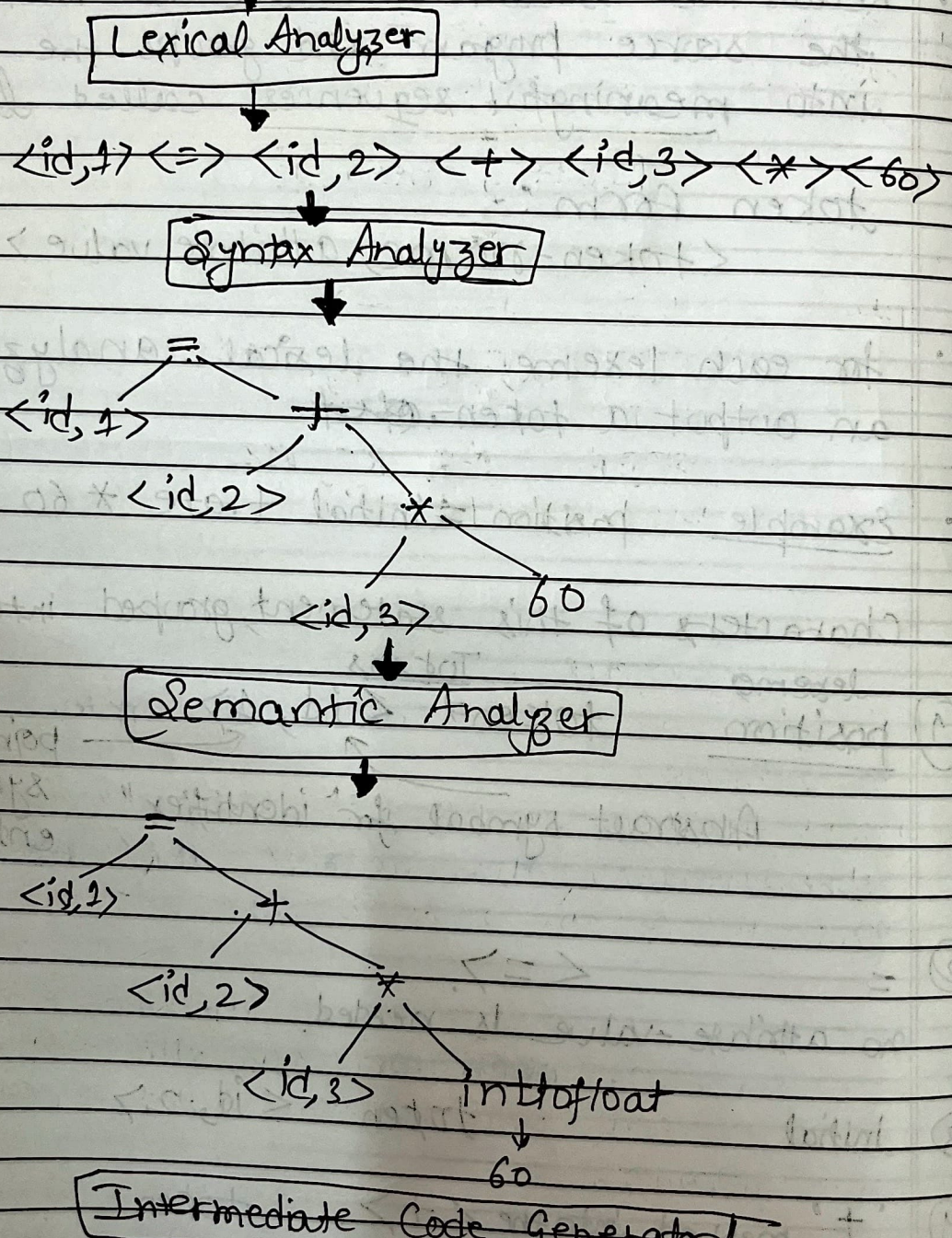
Abstract symbol for "identifier"

no attribute-value is needed.

Blank space will be discarded by LA.

Translation of an Assignment statement

position = initial + rate * 60



Three-Address Code

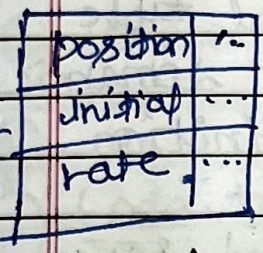
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

1	position
2	initial
3	rate

Symbol Table

(2) Syntax

- Recon
or Par
- Uses
represe
structu
- Syntax
represe
of th
the op
- Tree
shows
statem



Symbol Table

Code optimizer

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generator

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

(2) Syntax Analysis :

- Second phase of Compiler is Syntax Analysis or Parsing.
- Uses tokens to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- Syntax Tree, in which each interior node represents an operation and the children of the node represent the arguments of the operation.
- Tree shows the order in which the operations in statement are to be performed.

(3) Semantic Analysis:

Semantic Analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.

- Also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate code generation.

→ Typechecking: Compiler checks that each operator has matching operands.

(4) Intermediate Code generation:

Many compiler generate an explicit low-level or machine-like intermediate representation after syntax and semantic analysis, which we can think of as a program for an abstract m/c.

- It should have 2 properties:-

(1) It should be easy to produce.
(2) It should be easy to translate into the target machine.

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

Noting points:- (1) &
(2) Inter
(3) The compiler
the value
(4)
(5) Code Optim
M/c Indep
intermediate
result.
- Better mean
power.

(6) Code gener
- I/p is an int
and maps
- If the targ
memory loc
used by the
- Crucial aspe
variables.
LD
MU
LDF
AD
STF
- first operand
- "F" denote
- "#" signifies

Noting points:- ① Each instruction has atmost one operator on R.S
② Instructions fix the order.
③ The compiler must generate a temporary name to hold the value computed by a three-address instruction.

(5) Code optimization:
M/c Independent Code-optimization, improve the intermediate code so that better target code will result.
- Better means faster, shorter than consumes less power.

```
t1 = id3 * 60.0
id1 = id2 + t1
```

(6) Code generation: The code generator takes an
- I/p is an intermediate representation of source program and maps it into the target language.

- If the target lang. is m/c code, registers or memory locations are selected for each of the variables used by the program.

- Crucial aspect is assignment of registers to hold variables.

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

- first operand of each instruction specifies a destination.
- "F" denotes that it deals with floating-point numbers.
- "#" signifies that "60.0" is to treated as an constant.

⇨ Compiler-Construction Tools:

Most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler

Tools:- ① Parser Generators
Automatically produces syntax Analyzers from a grammatical description of a programming language. Example:- EQN, PIC

② Scanner generators
produces lexical analyzers from a regular expression description of the tokens of a language.
Ex:- LEX

③ Syntax-directed translation Engines.
Produce collections of routines for walking a parse tree and generating intermediate code.

④ Code-generator
Produces a code generator from a collection of rules for translating each operation of the intermediate lang. into the m/c lang. for a target m/c.

⑤ Data Flow analysis engines
- Facilitate the gathering of information about how values are transmitted from one part of a program to other part.

- Key po
⑥ Compil
provide
various

⇨ The R
- Main tas
character
lexemes
- Whenever
identifier
symbol

Source program → LA

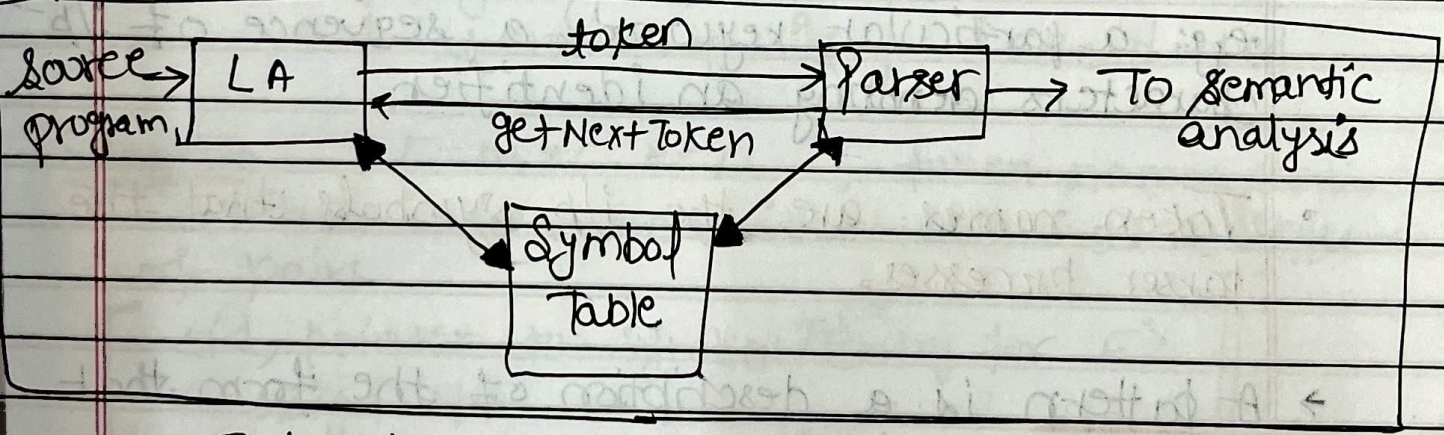
Intera
- It perform
① Stripping
tab)
② Correlating
with the
LA may ke
characters
error messa

- Key part of code optimization.

(6) Compiler-construction tool/kits
Provide an integrated set of routines for constructing various phases of a compiler.

⇨ The Role of Lexical Analyzer :

- Main task of lexical analyzer is to read the i/p characters of the source program, group them into lexemes & produce a sequence of tokens for each lexeme.
- Whenever LA discovers a lexemes constituting an identifier, it needs to enter that lexeme into the symbol table.



Interaction btw the LA and the Parser.

- It perform certain other tasks :-

(1) Stripping out comments and whitespace (blank, newline, tab)

(2) Correlating error messages generated by the compiler with the source program.
LA may keep track of the number of newline characters seen, to relate the line no. with each error message.

Sometimes LAs are divided into cascade of two processes :-

(1) Scanning : simple processes that do not require tokenization, such as deletion of comments and compaction of consecutive whitespace characters into one

(2) Lexical Analysis
- More complex portion.
- which produced tokens from the o/p of the Scanner

(1) Lexical Analysis versus Parsing

(2) ⇨ Tokens, Patterns and Lexemes :

→ A Token is a pair consisting of a token-name and an optional attribute value.
- Token name represents a kind of lexical unit e.g: a particular keyword, a sequence of i/p characters denoting an identifier.

- Token named are the i/p symbols that the parser processes.

→ A pattern is a description of the form that the lexemes of a token may take.
- In case of ~~token~~ keyword as a token, the pattern is just the sequence of the characters that form the keyword.

→ A lexeme is a sequence of characters in the source program that matched the pattern for a token and is identified by the LA as an instance of ~~an~~ that token.

Example:- printf ("Total = %d \n", score);

Here, both printf and score are lexemes matching the pattern for token id and "Total = %d \n" is a lexeme matching literal.

Tokens	Informal description	Sample Lexemes
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 6, 6.02e23
literal	anything but ", surrounded by "	"core dumped"

Example:- E = M * C **2, token names and attribute values are written below as a sequence of pairs

<id, pointer to symbol-table entry for E>
<assign_op>
<id, pointer to symbol-table entry for M>
<mult_op>
<id, pointer to symbol-table entry for C>
<exp_op>
<number, integer value 2>

③ ⇨ Attributes for Tokens

When more than one lexeme can match a pattern, the LA must provide the subsequent compiler phases additional information about the particular lexeme that matched.

The pattern for token number matches both 0 and 1.

LA returns <token name, attribute value> that describes the lexeme represented by token. Token-name influences parsing decisions.

In certain pairs, especially operators, punctuation and keywords, there is no need for an attribute value.

④ Lexical Errors

- Suppose a case arises in which LA is unable to proceed because none of the patterns for tokens matches any prefix of the remaining i/p.
Simplest recovery strategy is "panic mode" recovery. LA delete successive characters from the remaining i/p, until the analyzer can find a well-formed token at the begining of what i/p is left.

other act
① Delete
② Insert a
③ Replace a
④ Transpo

③ Input Buf

Tde cannot
an identifie
is not a
part of th

- A single c
also be th
operator

- for this u
handled

① Buffer pa
Introduced
process a
alternately

Two pointe

① Pointer
marks t
whose ext

② Pointer for
Scans ah

Other actions are :-

1. Delete one character from the remaining i/p
2. Insert a missing " into the remaining i/p.
3. Replace a character by another
4. Transpose two adjacent characters.

The lexical - Analyzer Generator - Lex

- Lex or FLEX, allows to specify a lexical analyzer by specifying regular expressions to describe pattern for tokens.

- The I/p notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex Compiler.

- Lex Compiler transforms the i/p pattern into a transition diagram and generates code, in a file called `lex.yy.c` that simulates this transition diagram.

Finite Automata

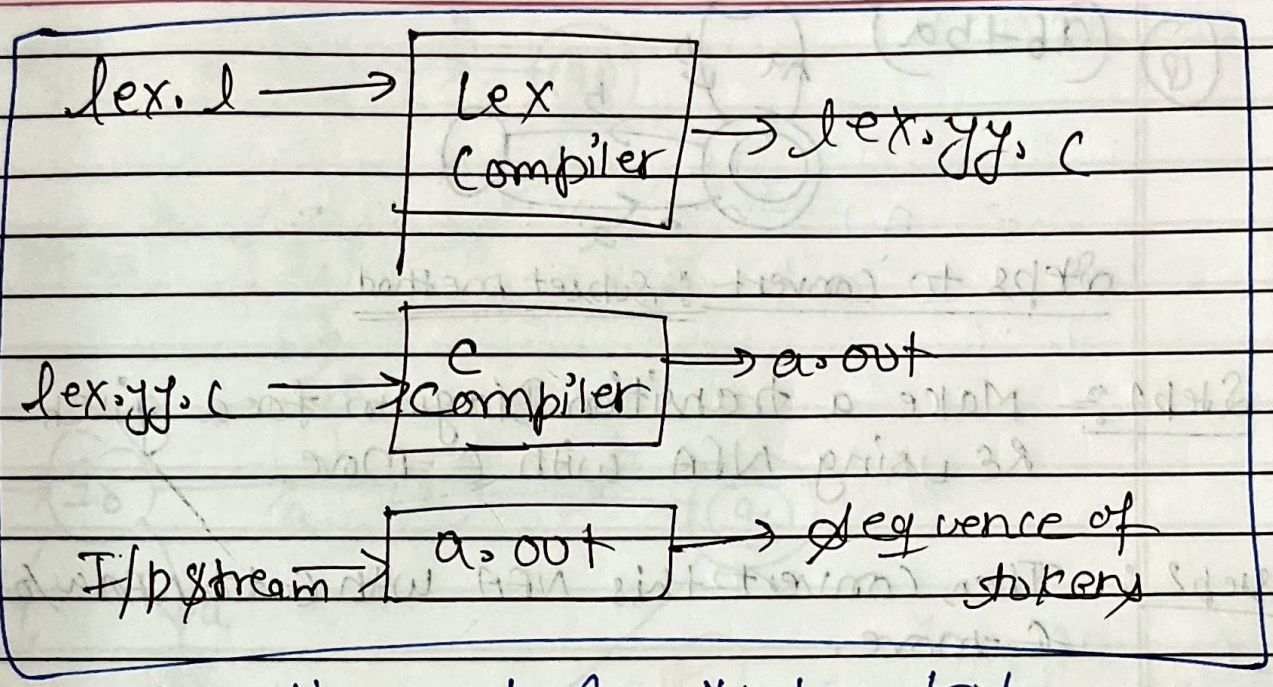
Lex turns its i/p program into a LA.

(1) Finite automata are recognizers, they simply "yes" or "no" about each posible string.

(2) Two types of FA :-

(a) (NFA) Non-Deterministic FA
- No restrictions on the labels of their edges.
- A symbol can label several edges out of the same state, and ϵ the empty string is a possible label.

(b) Deterministic finite Automata (DFA)
- For each state and for each symbol of its i/p alphabet exactly one edge with that symbol leaving that state.
- Both are capable of recognizing the same languages.



Creating a LA with lex tool

⇨ Regular Expression to FA

① ϕ → ○

② a → ○ —a→ ◎

③ a+b → ○ —a,b→ ◎

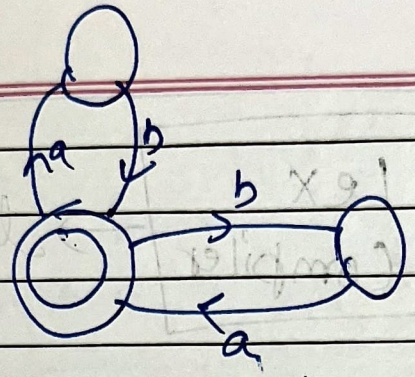
④ ab → ○ —a→ ○ —b→ ◎

⑤ a^* → ◎ with a self-loop a

⑥ ab^* → ○ —a→ ◎ with a self-loop b

⑦ $(ab)^*$ → ◎ —a→ ○ —b→ ◎

(a) $(ab+ba)^*$



Steps to convert :- Subset method

Step1 :- Make a transition Diagram for a given RE using NFA with ∈-move.

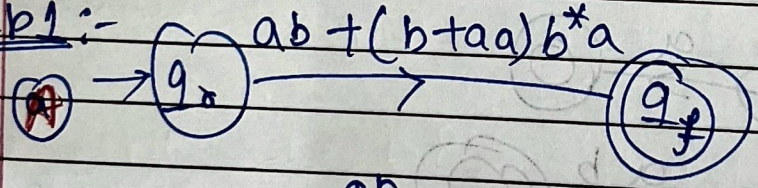
Step2 :- Then convert this NFA with ∈ to NFA w/o ∈-move.

Step3 :- Finally, convert the obtained NFA into DFA.

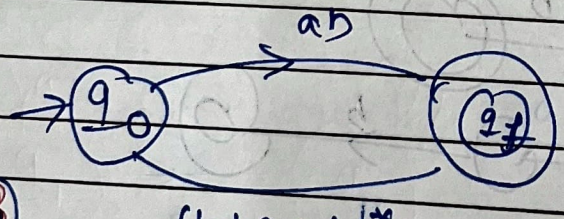
Ex:- RE = $(ab+(b+aa)b^*a)$ into FA.

Step1 :-

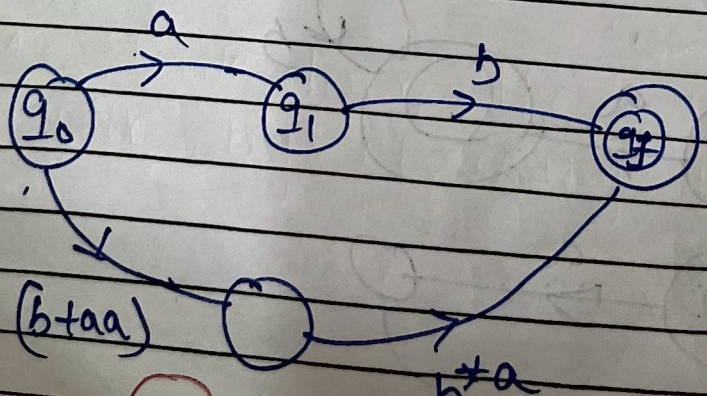
A

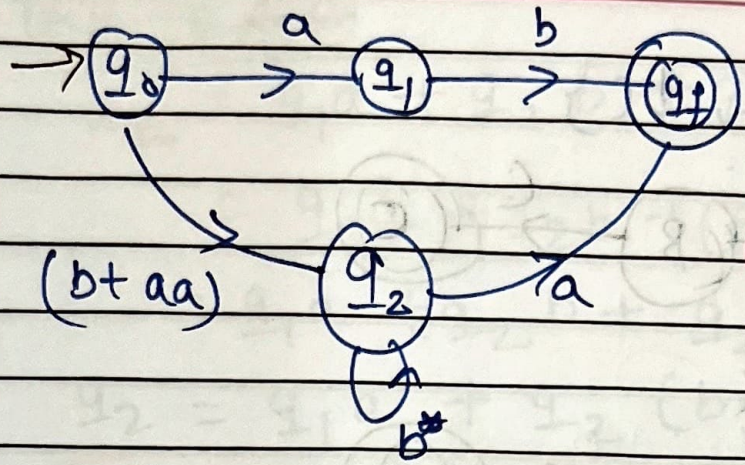


B

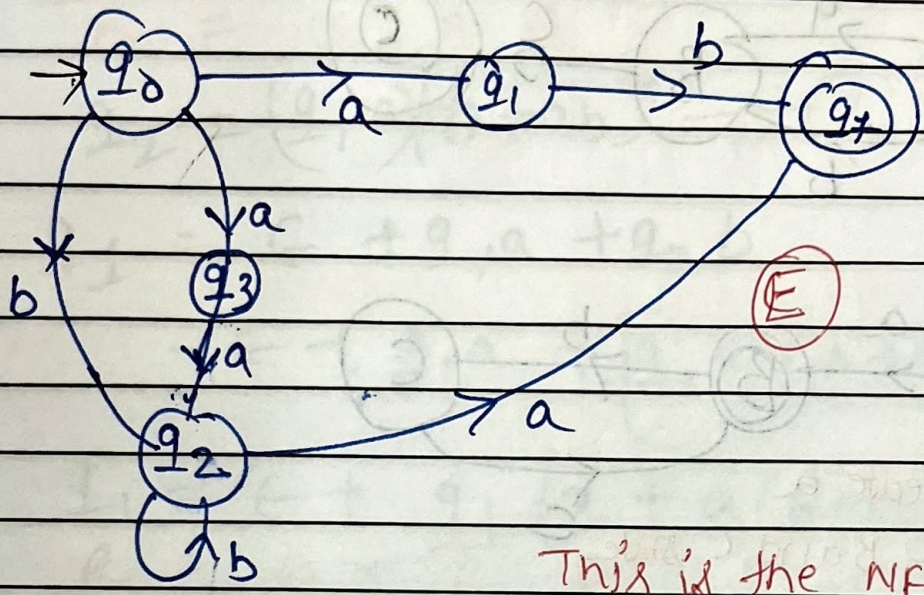


C





D



E

This is the NFA w/o ∈ move, now we will convert it into required DFA if ask to do so.

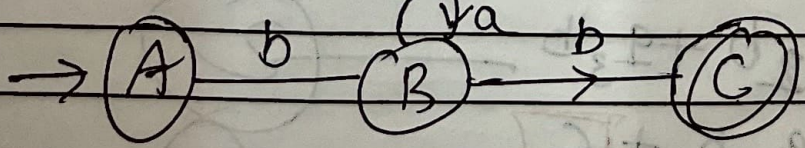
NFA → DFA

State	a	b
→q_0	$\{q_1, q_3\}$	q_2
q_1	ϕ	q_f
q_2	q_f	q_2
q_f	ϕ	ϕ
q_3	q_2	ϕ

State	a	b
→q_0	$\{q_1, q_3\}$	q_2
$\{q_1, q_3\}$	q_2	q_f
q_2	q_f	q_2
q_f	ϕ	ϕ
ϕ	ϕ	ϕ

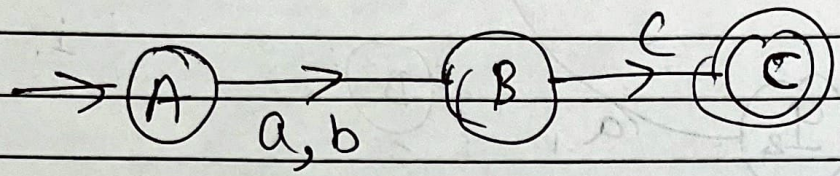
(2) RE: $b\,a^*\,b$

$L = \{baab, bb, bab, \ldots\}$

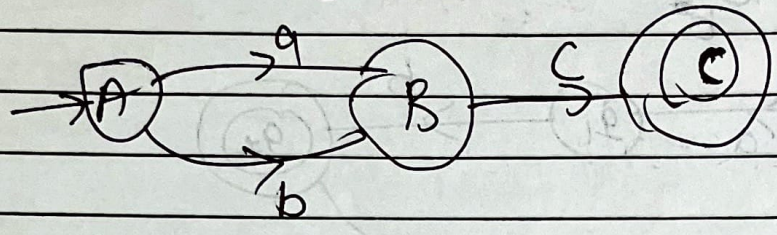


Q - $(a+b)c$

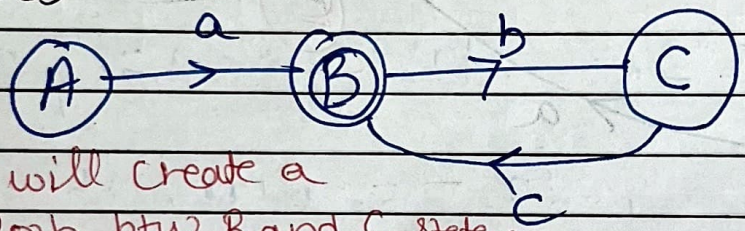
$L = \{ac, bc\}$



OR



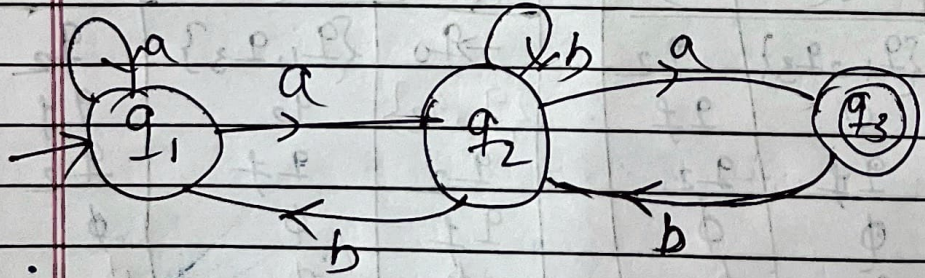
Q $a(bc)^*$



bc^* will create a self loop btw B and C states.

⇨ ~~Regular~~ Designing RE

NFA to RE using Arden's Theorem



Step 1:- Write down the equations for each state, by checking incoming edges.

$q_3 = q_2 a$ ——— (1)

$q_2 = q_1 a + q_2 b + q_3 b$ ——— (2)

$q_1 = q_1 a + q_2 b + \epsilon$ ——— (3)

~~DFA TO RE~~

$q_2 = q_1a + q_2b + q_3b$

$= q_1a + q_2b + (q_2 \cdot a)b$ [q_3 from eq 1]

$= q_1a + q_2b + q_2ab$

$q_2 = q_1a + q_2(b+ab)$

$R = Q + RP$ [$R = Q + RP$, $R = QP^*$]

$q_2 = (q_1a)(b+ab)^*$ — (4)

$q_1 = \in + q_1a + q_2b$

$= \in + q_1a + [q_1a(b+ab)^*]b$ — (by from eq 1.4)

$q_1 = \in + q_1[a + a(b+ab)^*]b$

$R = Q + RP$

$q_1 = \in + a + a$

$q_1 = (a + a(b+ab)^*b)^*$ — (5)

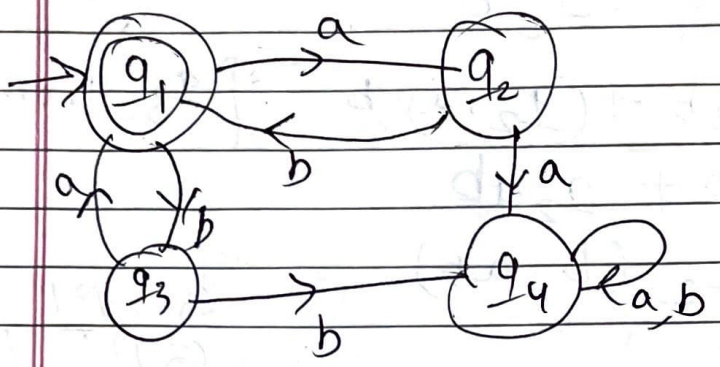
$q_3 = q_2a$

$q_3 = [(q_1a)(b+ab)^*]a$

$q_3 = [[a + a(b+ab^*)b]a \cdot (b+ab)^*]a$

RE

DFA to RE



R.E = q_1 / final state

$q_1 = \epsilon + q_2 b + q_3 a$ — (1)

$q_2 = q_1 a$ — (2)

$q_3 = q_1 b$ — (3)

$q_4 = q_2 a + q_3 b + q_4 a + q_4 b$ — (4)

→ Analyze and solve it,

$q_1 = \epsilon + q_2 b + q_3 a$

$q_1 = \epsilon + (q_1 a) b + (q_1 b) a$

$q_1 = \epsilon + q_1 ab + q_1 ba$

$q_1 = \epsilon + q_1 (ab + ba)$

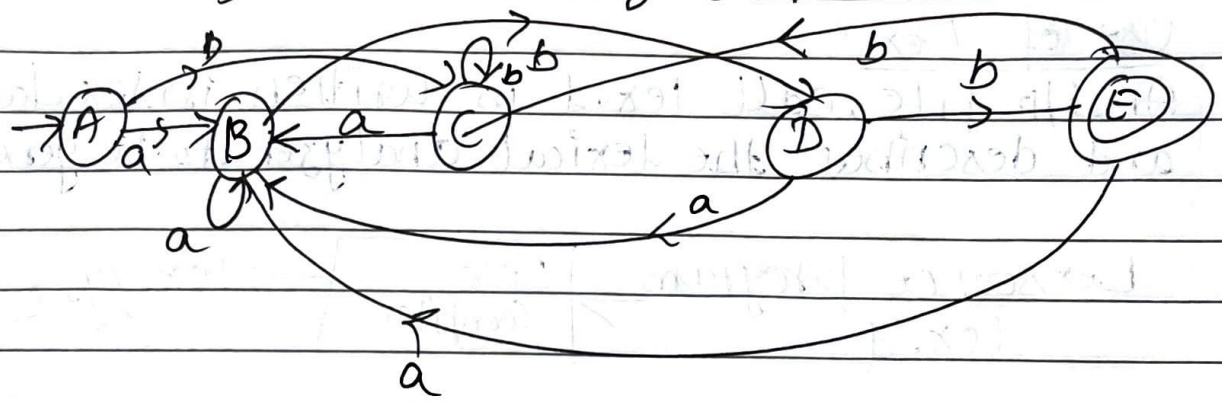
$R = Q + R P$

$q_1 = \epsilon + (ab + ba)^*$

$q_1 = (ab + ba)^*$

R.E = $q_1 = (ab + ba)^*$

Q Convert DFA to minimized DFA



o/p → {E} {A C} {B} {D}

Patterns for tokens

token ← pattern

digit → [0-9]

digits → digit+

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

letter → [A-Z a-z]

id → letter (letter | digit)*

if → if

then → then

else → else

relop → < | > | <= | >= | = | <>

ws → (blank | tab | newline)+

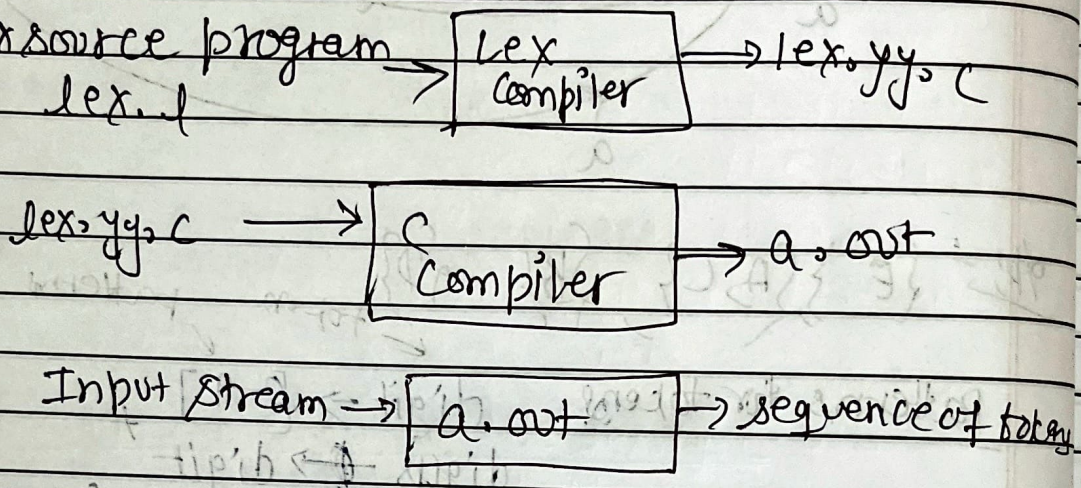
Tokens, their patterns and attribute values

Lexeme	Token Name	attribute Value
Any ws		
if	if	
then	then	
else	else	
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
>=	relop	GE
=	relop	EQ

The Lexical-Analyzer Generator Lex

① Use of Lex

An i/p file, call lex.l is written in lex language and describes the lexical analyzer to be generated.



Creating a lexical analyzer with Lex

- Lex Compiler transforms lex.l to a C program, in a file that is always named lex.yy.c
- later this file is compiled by C Compiler into a file called a.out as always.
- The C-compiler output is a working lexical analyzer that can take a stream of i/p char. and produce a stream of tokens.
- a.out is as a subroutine of the parser, it is a C function that returns an integer, which is a code for one of the possible token names.

- The attribute value, a numeric code, a pointer to the symbol table, or nothing is placed in a global variable yylval, shared btw the LA and parser

(2) Structure of Lex program

```
declarations          // declarations of variables, constants
%%
translation rules
%%
auxiliary functions
```

Translation rules each have the form:-

Pattern { Action }

Each pattern is a regular expression, may be use the regular definitions of the declaration section.

The actions are fragements of code, typically written in C.

The 3rd section holds additional functions are used in actions, these functions can be compiled seperately and loaded with the LA.

- LA created by Lex behaves with the parser as follows:
- When called by parser, the LA begins reading its remaining input, 1 character at a time.
- Until it finds the longest prefix of the i/p that matches one of the patterns Pi,

- It then executes the associated action Ai. Ai will return to the parser.
- But if it does not (because Pi describes comments or whitespace), then the LA proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser.
- LA returns a single-value, the token-name but uses the shared integer variable yylval to pass additional information about the lexeme found, if needed.

Directly copied to the file lex.yy.c and not treated as Regular definition

```
%{
/* definition of manifest constants
   LT, LE, EQ, NE, GT, GE, IF, THEN,
   ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter} ({letter}|{digit})*
number    {digit}+ (\. {digit}+)? (E [+-]? {digit}+)?

%%
```

declaration section

Pattern | Action → return token

{ws} | {/* no action and no return */}
if | {return (IF);}
then | {return (THEN);}
else
{id} | {yylval = (int) installID(); return(ID);}
{number} | {yylval = (int) installNum(); return(NUMBER);}
"<" | {yylval = LT; return (RELOP);}

%%

int installID() { /* function to install the lexeme, whose first character is pointed to by yytext, and whose length is yyleng, into the symbol table and return a pointer there to */
}

int installNum() { /* similar to installID, but puts numericals constants into a seperate table */
}

Anything in the auxiliary section is copied directly to file lex.yy.c, but may be used in actions.

Action taken when id is matched is threefold:

① function installID() is called to place the lexeme found in the symbol table.

② f" returns a pointer to the symbol table, which is placed in global variable yylval, f" is available two to variables of LA :-

(a) yytext :- pointer to the beginning of the lexeme Similar to beginptr

(b) yyleng :- length of the lexeme found.

③ Token name is returned to the parser.

Conflict Resolution in Lex

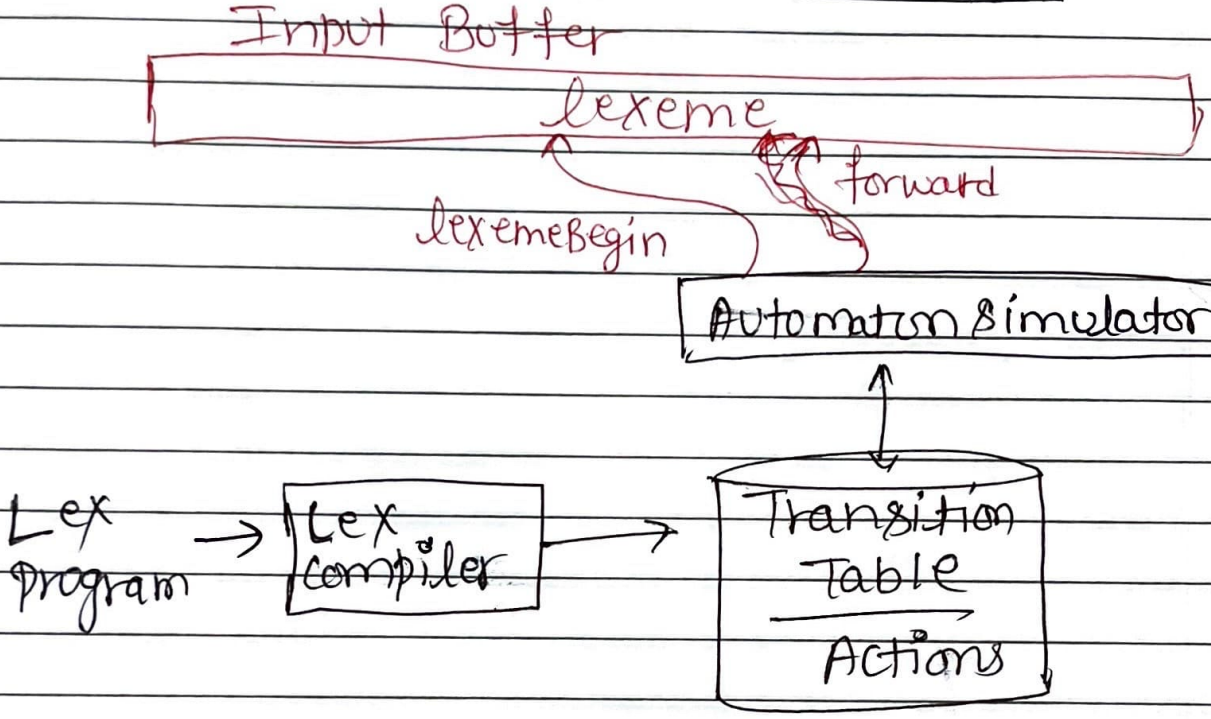
Rules that Lex used to decide a proper lexeme to select, when several prefixes of i/p match one or more patterns :-

① Always prefer a longer prefix.

② If the longest prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

Design of a Lexical Analyzer Generator

(1) Structure of the Generated Analyzer :



- A Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator.

The rest of the LA consists of components that are created from the Lex program by Lex itself. Components are :-

1. A transition table for the automaton.
2. Those f'ns that are passed directly through Lex to the output.
3. Actions from the i/p program, fragments of code to be invoked at appropriate time by to automaton simulator.