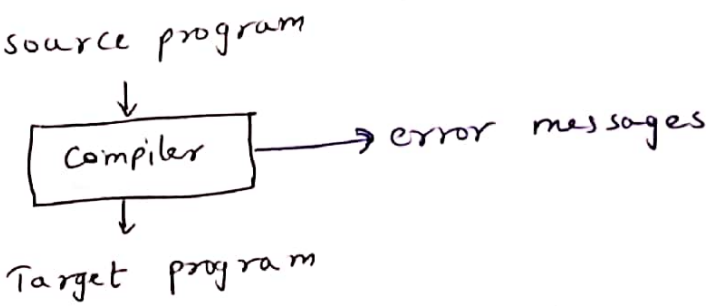
Unit - I

Language processors:

→ A compiler is a program that can read a program in one language -(source language) and translate it into an equivalent program in another Language - (Target Language)

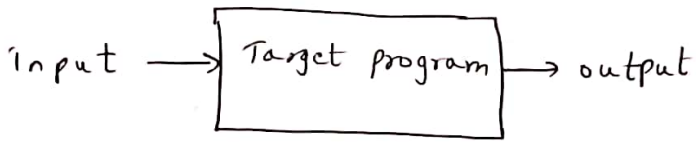
→ Important role of the Compiler is to report any errors in the source program that it detects during the translation process.



→ Generally, the source Language is high-level Language, such as C or C++, and Target Language is object Code called as Executable Code.

→ If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce

outputs.

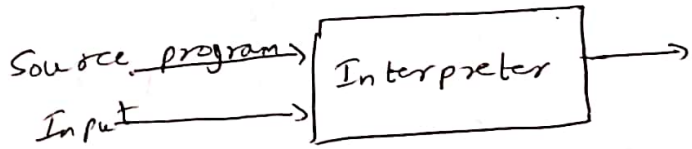


Running a Target program

→ An Interpreter is another common kind of Language processor.

→ Interpreter is also a Language Translator like Compiler.

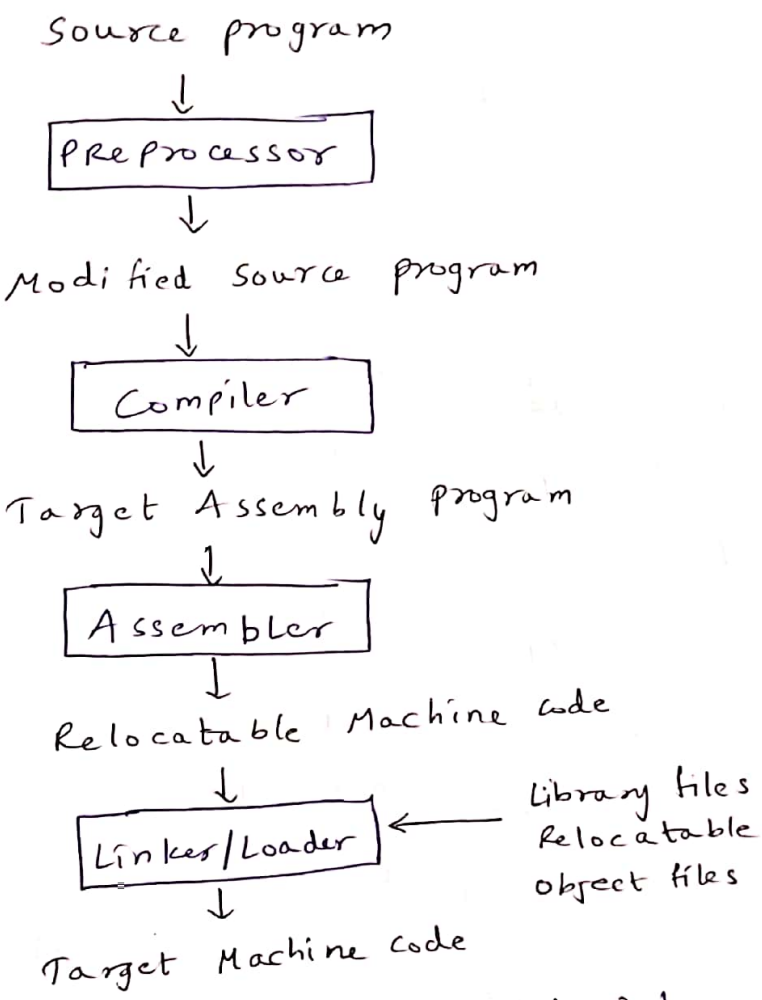
→ Interpreter directly executes the operations in the source program on inputs provided by User rather than producing a target program as a translation.



→ Interpreter is a common type of Language processor.

→ Interpreter executes the source program statement by statement and therefore it provides better error diagnostics in comparision with compiler.

A typical Language processing system



→ A source program can be divided into modules stored in separate files.

Preprocessor

→ It is a separate program that is called by the compiler before actual translation starts.

→ Main Task of preprocessor is to collect source programs.

→ Preprocessor can delete comments, include other files and perform Macro substitutions.

→ The output of preprocessor is a modified source program that is fed to compiler.

Compiler

→ It accepts modified source program, that is normally a program written in HLL as input and produces an assembly Language program (ALP) as output.

Assembler

→ It accepts assembly Language as input and produces relocatable machine code as output.

→ Assembly Language is a symbolic form of machine Language of the computer and is easy to translate.

→ Assembler is a translator for the Assembly Language of a particular computer.

Linker / Loaders.

→ A Linker collector code separately compiled

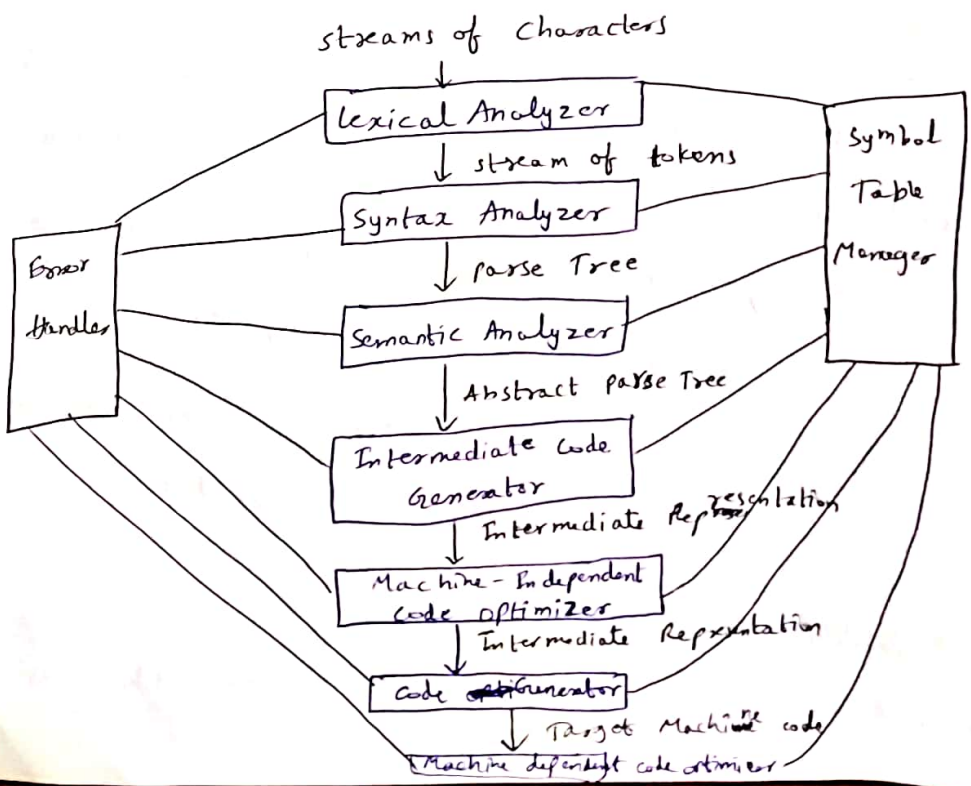
or) assembled in different object files into a file that is directly executable.

→ Linker resolves external memory addresses, where the code in one file may refer to a location in another file.

Loader is one which combines all the executable object files into memory for execution.

Note
→ Preprocessor, Assembler, Linkers & Loaders are also known as Cousins of Compiler.

Phases of a Compiler

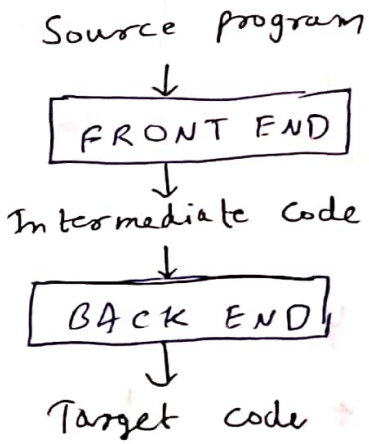


→ The compiler accepts the pre-processed file as input and translates it into an equivalent assembly Language file.

→ The process of translation of Input source into its target Assembly language file can be divided into two major phases:

1) Analysis phase: (also called as FRONT END of a Compiler)
It Consists of Lexical Analysis, Syntax Analysis, and Semantic Analysis phases.

2) Synthesis phase: (also called as BACK END of a Compiler)
It Consists of Intermediate code generator, code generator and code optimization phases.



(1) Lexical Analysis (or) Scanning (or) Scanner phase)

→ Lexical Analysis phase is also known as SCANNING phase.

→ The main task of lexical analysis is to read the stream of characters of the source program as input and groups the characters into meaningful sequences called LEXEMES. eg preprocessor directives, operators etc.

→ It is similar to finding words and punctuation in English text.

→ This phase accepts the source program and seperates it into logical groups called TOKENS.

Eg:-

LEXEME	TOKEN
main	MAIN
(	Left parenthesis
)	Right parenthesis
a	(Identifier, 100)
=	ASSIGN
}	Right bracket
*	Multiplication sign

→ A token for a variable is an ordered pair (id, address) where 'id' = unique integer code for the variables and address = pointer to the symbol table where variable is stored.

→ All identifiers will be entered into the symbol table.

→ Functions of Lexical Analyzer

1) Eliminates comments and removes extra white space (ie tab, space, new line, carriage return etc)

2) Keeps track of line numbers and column numbers and passes them as parameters to the other phases to enable error-reporting to the user.

3) Converts sequence of characters into Tokens.

→ For each Lexeme, the lexical analyzer produces as output a token of the form

< token-name, attribute-value >

token-name is an abstract symbol that is used during syntax Analysis

attribute-value points to an entry in the symbol for this token.

Note :

→ Regular Expressions are used to describe Tokens (Lexical Constructs).

→ Identification of tokens is done by Deterministic Finite Automata (DFA).

→ The set of tokens of a Language is represented by a large Regular Expression (RE).

→ This RE is fed to the lexical analyzer generator such as Lex, Flex or ML-Lex.

→ A DFA is created by the Lexical Analyzer Generator.

→ For eg, suppose a source program contains the assignment statement.

position = initial + rate * 60

1) Position is a lexeme that would be mapped into a token <id, 1>, where id is an abstract symbol standing for identifier and 1 points to the symbol-table entry for position.

→ Blank

by

2) The assignment symbol '=' is a lexeme that is mapped into the token ⟺.

3) initial is a lexeme that is mapped into the token <id,2>, where 2 points to symbol-table entry for initial.

4) '+' is a lexeme that is mapped into the token <+>.

5) rate is a lexeme that is mapped into the token <id,3>, where 3 points symbol-table entry for rate.

6) '*' is a lexeme that is mapped into the token <*>.

7) 60 is a lexeme that is mapped into the token <60>

→ Blanks seperating the lexemes would be discarded by the Lexical analyzer.

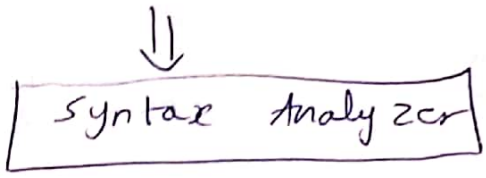
<id,1> ⟺ <id,2><+> <id,3> <*> <60>

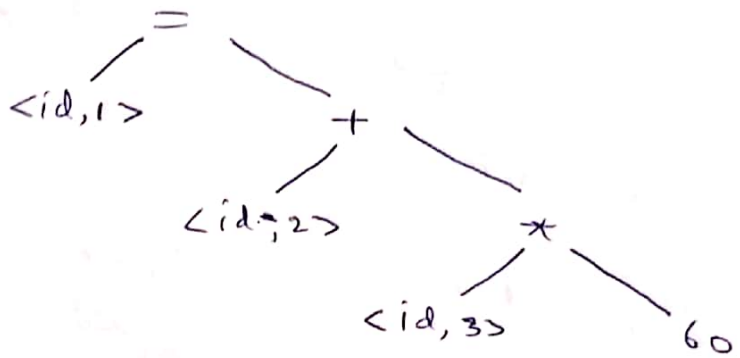
Syntax Analysis

→ The second phase of compiler is syntax analy
(or) parsing. The parser uses the first compone
of the tokens produced by lexical analyzer to
create a tree-like intermediate representation tha
depicts the grammatical structure of token strea

→ A typical representation is a syntax tree in
which each interior node represents an operati
and the children of the node represent
arguments of the operation.

→ <id,1> <id,2> <+> <id,3> <*> <60>





Syntax tree

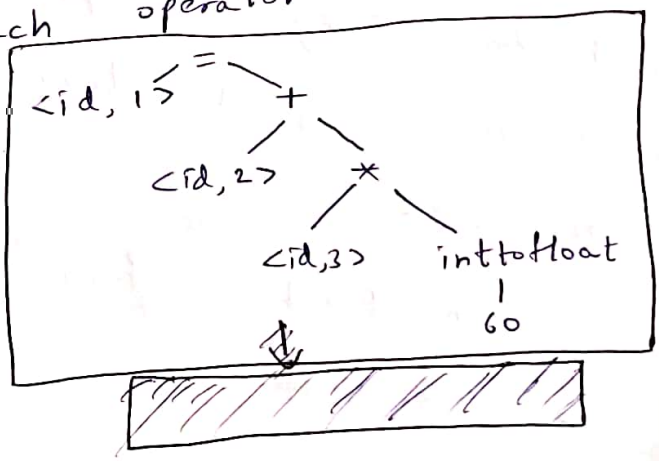
→ Syntax Analysis is similar to constructing
sentences from words and punctuation.

Semantic Analysis

→ The 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.

→ It also gathers type information and saves it in either the syntax tree or symbol tables for subsequent use during intermediate code generation.

→ Important task of semantic analysis is type checking, where the compiler checks that each operator has matching operands.



o/p after semantic Analysis.

Intermediate Code Generation

→ In process of translating a source program into target program, a compiler may construct one or more intermediate representations.

→ Syntax trees are a form of intermediate representation used during syntax & semantic Analysis.

→ After syntax and semantic Analysis of the source program, many compilers generate an explicit low-level or machine like intermediate representation.

→ This Intermediate representation ~~has~~ should have two important properties

(i) It should be easy to produce.

(ii) It should be easy to translate into target machine.

→ we use an intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.

output of Intermediate Code Generation

$t_1 = \text{inttofloat}(60)$

$t_2 = id3 * t_1$

$t_3 = id_2 * t_2$

$id_1 = t_3$

→ Each three address assignment instruction has at most one operator on the right side.

→ These instructions fix the order in which operations are to be done (multiplication precedes addition in source program).

→ Compiler must generate a temporary name to hold the value computed by a three address instruction.

Code optimization

→ The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.

→ A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code.

→ The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the inttofloat operation can be eliminated by replacing integer 60 by floating point 60.0.

→ t_3 is used only once to transmit its value to id_1.

output of Code optimization

$$t_1 = id_3 * 60.0$$

$$id_1 = id_2 + t_1$$

→ "optimizing compilers" spend significant amount of time on this phase.

Code Generation

→ The Code generator takes as input an Intermediate representation of the source program and maps it into target language.

→ If target language is machine code, registers (or) memory locations are selected for each of the variables used by the program.

→ Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

→ Important aspect of code generation is Judicious assignment of registers to hold variables.

output of Code generation

```
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.

→ The F in each instruction tells us that it deals with floating-point numbers.

→ The code loads the contents of address id_3 into register R_2, then multiplies it with floating point constant 60.0.

→ The # signifies that 60.0 is to be

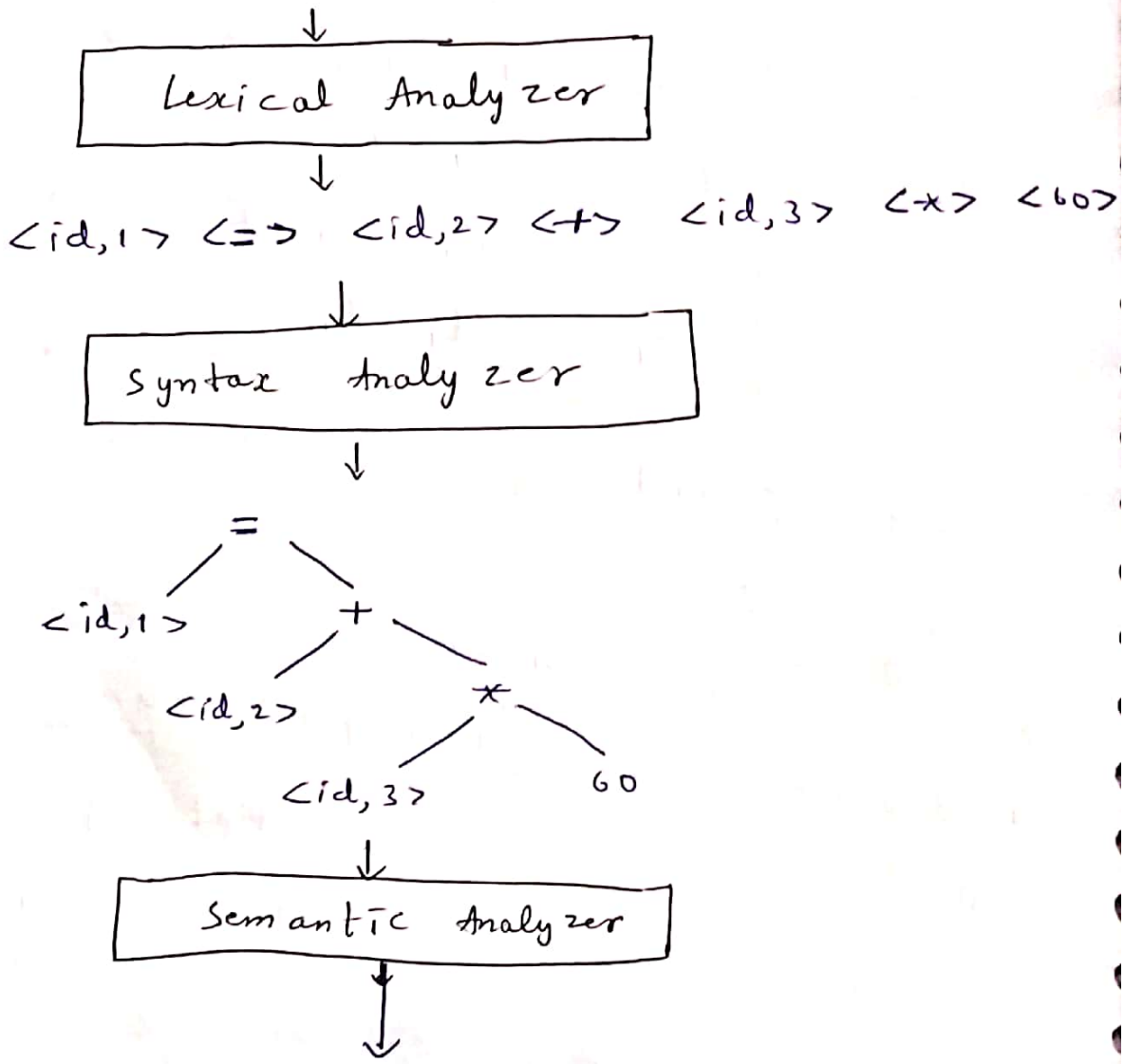
treated as immediate constant.

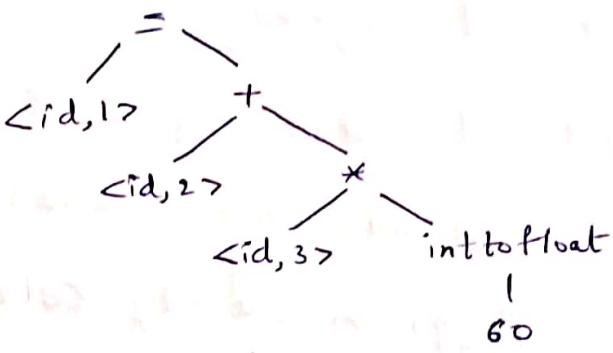
→ The third instruction moves id_2 into register R_1 and the fourth adds to it the value previously computed in register R_2.

→ Finally, the value in Register R_1 is stored into address of id_1.

Translation of an assignment statement

position = initial + rate * 60





↓

Intermediate Code Generator

↓

$t_1 = \text{int to float}(60)$

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

$id_1 = t_3$

↓

Code optimizer

↓

$t_4 = id_3 * 60.0$

$id_1 = id_2 + t_1$

↓

Code Generator

↓

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

Symbol Table Management

→ An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.

→ These attributes may provide information about the storage allocated for a name, its type, its scope and in case of procedure names, such things as the number and types of its arguments, the method of passing each argument and the type returned.

→ The symbol Table is a data structure containing a record for each variable name, with fields for the attributes of the name.

Compiler Construction Tools

1) Parser generators - that automatically produce syntax analyzers from a grammatical description of a programming Language.

Scanner generators that produce lexical analysers from regular-expression description of the tokens of a programming language.

Syntax-Directed translation Engines that produce collections of routines for walking a parse tree and generating intermediate code.

Code generator generators that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

Data-Flow Analysis engines that facilitate the gathering of information about how values are transmitted from one part of program to each other part. Data-Flow analysis is a key part of code optimization

Compiler-construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

The Science of Building a Compiler

→ A compiler must accept all source programs that conform to the specification of language.

→ Any transformation performed by compiler while translating a source program must preserve the meaning of the program being compiled.

→ Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile.

Modeling in Compiler Design and Implementation

→ The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms.

→ Some of the most fundamental models are finite state Machines and regular expressions.

These models are useful for describing the lexical units of program (keywords, identifiers etc) and for describing the algorithms used by compiler to recognize those units.

→ Most fundamental models are context-free grammars, used to describe the syntatic structure of programming languages such as nesting of parentheses (or) control constructs.

→ Trees are an important model for representing the structure of programs and their translation into object code.

The Science of Code Optimization

→ The word "optimization" in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than obvious code.

→ Optimization of code that a compiler performs has become both more important and more complex.

→ It is more important because massively parallel computers require substantial optimization (or) their performance suffers by orders of magnitude.

→ It is more complex because processor architectures have become more complex, yeilding more oppurtunities to improve the way code executes.

→ Compiler optimizations must meet the following design objectives.

- The optimization must be correct, that is, preserve the meaning of compiled program.
- The optimization must improve the performance of many programs.
- The compilition time must be kept reasonable.
- The engineering effort required must be manageable.

Programming Language Basics

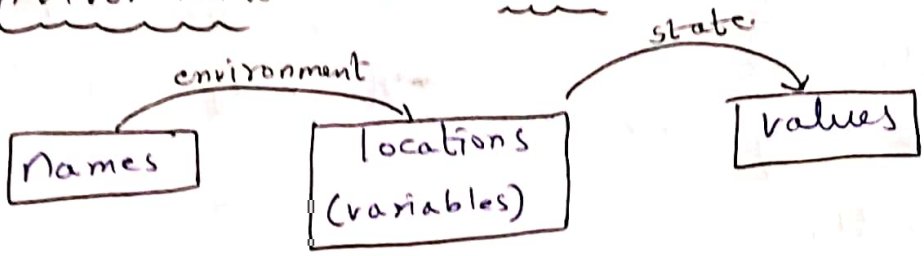
1) static/Dynamic Distinction

→ If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses static policy or that the issue can be

decided at compile time.

→ A policy that only allows a decision to be made when we execute the program is said to be dynamic policy or to require a decision at run time.

2) Environments and states



→ The Environment is a mapping from names to locations in the store.

→ The state is a mapping from locations in store to their values.

(Mapping L-values to R-values)

3) static versus dynamic binding of names to locations

→ Most of names to locations is dynamic.

→ The global declarations can be given a location in the store once and for all; as compilers generates object code.

Static versus dynamic binding of locations to values

→ The binding of locations to values is generally dynamic as well, since we cannot tell the value in the location until we run the program.

→ Declared constants are an exception as binding for macro to body is done statically.

Identifier - An Identifier is a string of characters (letters or digits) that refers to an entity, such as a data object, a procedure, a class or a type.

→ All identifiers are names, but all names are not identifiers

Eg:- x.y (x structure, y member) is name but not identifier (x & y are names).

Variable - It refers to particular location of store

→ It is common for the same identifier to be declared more than once, each such declaration introduces new variable.

→ Even if each identifier is declared just once, an identifier local to a recursive procedure will refer to different locations of the store at different times.

3) static scope and Block structure

→ The 'c' static policy is as follows

i) A cprogram consists of sequence of top-level declarations of variables and functions.

(ii) Functions may have variable declarations within them. The scope of each such declaration is restricted to the function in which it appears.

(iii) The scope of top-level declaration of name x consists of the entire program that follows, with the exception of those statements that lie within a function that has a declaration of x.

→ In 'c' syntax of blocks is given by

(i) one type of statement is a block. Blocks can appear anywhere that ~~order~~ other types of statements, such as assignment statements, can appear.

(ii) A block is a sequence of declarations followed by a sequence of statements, all surrounded by braces ({ })

→ This syntax allows blocks to be nested inside each other. This nesting property is referred to as Block structure.

Procedure – sub program can be referred to as "Procedure"

In 'C' Language, we shall refer them as "functions".

In Java, we shall term them as "methods"

→ A function generally returns a value but a procedure will not return a value.

→ object oriented Languages like Java & C++ use the term "methods". These can behave like either functions or procedures, but are associated with a particular class

(4) Explicit Access Control

→ classes and structures introduce a new scope for their members.

→ Through the use of keywords like public, private and protected, objects oriented Languages

such as C++ or) Java provide explicit control access to member names in a superclass.

→ These keywords support encapsulation by restricting access.

→ The private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any friend class (in C++).

→ The protected names are accessible to sub classes.

→ The public names are accessible from outside the class.

⑤ Dynamic Scope

→ The term dynamic scope, however, usually refers to the following policy: a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration.

→ Two dynamic policies: Macro expansion in C Method resolution in object-oriented programming.

→ Dynamic Scope resolution is also essential for polymorphic procedures, those have two (or) more definitions for the same name, depending on the types of the arguments.

Eg:-
```
#define a (x+1)
int x=2;
void b() { int x=1; printf("%d\n",a);}
void c() { printf("%d\n", a); }
void main() { b(); c();}
```

Declaration tell us about the types of things.

Definitions tell us about their values.

Eg:- int i; (declaration of i)
i=1; (definition of i)

→ In C++, a method is declared in a class definition, by giving the types of arguments and result of the method. (Signature)

→ The method is then defined (i.e) the code for executing the method is given in another place

⑥ Parameter Passing Mechanisms

→ Call by value

→ call by reference

⑦ Aliasing

→ In Java, where references to objects are passed by value. It is possible that two formal parameters can refer to same location. such variables are said to be aliases of one another.

→ Any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other.

Eg:- a-array belonging to procedure P P calls another procedure q(x,y) with a call q(a,a) (x & y have become aliases of each other). In q if x[10]=2 then y[10] also become 2.

Role of Lexical Analysis

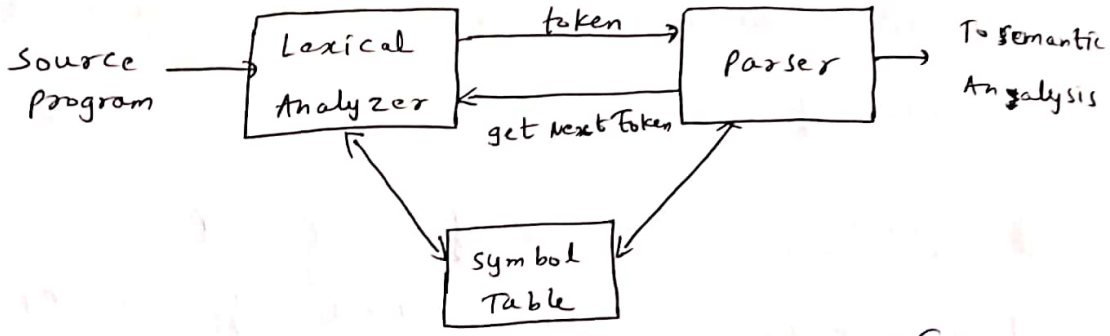
(1) Role of Lexical Analyzer

→ Main task of lexical analyzer is to read the input characters of the source program, group them into lexemes and produce as output a sequence of tokens for each lexeme in the source program.

→ The stream of tokens is sent to the parser for syntax analysis.

→ When lexical analyzer discovers a lexeme constituting a identifier, it needs to enter that lexeme into the symbol table.

→ In some cases, information regarding the kind of identifier may be read from the symbol table by lexical analyzer to assist it in determining the proper token it must pass to the parser.



→ The call, suggested by the getNextToken command, causes the Lexical Analyzer to read characters from its input until it can identify the next lexeme and produce for it the next Token, which it returns to the parser

→ Apart from identification of lexemes, the other task is stripping out comments and whitespace (blank, newline, tab and other characters)

→ Another task is correlating error messages generated by the compiler with the source program.

→ Lexical Analyzer keeps track of no. of new line characters seen, so it can associate a line number with each error message.

→ It also performs expansion of macros

→ Lexical Analyzers are divided into a cascade of two processes.

(a) Scanning consists of simple processes that donot require tokenization of the input, such as deletion of Comments and compaction of Consecutive white spaces into one.

(b) Lexical analysis proper is the more complex portion, where Scanner produces sequence of tokens as output.

1.1 Lexical Analysis Versus Parsing

→ simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify the task

Eg:- (Removing whitespaces & comments and sending to parser rather than sending them without removing)

→ Compiler efficiency is improved. A seperate lexical Analyzer allows us to apply

specialized techniques that serve only the lexical task, not job of parsing.

specialized buffering Techniques for reading input characters can speed up the compiler significantly.

→ Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical Analyzer.

1.2 Tokens, Lexemes & Patterns

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

→ The token name is an abstract symbol representing a kind of lexical unit (eg keyword, identifier)

→ we will often refer to token by token name

patterns - A pattern is a description of the form that the lexemes of a token may take.

→ In case of keyword as token, the pattern is just the sequence of characters that form keyword.

→ For identifiers, the pattern is a more complex

structure that is matched by many strings.

Lexeme - A lexeme is a sequence of characters in the source program that matches the pattern for a token.

→ It is identified by the lexical Analyzer as an instance of that token.

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

→ printf and score are lexemes matching the pattern for token id

→ "Total = %d\n" is a lexeme matching literal.

Examples of tokens

Token	Informal Description	Sample Lexemes
if	characters i, f	if
else	characters e, l, s, e	else
comparision	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters & digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but surrounded by " "s	"core dumped"

→ one of the Token for each keyword. The pattern for a keyword is the same as the keyword itself.

→ Tokens for the operators, either individually (or) in classes such as token comparision

→ One token representing all identifiers.

→ one or more tokens representing constants, such as numbers and literal strings.

→ Tokens for each punctuation symbol, such as left and right parantheses, comma & semicolon.

1-3

Attributes for Tokens

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

→ The token name influences parsing decisions, while attribute value influences translation of tokens after the parse.

→ The information about an identifier (lexeme, type, location at which it is found first) is kept

In the Symbol Table.

→ The most appropriate attribute value for an identifier is a pointer to symbol Table entry for that identifier

Eg: Token names and associated Attributes

$E = M * C * * 2$

<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>

→ For operators, punctuation and keywords there is no need for an attribute value.

1.4 Lexical Errors

→ It is hard for a lexical analyzer to tell, without the aid of other components that there is a source-code error.

Eg:- If fi string is encountered for first time in C-program

```
fi (a == f(x)) ....
```

→ A lexical analyzer cannot tell whether fi is a misspelling of the keyword if (or) an undeclared function identifier.

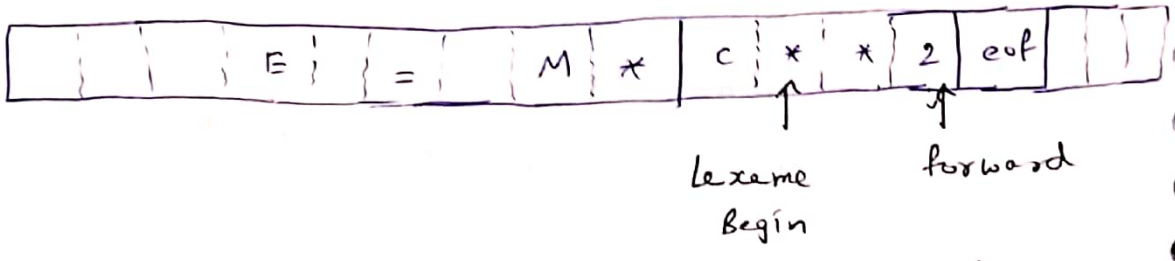
→ Since fi is a valid lexeme for token id, the lexical Analyzer must return the token id to the parser and let some other phase of compiler handle the error.

(2) Input Buffering

(2.1) Buffer pairs

→ Because of amount of time taken to process characters and large no. of characters that must be processed during the compilation of large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.

→ An Important scheme involves two buffers that are alternately reloaded.



using a pair of input buffers

→ Each buffer is of same size N, and N is usually the size of disk block, eg 4096 bytes

→ Using one system read command we can read N characters into a buffer, rather than using one system call per character.

→ If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of source file and is different from any possible character of source program.

→ Two pointers to input are maintained:

1) Pointer: lexeme Begin, marks the beginning

of current lexeme, whose extent we attempting to determine

2) Pointer forward scans ahead until a pattern match is found.

→ Once the next lexeme is determined, forward is set to the character at its right end.

→ Then after the lexeme is recorded as an attribute value of a token returned to the parser, lexeme begin is set to next character immediately after the lexeme is just found.

→ Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of newly loaded buffer.

~~Fetch~~

(2.2) Sentinels

→ For each character we read, we make two tests

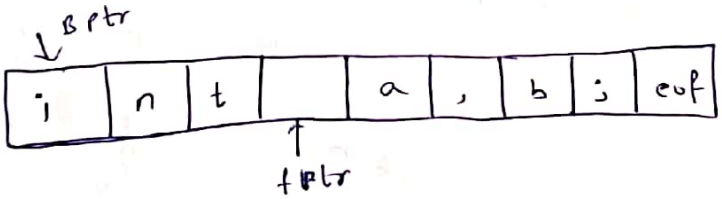
(i) For end of the buffer

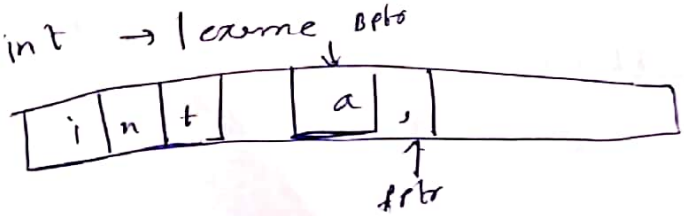
(ii) To determine what character is read.

→ we can combine the bufferend test with the test for current character if we extend each buffer to hold a sentinel character at the end.

→ The sentinel is a special character that cannot be part of source program, and a natural choice is character eof.

Eg: Input buffer





a → lexeme

① Single buffer → buffer (Forward pointer) will point to starting again (reading some data again)

② Double buffer

③ Specification of Tokens

→ Regular Expressions are an important notation for specifying lexeme patterns.

→ They (RE) are very effective in specifying those types of patterns that are needed for Tokens

3.1 Strings and Languages

→ An Alphabet is any finite set of symbols.

→ symbols are letters, digits, & punctuation.

→ Set {0,1} is a binary alphabet.

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

→ Length of string |s| (no of characters in the given string).

→ Empty string (ε), length of empty string is (zero).

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

3.2 Operations on Languages

→ In lexical Analysis, the most important operations on languages are union, concatenation and closure

→ L^*, (Kleen closure) is the set of strings you get by concatenating L zero (or) more times.

$$L^i = L^{i-1} L$$

Operation	Definition and Notation
Union of L and M	$L \cup M = \{ s \mid s \text{ in } L \text{ (or) } s \text{ is in } M \}$
Concatenation of L and M	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Posible closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

3.3 Regular Expressions

→ The regular expressions are built recursively out of small regular expressions, using the following rules :-

Rule 1: ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is empty string.

Rule 2: If a is a symbol in Σ, then a is a regular expression, and $L(a) = \{a\}$, that is the language with one string of length one

(1) $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$

(2) $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$

(3) $(r)^*$ is a regular expression denoting $(L(r))^*$

(4) (r) is a regular expression denoting $L(r)$

Eg:

R.E	Language
$a\|b$	$\{a, b\}$
$(a\|b)(a\|b)$	$\{aa, ab, ba, bb\}$
a^*	$\{\epsilon, a, aa, aaa \ldots\}$
$(a\|b)^*$	$\{\epsilon, a, b, aa, ab, ba, bb, aaa \ldots\}$
$a\|a^*b$	$\{a, b, aab, ab, aaab, \ldots\}$

(3.4) Regular Definitions

→ If Σ is an alphabet of basic symbols, then a Regular definition is a sequence of definitions of the form.

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\cdots$$
$$\cdots$$
$$d_n \rightarrow r_n$$

where

1) Each d_i is a new symbol, not in Σ and not the same as any other of the d's

2) Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2 \cdots d_{i-1}\}$

Eg:

$$\text{letter\_} \rightarrow A \mid B \mid \cdots \mid z \mid a \mid b \mid \cdots \mid z \mid \_$$

$$\text{digit} \rightarrow 0 \mid 1 \mid \cdots \mid 9$$

$$\text{id} \rightarrow \text{letter\_}(\text{letter\_} \mid \text{digit})^*$$

(Identifiers)

(3.5) Extentions of Regular Expressions

(1) One or more instances: (Positive closure)

+ r^+ denotes the Language $(L(r))^+$

$$r^* = r^+ | \epsilon \qquad r^+ = rr^* = r^*r$$

(2) Zero (or) one instance

? $r?$ is equivalent to $r | \epsilon$

(3) Character classes

→ A regular expression $a_1|a_2|\cdots a_n$ where the a_i's are each symbols of the alphabet, can be replaced by shorthand $[a_1 a_2 \cdots a_n]$

Eg: $[abc]$ $a|b|c$

$[a-z]$ $a|b|\cdots|z$

(4) Recognition of Tokens

Eg: Grammar for branching statements.

PASCAL

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | ∈
expr → term relop term
     | term
term → id/number
```

→ For relop, we use the comparision operators of Languages like Pascal or SQL, where = is "equals" and <> is "not equals"

→ The Terminals of the grammar, which are if, then, else, relop, id and number, are names of the tokens

→ The patterns for these tokens are described using regular definitions

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 → < | > | <= | >= | = | <>

→ For this Language, lexical analyzer will recognize the keywords if, then and else, as well as lexemes that match the patterns for relop, id and number.

ws → (blank | tab | newline)+

→ Token (ws) is different from the other tokens in that, when we recognize it, we donot return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace.

Lexemes	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	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Tokens, patterns and attribute values.

Transition Diagrams

→ As an intermediate step in the construction of a lexical Analyzer, we first convert patterns into stylized flowcharts called "Transition Diagrams"

→ Transition Diagrams have a collection of nodes (or) circles, called states.

→ Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

→ Edges are directed from one state of the Transition diagram to another.

→ Each edge is labeled by a symbol (or) set of symbols.

→ Some important Conventions about Transition Diagrams are:

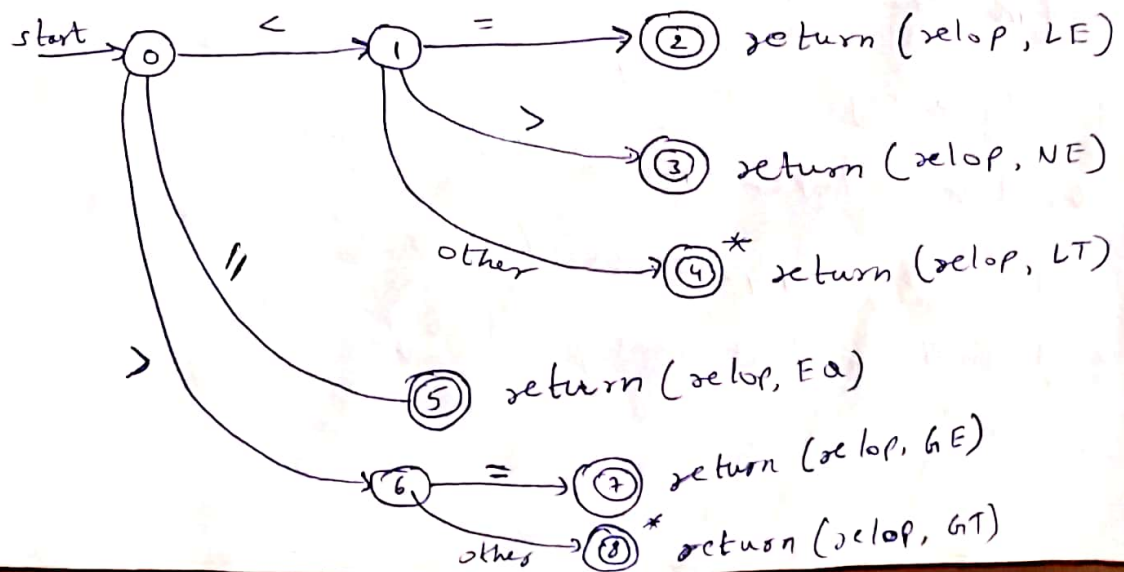
(i) Certain states are said to be accepting or final. These states indicate that a lexeme has been found, although lexeme may not consist of all positions between the lexeme Begin and forward pointers

→ We always indicate an accepting state by a double circle, and if there is an action to be taken - typically returning a token and an attribute value to the parser - we shall attach that action to the accepting state.

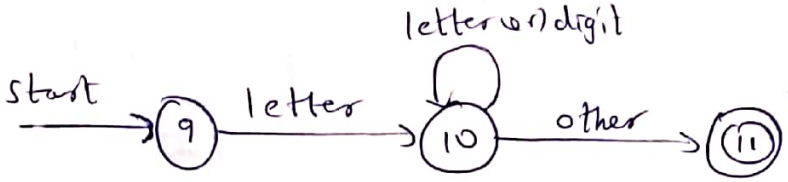
(ii) If it is necessary to retract the forward pointer one position (i.e lexeme does not include the symbol that got us to the accepting state), we shall include '*' near that accepting state.

(iii) One state is designated the start state or initial state, it is indicated by an edge labeled "start".

→ The Transition diagram always begins in the start state before any input symbols have been read.



Recognition of Reserved words and Identifiers



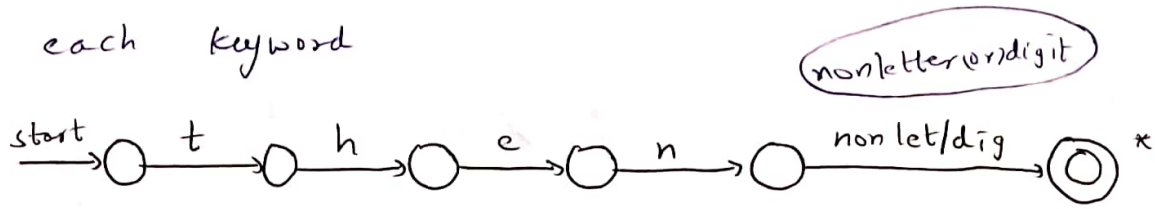
→ There are two ways to handle reserved words that look like identifiers:

(i) Install the reserved words in symbol Table initially. A field of the symbol-Table entry indicates that these strings are never ordinary identifiers, and tells which token they represent.

→ when we find a identifier, a call to installID() places it in symbol Table if it is not already there and returns a pointer to symbol-Table entry for the lexeme found.

→ The function getToken examines the symbol Table-entry for lexeme found, and returns whatever token name the symbol Table says this lexeme-represents (either id or) one of key word tokens that was initially installed in Table)

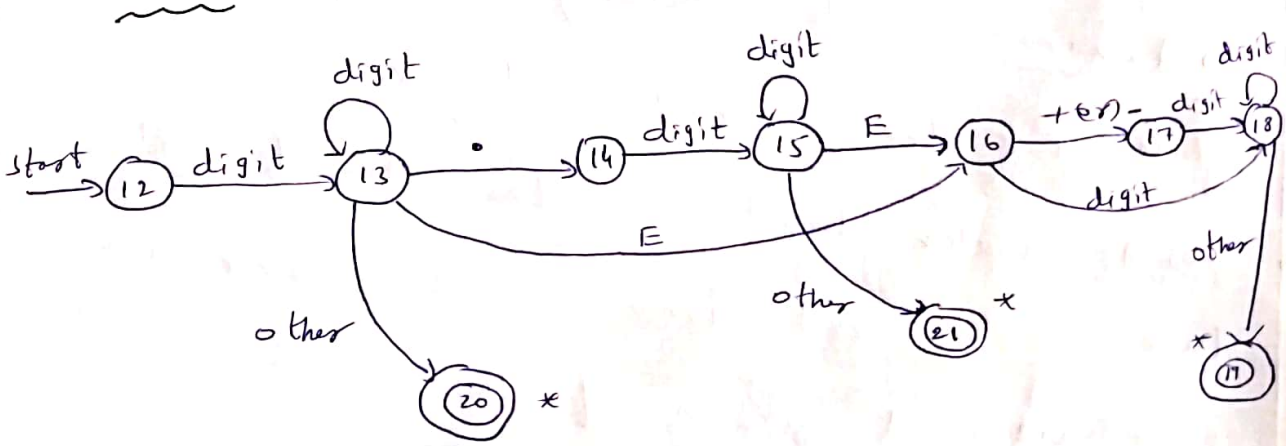
(ii) Create seperate transition diagrams for each keyword



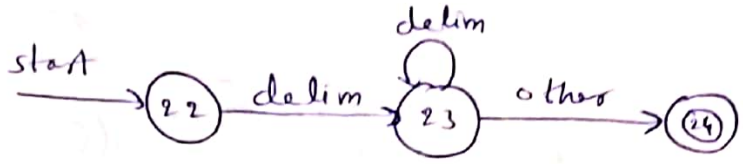
→ It is necessary to check that the identifier has ended, (or) else we would return token then in situations where the correct token was id, with a lexeme the next value that has then as proper prefix.

→ If we adopt this approach, we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when a lexeme matches both patterns.

Transition diagram for token number



→ The final transition diagram for whitespace



delim - blank, tab, newline

→ Note that in state 24, we have found a block of consecutive whitespace characters, followed by a non-whitespace character. we retract the input to begin at non-whitespace, but we donot return to the parser.

Architecture of a Transition-Diagram-Based Lexical Analyzer

→ Each state is represented by a piece of code.

→ A variable 'state' holding the no of current state for the Transition diagram.

→ A switch based on the value of state takes us to code for each of possible states, where we find action of that state.

```
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
                                  relop */
                 break;

      case   1:  ....
       ....

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

     }
   }
}
```

Lexical-Analyzer Generator Lex

→ Lex is tool that allows one to specify a lexical Analyzer by specifying regular expressions to describe patterns for tokens

→ The input notation for the Lex tool is referred to as Lex Language and the tool itself is a Lex Compiler.

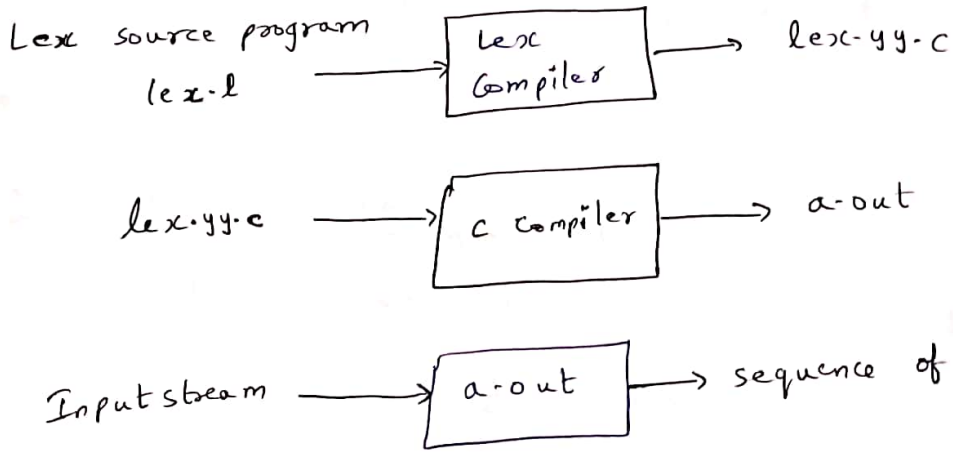
→ Lex Compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy.c

Use of Lex

→ An input file, which we call lex.l, is written in the lex Language

→ The lex compiler transforms lex.l to a C-program, in a file that is always named lex.yy.c

→ This lex.yy.c file is compiled by Ccompiler into a file called a.out



(i) Creating a Lexical Analyzer with Lex.

→ The normal use of the Compiled C-program, (a.out) is as 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, whether it be another numeric code, a pointer to the symbol table, is placed in a global variable yylval which is shared between Lexical Analyzer and parser

Structure of Lex program

```
%{
  Declaration section
%}
%%
translation rules
%%
```

Auxiliary functions

→ The declarations section include declarations of variables, manifest constants (identifiers declared to stand for a constant eg name of token), and regular Definitions.

→ The Translation rules section have the form

Pattern {Action}

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

→ The Actions are fragments of code, typically written in C.

→ The third section holds whatever additional functions are used in Actions.

→ The Lexical analyzer created by Lex behaves in concert with the parser as follows.

→ When called by parser, the lexical analyzer begins reading its remaining input one character at a time, until it finds the longest prefix of the input that matches one of the patterns P_i.

→ It then executes the associated action A_i.

→ Typically A_i will return to the parser.

→ If it does not return (P_i is whitespace (or) comments), then Lexical Analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to parser.

→ The Lexical analyzer returns a single value, the token name to the parser but uses the shared integer variable yylval to pass additional information about the lexeme found, if needed.

Eg: Lex program to recognize tokens

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

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

%%
{ws}      { /* no action and no return */}
if        { return (IF); }
then      { return (THEN); }
else      { return (ELSE); }
```

```
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNum(); return(NUMBER);}
"<"         {yylval = LT; return(RELOP);}
"<="        {yylval = LE; return(RELOP);}
"="         {yylval = EQ; return(RELOP);}
"<>"        {yylval = NE; return(RELOP);}
">"         {yylval = GT; return(RELOP);}
">="        {yylval = GE; 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
thereto */
}
int installNum() {/* similar to installID,
but puts numerical constants into a
seperate table */
}
```

→ Anything that is written between %{ and %} will be copied directly to the file lex.yy.c

→ Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by { } (curly braces)

→ If we wish to use one of the lex meta symbols such as parenthesis, +, * or ?, to stand for themselves, we must precede them with a backslash. (. is preceded by \.)

→ The action taken when id is matched is threefold:

(i) Function installID() is called to place the lexeme found in the symbol table.

(ii) This function returns a pointer to symbol table, which is placed in global variable yylval, it can be used by parser.

→ install ID() has available to it two variables that are set automatically by the lexical Analyzer that Lex generates:

(a) yytext is a pointer to the beginning of the lexeme

(b) yyleng is the length of lexeme found.

(iii) The token name ID is returned to the parser.

Conflict Resolution in Lex

→ Two Rules that Lex uses when several prefixes of the input match one or more patterns:

(i) Always prefer a longer prefix to a shorter prefix

(ii) If the Longest possible prefix matches two (or) more patterns, prefer the pattern listed first in Lex program.

Eg:- <=

then as keyword (must be written before id)

The Look a head Operator

→ Lex automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from input.

→ Sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters. If so, we may use the slash in a pattern to indicate the end of the part of the pattern that matches the lexeme.

→ What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

Eg: In Fortran and some other Languages keywords and not reserved.

IF(I,J) = 3 (IF is name of array, not keyword)

IF (condition) THEN .... (IF is a keyword)

↳ lex rule

IF / \(.*\) {letter}

.* → Any string without a newline

. → (lex Metasymbol) any character except newline

Finite Automata

→ Difference between Transition diagrams and Finite Automata

(i) Finite Automata are recognizers, they simply say "yes" (or) "no" about each possible input string.

(ii) Finite automata has two flavours

(a) Nondeterministic Finite Automata (NFA) have no restrictions on labels and 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) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

→ Both D-F-A and N-F-A are capable of recognizing the same languages.

→ These Languages are exactly the same Languages, called Regular Languages.

Non Deterministic Finite Automata (NFA)

→ The N-F-A consists of:

1) A Finite set of states S:

2) A set of Input symbols Σ, the input alphabet - ϵ (Empty string) is never a member of Σ.

3) A Transition function that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of next states.

4) A state s_0 from S that is distinguished as the start state (or) initial state.

5) A set of states F, a subset of S, that is distinguished as Accepting states (Final states)

→ We can represent NFA (or) DFA by a Transition graph, where the nodes are states and the labeled edges represent Transition function.

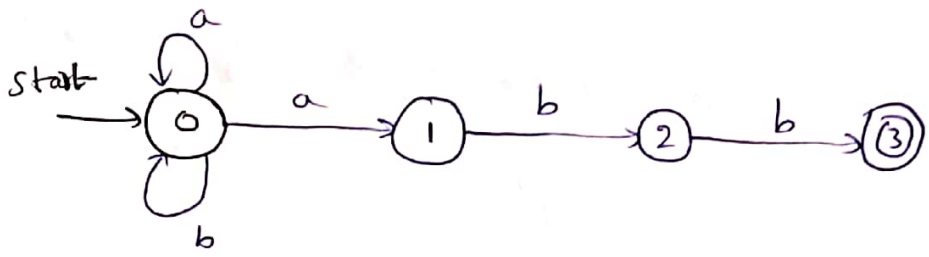
→ There is an edge labeled 'a' from state 's' to state 't' if and only if 't' is one of the next states for state 's' and input 'a'.

→ This is like Transition diagram except:

(a) The same symbol can label edges from one state to several different states.

(b) An edge may be labeled by ϵ, the empty string, instead of, (or) in addition to, symbols from the input alphabet.

Eg:- Regular expression $(a|b)^* abb$



Transition Tables

→ We can also represent an NFA by a Transition Table, where rows correspond to states, and whose columns correspond to input symbols and ϵ

→ The entry for a given state and input is the value of transition function applied to those arguments.

→ If the transition function has no information about the state-input pair, ϕ is put in the table for the pair.

Adv:- Easily find the transitions on given state & input

Disadv:- Takes lot of space when input symbol is large, yet most states donot have any moves on most of the input symbols

state	a	b	ϵ →
0	{0,1}	{0}	∅
1	∅	{2}	∅
2	∅	{3}	∅
3	∅	∅	∅

Acceptance of Input strings by Automata

→ An NFA accepts input string x if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out x.

→ Note: ϵ labels along the path are effectively ignored, since the empty string does not contribute to string constructed along the path.

Eg: a abb

Accepting path: 0 —a→ 0 —a→ 1 —b→ 2 —b→ 3

Another path: 0 —a→ 0 —a→ 0 —b→ 0 —b→ 0

→ The Language defined (or) accepted by an NFA is the set of strings labelling some path from the start to an accepting state:

Ex: R.E (a/b)* abb

→ Language is set of all strings from alphabet {a,b} that end in abb.

→ L(A) to represent Language accepted by A.

Deterministic Finite Automata

→ A D.F.A is special case of an NFA where:

(i) There are no moves on input ∈, and

(ii) For each state 's' and input symbol 'a', there is exactly one edge out of s labeled 'a'.

→ DFA is implemented (or) simulated when building Lexical Analyzers.

Algorithm for simulating DFA

I/p :- An input string 'x' terminated by end-of-file character eof. A DFA D with start state s_0, accepting states F, and transition function move.

o/p :- Answer "yes" if D accepts x ; "no" otherwise

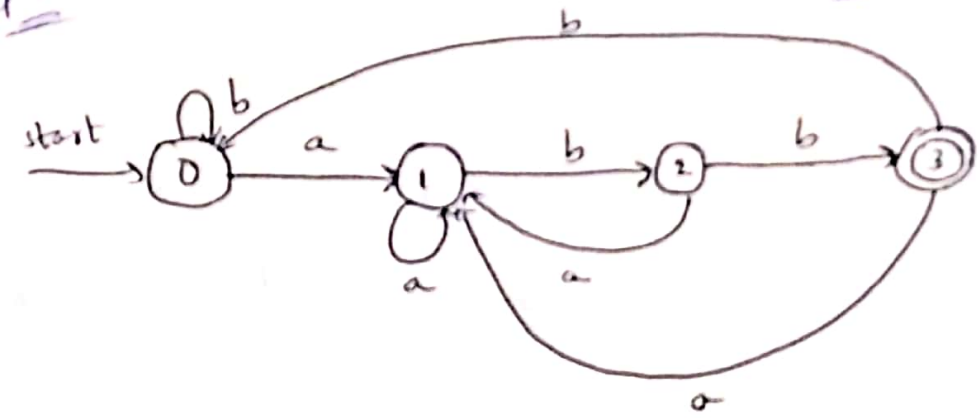
Method :- Apply the following algorithm to input string 'x'.

→ The function move(s,c) gives the state to which there is an edge from state 's' on input 'c'

→ The function nextchar() returns next character of the input string x.

```
s = s0;
c = nextchar();
while (c != eof)
{
  s = move(s,c);
  c = nextchar();
}
if (s is in F) return "yes";
else return "no";
```

Eg: DFA for accepting $(a/b)^*abb$



From Regular Expression to Automata

→ Regular Expression is the notation of choice for describing lexical analyzers and other pattern-processing software.

NFA is represented by Tuple

$(Q, \Sigma, \delta, q_0, F)$

- Q - set of states (Finite)
- Σ - ~~set of~~ Input alphabet (Finite)
- q_0 - start state
- F - set of Final states
- δ - $Q \times \Sigma \rightarrow 2^Q$

D.F.A is represented by Tuple.

$(Q, \Sigma, \delta, q_0, F)$

Q - Finite set of all states

Σ - Input Alphabets

q_0 - start state

F - set of all Final states

δ - $Q \times \Sigma \longrightarrow Q$

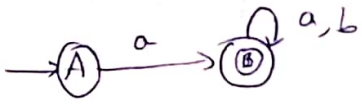
Conversion of N.F.A to D.F.A

Subset Construction Method

Eg ①: $\Sigma = \{a, b\}$

$L_1 = \{$ starts with 'a' $\}$

N.F.A



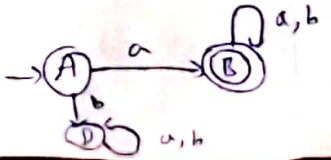
N.F.A Transition Table

	a	b
A	B	ϕ
B	B	B

D.F.A Transition Table

	a	b
A	B	D
*B	B	B
D	D	D

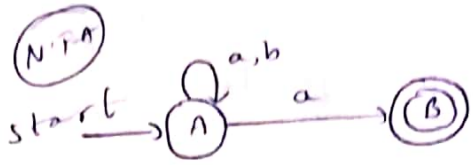
D.F.A



Eg:

Note:- N.F.A and D.F.A are equal.

Eg 2:- (N.F.A)



[L = {ends with 'a'}]

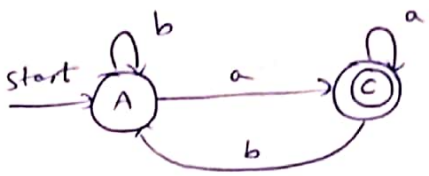
N.F.A

	a	b
→ A	{A, B}	A
B	∅	∅

D.F.A

	a	b
→ A	[A B]	[A]
* [A B]	[A B]	[A]
* C		

D.F.A



Eg 3:- N.F.A

L = { Third symbol from R.H.S is 'a'}

N.F.A

(N.F.A)

	a	b
A	[A B]	[A]
B	[C]	[C]
C	[D]	[D]
* D	∅	∅

(D.F.A)

	a	b
A	[AB]	[A]
[A B]	[A B C]	[A C]
[A C]		
D		

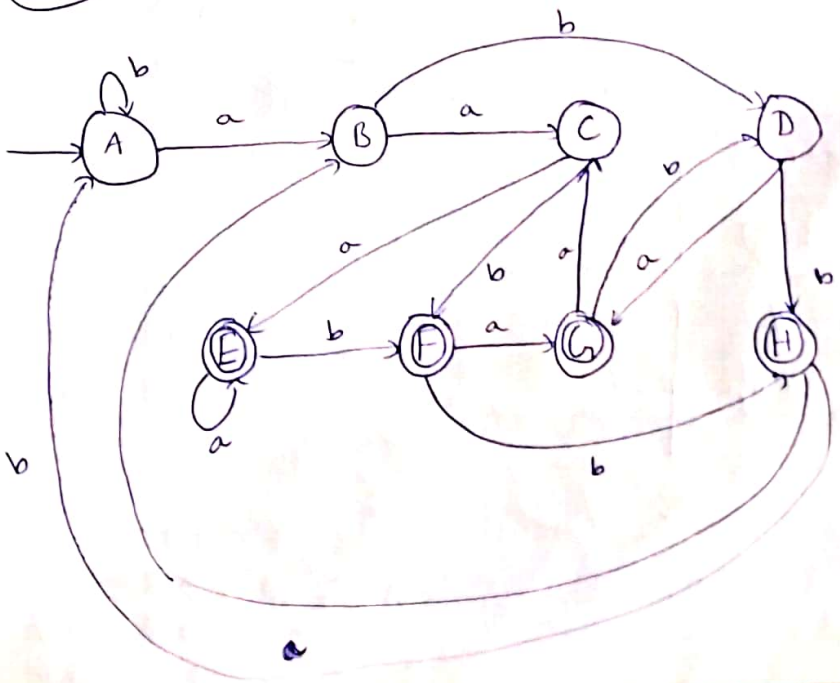
D.F.A [Transition Table]

		a	b
→	A	[AB]	[A]
(B)	[AB]	[ABC]	[AC]
(C)	[ABC]	[ABCD]	[ACD]
(D)	[AC]	[ABD]	[AD]
(E) *	[ABCD]	[ABCD]	[ACD]
(F) *	[ACD]	[ABD]	[AD]
(G) *	[ABD]	[ABC]	[AC]
(H) *	[AD]	[AB]	[A]

* → final state

→ → start state

D.F.A is



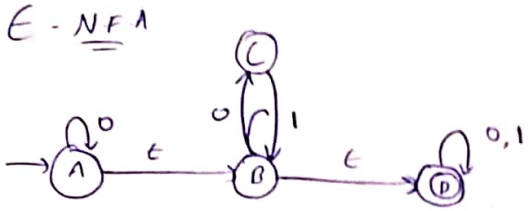
Conversion of ϵ-NFA to NFA

ϵ-NFA (NFA with epsilon Moves)

$(Q, \Sigma, \delta, q_0, F)$

$$\delta : Q \times \Sigma \cup \{\epsilon\} \to 2^Q$$

Eg: ϵ-NFA



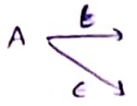
ϵ-closure (A) = {A, B, D}

ϵ-closure (B) = {B, D}

ϵ-closure (C) = {C}

ϵ-closure (D) = {D}

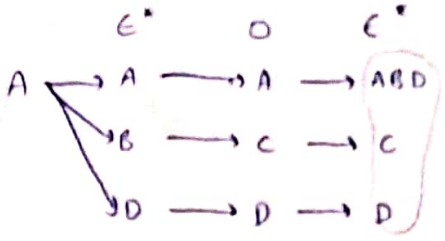
ϵ-closure (A) = {A, B, D}

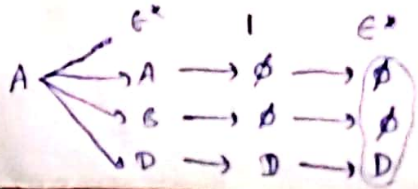


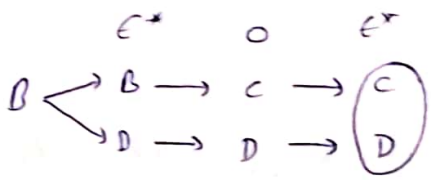
ϵ-closure (δ(ϵ-closure(A), 0))

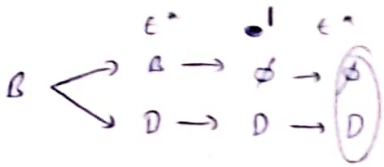
~~ϵ-closure~~ ~~δ(A, 0)~~

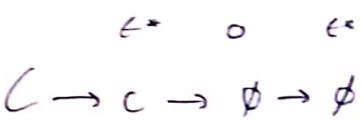
	0	1
A	{A, B, C, D}	{D}
B	{C, D}	{D}
C	ϕ	{B, D}
D	{D}	{D}

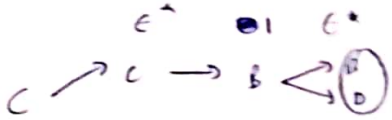


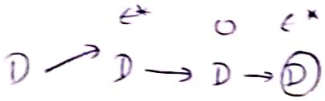


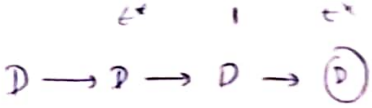


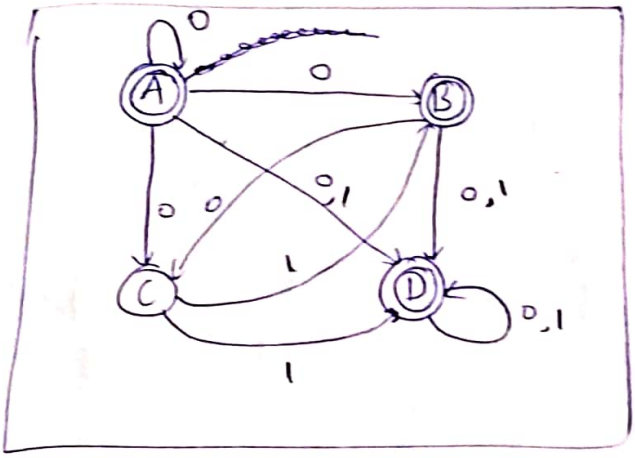










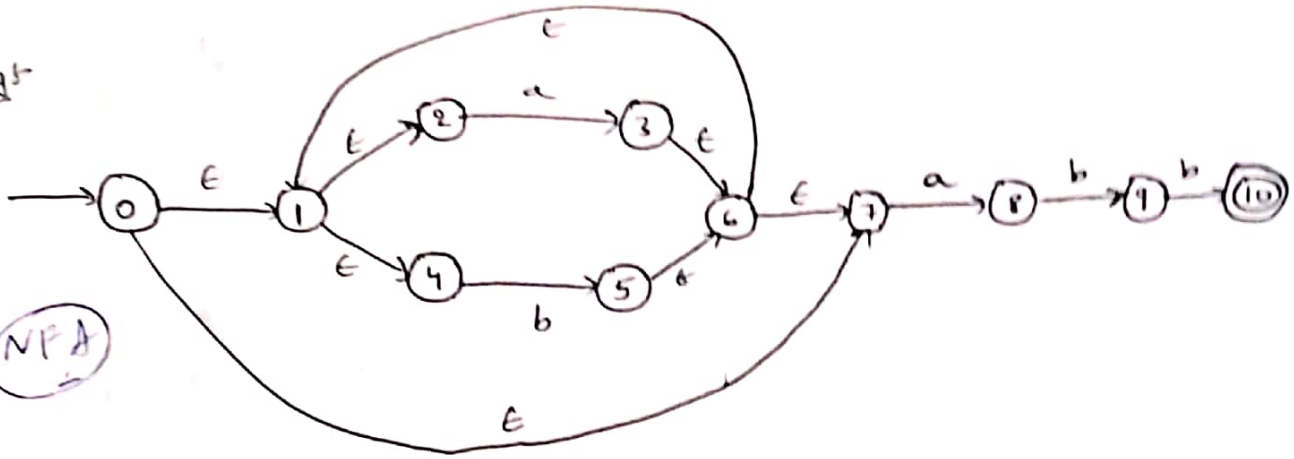


N·F·A

Simulation of N·F·A

```
S = ε-closure(S0);
c = nextchar();
while (c != eof)
{
  S = ε-closure(move(S, c));
  c = nextchar();
}
if (S ∩ F != ∅) return "yes";
else return "no";
```

Fig:



NFA

ε-closure (0) = {0, 1, 2, 4, 7}

ε-closure (1) = {1, 2, 4}

ε-closure (2) = {2}

ε-closure (3) = {3, 6, 7, 1, 2, 4}

ε-closure (4) = {4}

ε-closure (5) = {5, 6, 7, 1, 2, 4}

ε-closure (6) = {6, 7, 1, 2, 4}

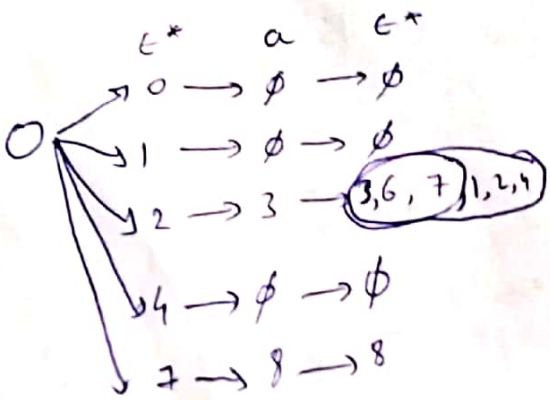
ε-closure (7) = {7}

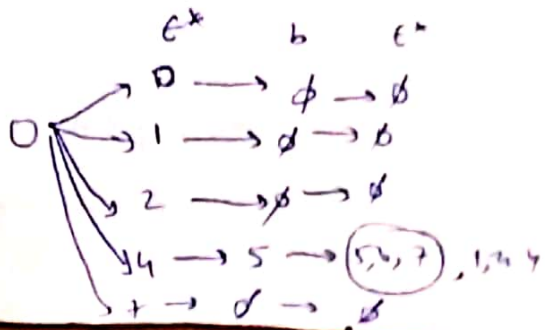
ε-closure (8) = {8}

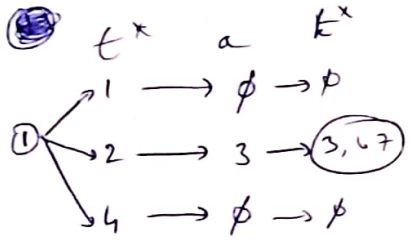
ε-closure (9) = {9}

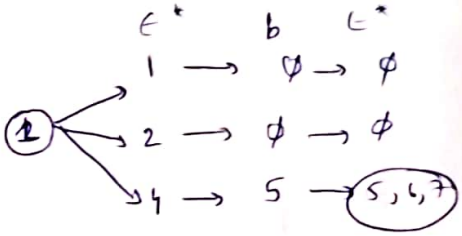
ε-closure (10) = {10}

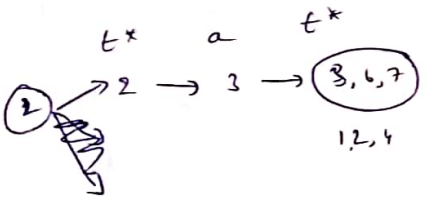
	a	b
0	{3, 6, 7, 1, 2, 4, 8}	{5, 6, 7, 1, 2, 4}
1	{3, 6, 7, 1, 2, 4}	{5, 6, 7, 1, 2, 4}
2	{3, 6, 7, 1, 2, 4}	∅
3	{1, 2, 3, 4, 6, 7, 8}	{1, 2, 4, 5, 6, 7}
4	∅	{1, 2, 4, 5, 6, 7}
5	{1, 2, 3, 4, 6, 7, 8}	{1, 2, 4, 5, 6, 7}
6	{1, 2, 3, 4, 6, 7, 8}	{1, 2, 4, 5, 6, 7}
7	{8}	∅
8	∅	{9}
9	∅	{10}
10	∅	∅

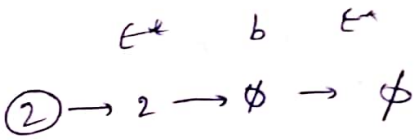


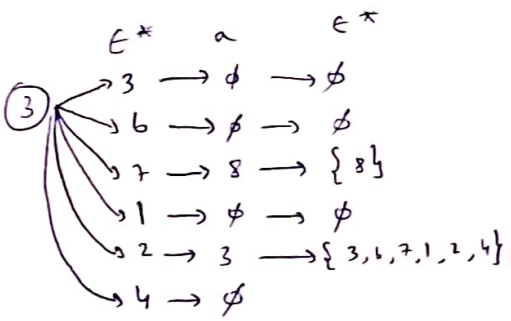


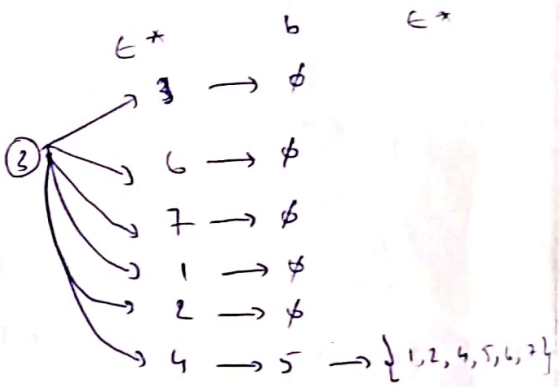


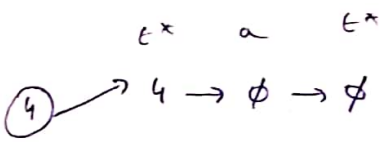


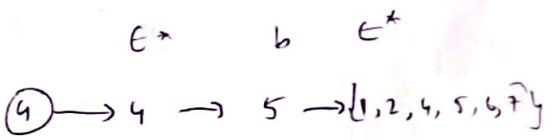


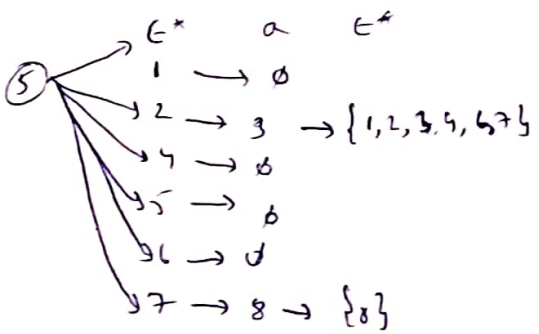


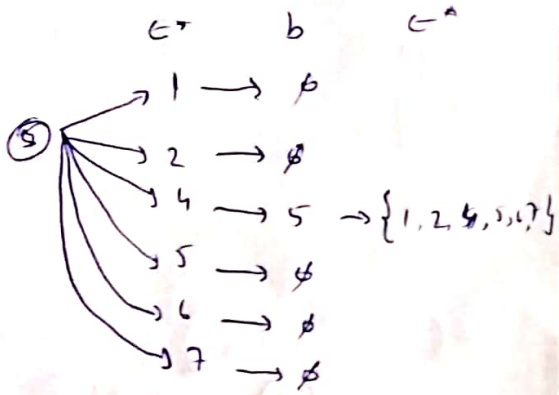


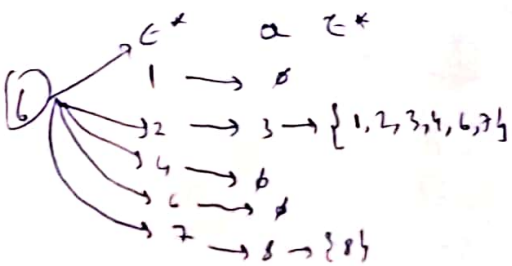


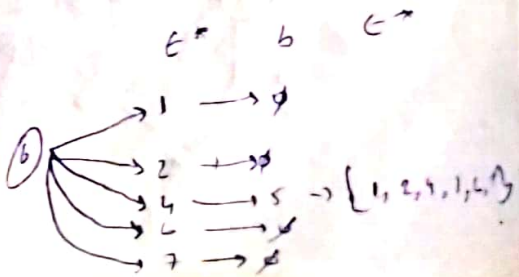












ϵ^* a ϵ^*

7 → 7 → 8 → 8

ϵ^* b ϵ^*

7 → 7 → ϕ → ϕ

ϵ^* a ϵ^*

8 → 8 → ϕ → ϕ

ϵ^* b ϵ^*

8 → 8 → 9 → 9

ϵ^* a ϵ^*

9 → 9 → ϕ → ϕ

ϵ^* b ϵ^*

9 → 10 → 10 → 10

ϵ^* a ϵ^*

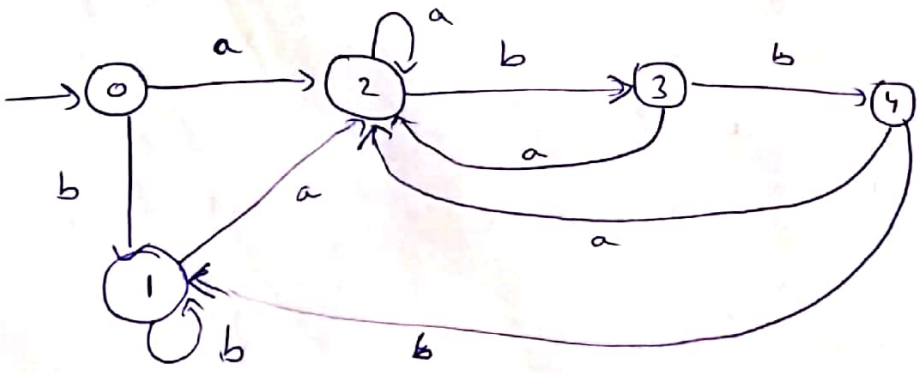
10 → 10 → ϕ → ϕ

ϵ^* b ϵ^*

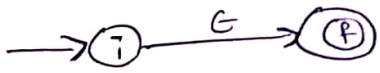
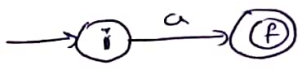
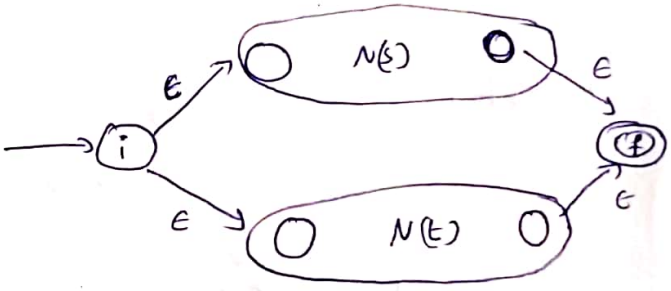
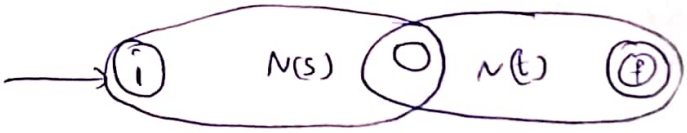
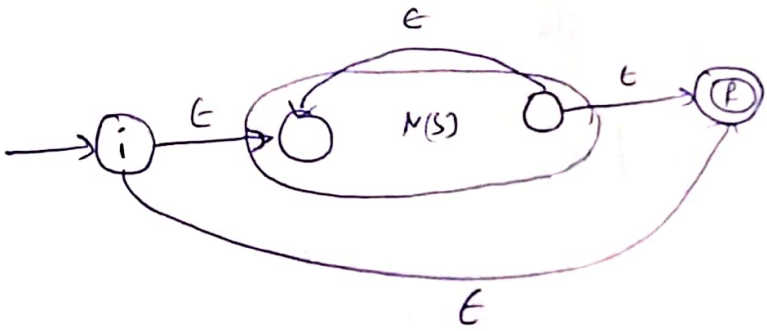
10 → 10 → ϕ → ϕ

NFA to D.F.A

	a	b
→ 0	[1 2 3 4 6 7 8] 2	[1 2 4 5 6 7] 1
1 [1 2 4 5 6 7]	[1 2 3 4 6 7 8] 2	[1 2 4 5 6 7] 1
[1 2 3 4 6 7 8] 2	[1 2 3 4 6 7 8] 2	[1 2 4 5 6 7 9] 3
3 [1 2 4 5 6 7 9]	[1 2 3 4 6 7 8] 2	[1 2 4 5 6 7 10] 4
[1 2 4 5 6 7 10] 4	[1 2 3 4 6 7 8] 2	[1 2 4 5 6 7] 1



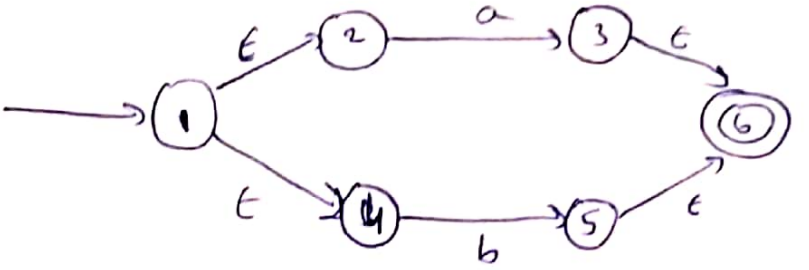
Construction of N.F.A from Regular Expression

R.E	N.F.A
ϵ	
a	
s | t	
s t	
s*	

Eg:- N.F.A for R.E = $(a|b)^* a b b$

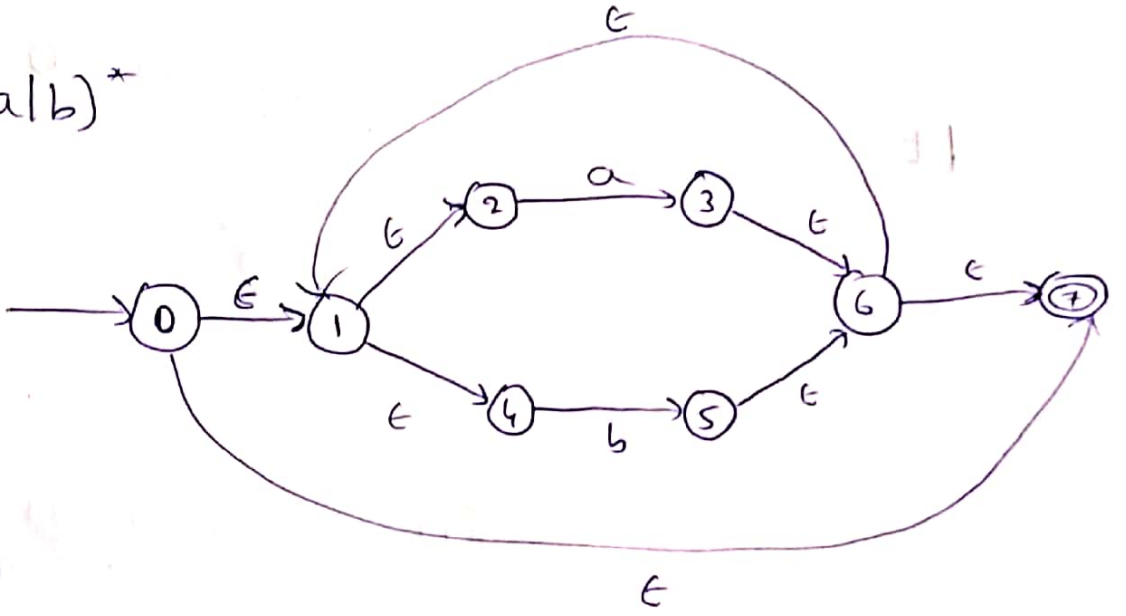
$a|b$

N.F.A



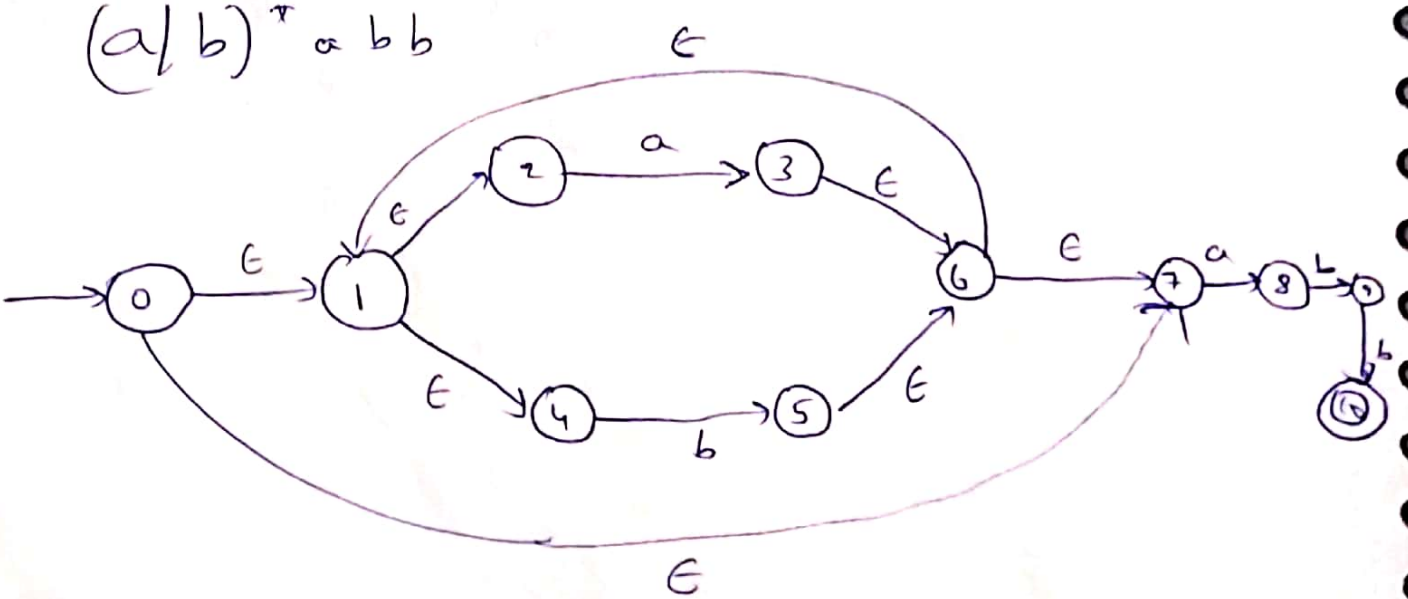
N.F.A for

$(a|b)^*$



Final N.F.A for

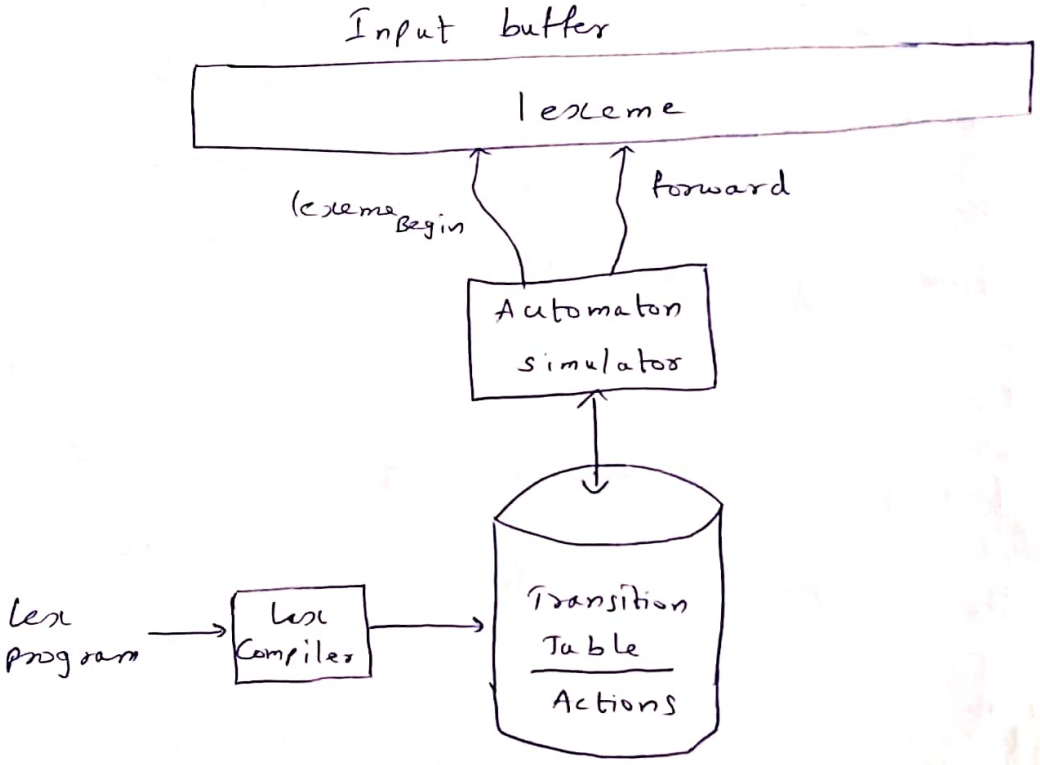
$(a|b)^* a b b$



Design of Lexical Analyzer Generator

Structure of a Generated Analyzer

→ DFA is used for the implementation of Lex.



→ The program that serves as the Lexical Analyzer includes a fixed program that simulates an automaton.

→ The rest of lexical Analyzer consists of components that are created from Lex program by Lex itself.

The components are:

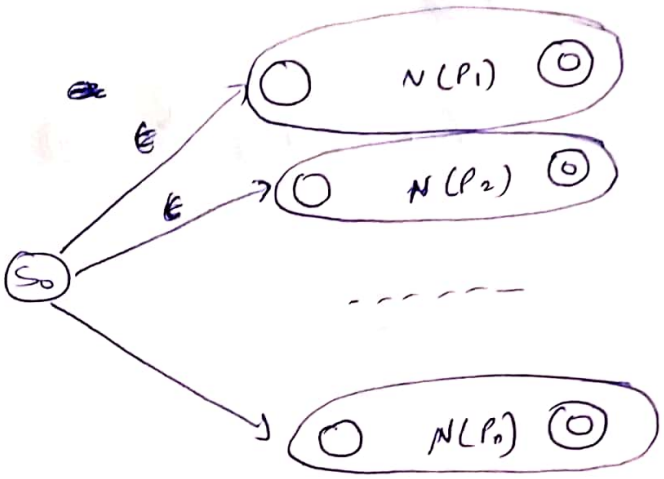
1) A Transition table for the automaton
2) Those functions that are passed directly through Lex to the output.
3) The actions from the input program, which appear as fragments of code to be invoked at the appropriate time by the automaton simulator.

→ To construct the automaton, we begin by taking each regular expression pattern in the Lex program and converting it to NFA

→ we need a single automaton that will recognize lexemes matching any of the patterns in the program, so we combine all the NFA's into one by introducing a new start state with ε-transitions to each of start states of NFA's N_1.

for pattern P_i :

N-F-A constructed from Lex Program



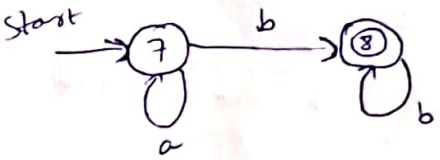
a { action A_1 for pattern P_1 }

abb { action A_2 for pattern P_2 }

$a^* b^+$ { action A_3 for pattern P_3 }

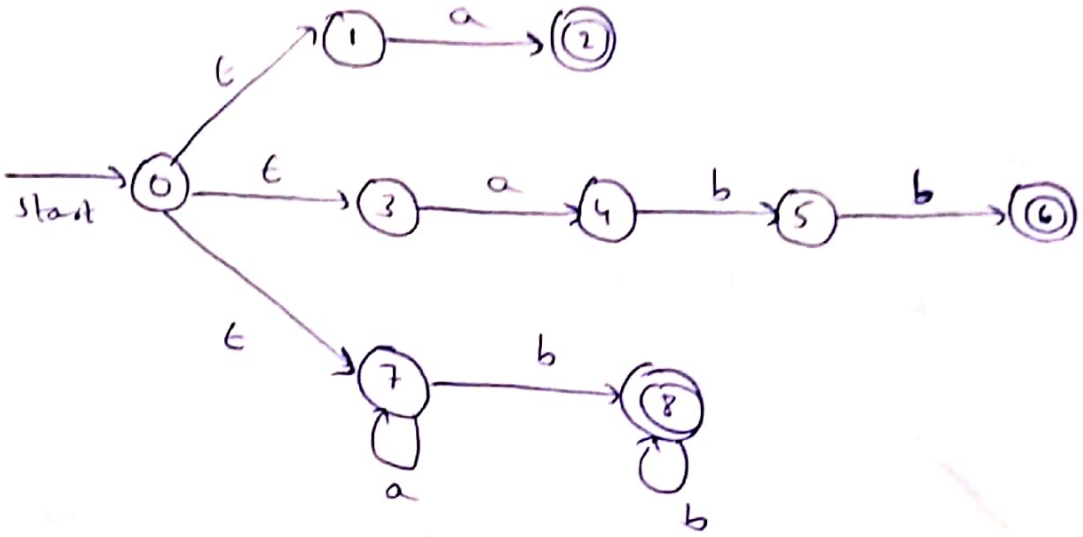
(NFA for a)

(N-F-A for abb)



(NFA for $a^* b^+$)

Combined NFA



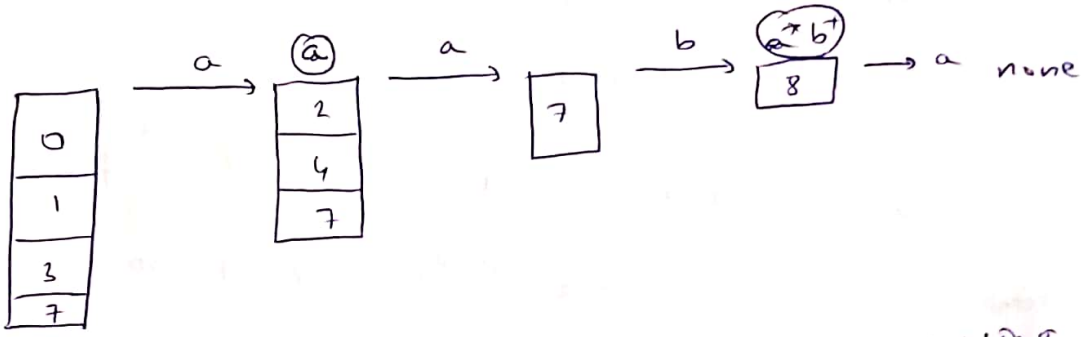
Pattern Matching Based on NFA's

→ If lexical Analyzer simulates an NFA then it must read input beginning at a point on its input which we have refferred to as lexemeBegin.

→ As it moves the pointer called forward ahead in the input, it calculates the set of states it is in at each point.

→ Eventually, the NFA simulation reaches a point on the input where there are no next states.

→ At that point, there is no hope that any longer prefix of input would ever get the N.F.A to an accepting state, rather, the set of states will always be empty.



Sequence of set of states entered when processing input aaba

→ we look backwards in the sequence of set of states, until we find a set that includes one or more accepting states.

→ If there are several accepting states in that set, pick the one associated with the earliest pattern P_i in the list from Lex program.

→ Move the forward pointer back to end of end of lexeme and perform action A_i associated with pattern P_i.

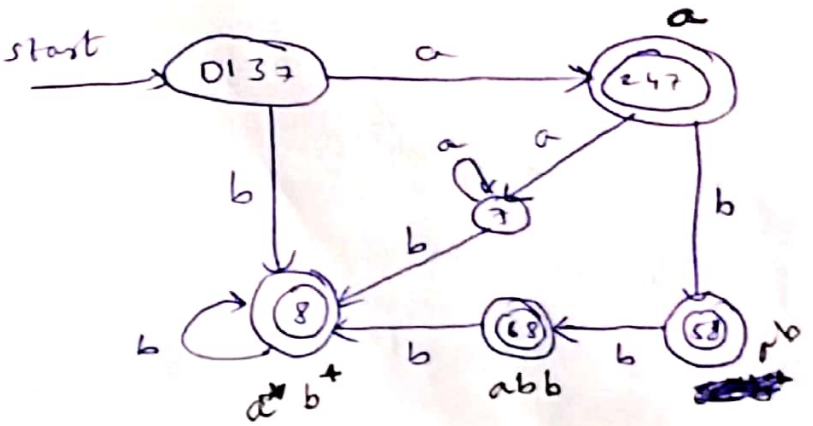
DFA's for lexical Analyzers

→ Another Architecture, resembling the output of Lex, is to convert NFA for all patterns into equivalent DFA using subset Construction Method.

→ with in each D-FA state, if there are one (or) more accepting N-F-A states, determine the first pattern whose accepting state is represented, and make pattern the output of D-F-A state.

→ we simulate D-F-A until there is no next state.

Transition graph for DFA handling the patterns a, abb and $a^* b^+$



Implementing the lookahead operator

→ The Lex lookahead operator / in a Lex pattern r_1/r_2 is sometimes necessary, because the pattern r_1 for a particular token may need to describe some trailing context r_2 in order to correctly identify the actual lexeme.

→ when converting pattern r_1/r_2 to an NFA, we treat '/' as if it were ϵ. So we do not actually look for a '/' on the input.

→ If the NFA recognizes a prefix xy of the input buffer as matching this regular expression, the end of lexeme is no where NFA entered its accepting state.

→ The end occurs when NFA enters a state 's' such that.

(i) 's' has ϵ transition on the (imaginary) /,

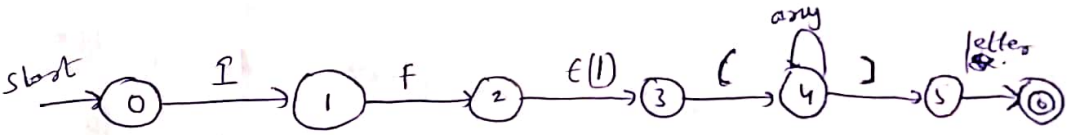
(ii) There is a path from the start state of NFA to state s that spells out x.

(iii) There is a path from start state 's' to accepting state that spells out y.

(iv) x is as long as possible for any xy satisfying conditions 1-3.

~~Recognition of DFA Based Pattern Matchers~~

Eg: NFA for recognizing keyword IF in Fortran



→ Notice that the ϵ-transition from (2→3) represents the lookahead operator.

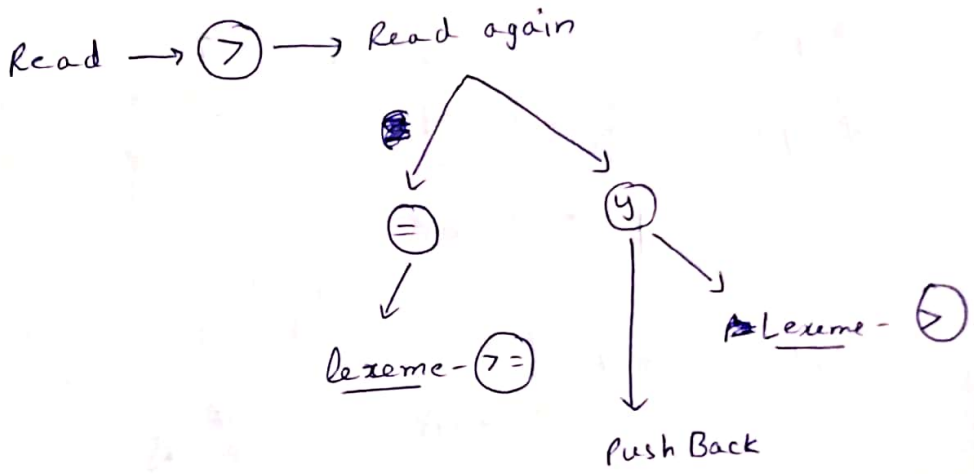
→ State 6 indicates the presence of keyword (IF).

→ we find the lexeme IF by scanning backwards to last occurrence of state 2, whenever

State 6 is entered.

Eg: ②

$x >= y$



Optimization of DFA Based Pattern Matchers.

(1) The first algorithm is useful in a lex compiler because it constructs a D.F.A directly from Regular Expression

(2) The second algorithm minimizes the no of states of any D.F.A, by combining states that have some future behaviour.

(3) The third algorithm produces more compact representations of transition Tables.

RE to DFA Directly

(i) nullable
(ii) firstpos
(iii) Last pos
(iv) Followpos

**Followpos

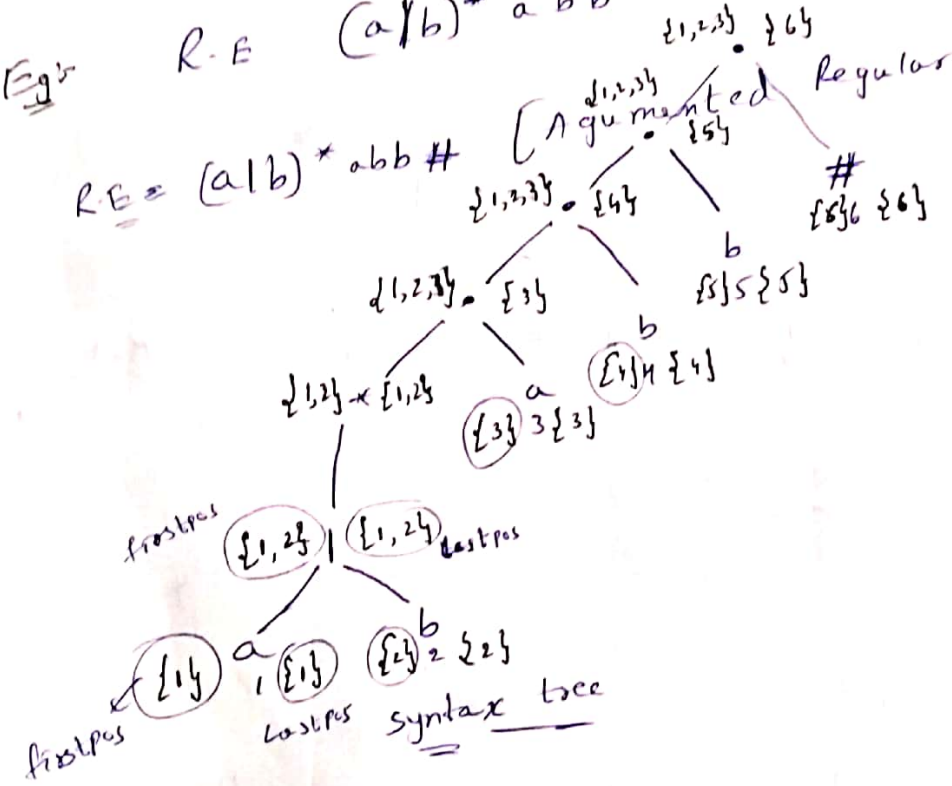
→ For cat node ($c_1 \cdot c_2$)
for every i in Lastpos(c_1) add Firstpos(c_2)

→ For * node
for every i in Lastpos(n) add firstpos(n)

Eg: R.E $(a|b)^* a b b$

[Augmented Regular Expression]

RE = $(a|b)^* abb\#$

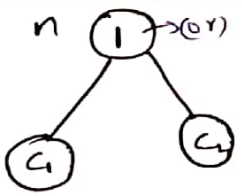
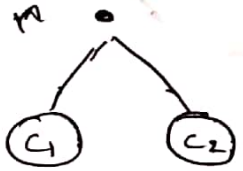
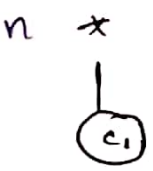


6 Leaf nodes

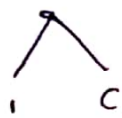
→ nullable True only for '*'

→ If node is * node
for each i in Lastpos(n)
follow(i) = followpos(i) ∪ firstpos(n)

Scanned with CamScanner

node n	nullable(n)	firstpos(n)	Lastpos(n)
If n is leaf labelled with transition 'i'	false	{i}	{i}
	nullable(c_1) (or) nullable(c_2)	firstpos(c_1) union firstpos(c_2)	Last ~~first~~pos(c_1) union Last ~~first~~pos(c_2)
	nullable(c_1) and nullable(c_2)	if nullable(c_1) then firstpos(c_1) union firstpos(c_2) else firstpos(c_1)	if nullable(c_2) then Lastpos ~~firstpos~~(c_1) union Lastpos ~~firstpos~~(c_2) else Lastpos(c_2) ~~firstpos~~
	true	firstpos(c_1)	Last ~~pos~~(c_1)

Followpos(i)

(i) If n is cat node  and i is a position in lastpos(c_1) then all position in firstpos(c_2) are in followpos(i)

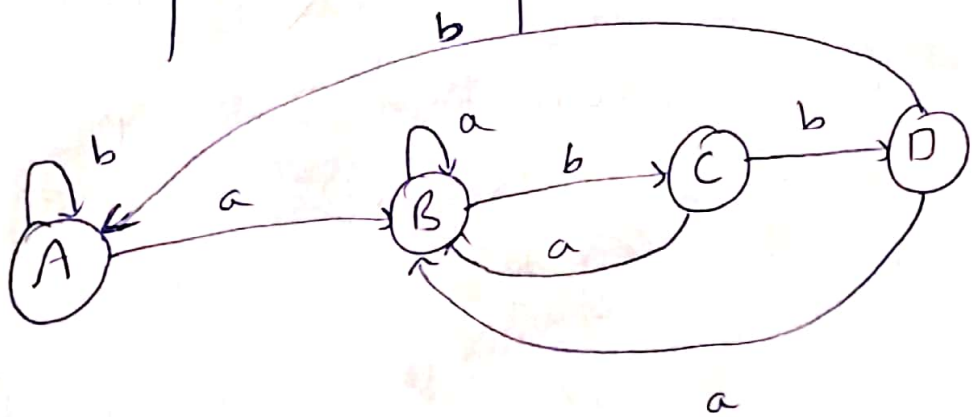
(ii) If n is * node and i is a position in lastpos(n) then all position in firstpos(n) in followpos(i)

Node n	followpos(n)
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	∅

D.F.A Construction — union of followpos(pos) and followpos(pos) corresponding to a / corresponding to b

state	DFA state	a	b
{1, 2, 3}	A	B	A
{1, 2, 3, 4}	B	B	C
{1, 2, 3, 5}	C	B	D
{1, 2, 3, 6}	D	B	A

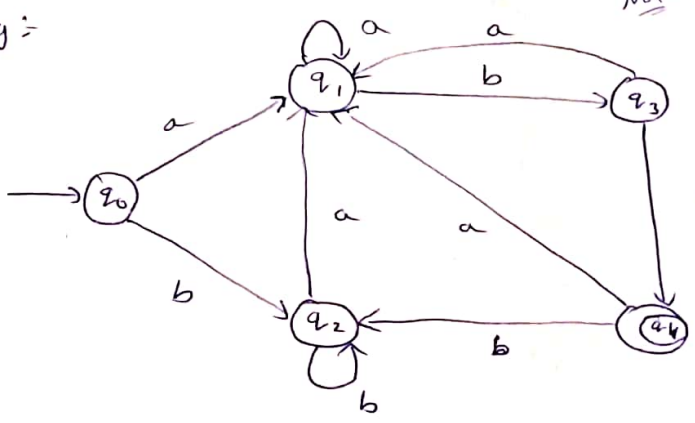
DFA



Minimizing no of states in DFA.

Partitioning method

Eg:-



Note

states

P, q are equal

if $\delta(p,w) \in F$

$\Rightarrow \delta(q,w) \in F$

if $\delta(p,w) \notin F$

$\Rightarrow \delta(q,w) \notin F$

step 1 :- Try to delete all unreachable states

step 2 :- Draw state Transition Table

step 3 :- Try to find out 0-equivalent sets

	a	b
→ q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4
* q_4	q_1	q_2

Non Final states: $[q_0 q_1 q_2 q_3]$

Final state: $[q_4]$

step 3:- Try to seperate final & Non-Final states

0 equivalent

$[q_0\ q_1\ q_2\ q_3]$ $[q_4]$

$(q_0\ q_1)$ —a→ (q_1, q_2) (Non final) ✓
—b→ (q_1, q_3) (Non Final)

$(q_0\ q_2)$ —a→ (q_1, q_1) (NF) ✓
—b→ (q_2, q_2) (NF)

$(q_0\ q_3)$ —a→ (q_1)
—b→ (q_2, q_4) (NF, F) ×

1 - equivalent

$[q_0\ q_1\ q_2]$ $[q_3]$ $[q_4]$

(q_0, q_1) —a→ (q_1, q_2) (NF)
—b→ (q_3, q_2) (Diff states) ×

2 - equivalent

$[q_0\ q_2]$ $[q_1]$ $[q_3]$ $[q_4]$

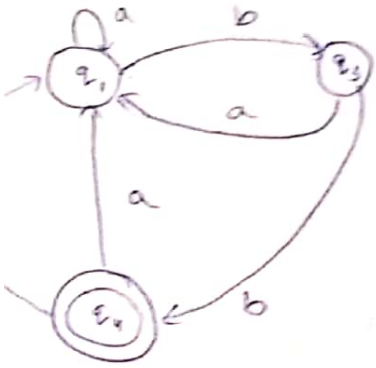
(q_0, q_2) —a→ (q_1) ✓ (No problem)
—b→ (q_2)

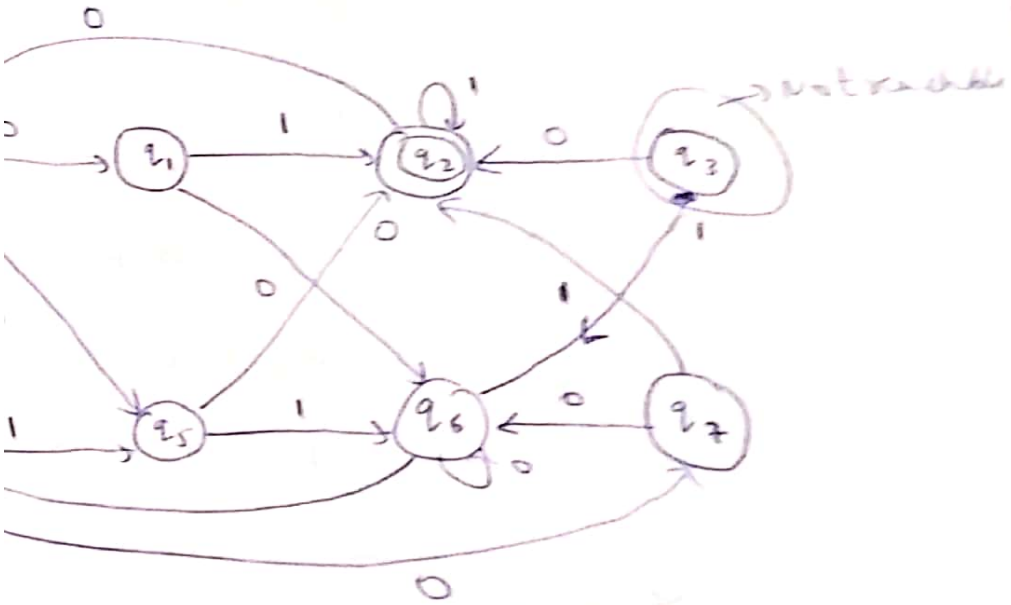
Both equal so stop

3 - equivalent

$[q_0\ q_2]$ $[q_1]$ $[q_3]$ $[q_4]$

D.F.A





0	1	
q_1	q_5	
* q_6	* q_2	
q_0	* q_3	
* q_2	q_6	unreachble
q_7	q_5	
* q_3	q_0	
q_6	q_4	
q_6	* q_2	

0-equivalent $[q_0 q_1 q_4 q_5 q_6 q_7]$ $[q_2]$

× (q_0, q_1) 0 → (q_1, q_6) ✓ ; 1 → (q_5, q_2^*) ×

✓ (q_0, q_4) 0 → (q_1, q_7) ✓ ; 1 → (q_5) ✓

× (q_0, q_5) 0 → (q_1, q_2^*) × ; 1

✓ (q_0, q_6) 0 → (q_1, q_6) ✓ ; 1 → (q_5) ✓

× (q_0, q_7) 0 → (q_1, q_6) ✓ ; 1 → (q_5, q_2^*) ×

× (q_1, q_4) 0 → (q_6, q_7) ✓ ; 1 → (q_2^*, q_5) ×

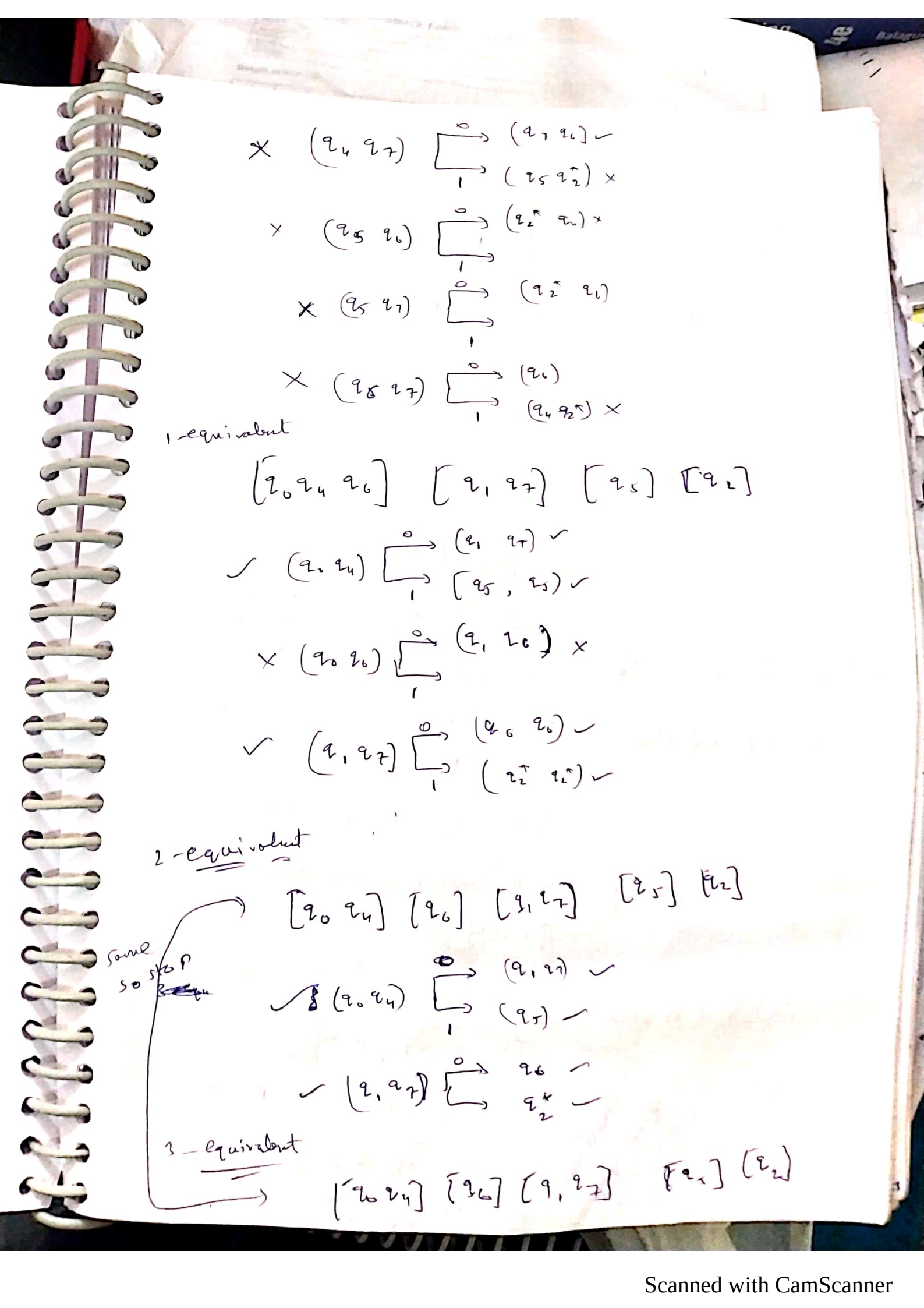
× (q_1, q_5) 0 → (q_6, q_2) * ; 1

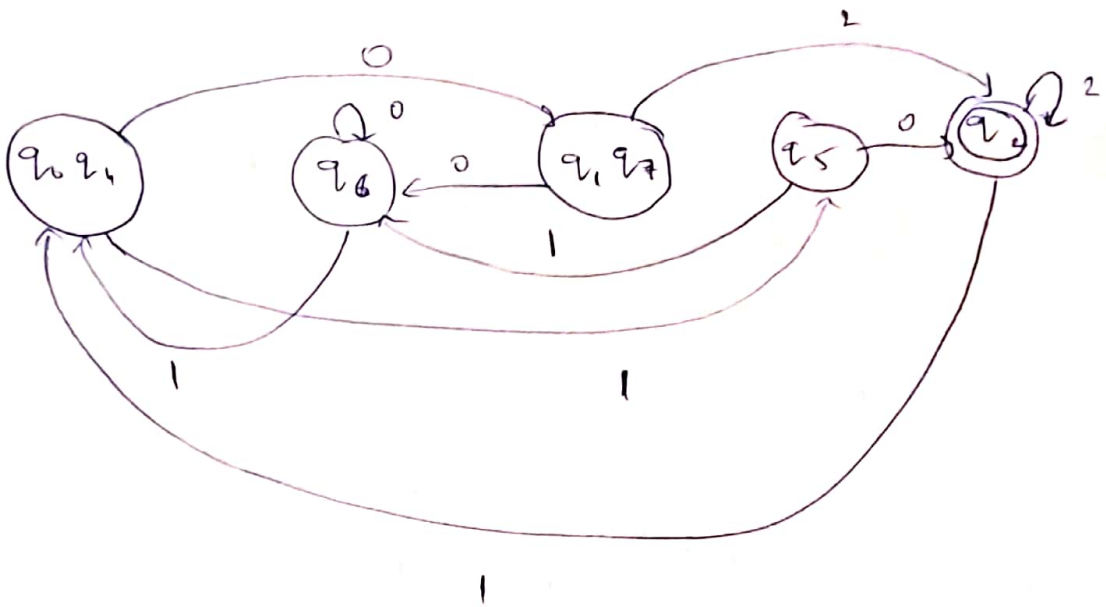
× (q_1, q_6) 0 → (q_6, q_6) ✓ ; 1 → (q_2^*, q_7) ×

✓ (q_1, q_7) 0 → (q_6, q_1) ✓ ; 1 → (q_2^*) ✓

× (q_4, q_5) 0 → (q_7, q_2^*) × ; 1

✓ $q_4 q_6$ 0 → (q_7, q_6) ✓ ; 1 → (q_5, q_4) ✓





Minimal D.F.A

D.F.A from follow pos table

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

a - 1.
b - 2.
a - 3.
b - 4
b - 5
- 6

Initial state $A = \{1, 2, 3\}$

$(A, a) = A = \text{Followpos}(1) \cup \text{Followpos}(3)$

$= \{1, 2, 3\} \cup \{4\}$

$= \boxed{\{1, 2, 3, 4\} = B}$ new state

$(A, b) = \text{Followpos}(2)$

$= \boxed{\{1, 2, 3\}}$ A

$(B, a) = \text{Followpos}(1) \cup \text{Followpos}(3)$

$= \boxed{\{1, 2, 3, 4\}}$ B

$(B, b) = \text{Followpos}(2) \cup \text{Followpos}(4)$

$= \{1, 2, 3\} \cup \{5\}$

$\boxed{\{1, 23, 5\} = C}$ new state

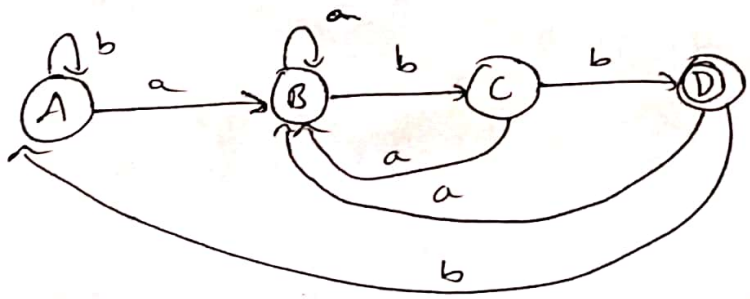
(C, a) = Followpos(1) ∪ Followpos(3)
= {1,2,3} ∪ {4}
(C, a) = {1,2,3,4} = B

(C, b) = Followpos(2) ∪ Followpos(5)
= {1,2,3} ∪ {6}
= {1,2,3,6} = D new state

(D, a) = Follow(1) ∪ Followpos(3)
= {1,2,3} ∪ {4}
= {1,2,3,4} = B

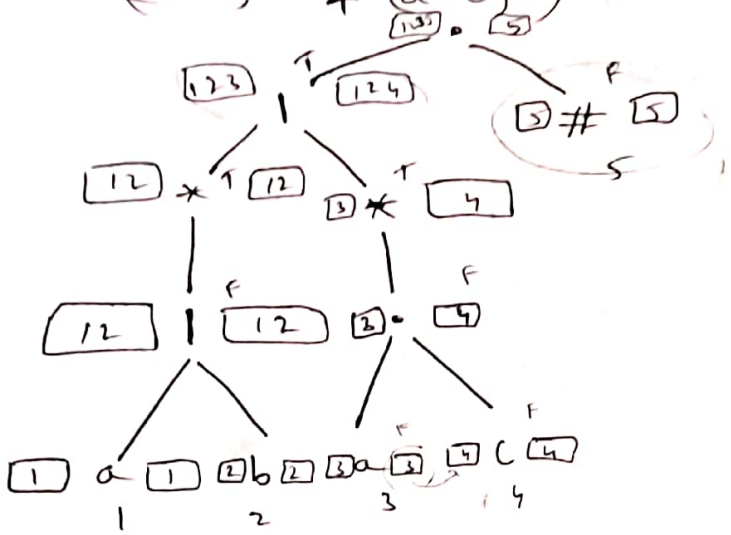
(D, b) = Follow(4)
= {1,2,3} = A

where ever 6 (#node) is there then it is final state.



Eg 3: $a^* b^* a (a|b)^* b^* a \#$

Eg 2:- $(a+b)^* + (a \cdot c)^*) \cdot \#$



		followpos
a	1	{1, 2, 5}
b	2	{1, 2, 5}
a	3	{4}
c	4	{3, 5}
#	5	ϕ

A = 1235

(A, a) = Followpos(1) ∪ Followpos(3)
= {125} ∪ {4}
= {1, 2, 4, 5} D

(A, b) = Followpos(2)
= {1, 2, 5} C

(A, c) = Followpos(4)
= {3, 5} D

(B, a) = {1, 2, 5} C

(B, b) = {1, 2, 5} C

(B, c) = {3, 5} D

(C, a) = {1, 2, 5} C

(C, b) = {1, 2, 5} C

(C, c) = {ϕ}

(D, a) = {4} E

(D, b) = ϕ

(D, c) = ϕ

(E, a) = ϕ

(E, b) = ϕ

(E, c) = {3, 5} D

Final states

C, D, B, A