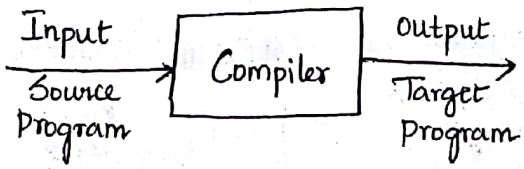
UNIT-I INTRODUCTION TO COMPILER AND AUTOMATA.

1. Compilers - Analysis of the source program.
2. Phases of compiler - cousins of the compiler
3. Grouping of phases - Compiler construction tools.
4. Lexical analysis - Role of lexical Analyzer
5. Input buffering - Specification of tokens - Design of lexical analysis(LEX)
6. Finite automation (Deterministic and non-deterministic)
   Conversion of regular expression of NDFA - Thompson's.
7. Conversion of NDFA to DFA - Minimization of NDFA.
8. Derivation - parse tree - ambiguity.

Compilers:-

⇒ Compiler is a program which takes one language (source program) as input and translates it into an equivalent another language (Target program)



⇒ Compiler converts high level language to machine level language.

⇒ During this process of translation if some errors are encountered then compiler displays them as error messages.

⇒ The compiler takes a source program as high level language such as C, PASCAL, FORTRAN and converts it into low level language or a machine level language such as assembly language.

Analysis of the source program :-

⇒ The compilation can be done in two parts.

1. Analysis part - the source program is read and broken down into constituent pieces.
   - The syntax and the meaning of the source string is determined and then an intermediate code is created from the input source program.

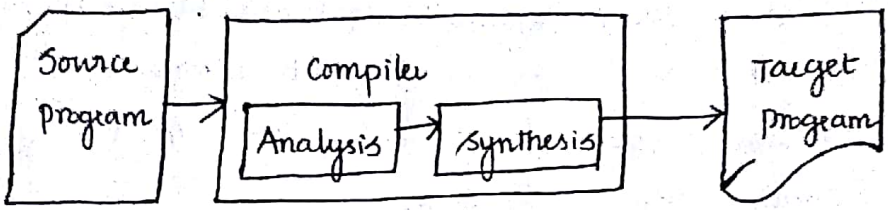


Fig: Analysis and synthesis model.

2. Synthesis part - Intermediate form of the source language is taken and converted into an equivalent target program.

Execution of program

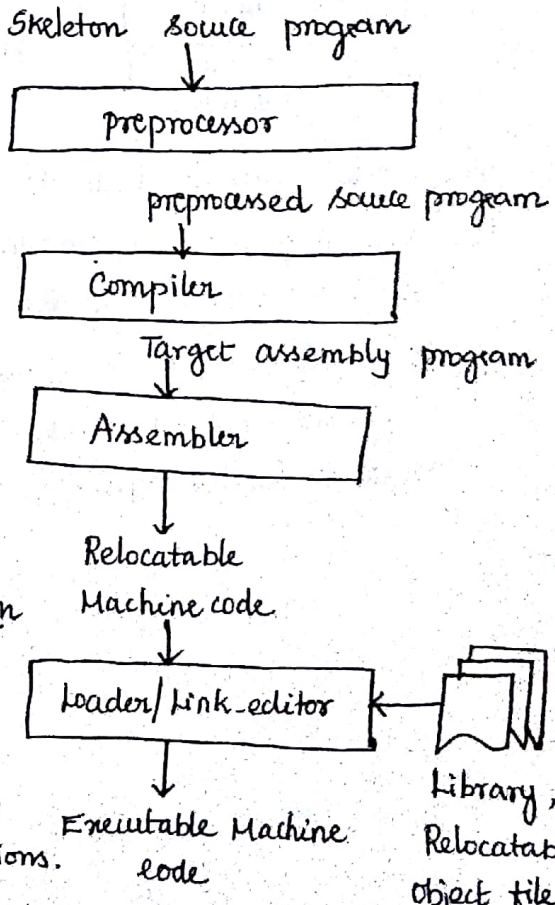
⇒ Compiler takes a source program written in high level language as an input and converts it into a target assembly language.

⇒ The assembler then takes this target assembly code as input and produces a relocatable machine code as an input.

⇒ Then a program loader is called for performing the task of loading and link editing.

⇒ Task of loader is to perform the relocation of an object code.

Allocation and placement of load time address and data in memory at proper locations.



⇒ The link editor links the object modules and prepares a single module from several files of relocatable object modules to resolve the mutual references.

⇒ These files may be library files and these library files may be referred by any program.

Properties of compiler

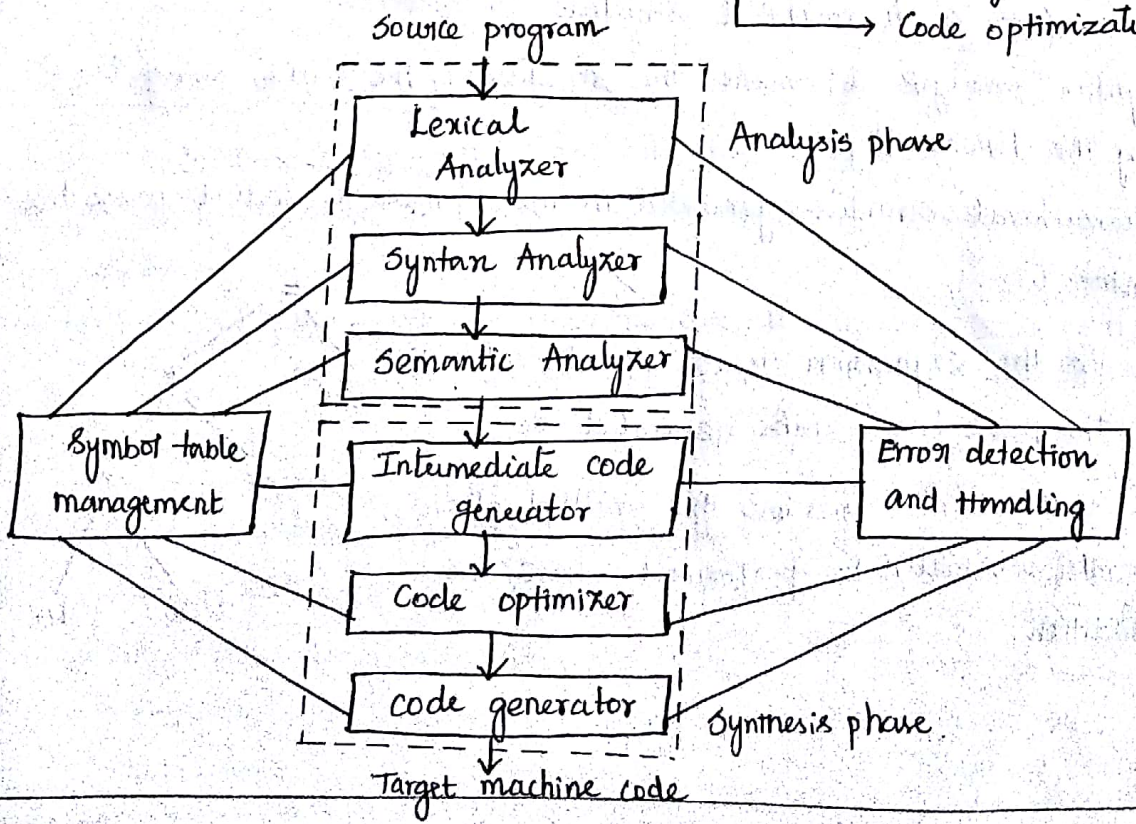
1. The compiler itself must be bug-free
2. It must generate correct machine code.
3. The generated machine code must run fast
4. The compiler must be portable
5. It must give good diagnostics and error messages.
6. It must have consistent optimization.

Phases of compilers :-

⇒ The process of compilation is carried out in two parts analysis and synthesis

⇒ Analysis is carried out in three phases → Lexical analysis, Syntax analysis, Semantic analysis.

⇒ Synthesis is carried out in three phases → Intermediate code generation, Code generation, Code optimization.



Lexical Analysis:

⇒ The lexical analysis is also called scanning.

⇒ Complete source code is scanned and broken up into group of strings called tokens.

⇒ A token is a sequence of characters having a collective meaning.

Example: Consider the assignment statement

`total = count + rate * 10`

Then in lexical analysis phase this statement is broken up into series of tokens as follows,

1. The identifier total
2. The assignment symbol
3. The identifier count
4. The plus sign.
5. The identifier rate
6. The multiplication sign
7. The constant number 10.

⇒ The blank characters which are used in the programming statement are eliminated during lexical analysis phase.

Syntax Analysis:

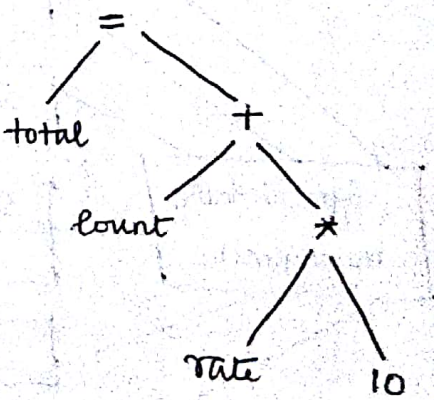
⇒ The syntax analysis is also called parsing.

⇒ In this phase the tokens generated by the lexical analyzer are grouped together to form a hierarchical structure.

⇒ The syntax analysis determines the structure of the source string by grouping the tokens together.

⇒ The hierarchical structure generated in this phase is called parse tree or syntax tree.

Example: For the expression total = count + rate * 10
The parse tree can be generated as,



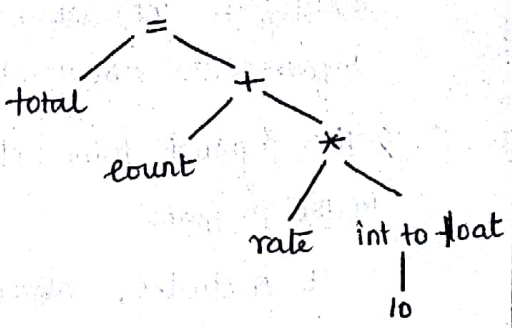
Here, in arithmetic expression the multiplication operation should be performed before the addition.

Semantic Analysis:-

⇒ Semantic analysis determines the meaning of the source string.

⇒ For example meaning of source string means matching of parenthesis in the expression, or matching of if....else statements or performing arithmetic operations of the expressions that are type compatible or checking of scope of operation.

⇒ Example: Type compatibility checking of the expression total = count+rate*10



Intermediate code Generation :-

⇒ The intermediate code is a kind of code which is easy to generate & it is in variety of forms such as three address code, quadruple, triple, Posix.

⇒ The three address code consists of instructions each of which has at the most three operands.

⇒ Example:

$t_1 := int\ to\ float(10)$

$t_2 := rate * t_1$

$t_3 := count + t_2$

$total := t_3$

Code optimization :-

⇒ The code optimization phase attempts to improve the intermediate code.

⇒ This is necessary to have a fast executing code or less consumption of memory.

⇒ Thus by optimizing the code the overall running time of the target program can be improved.

Code Generation :-

⇒ Code generation phase the target code gets generated.

⇒ The intermediate code instructions are translated into sequence of machine instructions.

Example:

```
MOV  rate, R1
MUL  #10.0, R1
MOV  count, R2
ADD  R2, R1
MOV  R1, total
```

Symbol table Management :-

⇒ To support these phases of compiler a symbol table is maintained. The task of symbol table is to store identifiers used in the program.

⇒ The symbol table also stores information about attributes of each identifier. The attributes of identifiers are usually its type, its scope, information about the storage allocated for it.

⇒ The symbol table also stores information about the subroutines used in the program.

It includes, Name of the subroutine
Number of arguments passed to it
Type of these arguments
Method of passing these arguments and return type.

⇒ Basically symbol table is a data structure used to store the information about identifiers.

⇒ The symbol table allows us to find the record for each identifier quickly and to store or retrieve data from that record efficiently.

Error detection and handling :-

⇒ In compilation, each phase detects errors. These errors must be reported to error handler whose tasks is to handle the errors.

⇒ Normally, the errors are reported in the form of message.

⇒ When the input characters from the input do not form the token, the lexical analyzer detects it as error. (lexical errors)

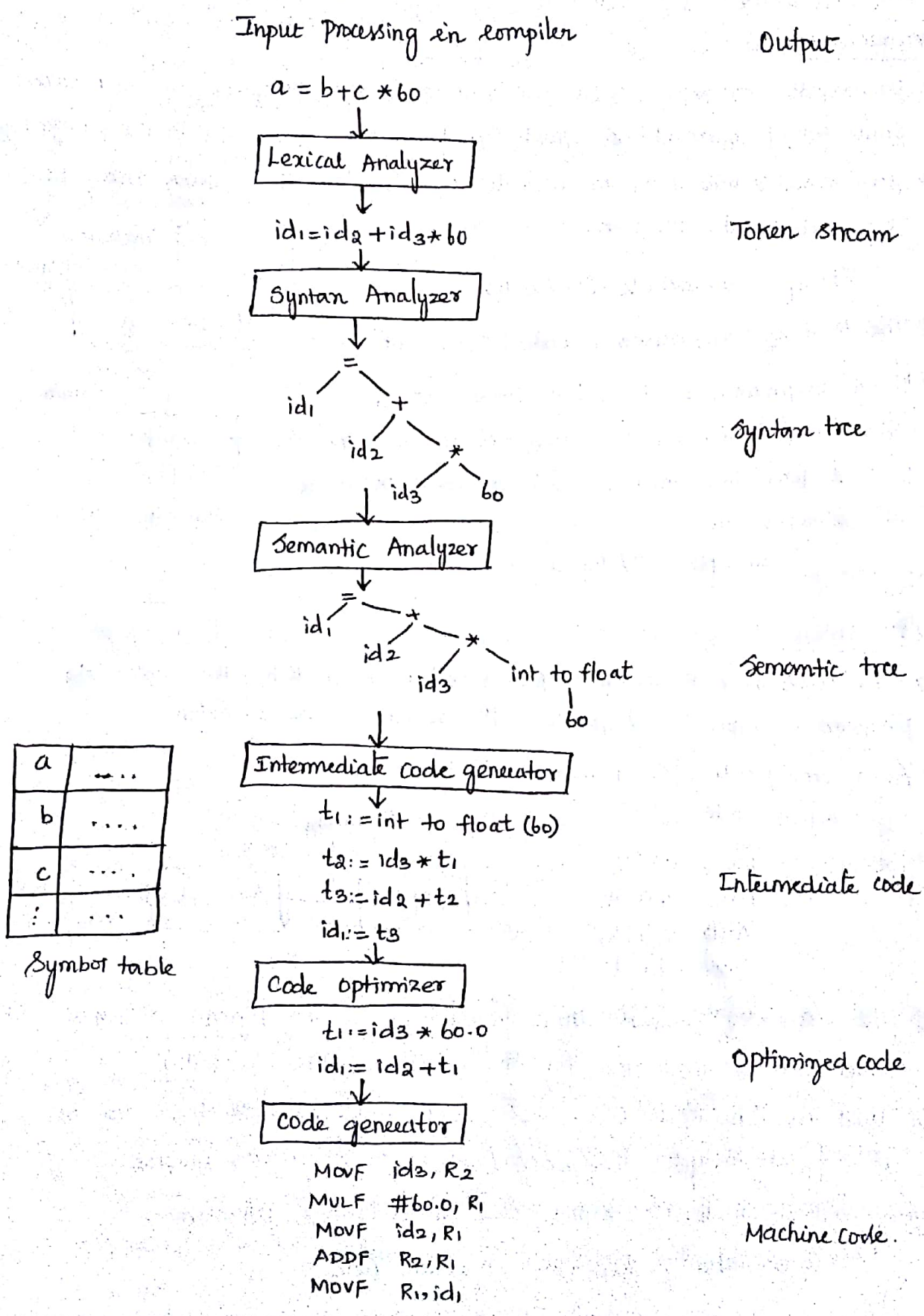
⇒ Large number of errors can be detected in syntax analysis phase.

- ✓ Errors in structure
- ✓ Missing operators
- ✓ Unbalanced parenthesis

} ⇒ Syntax errors

⇒ During semantic analysis; type mismatch kind of error is usually detected.

Example: Show how an input $a = b + c * 60$ get processed in compiler. Show that the output at each stage of compiler. Also show the contents of Symbol table.



Cousins of the compiler

Cousins of compiler means the content in which the compiler typically operates. They are preprocessor, assemblers, loaders and link editors.

1. **preprocessor:-**

⇒ preprocessors allows user to use macros in the program. Macro means some set of instructions which can be used repeatedly in the program.

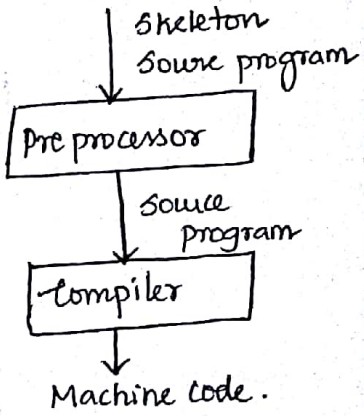
⇒ preprocessor allow users to include the header files which may be required by the program.

Example: #include <stdio.h>

⇒ The task of preprocessor is called file inclusion.

⇒ Macro preprocessor: Macro is a small set of instructions. Whenever in a program macro name is identified then that name is replaced by macro definition.

Example: #define PI 3.14

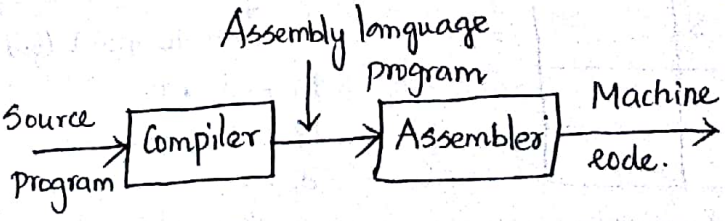


2. **Assemblers:-**

⇒ The assembler is a kind of translator which takes the assembly program as input and produces the machine code as output.

⇒ An assembly code is a mnemonic version of machine code.

Example:
```
MOV  a, R1
MUL  #5, R1
ADD  #7, R1
MOV  R1, b
```



⇒ The assembler converts these instructions in the binary language which can be understood by the machine. (Machine code)

⇒ This machine code is a relocatable machine code that can be passed directly to the loader/linker for execution purpose.

⇒ pass 1 assembler - complete scan of the input program.
pass 2 assembler - Relocatable machine code.

3. Loaders and Link Editors

⇒ Loader is a program which performs two functions loading and link editing.

⇒ Loading is the process in which the relocatable machine code is read and the relocatable addresses are altered. Then that code with altered instructions and data is placed in the memory at proper location.

⇒ The job of link editor is to make a single program from several files of relocatable machine codes.

Grouping of phases :-

⇒ Different phases of compiler can be grouped together to form a front end and back end.

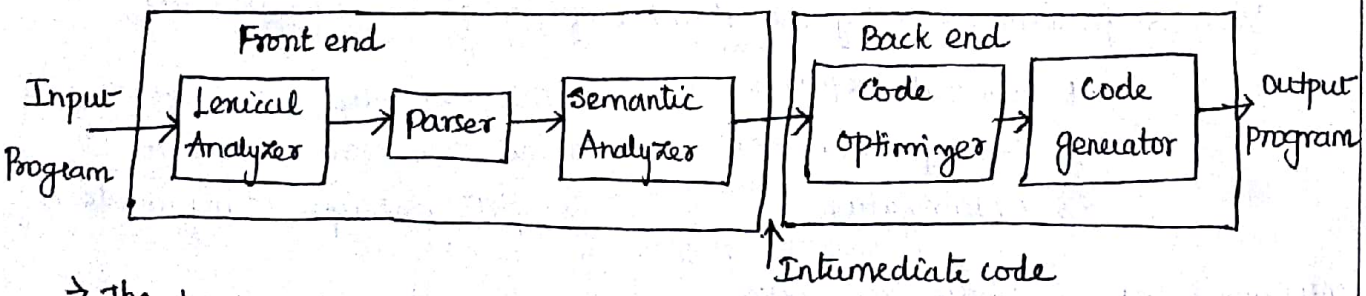
⇒ The front end consists of those phases that primarily dependent on the source language and independent on the target language.

⇒ The front end consists of analysis part. Typically it includes,

- Lexical Analysis, syntax analysis and semantic analysis.

⇒ The back end consists of those phases that are typically dependent upon the target language and independent on the source language.

⇒ It includes code generation and code optimization.



⇒ The front end and back end of the compiler is very much advantageous because of following reasons-

1. By keeping the same front end and attaching different back ends one can Produce a compiler for same source language on different machines.
2. By keeping different front ends and same back end one can compile several different languages on the same machine.

⇒ Machine Dependent phases → Code generation, Code optimization.

⇒ Machine Independent phases → Lexical Analysis, Syntax Analyzers, Semantic Analyzers.

Concept of pass:-

⇒ One complete scan of the source language is called pass. It includes reading and input file and writting to an output file.

⇒ Many phases can be grouped one pass. It is difficult to compile the source program into a single pass, because the program may have some forward reference

⇒ on the other hand, if we group several phases into one pass, we may be forced to keep the entire program in the memory. Therefore memory requirement may be large.

⇒ There may be a requirement of more than one pass in compilation process.

⇒ A typical arrangement for optimizing compiler is one pass for Scanning and parsing.

⇒ One pass for semantic analysis and third pass for code generation and target code optimization.

Factors affecting the number of passes:-

Various factors affecting the number of passes in compilers are,

1. Forward reference
2. Storage limitations
3. Optimization.

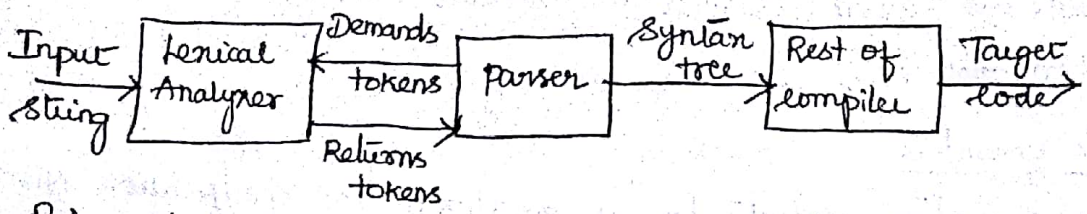
* The compiler can require more than one passes to complete subsequent information.

Difference between phase and pass

phase	pass
⇒ The process of compilation is carried out in various steps. These steps are referred as phases.	⇒ Various phases are logically grouped together to form a pass.
⇒ phases of compilation are lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code generation, code optimization.	⇒ process of compilation can be carried out in single pass or in multiple passes.

Lexical Analysis:-

⇒ Lexical Analyzer is the first phase of compiler. The lexical analyzer reads the input source program from left to right one character at a time and generate the sequence of tokens.



Role of lexical analyzer:-

1. It produces stream of tokens.
2. It eliminates blank and comments.
3. It generates symbol table which stores the information about identifiers, constants encountered in the input.
4. It keeps track of line numbers.
5. It reports the error encountered while generating the tokens.

Lexical errors:-

⇒ These types of errors can be detected during lexical analysis phase. Typical lexical phase errors are

- Spelling errors. Hence get incorrect tokens.
- Exceeding length of identifier or numeric constants.
- Appearance of illegal characters.

Error recovery action

1. If there are more than one pattern matching, then the lexical analyzer has to choose the longest lexeme matched.
2. Another rule is that the lexical rule written in LEX specification file must be ordered.

Compiler Construction Tools

⇒ Writing a compiler is tedious and time consuming task. There are some specialized tools for helping in implementation of various phases of compilers.

⇒ These tools are called compiler construction tools. Various compiler construction tools are,

1. Scanner generators:
   ⇒ These generators generate lexical analyzers. The specification given to these generators are in the form of regular expressions.
   ⇒ The UNIX has utility for a scanner generate called LEX. The specification given to the LEX consists of regular expressions for representing various tokens.
2. Parser generators:
   ⇒ These produce the syntax analyzer. The specification given to these generators is given in the form of context free grammar.
   ⇒ Typically UNIX has a tool called YAAC which is a parser generator.
3. Syntax-directed translation engines:-
   ⇒ In this tool the parse tree is scanned completely to generate an intermediate code.
   ⇒ The translation is done for each node of the tree.
4. Automatic code generator:-
   ⇒ These generators take an intermediate code as input and converts each rule of intermediate language into equivalent machine language.
   ⇒ The template matching technique is used. The intermediate code statements are replaced templates that represent the corresponding sequence of machine instructions.
5. Data flow Engines:-
   ⇒ The data flow analysis is required to perform good code optimization. The data flow engines are basically useful in code optimization.

Tokens, Patterns, Lexemes.

Tokens ⇒ It describes the class or category of input string.
Example: Identifiers, keywords, constants.

Patterns ⇒ Set of rules that describes the token.

Lexemes ⇒ Sequence of characters in the source program that are matched with the pattern of token.
Example: int, i, num, ans etc..

Issues in lexical analyzer:-

1. The lexical analysis and syntax analysis are separated out for simplifying the task of one or other phases. This separation reduces the burden on parsing phase.
2. Compiler efficiency gets increased if the lexical analyzer is separated. This is because a large amount of time is spent on reading the source file.
3. Input alphabet peculiarities and other device specific anomalies can be restricted to the lexical analyzer. the representation of non standard symbol can be isolated in lexical analyzer.

Input Buffering:-

⇒ The lexical analyzer scans the input string from left to right one character at a time. It uses two pointers begin-ptr(bp) and foward-ptr (fp) to keep track of the portion of the input scanned.

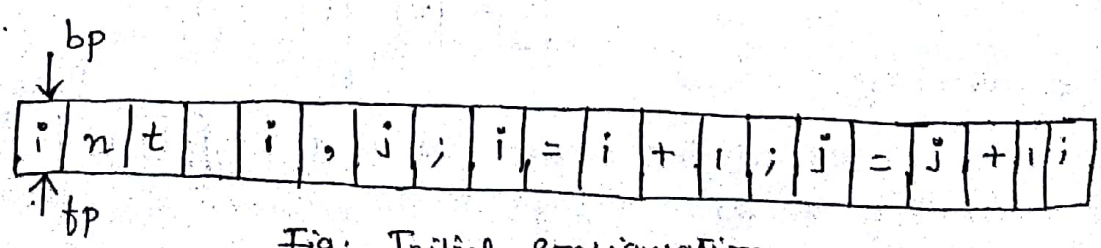
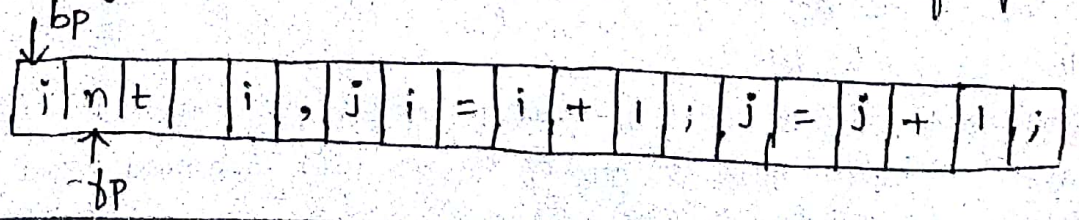


Fig: Initial configuration.

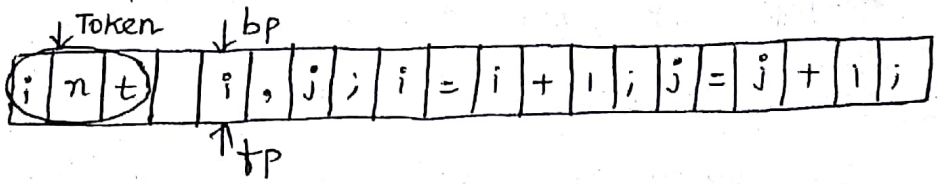
⇒ Initially both the pointers point to the 1st character of input string



⇒ The forward_ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme.

⇒ Here, forward_ptr (fp) encounters a blank space the lexeme "int" is identified.

⇒ The fp will be moved ahead at white space. When fp encounters white space, it ignore and moves ahead. Then both the begin_ptr(bp) and forward_ptr (fp) are set at next token i.



⇒ The input character is thus read from secondary storage. But reading in this way from secondary storage is costly. Hence buffering technique is used.

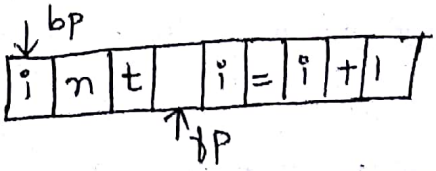
⇒ A block of data is first read into a buffer and then scanned by lexical analyzer. There are two methods used in this context.

- one buffer scheme and two buffer scheme.

1. one buffer scheme:-

⇒ In this one buffer scheme, only one buffer is used to store the input string.

⇒ But the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first part of lexeme

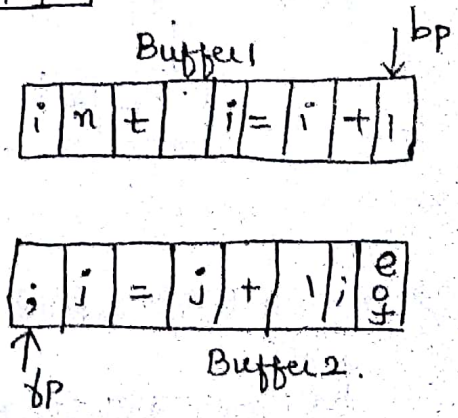


2. Two buffer scheme:-

⇒ In this method two buffers are used to store the input string.

⇒ The first buffer and second buffer are scanned alternately.

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



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

Specification of tokens:

⇒ To Specify the tokens regular expressions are used. When a pattern is matched by some regular expression then token can be recognized.

⇒ Regular expressions are mathematical symbolisms which describe the set of strings of specific language. It provides convenient and useful notation for representing tokens.

Definition of Regular expression

1. ϵ is a regular expression that denotes the set containing empty string.
2. If R_1 and R_2 are RE then $R = R_1 | R_2$ also RE (Union operation)
3. If R_1 and R_2 are RE then $R = R_1 . R_2$ also RE (Concatenation operation)
4. If R_1 is a RE then $R = R_1^*$ also regular expression (Kleen closure)

A language denoted by regular expressions is said to be a regular set or a regular language.

Example:

1. RE for a language containing the strings of length two over $\Sigma = \{0,1\}$.

   Solution: RE = (0+1)(0+1)

2. RE for a language containing strings which end with "abb" over $\Sigma = \{a,b\}$

   Solution: $(a+b)^*abb$

3. RE for recognizing identifier

   Solution: RE = letter (letter + digit)*

   Where letter = (A, B, ... Z, a, b, ... z) and digit = (0, 1, 2, ... 9)

4. RE for the language containing all the strings with any number of 'a's and 'b's Solution: RE = (a+b)*

5. RE for the language accepting all the strings which are ending with 00 over the set $\Sigma = \{0,1\}$

   Solution: RE = $(0+1)^*00$

6. RE to represent the valid date format in mm-dd-4444 format

   Solution: [1-12] - [1-31] - [0-9] [0-9] [0-9] [0-9]

Design of lexical analysis (LEX):

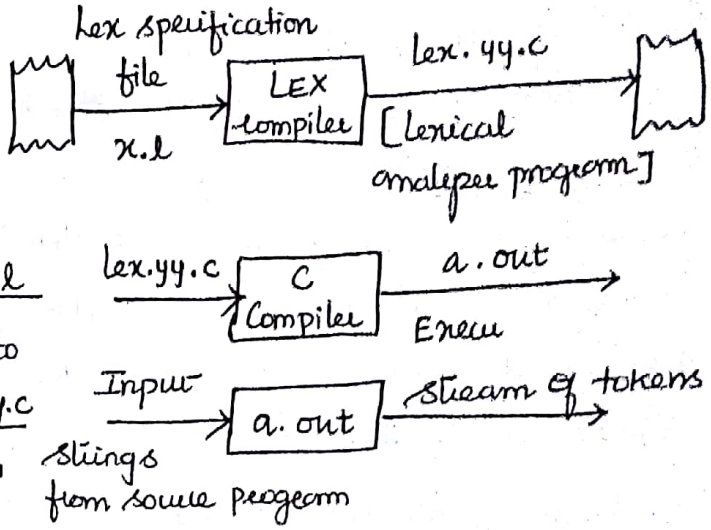
Introduction - LEX

⇒ LEX is a unix utility which generates lexical analyzer. lex scans the source program in order to get the stream of tokens and these tokens are related together.

⇒ The LEX specification file can be created using the extension .l. For example the specification file can be x.l

⇒ This x.l file is then given to LEX compiler to produce lex.yy.c

⇒ This lex.yy.c is a C program which is actually a lexical analyzer program.



⇒ The LEX specification file stores the regular expressions for the tokens and the lex.yy.c file consists of the tabular representation of the transition diagrams constructed for the regular expression.

⇒ Finally the compiler compiles this generated lex.yy.c and produces an object program a.out. When some input stream is given to a.out then sequence of tokens gets generated.

Structure of LEX

LEX program consists of three parts

1. Declaration section
   - ↳ declaration of variable, constants
   - ↳ regular definitions can also be written.
2. Rule section
   - ↳ consists of regular expressions with associated actions.
3. procedure section
   - ↳ All the required procedures are defined.

```
%{
   Declaration section
%}
%%
   Rule section
%%
Auxiliary procedure section.
```

Following commands are used to run the lex program x.l

```
$ lex. x.l        → Generates lex.yy.c
$ cc lex.yy.c     → compiles lex.yy.c
$ ./a.out         → Runs an executable file.
```

Example: LEX program for word counting.

```
%{
   int char_cnt=0, word_cnt=0, line_cnt=0;   // Declaration section
%}
   word [^ \t\n]+
%%
  {word}  {word_cnt++; char_cnt+=yyleng;}   // Rule section
  \n {char_cnt++; line_cnt++;}
   .{char_cnt++;}
%%
main()
{                                           // procedure section.
    yylex();
    Printf("\n The character count = %d", char_cnt);
    Printf("\n The word count = %d", word_cnt);
    Printf("\n The line count = %d", line_cnt);
    Printf("\n");
}
int yywrap()  // This function is called when end of file is
{                encountered.
  return 1;
}
```

Finite Automata

A finite automata is a collection of 5-tuple $(Q, \Sigma, \delta, q_0, F)$

Where, Q - finite set of states
Σ - input alphabet
q_0 - initial state
δ - Transition function.
F - set of final states.

Types of Automata

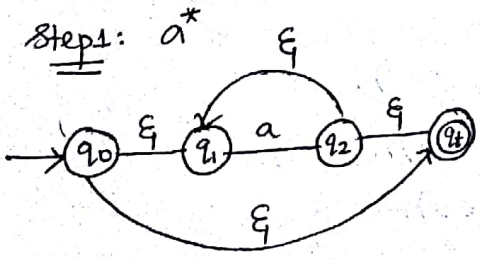
1. Deterministic Finite Automata ⇒ each move is uniquely determined on current input and current state.
2. Non-deterministic Finite Automata ⇒ Two or more path from current state to next state for a input.

Conversion of regular expression of NDFA - Thompson's construction

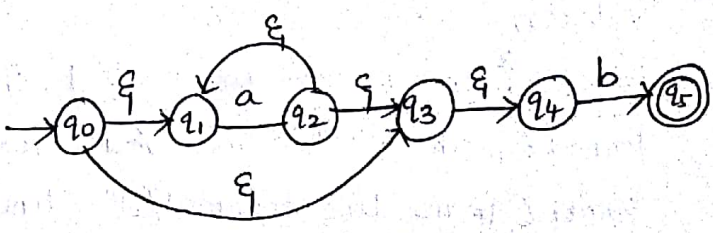
Pbm 1: construct NFA equivalent to $r = a^* b$

Solution:

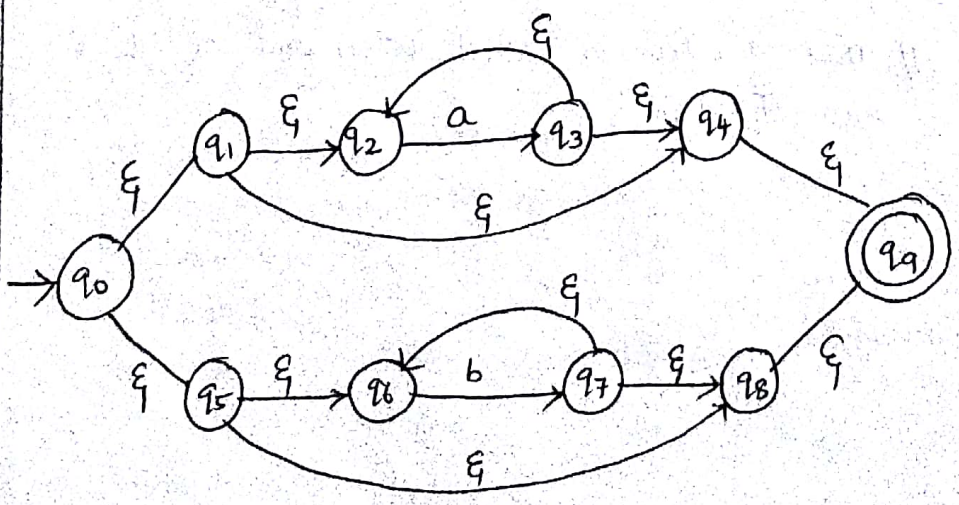
Step1: a^*



Step2: $r = a^* b$



Pbm 2: Draw NFA for $a^* | b^*$



Direct method for conversion of RE to FA

Step1: Design a transition diagram for given RE using NFA with ε-moves.

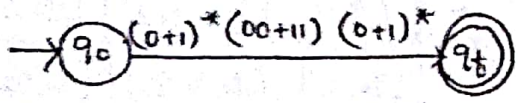
Step2: Convert this NFA with ε to NFA without ε.

Step3: Convert the obtained NFA to equivalent DFA.

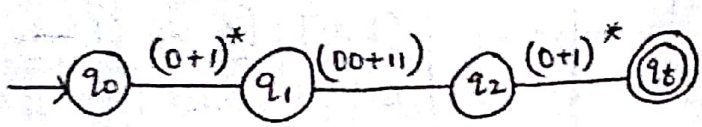
Prm: Construct DFA to accept the regular expression $(0+1)^*(00+11)(0+1)^*$

Solution: $RE = (0+1)^*(00+11)(0+1)^*$

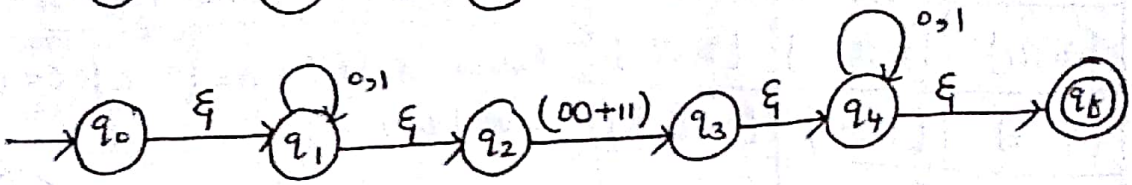
Step1:



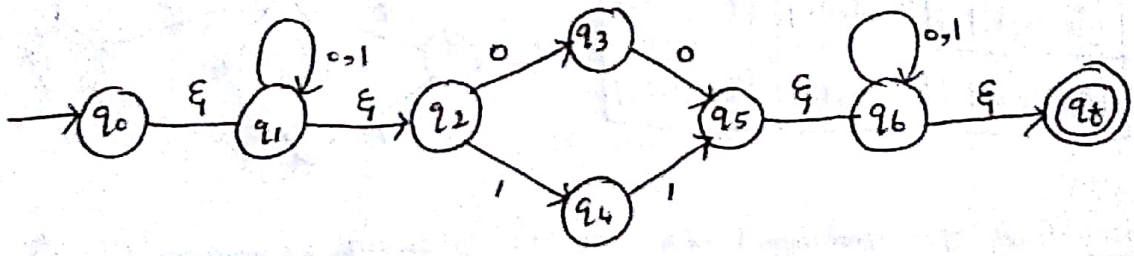
Step2:



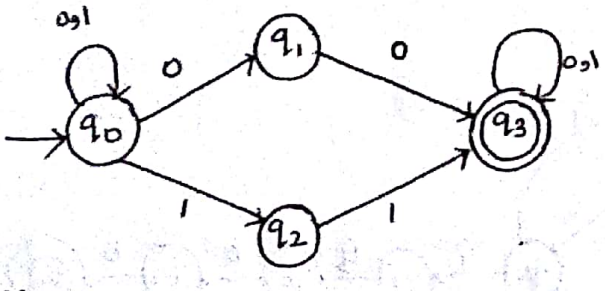
Step3:



Step4:



Step5: Eliminate ε-transitions,



Step6: Transition Table

Q \ Σ	0	1
q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$
q_1	q_3	-
q_2	-	q_3
q_3	q_3	q_3

Step 7: From the transition table

$\delta([q_0,q_1],0) = [q_0,q_1,q_3]$ → New State

$\delta([q_0,q_1],1) = [q_0,q_2]$ → New State

Now,

$\delta([q_0,q_1,q_3],0) = [q_0,q_1,q_3]$

$\delta([q_0,q_1,q_3],1) = [q_0,q_2,q_3]$ → New State

$\delta([q_0,q_2],0) = [q_0,q_1]$

$\delta([q_0,q_2],1) = [q_0,q_2,q_3]$

Now,

$$\delta([q_0,q_2,q_3],0) = [q_0,q_1,q_3]$$

$$\delta([q_0,q_2,q_3],1) = [q_0,q_1,q_3]$$

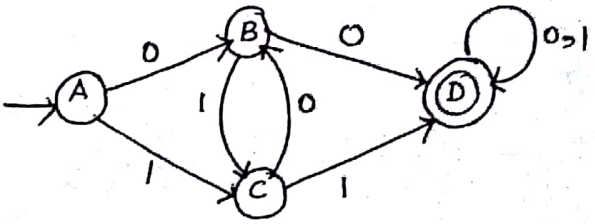
Transition table

	Q \ Σ	0	1
→	$[q_0]$	$[q_0,q_1]$	$[q_0,q_2]$
	$[q_1]$	q_3	-
	$[q_2]$	-	q_3
*	$[q_3]$	q_3	q_3
	$[q_0,q_1]$	$[q_0,q_1,q_3]$	$[q_0,q_2]$
	$[q_0,q_2]$	$[q_0,q_1]$	$[q_0,q_2,q_3]$
*	$[q_0,q_1,q_3]$	$[q_0,q_1,q_3]$	$[q_0,q_2,q_3]$
*	$[q_0,q_2,q_3]$	$[q_0,q_1,q_3]$	$[q_0,q_2,q_3]$

Minimized DFA

Q \ Σ	0	1
→ $[q_0]$	$[q_0,q_1]$	$[q_0,q_2]$
$[q_0,q_1]$	$[q_0,q_1,q_3]$	$[q_0,q_2]$
$[q_0,q_2]$	$[q_0,q_1]$	$[q_0,q_2,q_3]$ → $[q_0,q_1,q_3]$
$[q_0,q_1,q_3]$	$[q_0,q_1,q_3]$	$[q_0,q_1,q_3]$

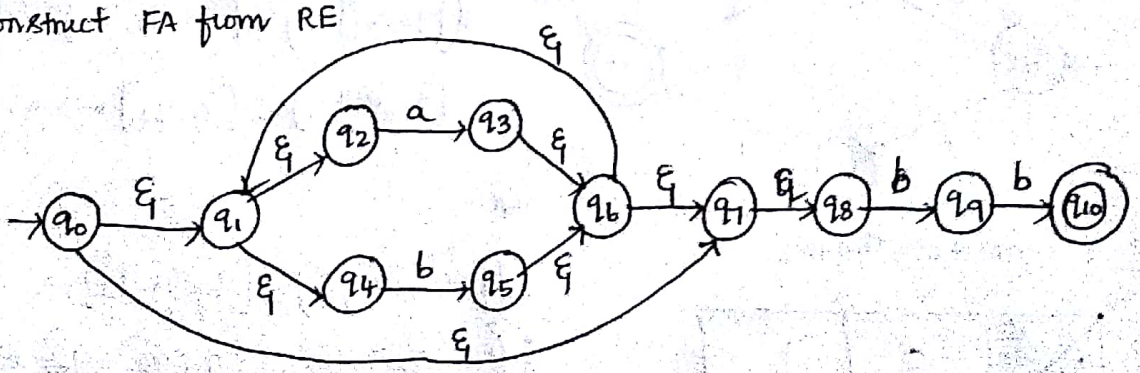
Assume $A=[q_0]$, $B=[q_0,q_1]$ $C=[q_0,q_2]$ $D=[q_0,q_1,q_3]$



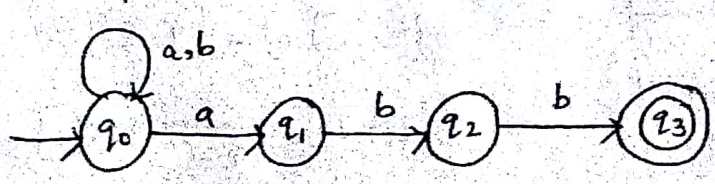
Prob: Give the minimized DFA for the following expression $(a|b)^*abb$.

Solution: $RE = (a|b)^*abb$

Step1: Construct FA from RE



Step2: Eliminate ε-transition.



Transition table:

Q \ Σ	a	b
q_0	q_0, q_1	q_0
q_1	ϕ	q_2
q_2	ϕ	q_3
q_3	ϕ	ϕ

From the table,

$\delta([q_0,q_1], a) = [q_0, q_1]$

$\delta([q_0,q_1], b) = [q_0, q_2]$ → New state

$\delta([q_0,q_2], a) = [q_0, q_1]$

$\delta([q_0,q_2], b) = [q_0, q_3]$ → New state

$\delta([q_0,q_3], a) = [q_0, q_1]$

$\delta([q_0,q_3], b) = [q_0]$

Equivalent DFA table

	Q \ Σ	a	b
→	q_0	$[q_0,q_1]$	q_0
	q_1	ϕ	q_2
	q_2	ϕ	q_3
*	q_3	ϕ	ϕ
	$[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_2]$
	$[q_0,q_2]$	$[q_0,q_1]$	$[q_0,q_3]$
*	$[q_0,q_3]$	$[q_0,q_1]$	q_0

Minimized DFA,

	Q \ Σ	a	b
→	q_0	$[q_0,q_1]$	q_0
	$[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_2]$
	$[q_0,q_2]$	$[q_0,q_1]$	$[q_0,q_3]$
*	$[q_0,q_3]$	$[q_0,q_1]$	q_0

Assume,

$q_0 = A$

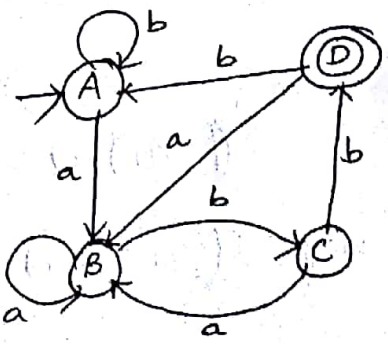
$[q_0,q_1] = B$

$[q_0,q_2] = C$

$[q_0,q_3] = D$

Q \ Σ	a	b
→ A	B	A
B	B	C
C	A	D
* D	B	A

Transition diagram,



Prob: Prove that the following two RE are equivalent by showing that the minimum state DFA's are same

(i) $(a/b)^*$ (ii) $(a^*/b^*)^*$

Prob: Illustrate Thomson's construction by drawing a NFA for the RE $(a/b)^*a$. Convert this NFA to DFA using Subset construction.

Way

Prob: Construct the minimized DFA for RE $= (0+1)^*(0+1)10$

Conversion of NDFA to DFA.

Prm: let $M = (\{q_0, q_1\}, \{0,1\}, \delta, q_0, \{q_1\})$ be NFA where $\delta(q_0,0) = \{q_0, q_1\}$ $\delta(q_0,1) = \{q_1\}$, $\delta(q_1,0) = \phi$, $\delta(q_1,1) = \{q_0, q_1\}$ construct its equivalent DFA.

Solution: Given, $M = (\{q_0, q_1\}, \{0,1\}, \delta, q_0, \{q_1\})$ be NFA

$\therefore$ DFA $M' = (Q', \Sigma, \delta', q_0', F')$

$\delta(q_0,0) = \{q_0, q_1\}$
$\delta(q_0,1) = \{q_1\}$
$\delta(q_1,0) = \phi$
$\delta(q_1,1) = \{q_0, q_1\}$

Transition table for NFA

	Q \ Σ	0	1
→	q_0	$\{q_0, q_1\}$	$\{q_1\}$
*	q_1	ϕ	$\{q_0, q_1\}$

compute transition from the state $[q_0, q_1]$

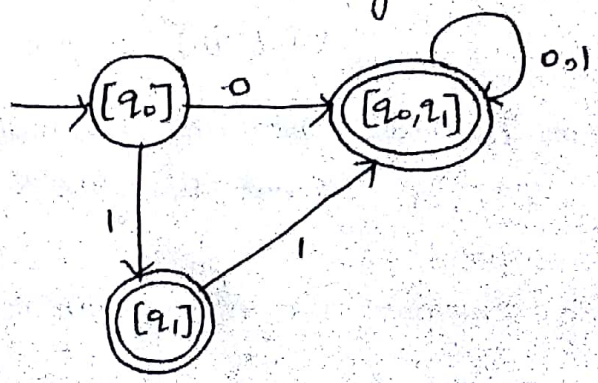
$$\delta(\{q_0, q_1\}, 0) = \delta(q_0,0) \cup \delta(q_1,0)$$
$$= \{q_0, q_1\} \cup \phi$$
$$= \{q_0, q_1\}$$
$$\therefore \delta'([q_0, q_1], 0) = [q_0, q_1]$$

Now $\delta(\{q_0, q_1\}, 1) = \delta(q_0,1) \cup \delta(q_1,1)$
$$= \{q_1\} \cup \{q_0, q_1\}$$
$$= \{q_0, q_1\}$$
$$\therefore \delta'([q_0, q_1], 1) = [q_0, q_1]$$

$\therefore$ Transition table of DFA

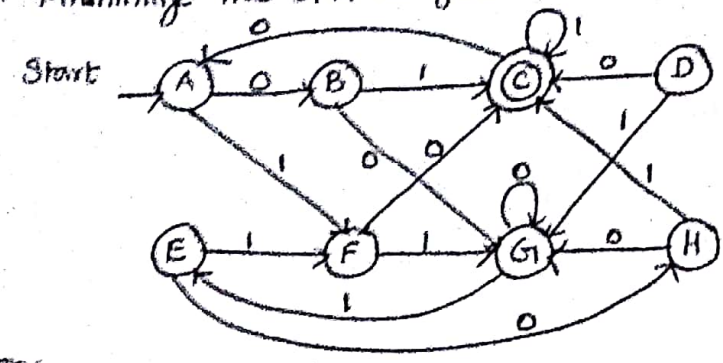
	Q \ Σ	0	1
→	$[q_0]$	$[q_0, q_1]$	$[q_1]$
*	$[q_1]$	ϕ	$[q_0, q_1]$
*	$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1]$

Transition diagram.



Minimization of NDFA

Prob 1: Minimize the DFA as given below,



Solution:

Transition table

	0	1
→ A	B	F
B	G1	C
(C)	A	C
D	C	G1
E	H	F
F	C	G1
G1	G1	E
H	G1	C

Find the equivalent state

Consider pair (B, H)

$\delta(B,0) = G_1$ | $\delta(B,1) = C$
$\delta(H,0) = G_1$ | $\delta(H,1) = C$

∴ (B, H) are equivalent

Since (B, H) are equivalent, consider the pair (A, E)

$\delta(A,0) = B$ (or) H | $\delta(A,1) = F$
$\delta(E,0) = H$ (or) B | $\delta(E,1) = F$

∴ (A, E) are equivalent

Consider the pair (D, F)

$\delta(D,0) = C$ | $\delta(D,1) = G_1$
$\delta(F,0) = C$ | $\delta(F,1) = G_1$

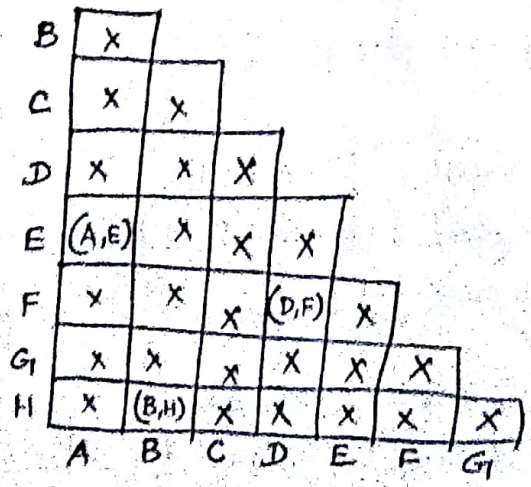
∴ (D, F) are equivalent

∴ The equivalent pairs are (A, E), (B, H) and (D, F)

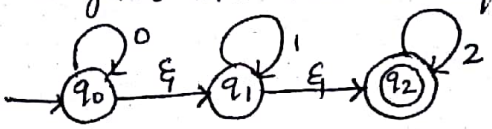
The minimized DFA will be,

Q \ Σ	0	1
→ A	B	D
B	G1	C
(C)	A	C
D	C	G1
G1	G1	A

Table filling,



Prob 2: Convert the given NFA into its equivalent DFA.



Solution:

Step1: Find ε-closure of each state

ε-closure $(q_0) = \{q_0, q_1, q_2\}$

ε-closure $(q_1) = \{q_1, q_2\}$

ε-closure $(q_2) = \{q_2\}$

Step2: Find δ' transition

Let ε-closure $(q_0) = \{q_0, q_1, q_2\}$ be state A.

$\delta'(A,0) = \varepsilon\text{-closure}\{\delta(q_0,q_1,q_2),0)\}$

$= \varepsilon\text{-closure}\{\delta(q_0,0) \cup \delta(q_1,0) \cup \delta(q_2,0)\}$

$= \varepsilon\text{-closure}\{q_0\}$

$= \{q_0,q_1,q_2\} \Rightarrow$ state A

$\delta'(A,1) = \varepsilon\text{-closure}\{\delta(q_0,q_1,q_2),1\}$

$= \varepsilon\text{-closure}\{\delta(q_0,1) \cup \delta(q_1,1) \cup \delta(q_2,1)\}$

$= \varepsilon\text{-closure}\{q_1\}$

$= \{q_1,q_2\} \Rightarrow$ state B

$\delta'(A,2) = \varepsilon\text{-closure}\{\delta(q_0,q_1,q_2),2\}$

$= \varepsilon\text{-closure}\{\delta(q_0,2) \cup \delta(q_1,2) \cup \delta(q_2,2)\}$

$= \varepsilon\text{-closure}\{q_2\}$

$= \{q_2\} \Rightarrow$ state C

Now,

$\delta'(B,0) = \varepsilon\text{-closure}\{\delta(q_1,q_2),0\}$

$= \varepsilon\text{-closure}(\delta(q_1,0) \cup \delta(q_2,0)\}$

$= \varepsilon\text{-closure}\{\phi\}$

$= \phi$

$\delta'(B,1) = \varepsilon\text{-closure}\{\delta(q_1,q_2),1\}$

$= \varepsilon\text{-closure}\{\delta(q_1,1) \cup \delta(q_2,1)\}$

$= \varepsilon\text{-closure}\{q_1\}$

$= \{q_1,q_2\} \Rightarrow$ state B

$\delta'(B,2) = \varepsilon\text{-closure}\{\delta(q_1,q_2),2\}$

$= \varepsilon\text{-closure}\{\delta(q_1,2) \cup \delta(q_2,2)\}$

$= \varepsilon\text{-closure}\{q_2\}$

$= \{q_2\} \Rightarrow$ state C

Now,

$\delta'(C,0) = \varepsilon\text{-closure}\{\delta(q_2,0)\}$

$= \varepsilon\text{-closure}\{\phi\}$

$= \phi$

$\delta'(C,1) = \varepsilon\text{-closure}\{\delta(q_2,1)\}$

$= \varepsilon\text{-closure}(\phi)$

$= \phi$

$\delta'(C,2) = \varepsilon\text{-closure}\{\delta(q_2,2)\}$

$= q_2$

$\therefore \delta'(A,0) = A \quad | \quad \delta'(B,0) = \phi$

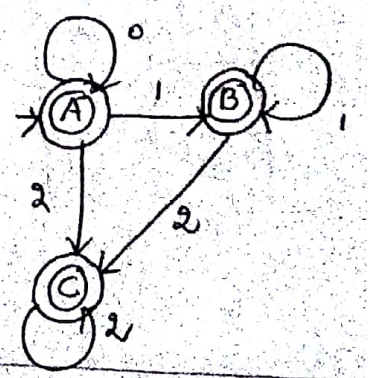
$\delta'(A,1) = B \quad | \quad \delta'(B,1) = B$

$\delta'(A,2) = C \quad | \quad \delta'(B,2) = C$

$\delta'(C,0) = \phi$

$\delta'(C,1) = \phi$

$\delta'(C,2) = q_2$



Prob2: using table filling method minimize the following DFA. Draw the transition diagram of resulting DFA.

	0	1
→ A	B	E
B	C	F
* C	D	H
D	E	H
E	F	I
* F	G1	B
G1	H	B
H	I	C
* I	A	E

Solution: Here final states are C, F, I

Since C, F, I are final states

consider the pair (E,H)

$\delta(E,0) = F$ | $\delta(E,1) = I$

$\delta(H,0) = I$ | $\delta(H,1) = C$

∴ (E,H) are equivalent state

consider the pair (B,E)

$\delta(B,0) = C$ | $\delta(B,1) = F$

$\delta(E,0) = F$ | $\delta(E,1) = E$

(B,E) are equivalent state

consider the pair (B,H)

$\delta(B,0) = C$ | $\delta(B,1) = F$

$\delta(H,0) = I$ | $\delta(H,1) = C$

∴ (B,H) are equivalent state.

Since (B,H) and (B,E) is equivalent

consider the pair (A,G1)

$\delta(A,0) = B(or)H$ | $\delta(A,1) = E$ or B

$\delta(G1,0) = H(or)B$ | $\delta(G1,1) = B$ or E

∴ (A,G1) is equivalent state.

Since (B,E) and (E,H) are equivalent

consider the pair (A,D)

$\delta(A,0) = B$ or E | $\delta(A,1) = E$ or H

$\delta(D,0) = E$ or B | $\delta(D,1) = H$ or E

∴ (A,D) is equivalent state

Since (E,H) and (B,H) are equivalent

consider the pair (D,G1)

$\delta(D,0) = E$ or H | $\delta(D,1) = H$ or B

$\delta(G1,0) = H$ or E | $\delta(G1,1) = B$ or H

∴ (D,G1) is equivalent state

Since (A,D) and (E,H) are equivalent

consider the pair (C,I)

$\delta(C,0) = D$ or A | $\delta(C,1) = H$ or E

$\delta(I,0) = A$ or B | $\delta(I,1) = E$ or H

∴ (C,I) is equivalent.

Since (D,G) and (B,H) are equivalent

consider the pair (C,F)

$\delta(C,0) = D$ or G | $\delta(C,1) = H$ or B

$\delta(F,0) = G$ or D | $\delta(F,1) = B$ or H

∴ (C,F) is equivalent

Since (A,G1) and (B,E) are equivalent

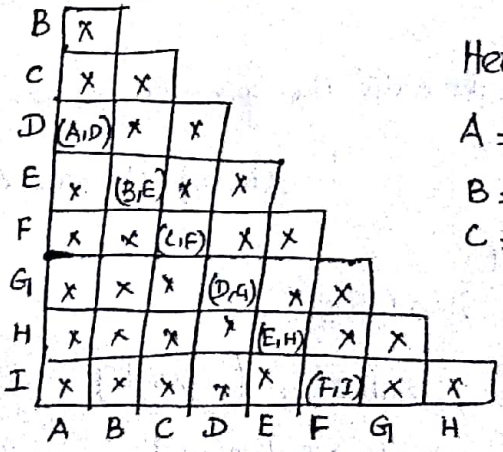
consider the pair (F,I)

$\delta(F,0) = G1$ or A | $\delta(F,1) = B$ or E

$\delta(I,0) = A$ or $G1$ | $\delta(I,1) = E$ or B

∴ (F,I) is equivalent state

Table filling



Here

A = G1 = D

B = H = E

C = I = F

Minimized DFA

Q \ Σ	0	1
A	B	B
B	C	C
→ C	A	B

Derivation, Parse tree and Ambiguity

Derivations ⇒ Use the production from head to body (ie) from start symbol expanding till reaches the given string.

Two types of derivations are,

(i) Left most Derivation (LMD)

(ii) Right most Derivation (RMD)

⇒ LMD is a derivation in which the leftmost nonterminal is replaced first from the sentential form. (ie) $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$, then α is left sentential form.

⇒ RMD is a derivation in which rightmost nonterminal is replaced from the sentential form (ie) $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha$, then α is right sentential form.

Parse tree (Derivation tree)

⇒ It is a graphical representation for the derivation of the given production rule for a given CFG.

Properties:

⇒ Root node is always a node indicating start symbol.

⇒ Derivation is read from left to right.

⇒ Leaf node is always terminal nodes.

⇒ Interior node are always non-terminal nodes.

Example:

Consider the grammar G has the production.

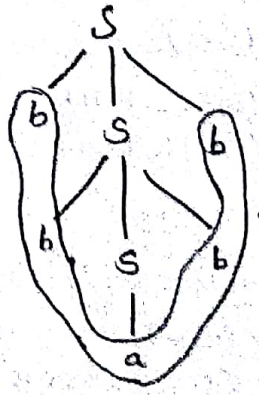
$S \rightarrow bSb \mid a \mid b$ and string "bbabb"

Derivation:

$S \rightarrow bSb$

$\rightarrow bbSbb$ [$S \rightarrow bSb$]

$\rightarrow bbabb$ [$S \rightarrow b$]



⇒ bbabb

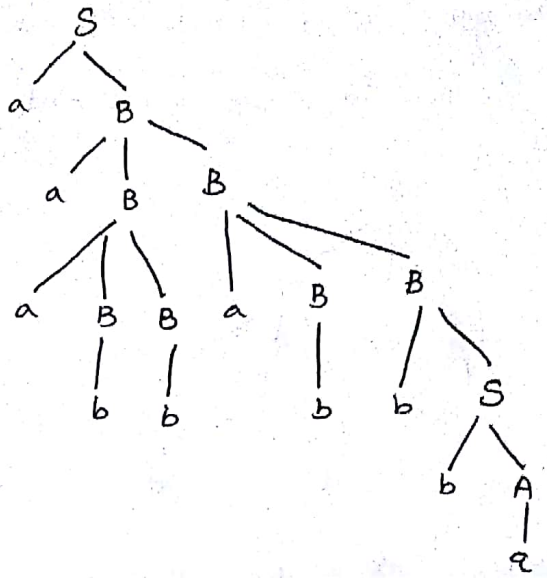
Problems: Construct the derivation tree for the string "aaabbabbba" using LMD and RMD using $S \to aB|bA$, $A \to a|aS|bAA$, $B \to b|bS|aBB$.

Solution:

LMD

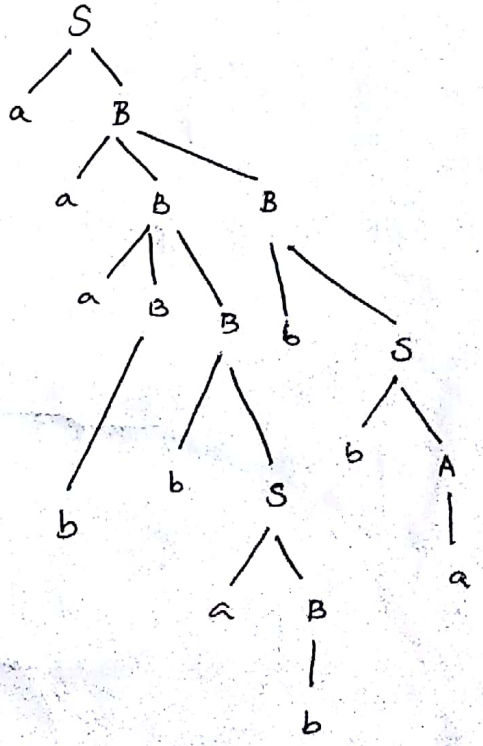
$S \Rightarrow aB$
$\Rightarrow aaBB$ [B → BB]
$\Rightarrow aaaBBB$ [B → aBB]
$\Rightarrow aaabBB$ [B → b]
$\Rightarrow aaabbB$ [B → b]
$\Rightarrow aaabbaBB$ [B → aBB]
$\Rightarrow aaabbabB$ [B → b]
$\Rightarrow aaabbabbS$ [B → bS]
$\Rightarrow aaabbabbbA$ [S → bA]
$\Rightarrow aaabbabbba$ [A → a]

Parse tree



RMD

$S \Rightarrow aB$
$\Rightarrow aaBB$ [B → aBB]
$\Rightarrow aaBbS$ [B → bS]
$\Rightarrow aaBbbA$ [S → bA]
$\Rightarrow aaBbbba$ [A → a]
$\Rightarrow aaaBBbbba$ [B → aBB]
$\Rightarrow aaaBbSbbba$ [B → bS]
$\Rightarrow aaaBbaBbbba$ [S → aB]
$\Rightarrow aaaBbabbbba$ [B → b]
$\Rightarrow aaabbabbbba$ [B → b]

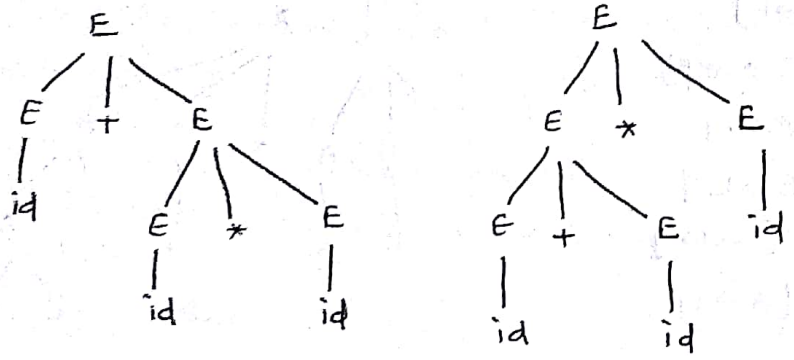


Ambiguity:

If there exists more than one parse tree for a given grammar, that means there could be more than one leftmost or rightmost derivation possible and then that grammar is said to be ambiguous grammar.

Example: $E \rightarrow E+E \mid E*E \mid id$

Then for string id + id * id



Problem: Show that following grammar is ambiguous.

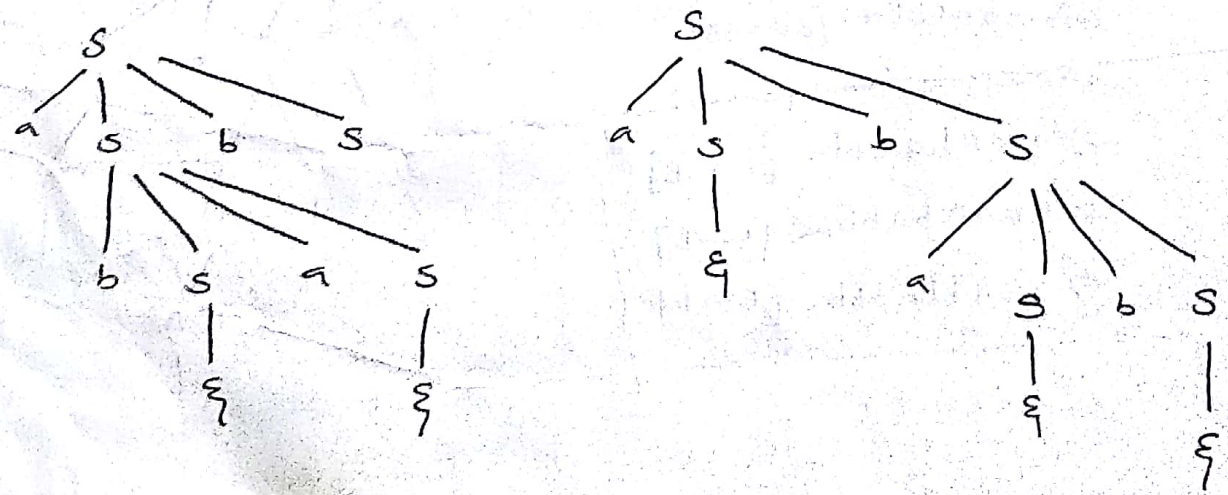
$S \rightarrow a\,S\,b\,S$

$S \rightarrow b\,S\,a\,S$

$S \rightarrow \varepsilon$

Solution:

Consider the string "abab"



From the above parse tree, we can conclude the given grammar is ambiguous.