United College of Engineering and Research,Naini Prayagraj

Computer Science and Engineering Department

Compiler Design

Semester 5th

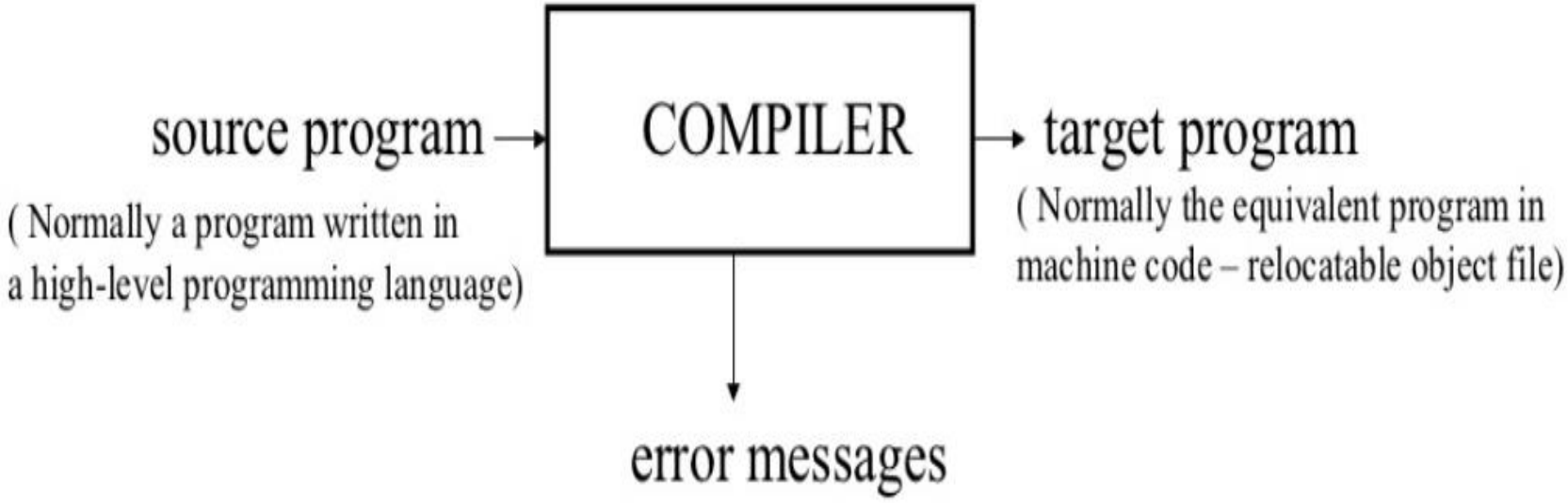
Prepared By

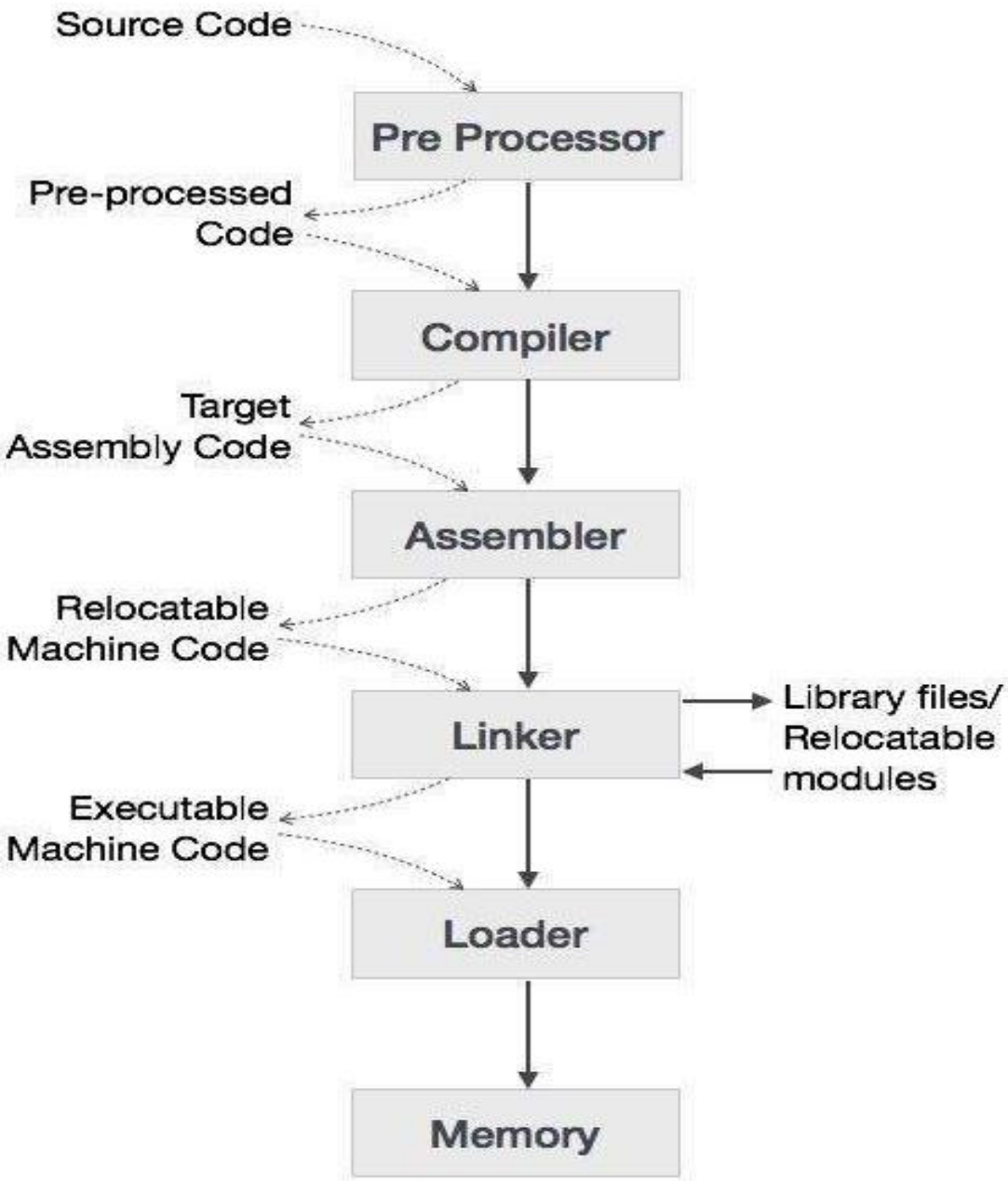
Prabhat Shukla

Lecture 1

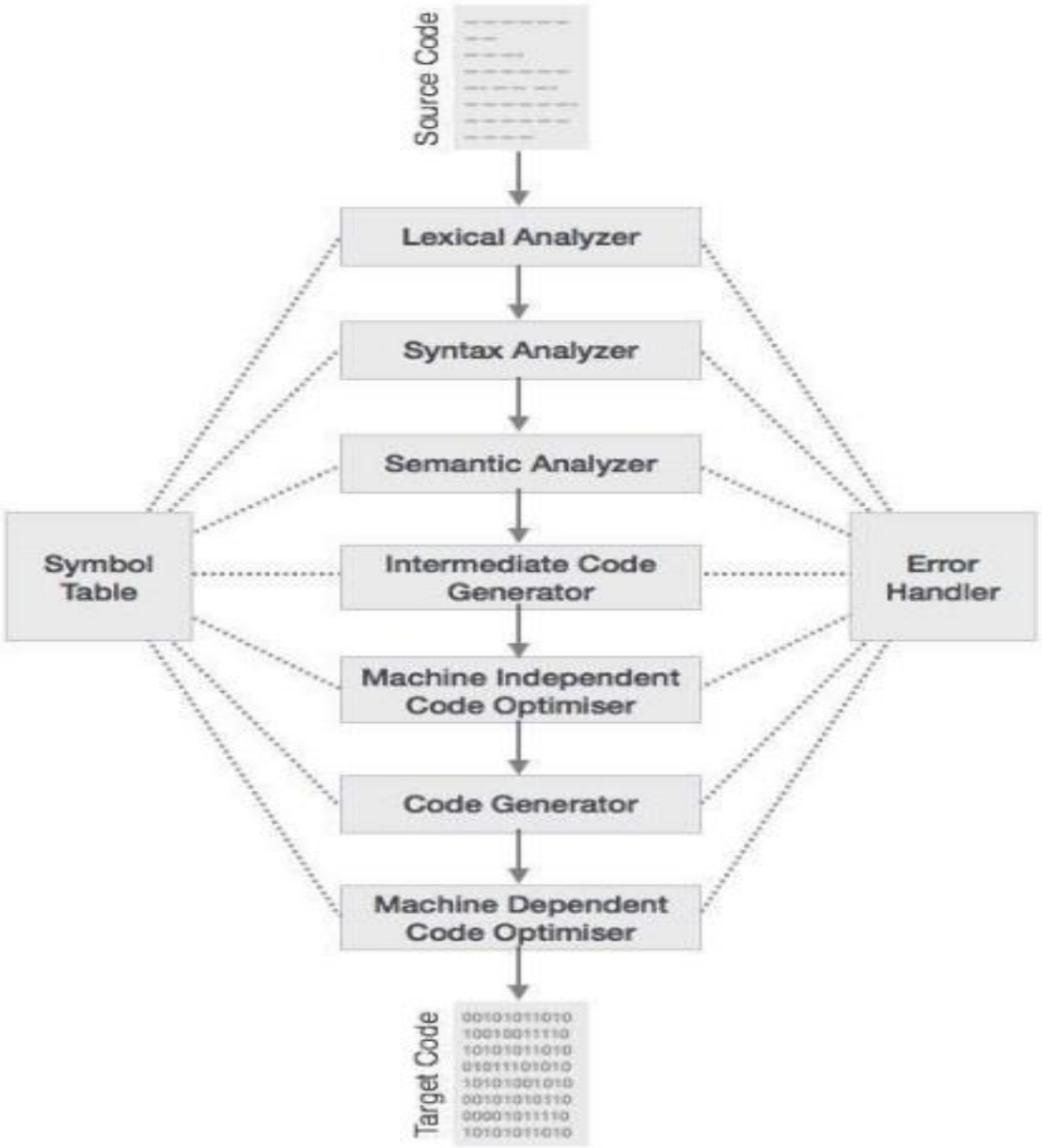
COMPILERS

- A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.

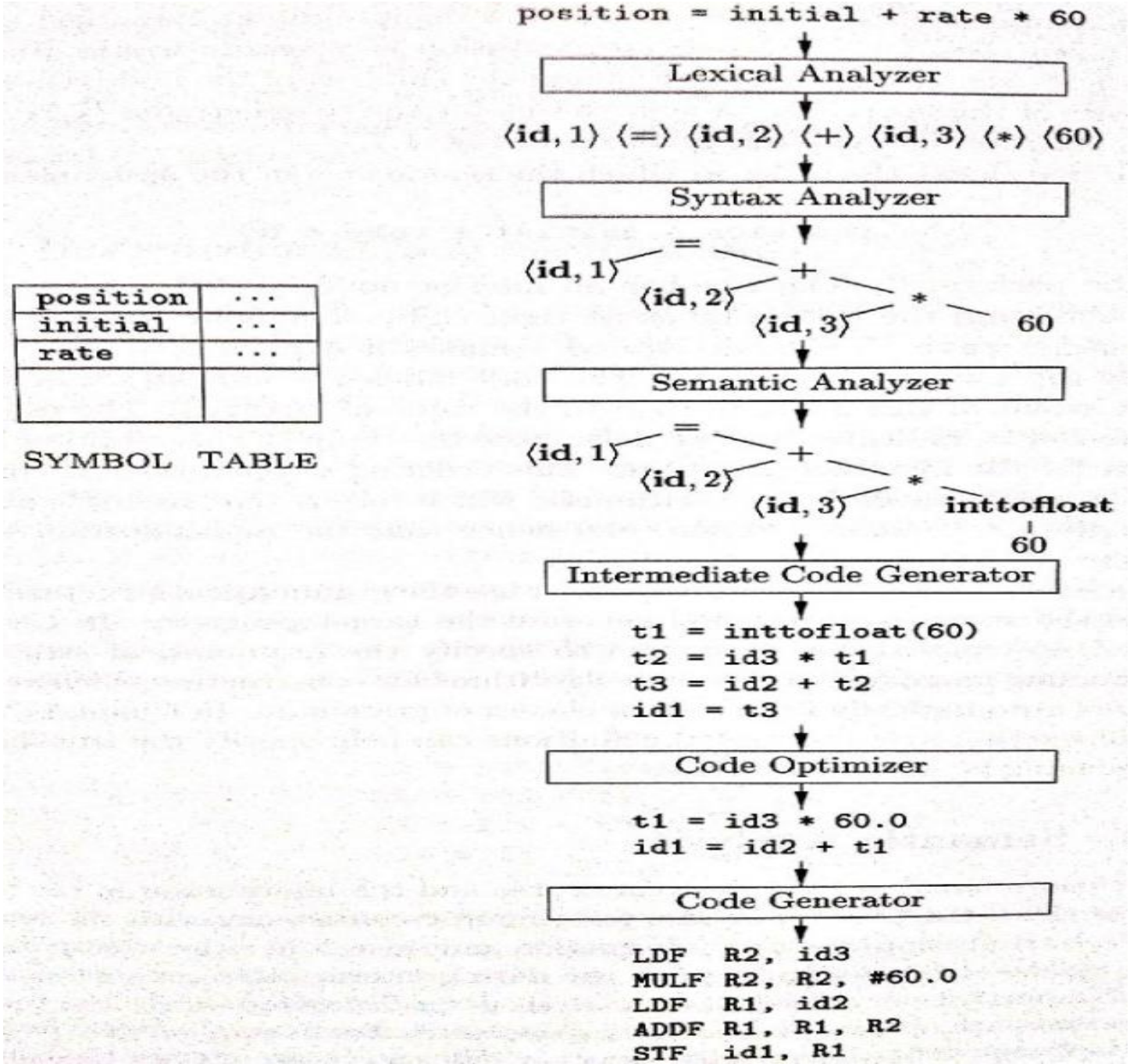



Source Code
Pre Processor
Pre-processed
Code
Compiler
Target
Assembly Code
Assembler
Relocatable
Machine Code
Linker
Library files/
Relocatable
modules
Executable
Machine Code
Loader
Memory

Phases of compiler



Working of compiler different phases

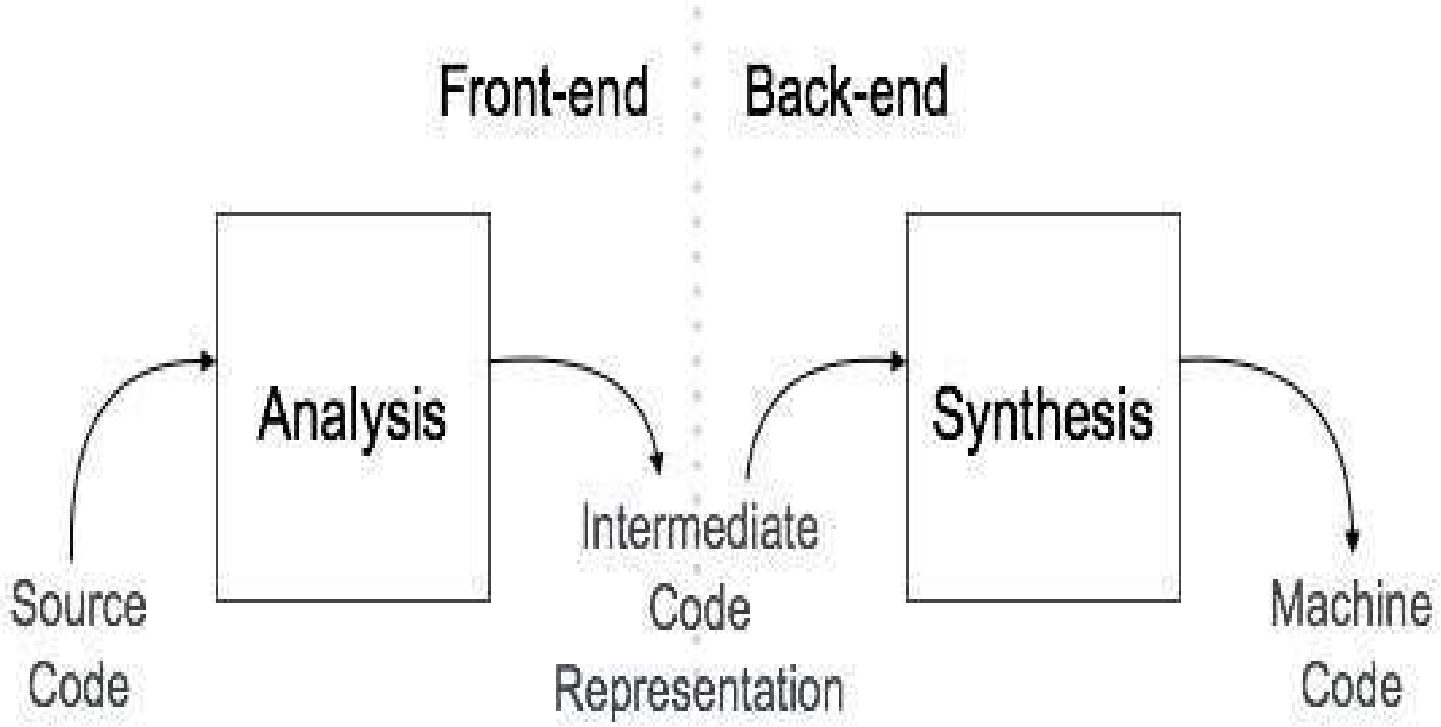


Compiler vs Interpreter

COMPILER	INTERPRETER
1. It translates the full source code at a time.	1. It translates one line of code at a time.
2. It translates one line of code at a time.	2. Comparatively slower.
3. It uses more memory to perform.	3. The interpreter uses less memory than the compiler to perform.
4. Error detection is difficult for the compiler.	4. Error detection is easier for the interpreter.
5. It shows error alert after scanning the full program.	5. Whenever it finds any error it stops there.

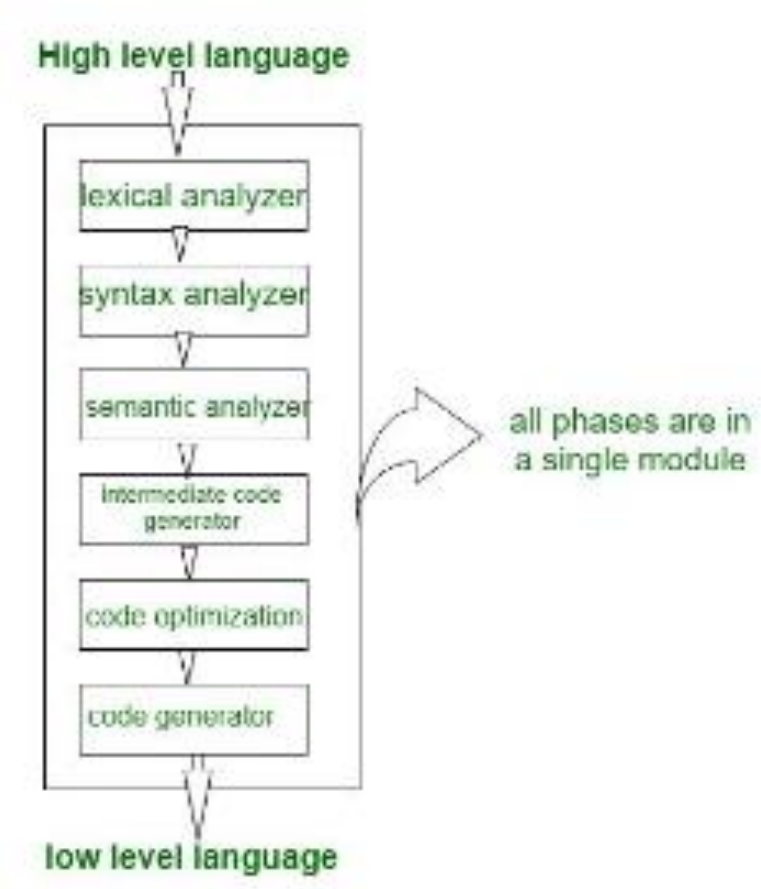
Lecture 2

Phases of Compiler



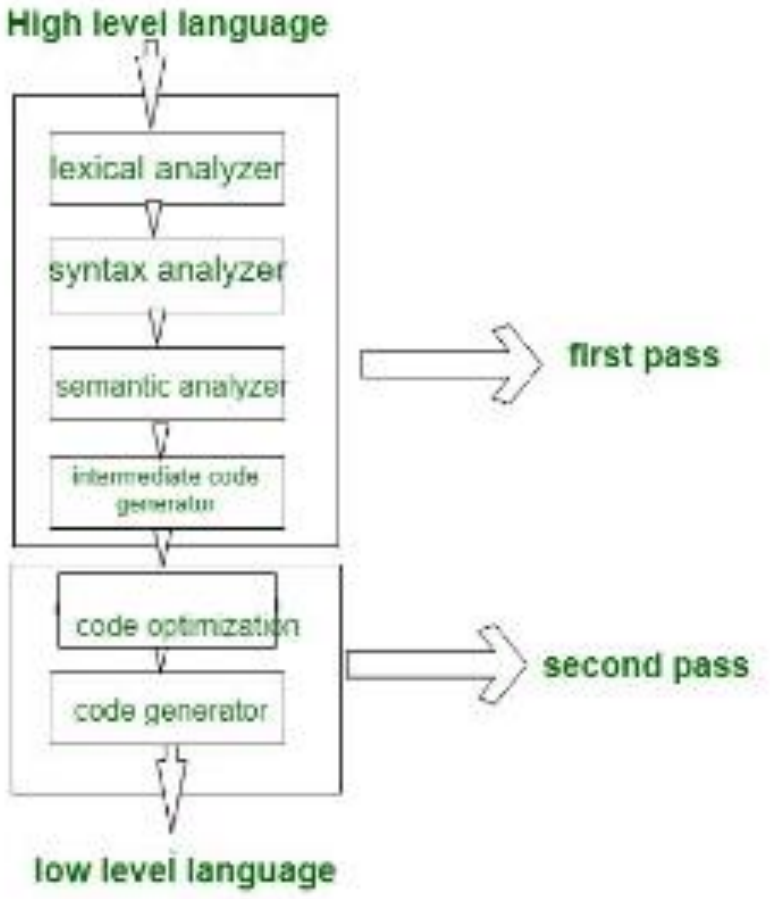
Single-pass Compiler

If we combine all the phases of compiler design in a **single** module is known as single pass compiler.



Multi-pass Compilers

A Two pass/multi-pass Compiler is a type of compiler that processes the *source code* or abstract syntax tree of a program multiple times. In multipass Compiler we divide phases in two pass as:



One pass	Multi pass
Passes through the source code of each compilation unit only once	Processes the source code of a program several times
Compilation time is faster	Compilation time is slower
Has limited scope of passes	Has wide scope of passes.
wide compilers	Narrow compilers
Pascal	Java

Bootstrapping and Cross Compiler

- Bootstrapping is widely used in the compilation development.
- Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.
- Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler to compile everything else as well as future versions of itself.

A compiler can be characterized by three languages:

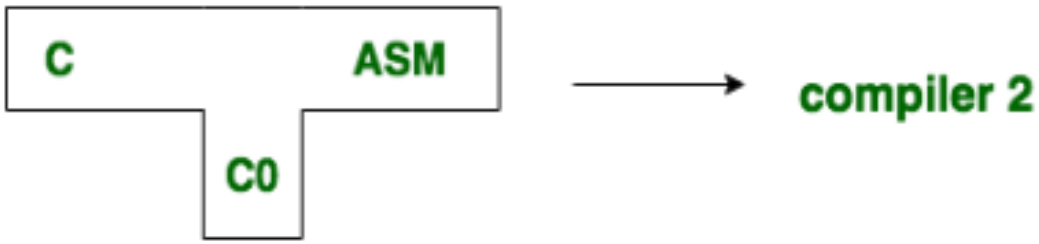
- Source Language
- Target Language
- Implementation Language

Working of Bootstrapping

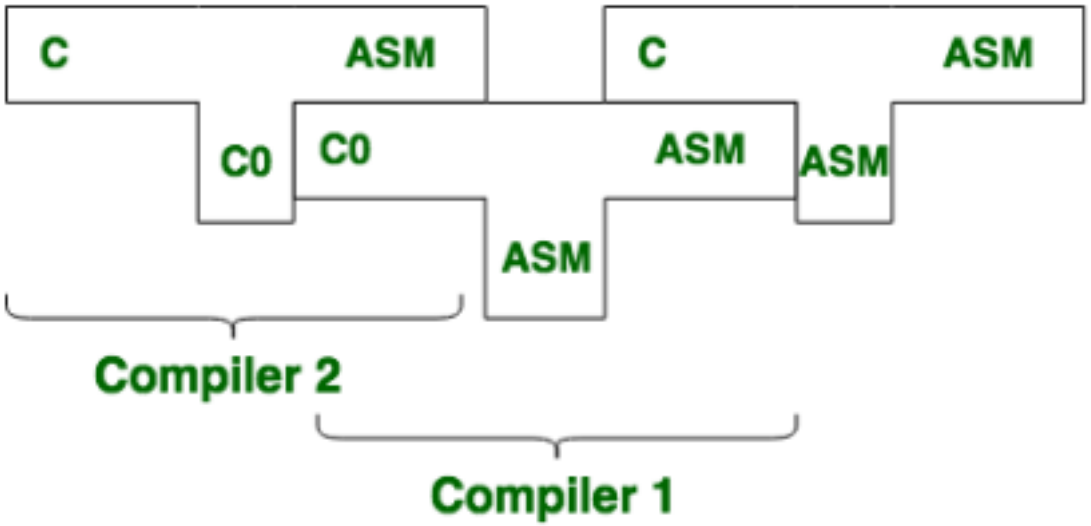
Compiler which takes C language and generates an assembly language as an output with the availability of a machine of assembly language.

- **Step-1:** First we write a compiler for a small of C in assembly language.

- **Step-2:** Then using with small subset of C i.e. C0, for the source language c the compiler is written.



- **Step-3:** Finally we compile the second compiler. using compiler 1 the compiler 2 is compiled.



- **Step-4:** Thus we get a compiler written in ASM which compiles C and generates code in ASM.

Cross Compiler

C**ross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler.

Lecture 3

Tokens, Patterns, and Lexemes

Tokens:

- 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.
  - Ex., a particular keyword, or a sequence of input characters denoting an identifier.
- The token names are the input symbols that the parser processes.
- We will often refer to a token by its token name.

Patterns:

- A pattern is a description of the form that the lexemes of a token may take.
- In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
- For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Lexemes:

- A lexeme is a sequence of characters in source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Example

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or <> or > or >=
id	pi, count	letter forllowed by letter and digits
num	3.14, 786	Any numeric constant
literal	"Tech Saleh"	Any character between " and " except "

Finite Automata

- Finite automata are used to recognize patterns.
- It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
- At the time of transition, the automata can either move to the next state or stay in the same state.
- Finite automata have two states, **Accept state** or **Reject state**. When the input string is processed successfully, and the automata reached its final state, then it will accept

Formal Definition of FA

A finite automaton is a collection of 5-tuple (Q, Σ, δ, q0, F), where:

Q: finite set of states

Σ: finite set of the input symbol

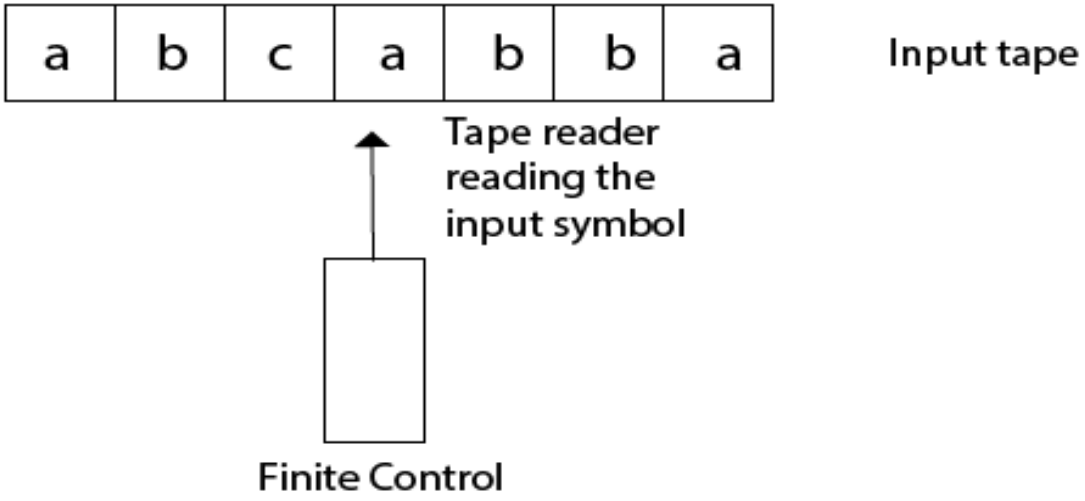
q0: initial state

F: **final** state

δ: Transition function

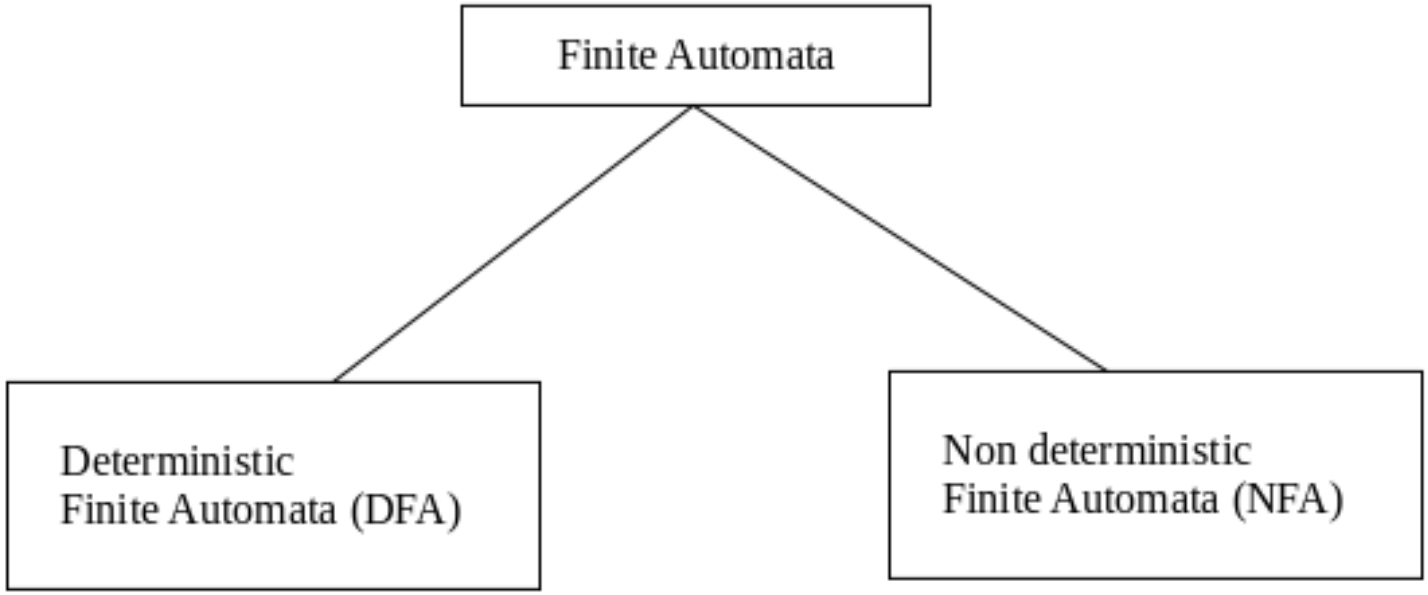
Finite Automata Model

- Finite automata can be represented by input tape and finite control.
- **Input tape:** It is a linear tape having some number of cells. Each input symbol is placed in each cell.
- **Finite control:** The finite control decides the next state on receiving particular input from input tape. The tape reader reads the cells one by one from left to right, and at a time only one input symbol is read.



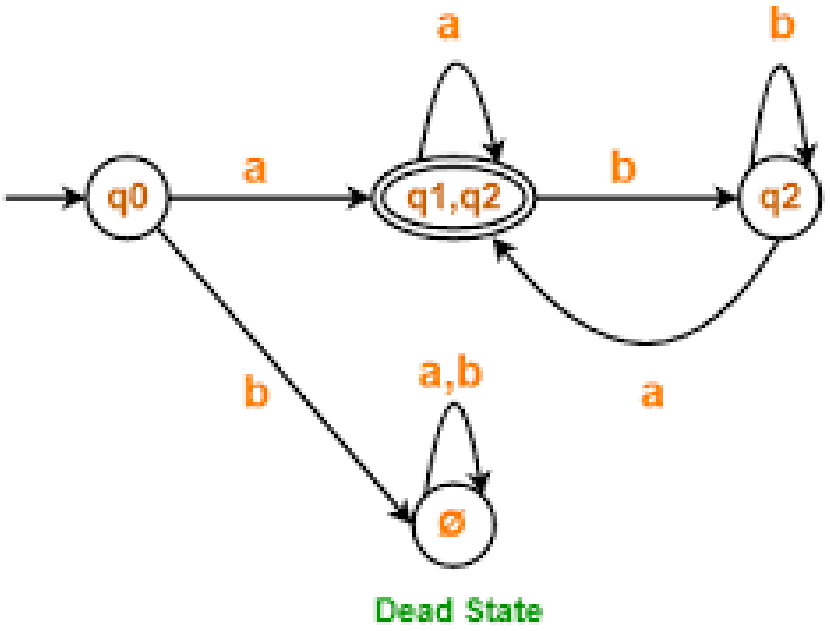
Types of Finite Automata:

- DFA(deterministic finite automata)
- NFA(non-deterministic finite automata)



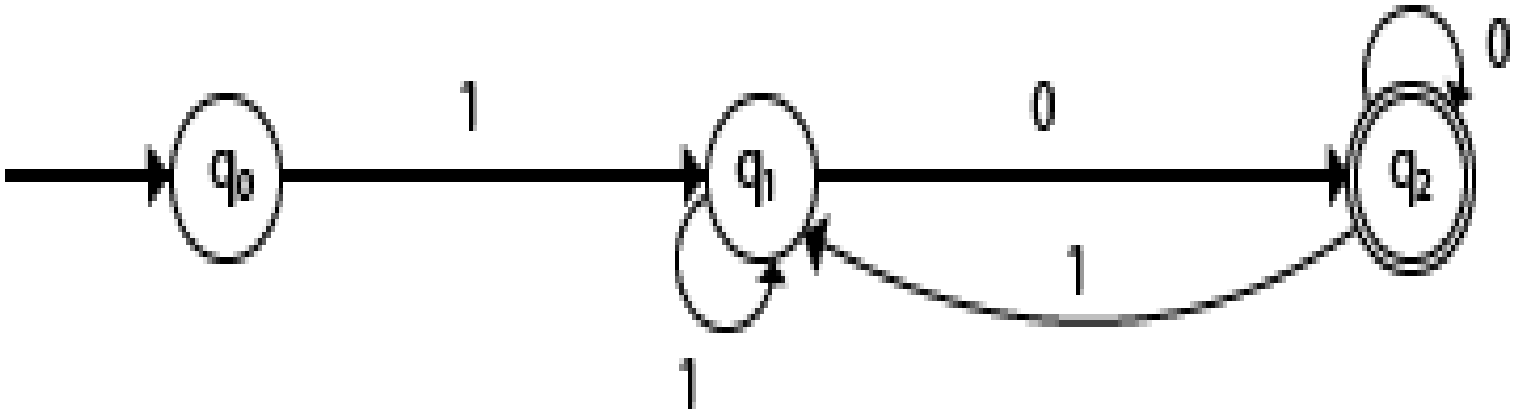
DFA

DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. In the DFA, the machine goes to one state only for a particular input character. DFA does not accept the null move.

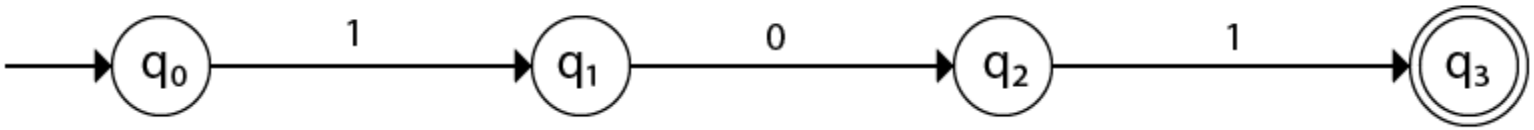


Deterministic Finite Automata (DFA)

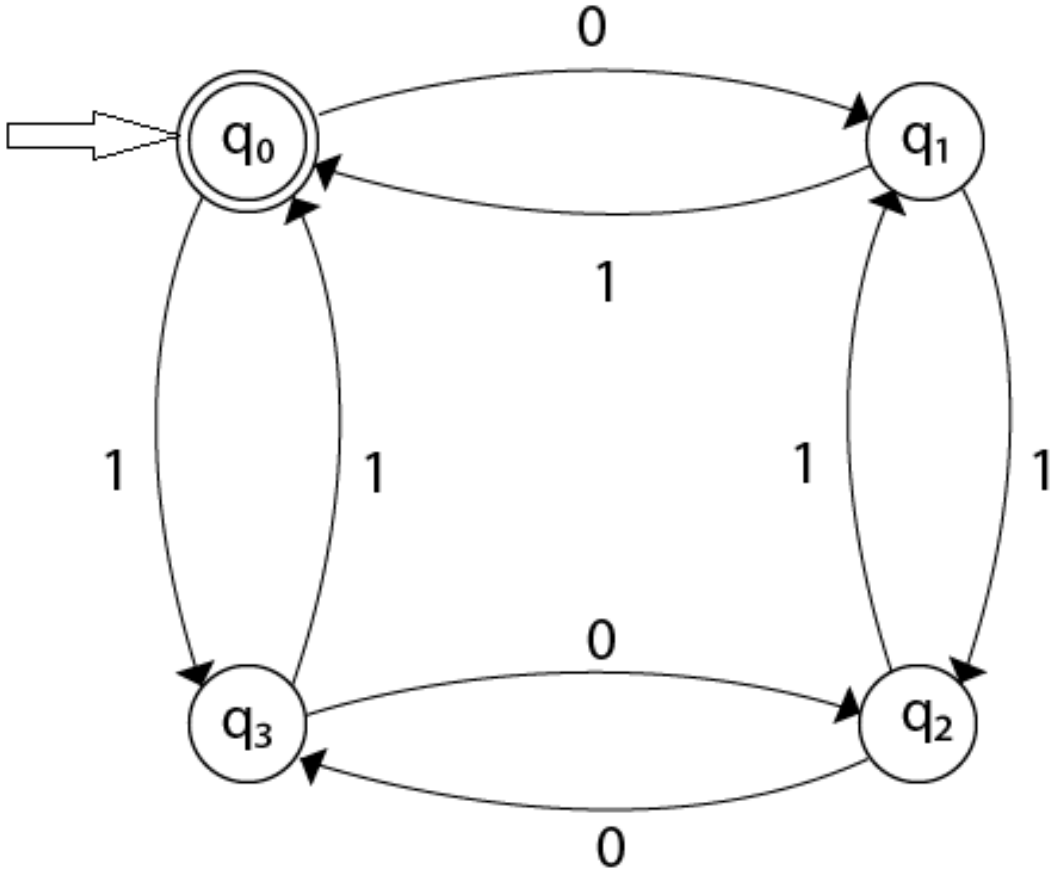
Design a DFA with ∑ = {0, 1} accepts those string which starts with 1 and ends with 0.



Design a DFA with ∑ = {0, 1} accepts the only input 101

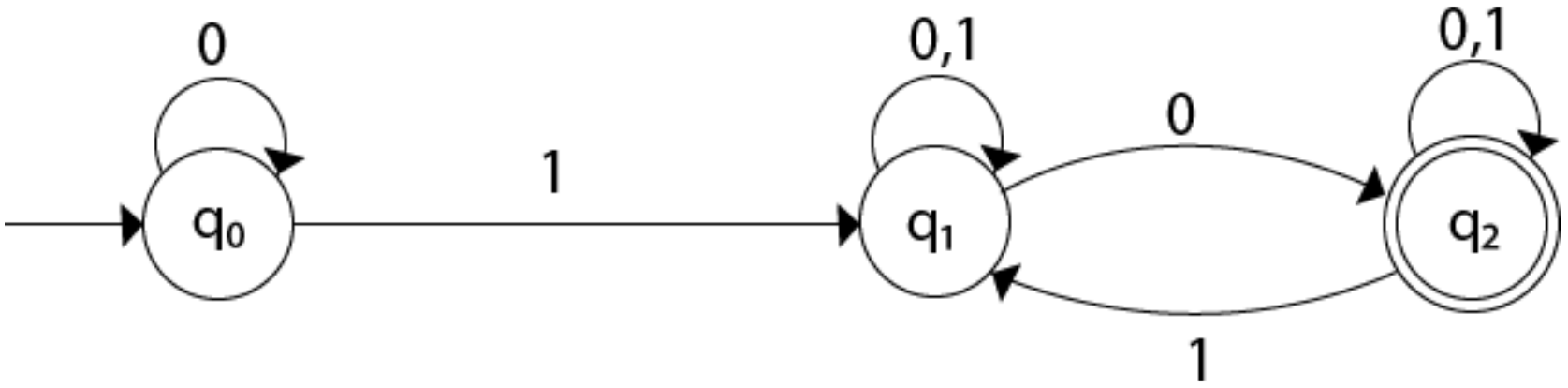


Design DFA with ∑ = {0, 1} accepts even number of 0's and even number of 1's.

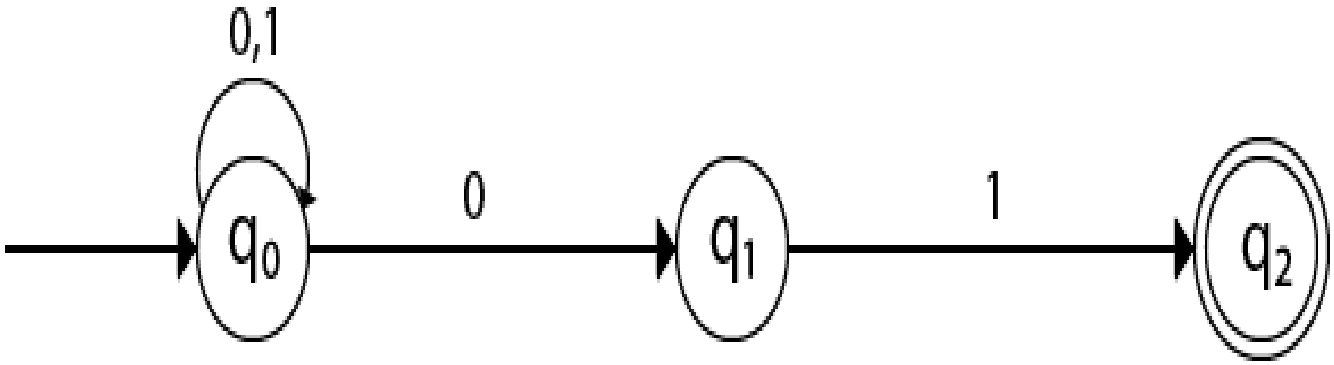


NFA

NFA stands for non-deterministic finite automata. It is used to transmit any number of states for a particular input. It can accept the null move.

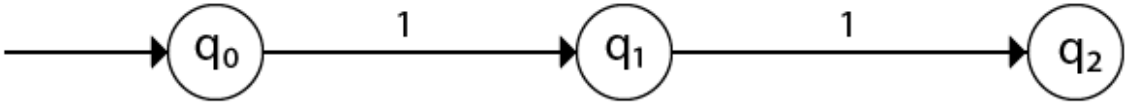


Design an NFA with ∑ = {0, 1} accepts all string ending with 01.



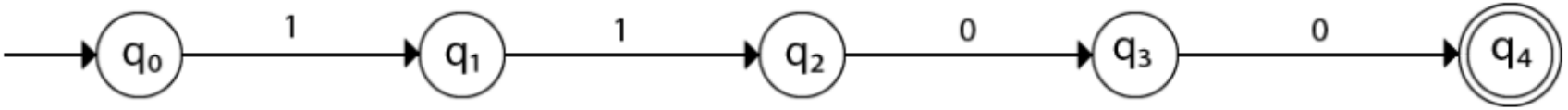
Design an NFA with ∑ = {0, 1} in which double '1' is followed by double '0'.

The FA with double 1 is as follows:



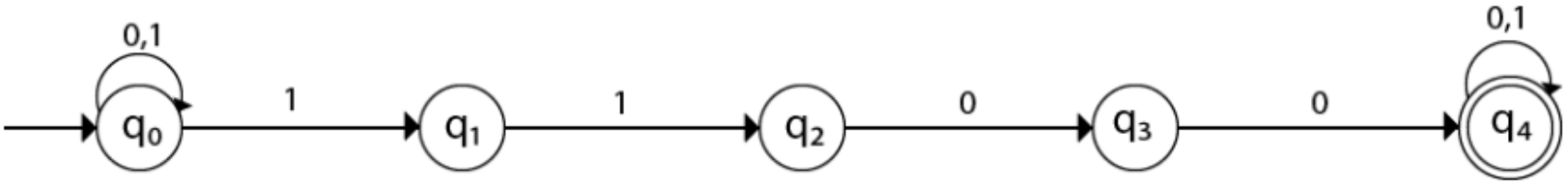
It should be immediately followed by double 0.

Then,



Now before double 1, there can be any string of 0 and 1. Similarly, after double 0, there can be any string of 0 and 1.

Hence the NFA becomes:



Thomson's Construction rule

Construction of an NFA from a Regular Expression

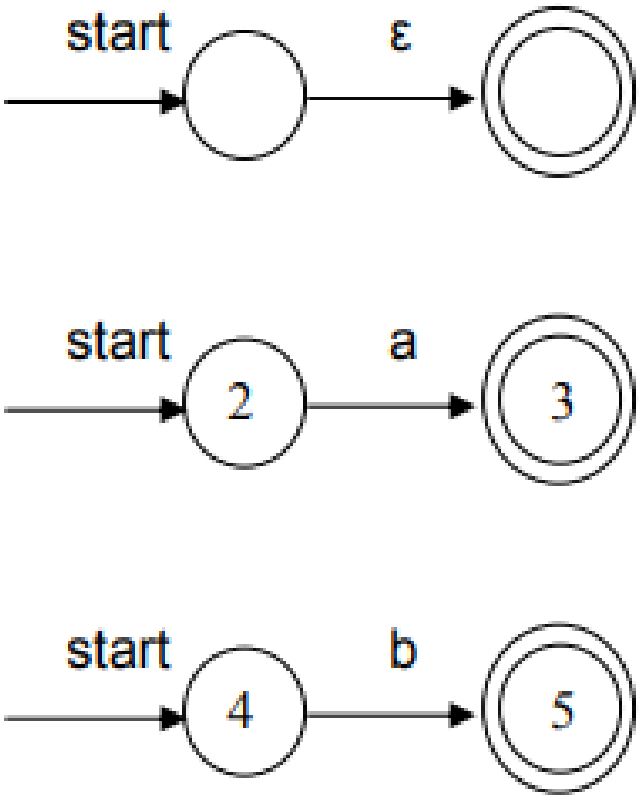
INPUT: A regular expression r over alphabet C.

OUTPUT: An NFA N accepting L(r)

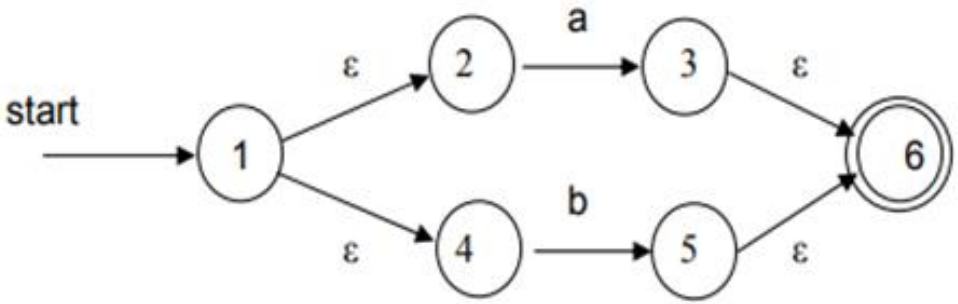
METHOD: Begin by parsing r into its constituent sub expressions. The rules for constructing an NFA consist of basis rules for handling subexpressions with no operators, and inductive rules for constructing larger NFA's from the NFA's for the immediate subexpressions of a given expression.

Construction Method

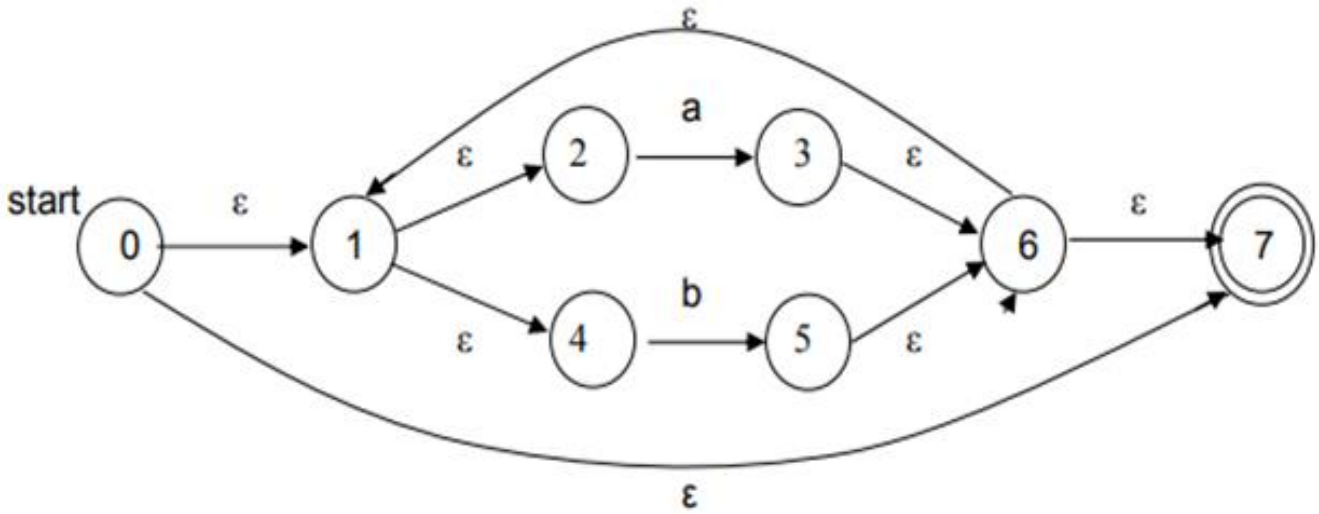
the NFA's for single character regular expressions ε, a, b



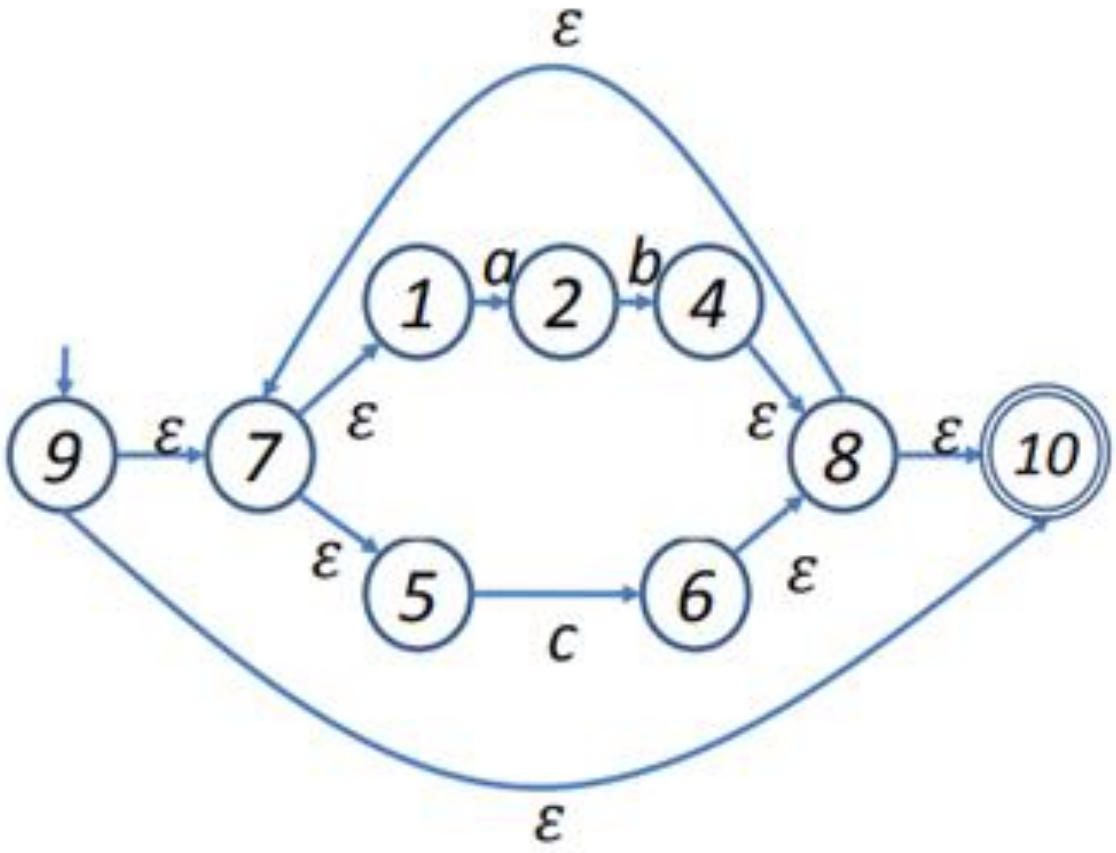
NFA for a|b



NFA (a|b)*



Draw the NFA for the regular expression (ab|c)*



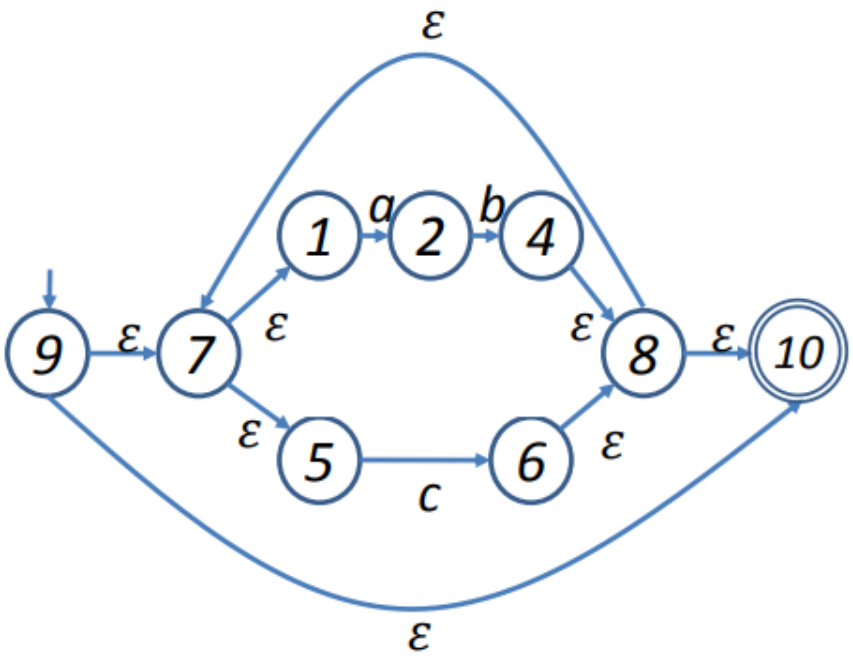
Subset Construction Algorithm

Step 1:- Put ϵ- closure (S_0) an unmarked state into the set of DFA.

Step 2:- while(there is one unmarked state s1 in DFA)

```
begin
        mark s1
        for each input symbol a do
        begin
                s2← ϵ-closure(move(s1,a))
                if(s2 is not in DFA)
                {
                   Then add s2 into DFA as an unmarked state
                }
                Trans[s1,a]←s2
        End
End
```

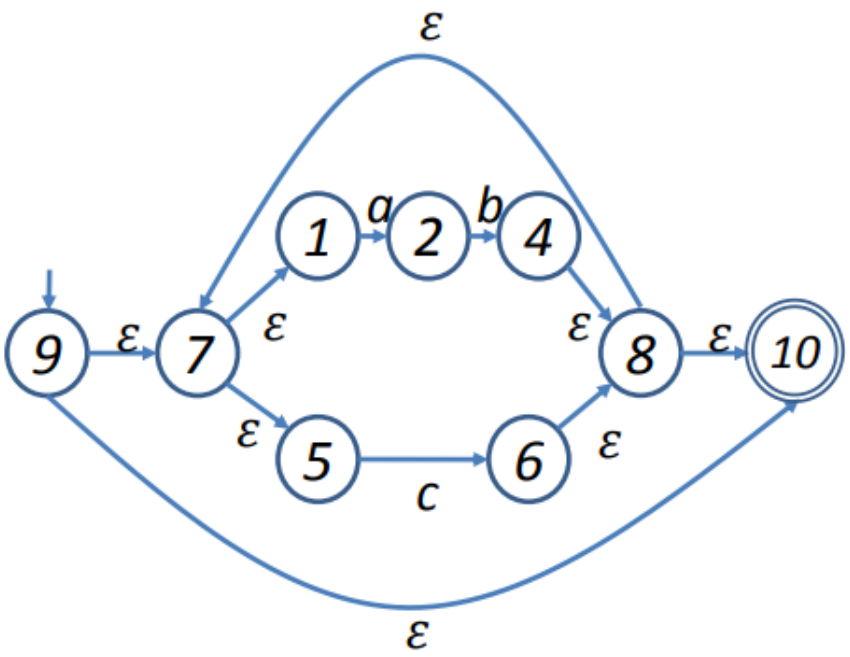
Mark state A



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓			

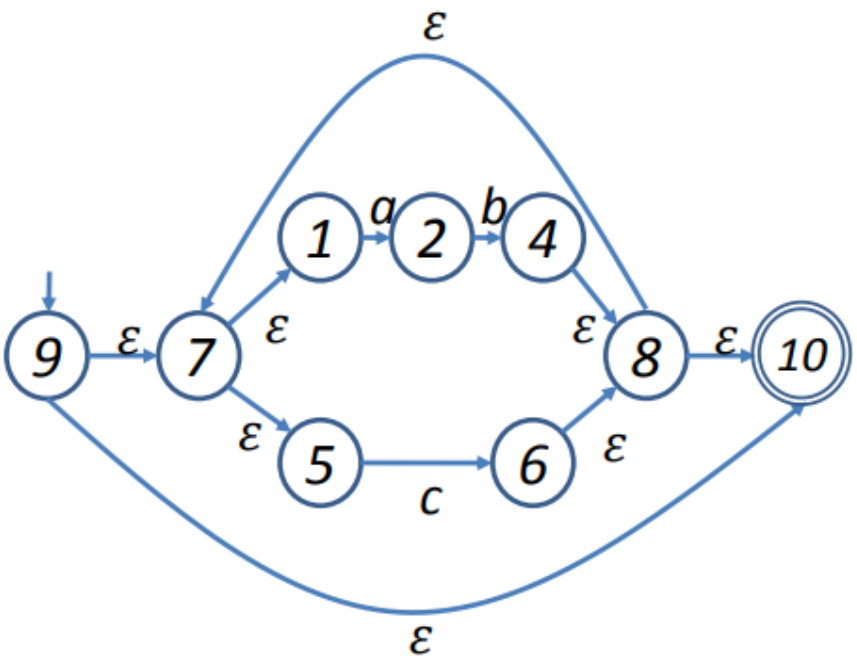
Compute *ε-closure(move(A, a))*



D_{states}

NFA States	DFA State	*Next State*		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B		
{2}	B			

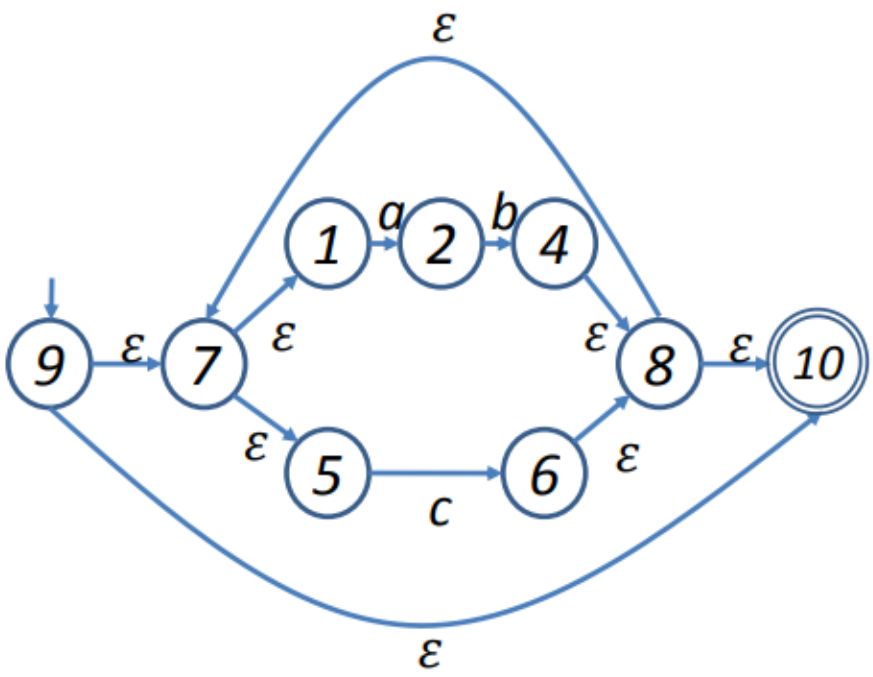
Compute ε-closure(move(A, b))



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	
{2}	B			

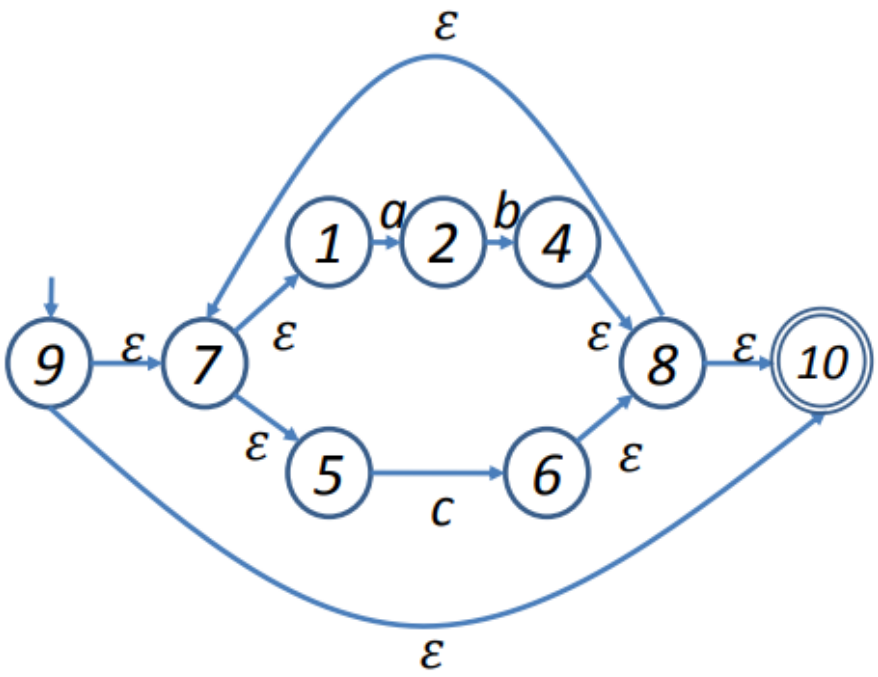
Compute ε-*closure(move(A, c))*



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B			
{6,8,10,7,1,5}	C			

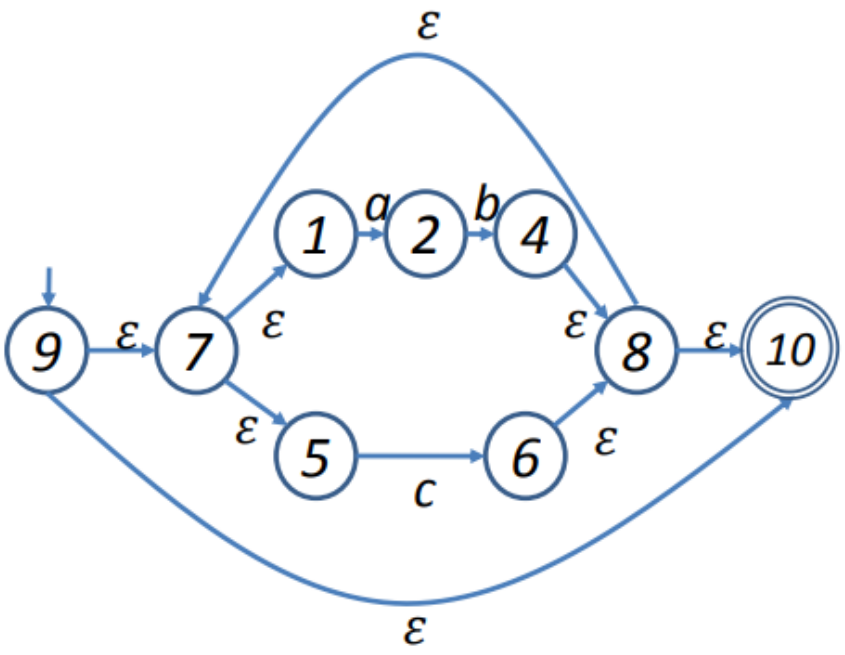
Mark B



D_{states}

NFA States	DFA State	*Next State*		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓			
{6,8,10,7,1,5}	C			

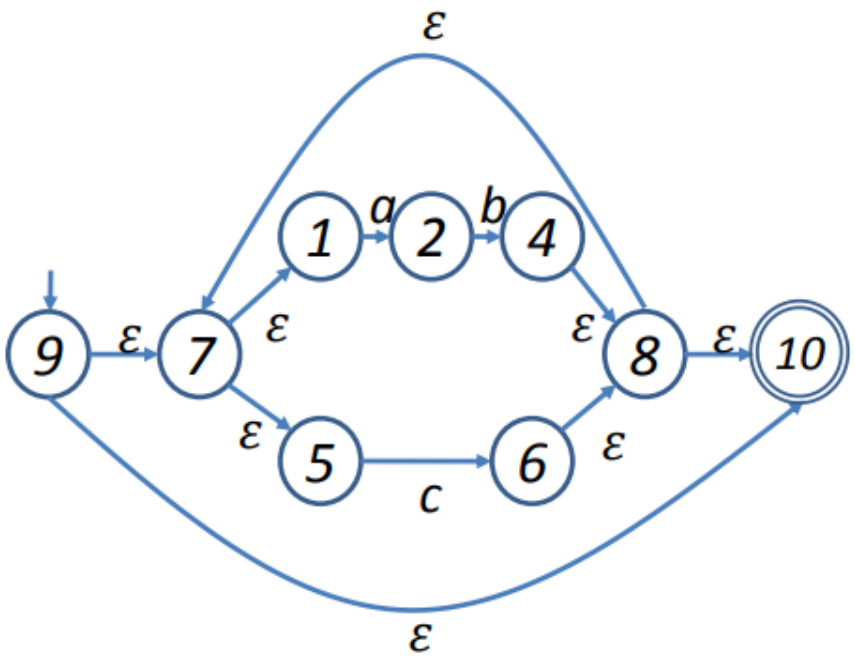
Compute ε-closure(move(B, a))



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-		
{6,8,10,7,1,5}	C			

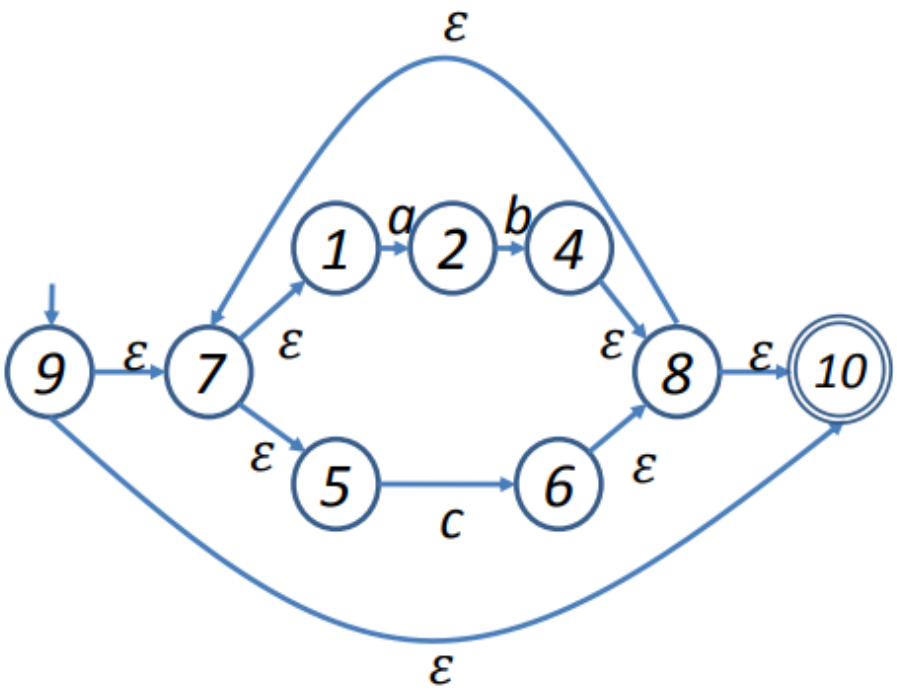
Compute ε-closure(move(B, b))



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	
{6,8,10,7,1,5}	C			
{4,8,7,1,5,10}	D			

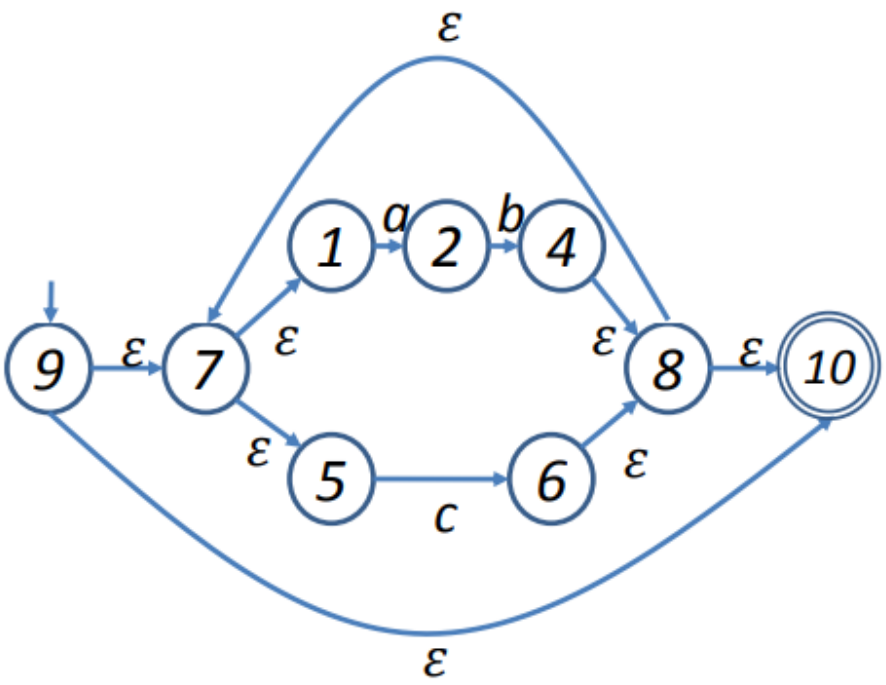
Compute ε-closure(move(B, c))



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C			
{4,8,7,1,5,10}	D			

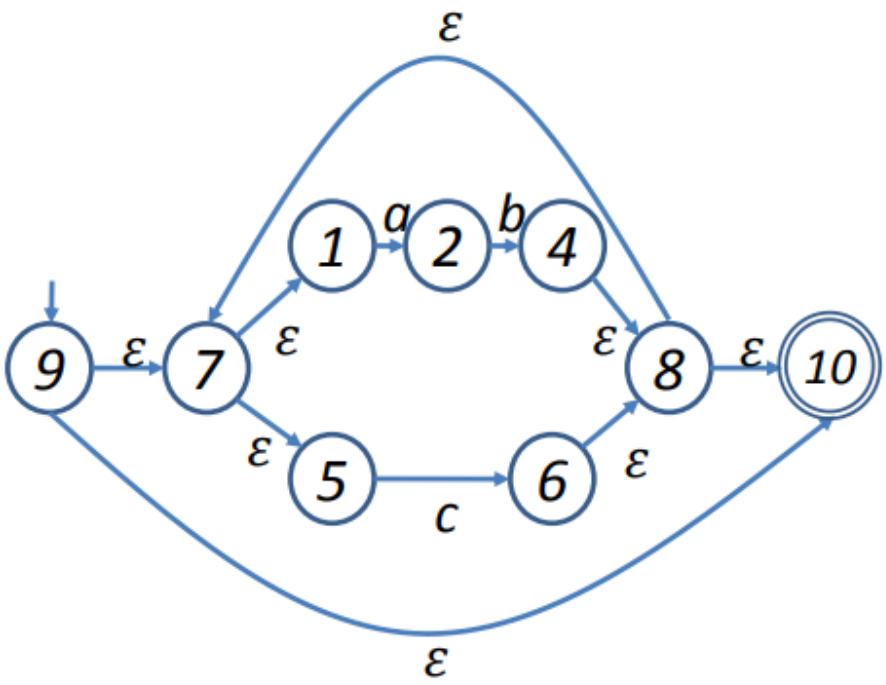
Mark C



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C ✓			
{4,8,7,1,5,10}	D			

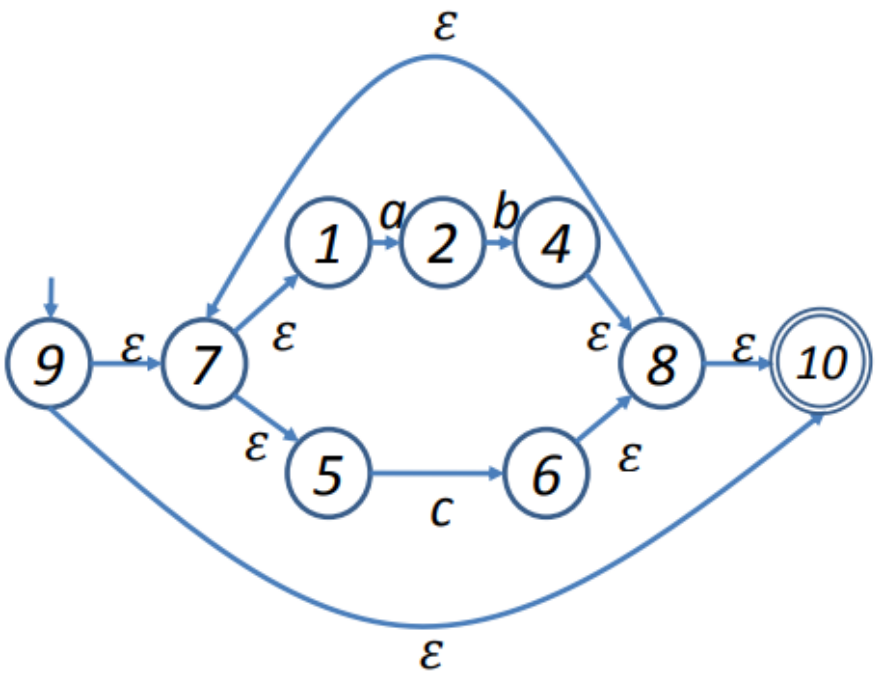
Compute ε-*closure(move(C, a))*



D_{states}

NFA States	DFA State	*Next State*		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C ✓	B		
{4,8,7,1,5,10}	D			

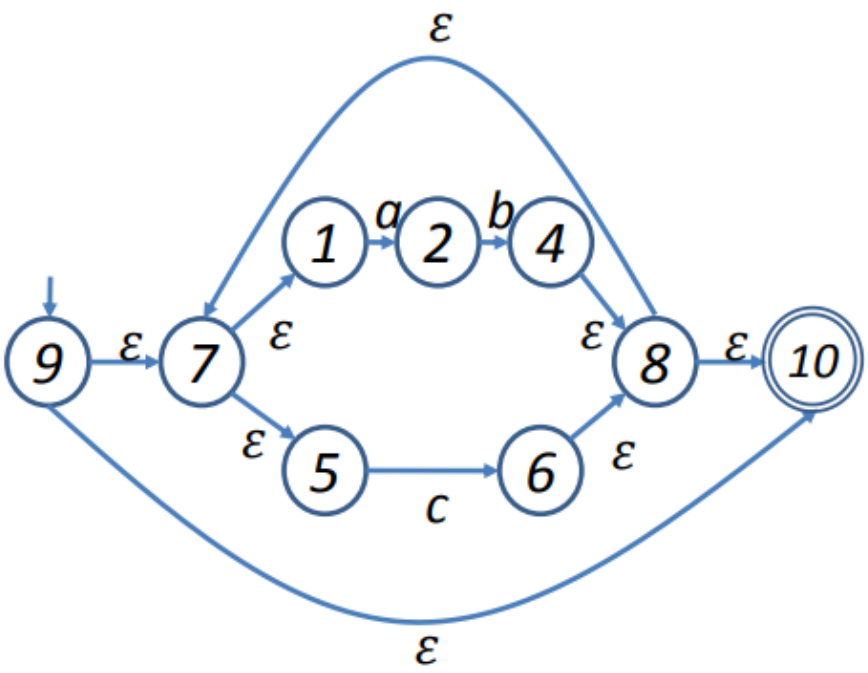
Compute ε-closure(move(C, b))



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C ✓	B	-	
{4,8,7,1,5,10}	D			

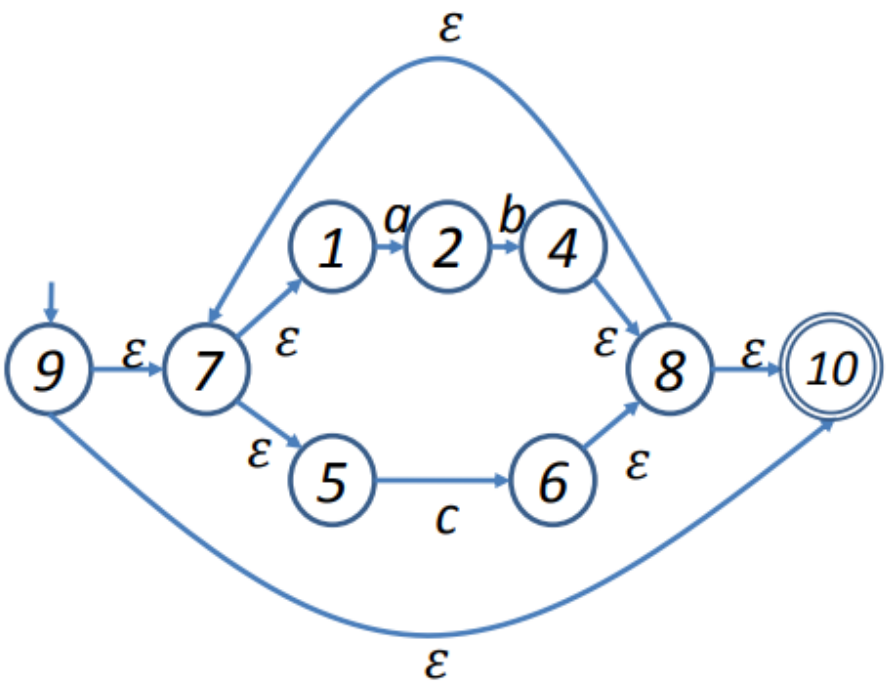
Compute ε-*closure(move(C, c))*



D_{states}

NFA States	DFA State	*Next State*		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C ✓	B	-	C
{4,8,7,1,5,10}	D			

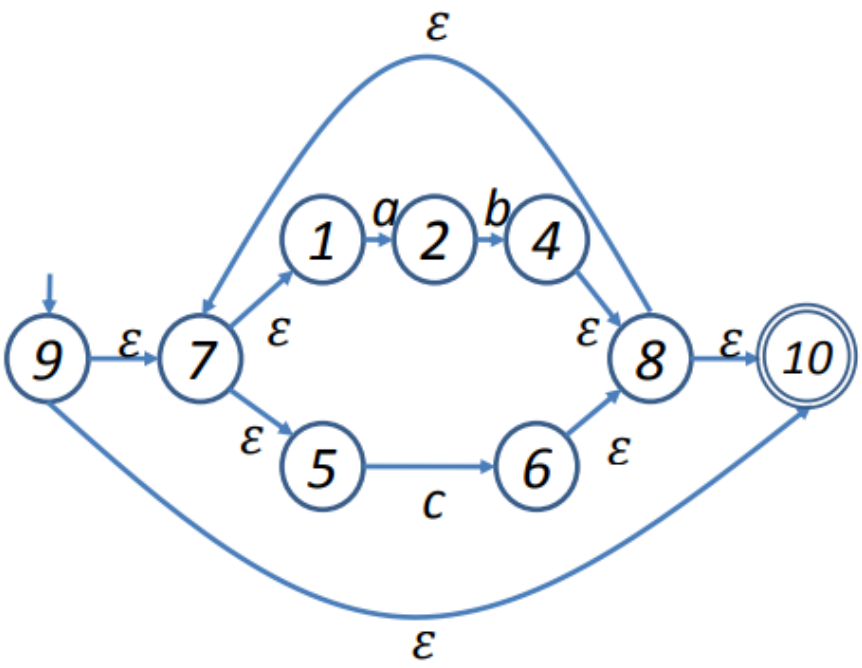
Mark D



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C ✓	B	-	C
{4,8,7,1,5,10}	D ✓			

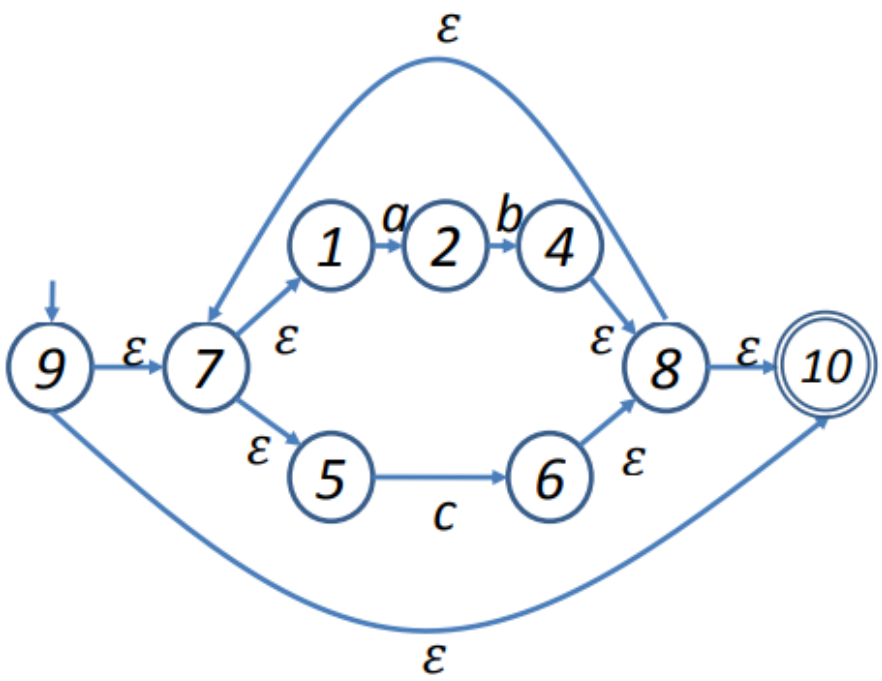
Compute ε-closure(move(D, a))



D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C ✓	B	-	C
{4,8,7,1,5,10}	D ✓	B		

Compute ε-closure(move(D, b))

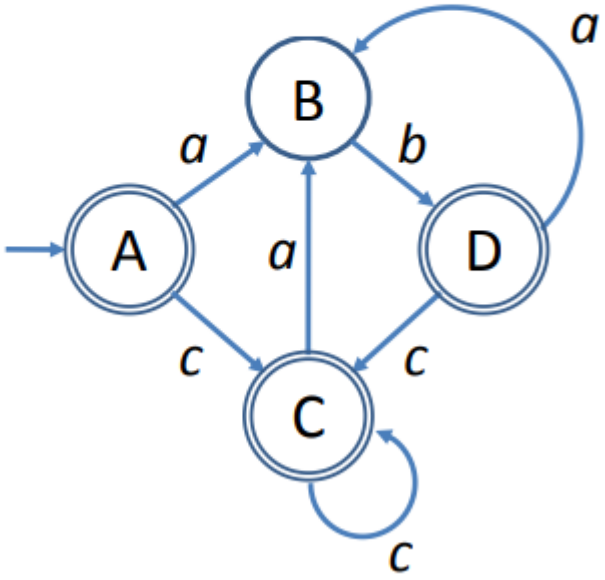


D_{states}

NFA States	DFA State	Next State		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C ✓	B	-	C
{4,8,7,1,5,10}	D ✓	B	-	C

DFA

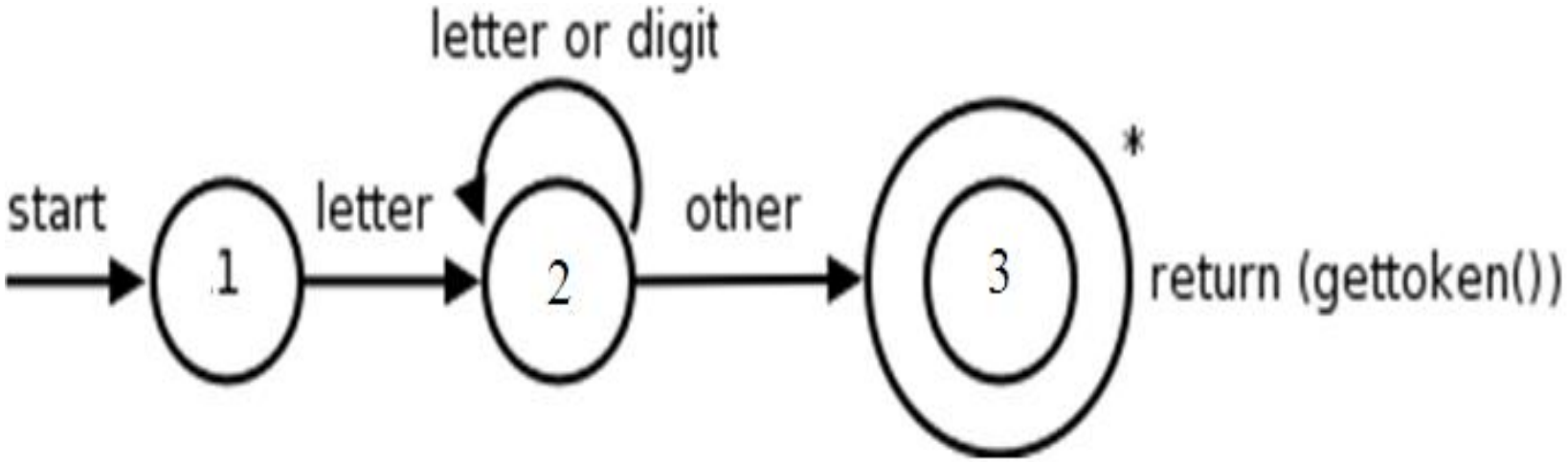
NFA States	DFA State	*Next State*		
		a	*b*	*c*
{9,7,1,5,10}	A ✓	B	-	C
{2}	B ✓	-	D	-
{6,8,10,7,1,5}	C ✓	B	-	C
{4,8,7,1,5,10}	D ✓	B	-	C



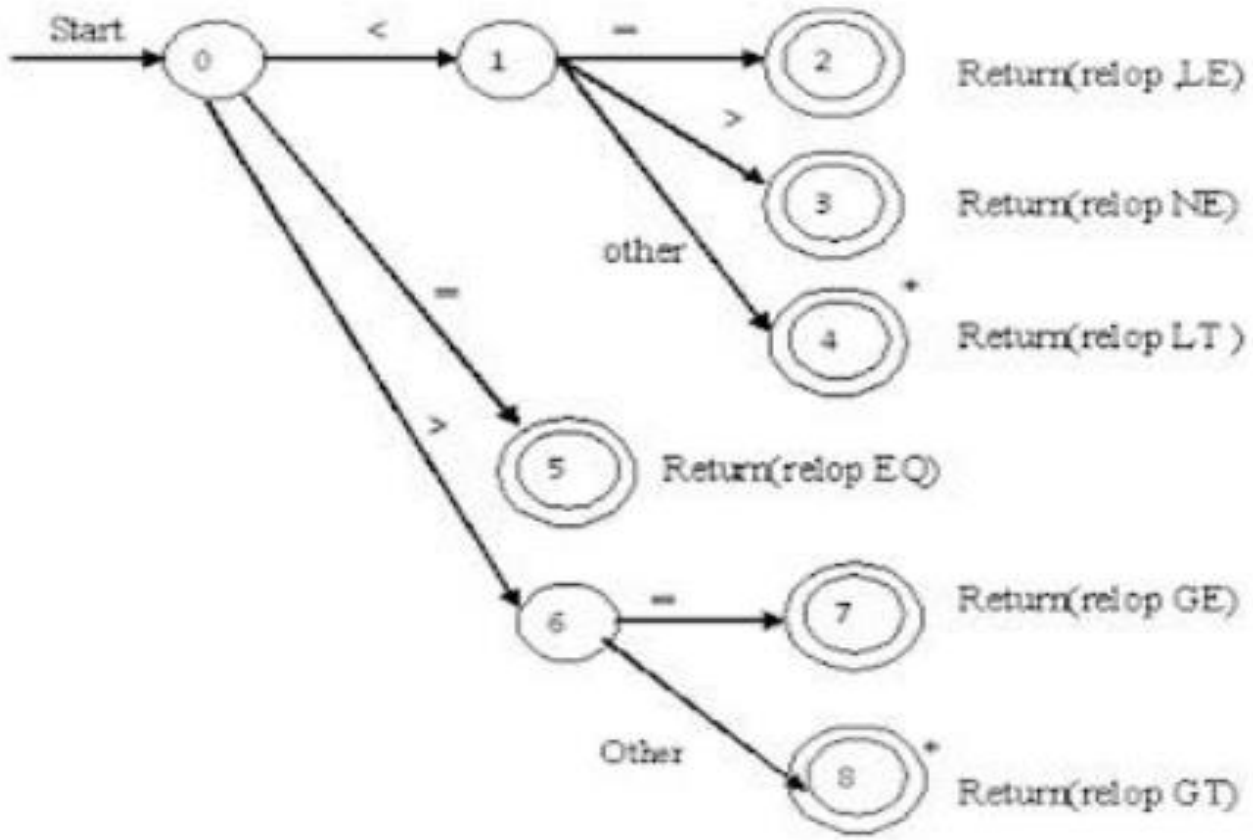
TRANSITION DIAGRAM

- Transition Diagram has 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.
- If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a.
- if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Transition Diagram for Identifier



Transition Diagram for Relational operator



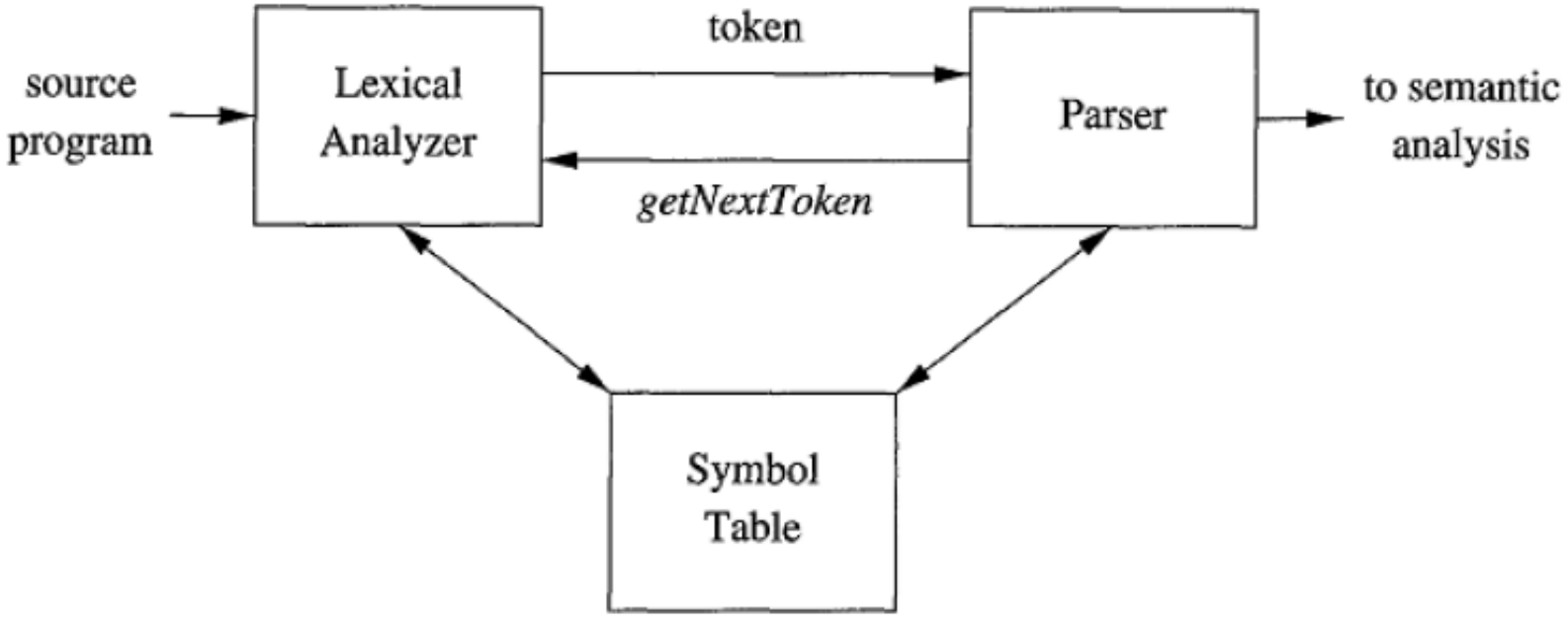
INPUT BUFFERING

- The Lexical Analyzer scans the characters of the source program one at a time to discover tokens.
- Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Buffering techniques:

1. Buffer pairs
2. Sentinels

Lexical Analyzer Generator



The Lexical Analyzer Generator LEX

- LEX, or in a more recent implementation FLEX is a 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 the LEX language and the tool itself is the LEX compiler.

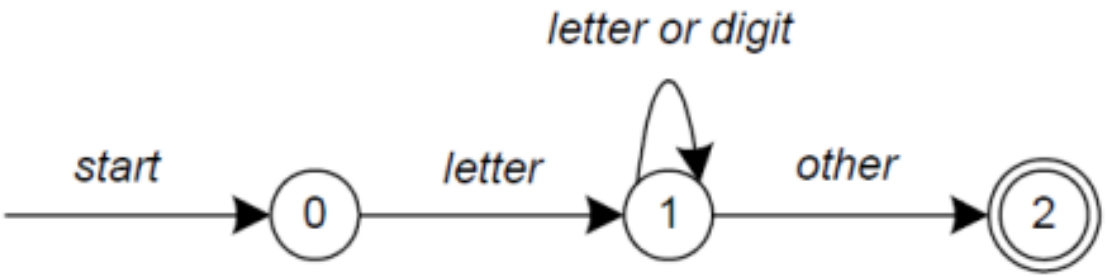
- The LEX compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy.c, that simulates this transition diagram.

Lex

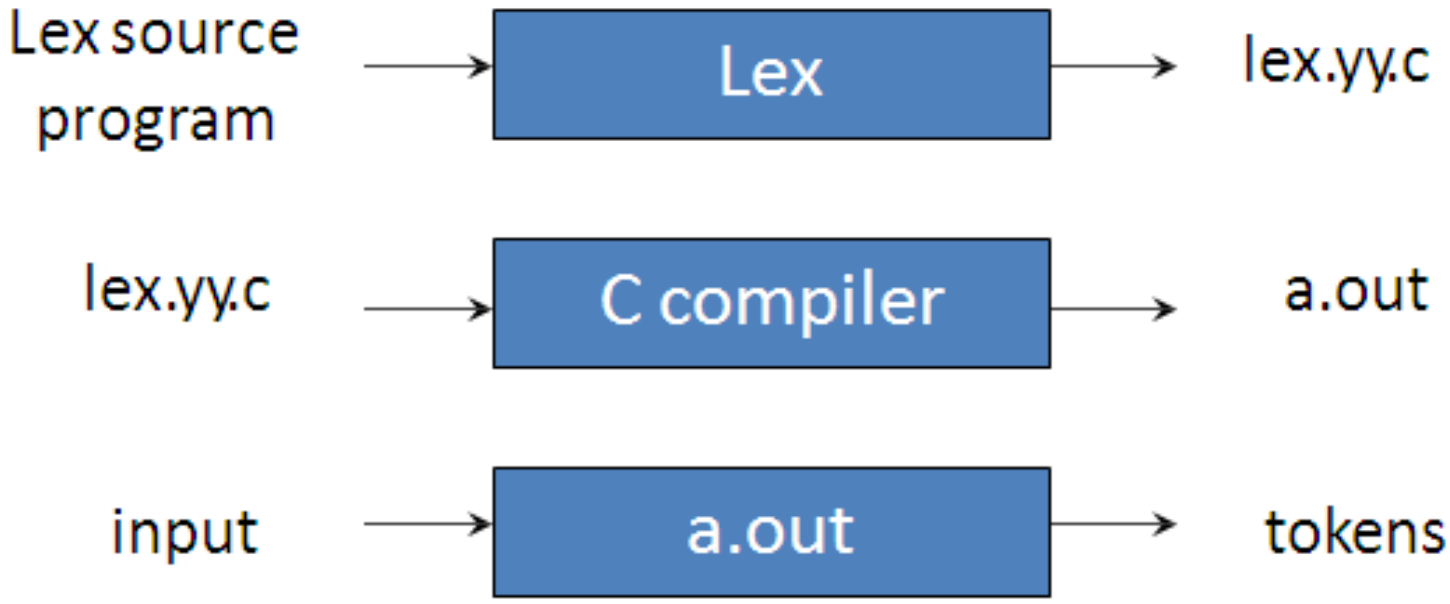
- lex is a program (generator) that generates lexical analyzers, (widely used on Unix).
- It is mostly used with Yacc parser generator.
- It reads the input stream (specifying the lexical analyzer) and outputs source code implementing the lexical analyzer in the C programming language.

Lex(cont.)

- Lex will read patterns (regular expressions); then produces C code for a lexical analyzer that scans for identifiers.
- A simple pattern: letter(letter|digit)*
- This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits.



An Overview of Lex



Structure of LEX programs

- Lex source is separated into three sections by %% delimiters
- The general format of Lex source is

```
{definitions}
%%                                  (required)
{transition rules}
%%
                                    (optional)
{user subroutines}
```

- The absolute minimum Lex program is thus

```
%%
```

Structure of LEX programs

Example:

```
digit  [0-9]
letter [a-zA-Z]
%%
({letter}|{digit})*        printf("id: %s\n", yytext);
\n                         printf("new line\n");
%%
main()
{
   yylex();
}
```

Metacharacter	Matches
`.`	any character except newline
`\n`	newline
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`?`	zero or one copy of the preceding expression
`^`	beginning of line
`$`	end of line
`a|b`	`a` or `b`
`(ab)+`	one or more copies of `ab` (grouping)
`"a+b"`	literal "`a+b`" (C escapes still work)
`[]`	character class

Expression	Matches
`abc`	`abc`
`abc*`	`ab abc abcc abccc ...`
`abc+`	`abc abcc abccc ...`
`a(bc)+`	`abc abcbc abcbcbc ...`
`a(bc)?`	`a abc`
`[abc]`	one of: `a`, `b`, `c`
`[a-z]`	any letter, a-z
`[a\-z]`	one of: `a`, `-`, `z`
`[-az]`	one of: `-`, `a`, `z`
`[A-Za-z0-9]+`	one or more alphanumeric characters
`[ \t\n]+`	whitespace
`[^ab]`	anything except: `a`, `b`
`[a^b]`	one of: `a`, `^`, `b`
`[a|b]`	one of: `a`, `|`, `b`
`a|b`	one of: `a`, `b`

Lex Predefined Function

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

Formal Grammar

- Formal grammar is a set of rules. It is used to identify correct or incorrect strings of tokens in a language. The formal grammar is represented as G.
- Formal grammar is used to generate all possible strings over the alphabet that is syntactically correct in the language.
- Formal grammar is used mostly in the syntactic analysis phase (parsing) particularly during the compilation.

Formal grammar G is written as follows:

G = <V, N, P, S>

Where:

N describes a finite set of non-terminal symbols.

V describes a finite set of terminal symbols.

P describes a set of production rules

S is the start symbol.

Example:

$S \longrightarrow aA|b$

$A \longrightarrow aA \mid a$

Backus Normal Form(BNF)

- Backus Normal Form is a notation techniques for the context free grammar.
- It is often used to describe the syntax of languages used in computing.
- Such as computer programming languages, document formats and so on..
- They are applied wherever exact descriptions of languages are needed.

Example:-Write a BNF grammar for the language of University of Lucknow course codes.

- **Example sentences:**
- **CSI3125**
- **MAT2743**
- **PHY1200**
- **CHE6581**
- **CSI9999**

Solution:

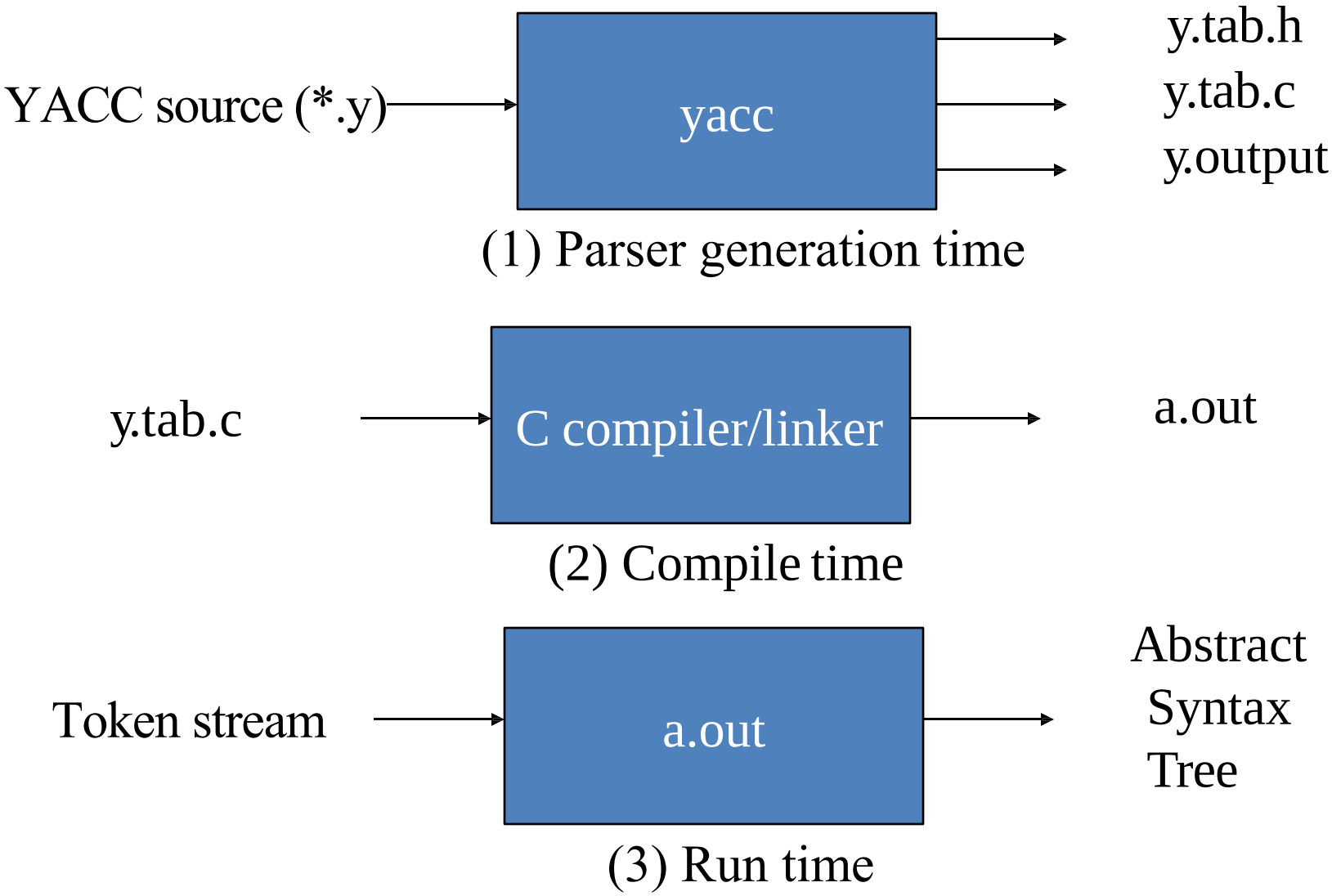
- <coursecode> ::= <acadunit> <coursenumber>
- <acadunit> ::= <letter> <letter> <letter>
- <coursenumber> ::= <year> <semesters> <digit> <digit>
- <year> ::= <ugrad> | <grad>
- <ugrad> ::= 0 | 1 | 2 | 3 | 4
- <grad> ::= 5 | 6 | 7 | 9
- <semesters> ::= <onesemester> | <twosemesters>
- <onesemester> ::= <frenchone> | <englishone> | <bilingual>
- <frenchone> ::= 5 | 7
- <englishone> ::= 1 | 3
- <bilingual> ::= 9
- <twosemesters> ::= <frenchtwo> | <englishtwo>
- <frenchtwo> ::= 6 | 8
- <englishtwo> ::= 2 | 4
- <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Introduction about YACC

What is **YACC** ?

- Tool which will produce a parser for a given grammar.
- YACC (Yet Another Compiler Compiler) is a program designed to compile LALR(1) grammar and to produce the source code of the syntactic analyzer of the language produced by this grammar.

How YACC Works



YACC File Format

%{

C declarations

%}

yacc declarations

%%

Grammar rules

%%

Additional C code

– **Comments enclosed in /* ... */ may appear in any of the sections.**

An YACC File Example

```
%{
#include <stdio.h>
%}

%token NAME NUMBER
%%

statement: NAME '=' expression
        | expression            { printf("= %d\n", $1); }
        ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
        |   expression '-' NUMBER { $$ = $1 - $3; }
        |   NUMBER                { $$ = $1; }
        ;
%%
int yyerror(char *s)
{
   fprintf(stderr, "%s\n", s);
   return 0;
}

int main(void)
{
   yyparse();
   return 0;
}
```

Ambiguity

- A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be **ambiguous**.

- To show that a grammar is ambiguous all we need to do is find a terminal string that is the yield of more than one parse tree.

- Since a string with more than one parse tree usually has more than one meaning, we need to design unambiguous grammars for compiling applications, or to use ambiguous grammars with additional rules to resolve the ambiguities.

Example 1:Let us consider a grammar G with the production rule

$E \rightarrow I$

$E \rightarrow E + E$

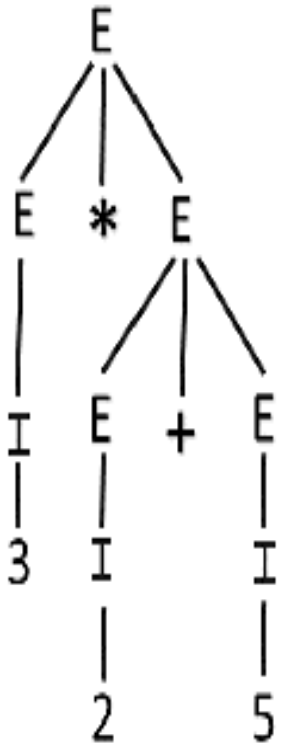
$E \rightarrow E * E$

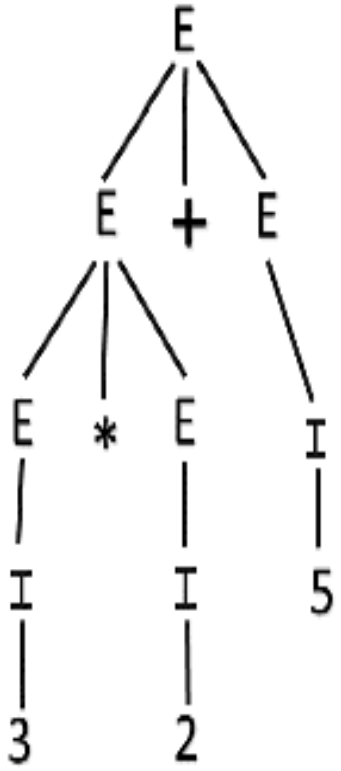
$E \rightarrow (E)$

$I \rightarrow \varepsilon \mid 0 \mid 1 \mid 2 \mid \ldots \mid 9$

Solution:

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:





Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is ambiguous.