



Course Name:
COMPILER DESIGN

Course code:
SCS1303



Course Objective:

- To explore the principles, algorithms, and data structures involved in the design and construction of compilers.
- To introduce the theory and tools that can be employed in order to perform syntax-directed translation of a high-level programming language into an executable code.

Prerequisite Courses:

- Theory of Computation
- Data Structures
- Programming Languages

[Syllabus:Compiler Design.doc](#)



SCS1303	COMPILER DESIGN	L	T	P	Credits	Total Marks
		3	0	0	3	100

COURSE OBJECTIVES

- To study the structure of compiler.
- To study the working principles compilation process.

UNIT 1 LEXICAL ANALYSIS

9 Hrs

Structure of compiler – Functions and Roles of lexical phase – Input buffering – Representation of tokens using regular expression – Properties of regular expression – Finite Automata – Regular Expression to Finite Automata – NFA to Minimized DFA.

UNIT 2 Parser

9 Hrs

CFG – Derivation – CFG vs R.E. - Types Of Parser –Bottom UP: Shift Reduce Parsing - Operator Precedence Parsing, SLR parser- Top Down: Recursive Decent Parser - Non-Recursive Decent Parser.

UNIT 3 INTERMEDIATE CODE GENERATION

9 Hrs

Syntax directed translation scheme - Three Address Code – Representation of three address code - Intermediate code generation for: assignment statements - Boolean statements - switch case statement –Procedure call - Symbol Table Generation.

UNIT 4 CODE OPTIMIZATION

9 Hrs

Optimization - issues related to optimization – Basic block – Conversion of basic block to flow graph - loop optimization & its types – DAG - peephole optimization - Dominators - Data Flow optimization

UNIT 5 CODE GENERATION

9 Hrs

Issues involved in code generation – Register allocation – Conversion of three address code to assembly code using code generation algorithm – examples – Procedure for converting assembly code to machine code – Case study

Max. 45 Hours

TEXT / REFERENCE BOOKS

1. Alfred V. Aho, Jeffery D. Ullman & Ravi Sethi, "Compiler Principles, Techniques & Tools", Addison-Wesley Publishing Company, 1986.
2. Alfred V. Aho, Jeffery D. Ullman, "Principles of Compiler Design", Narosa Publishing House, 15th reprint, 1996.
3. D M. Dhamdhere, "System Programming", 2nd Edition, Tata McGraw Hill Publishing, 1999.



UNIT 1 LEXICAL ANALYSIS

- Structure of compiler
- Functions and Roles of lexical phase
- Input buffering
- Representation of tokens using regular expression
- Properties of regular expression
- Finite Automata
- Regular Expression to Finite Automata
- NFA to Minimized DFA.



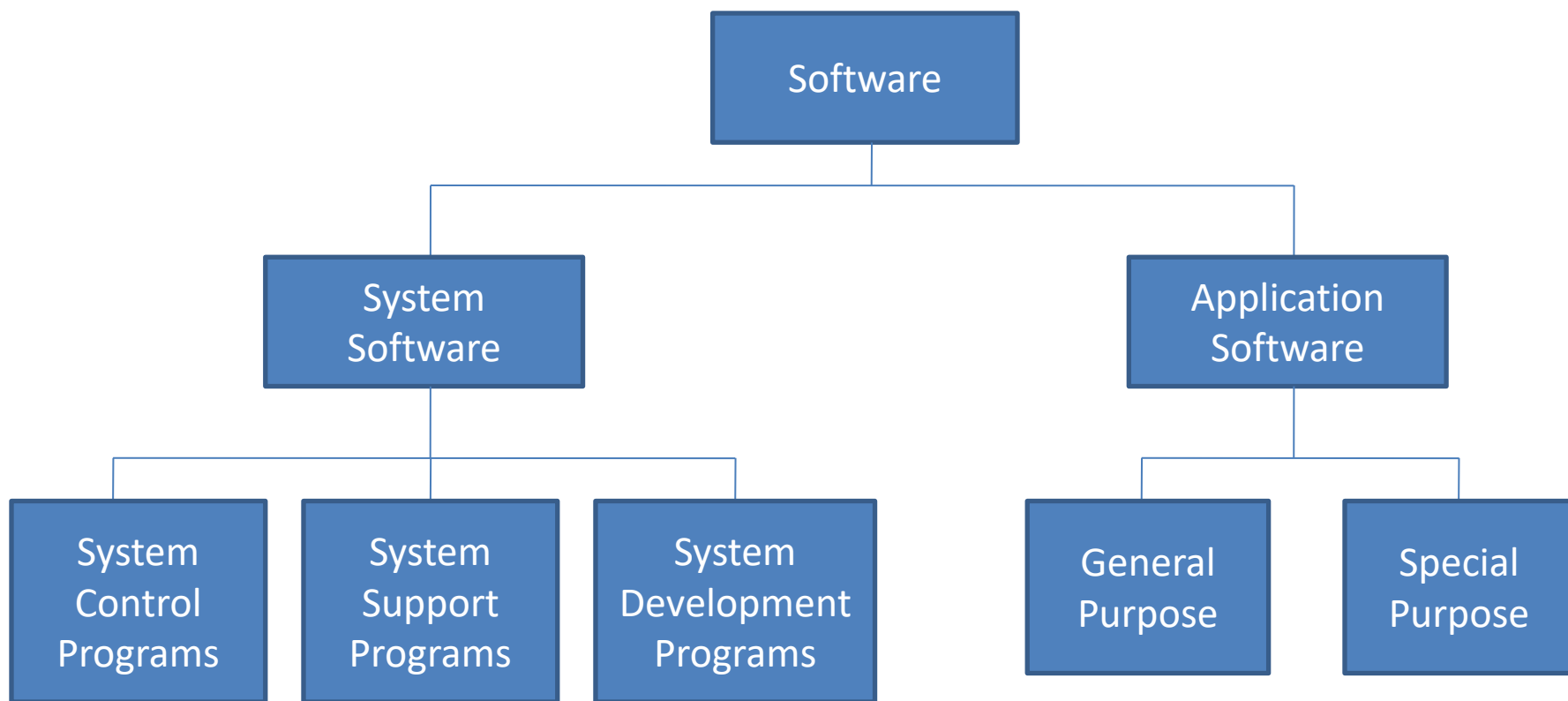
Outline

- Software
- Languages
- Language Processors
- Language Processing System
- Need for Compiler
- Structure of Compiler



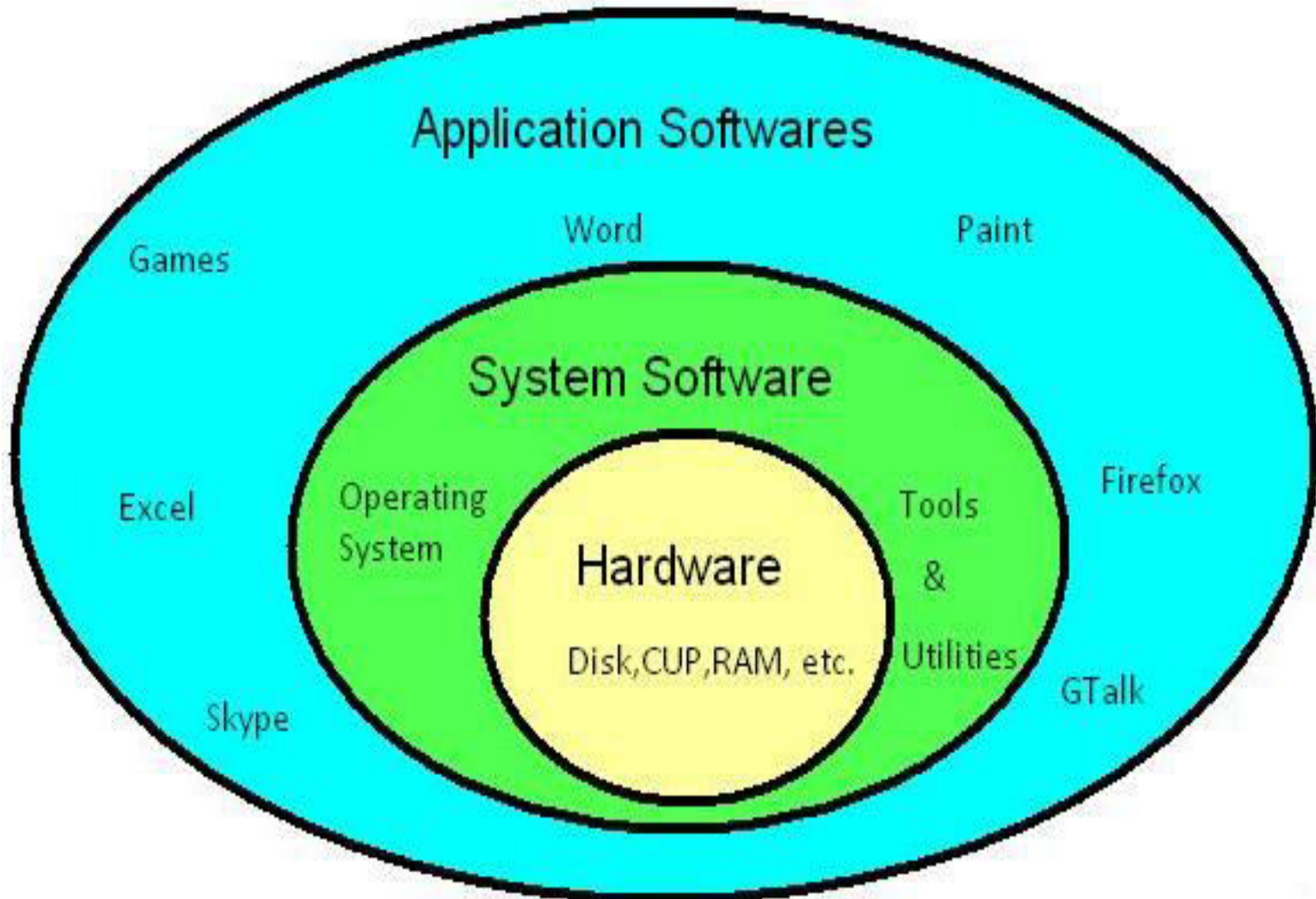
Software

- Software is a set of computer programs which are designed and developed to perform specific task desired by the user or the computer itself.





System S/W & Application S/W





- **System Control Programs:**

They control the execution of programs, manage the storage and processing resources of the computer and perform other management and monitoring functions. e.g.. OS

- **System Support Programs:**

They provide routine service functions to other computer programs and computer users. e.g.. Utility Programs

- **System Development Programs:**

They assist in the creation of publication programs.eg. Language translators like interpreters, compilers and assemblers.



Languages

- **Low Level language** is machine friendly. Commands or functions in the language map closely to processor instructions.
- **Assembly language** is machine dependent ,the mnemonics that are being used to represent instructions are not directly understandable by machine.
- **High Level language** is machine independent.
- A computer understands instructions in **machine code**, i.e. in the form of 0s and 1s. It is a tedious task to write a computer program directly in machine code.
- The programs are written mostly in high level languages like Java, C++, Python etc. and are called **source code**.



Language Processor

- Source code cannot be executed directly by the computer and must be converted into machine language to be executed.
- Hence, special translator system software is used to translate the program written in high-level language into machine code(object program / object code) is called **Language Processor**.

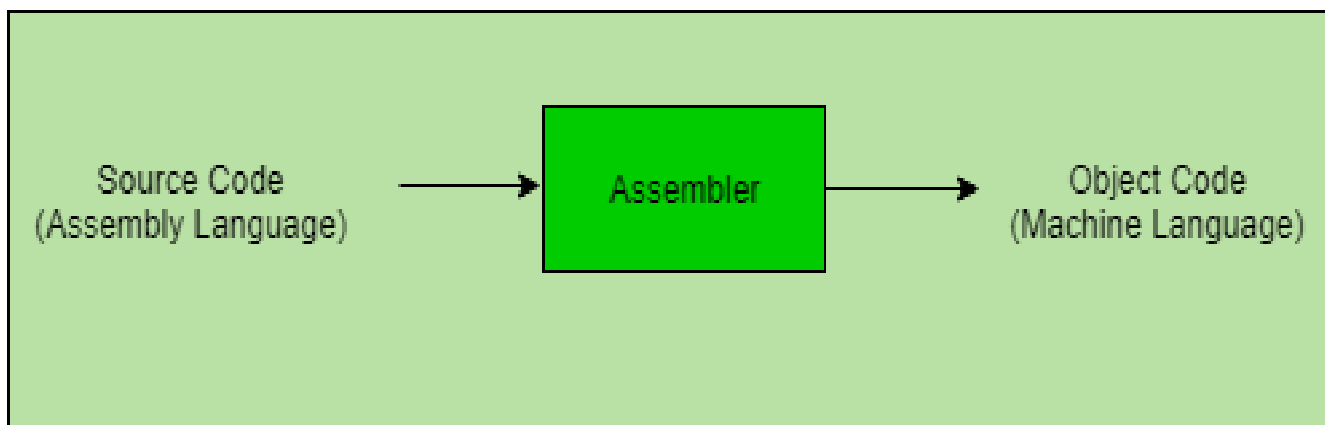
There are three types of translator programs:

- **Assembler**
- **Interpreter**
- **Compiler**



Assembler

- Translate the program written in Assembly language into machine code.

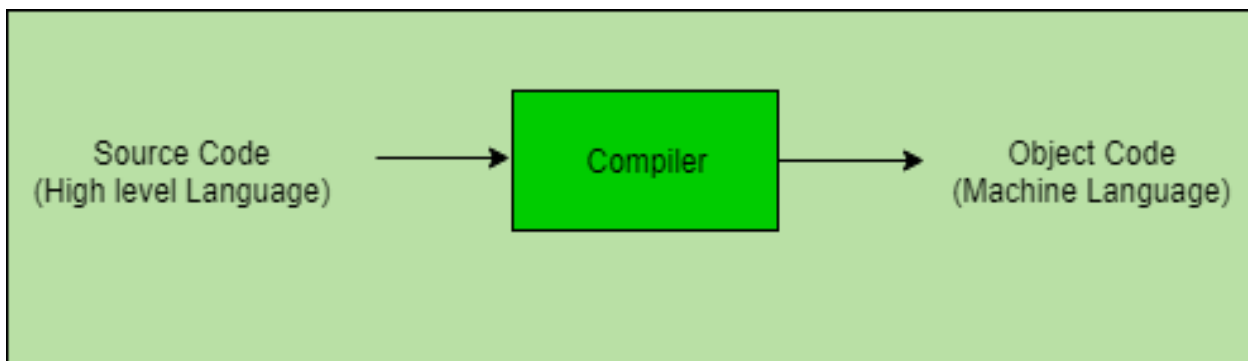




Compiler

- Reads the source program written in High Level Language(HLL) as a whole in one go and translates it into an equivalent machine language program.

Example HLL: C, C++, C#, Java...



- The compiler specifies the errors at the end of compilation with line numbers.
- The errors must be removed before the compiler can successfully recompile the source code again.



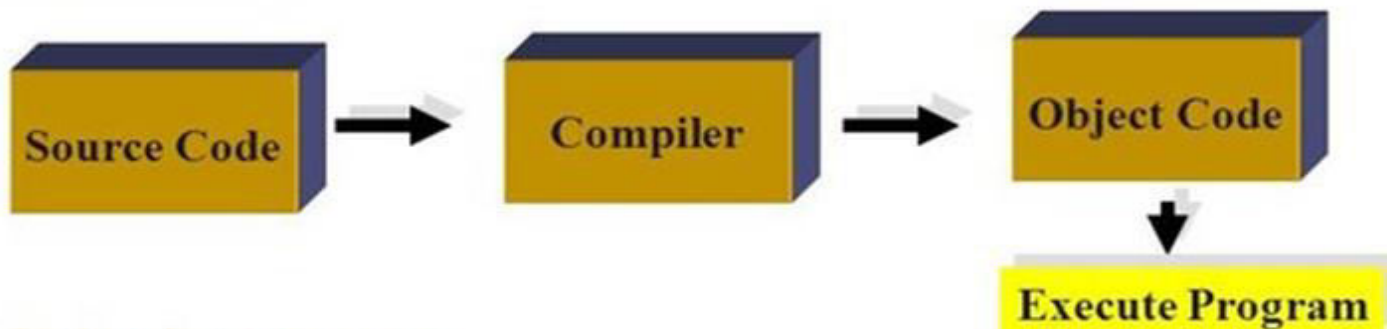
Interpreter

- The translation of single statement of source program into machine code and executes it immediately before moving on to the next line is called an interpreter.
- If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message.
- An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code. **Example:** Perl, Python and Matlab.
- An interpreter generally uses one of the following strategies for program execution:
 - Parse the source code and perform its behavior directly;
 - Translate source code into some efficient intermediate representation and immediately execute this;
 - Explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.

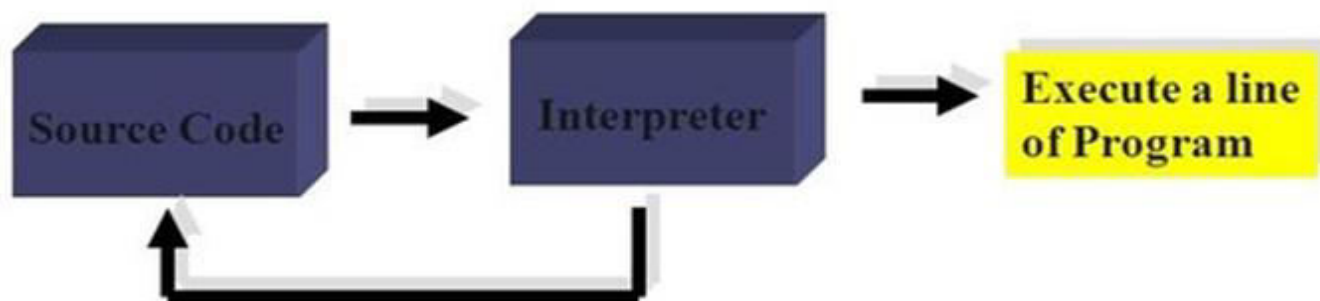


What is the difference?

- Using Compiler:



- Using Interpreter:



Once a program is compiled, its source code is not useful for running the code.

For interpreted programs, the source code is needed to run the program every time.

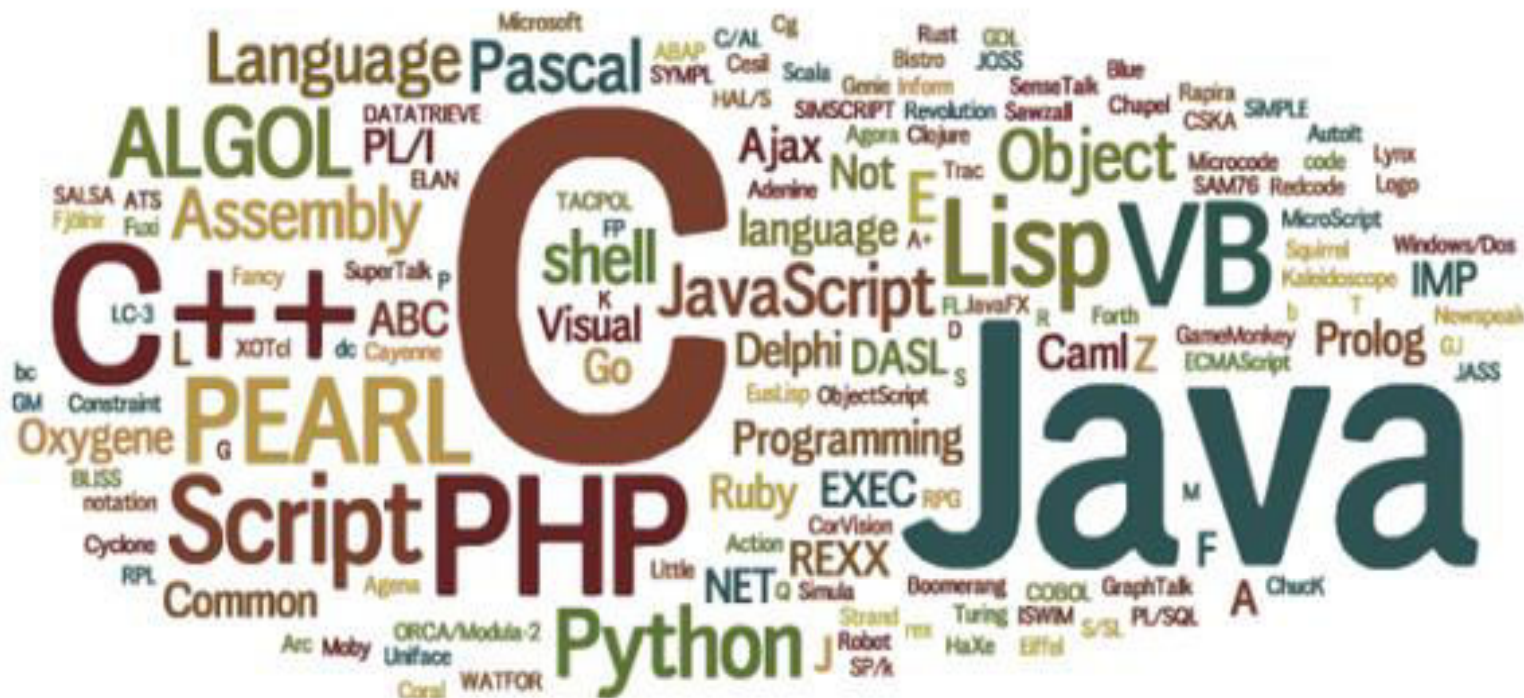


Compiler vs Interpreter

COMPILER	INTERPRETER
A compiler is a program which converts the entire source code of a programming language into executable machine code for a CPU.	Interpreter takes a source program and runs it line by line, translating each line as it comes to it.
Compiler takes large amount of time to analyze the entire source code but the overall execution time of the program is comparatively faster.	Interpreter takes less amount of time to analyze the source code but the overall execution time of the program is slower.
Compiler generates the error message only after scanning the whole program, so debugging is hard as the error can be present any where in the program.	Its Debugging is easier as it continues translating the program until the error is met.
Generates intermediate object code.	No intermediate object code is generated.



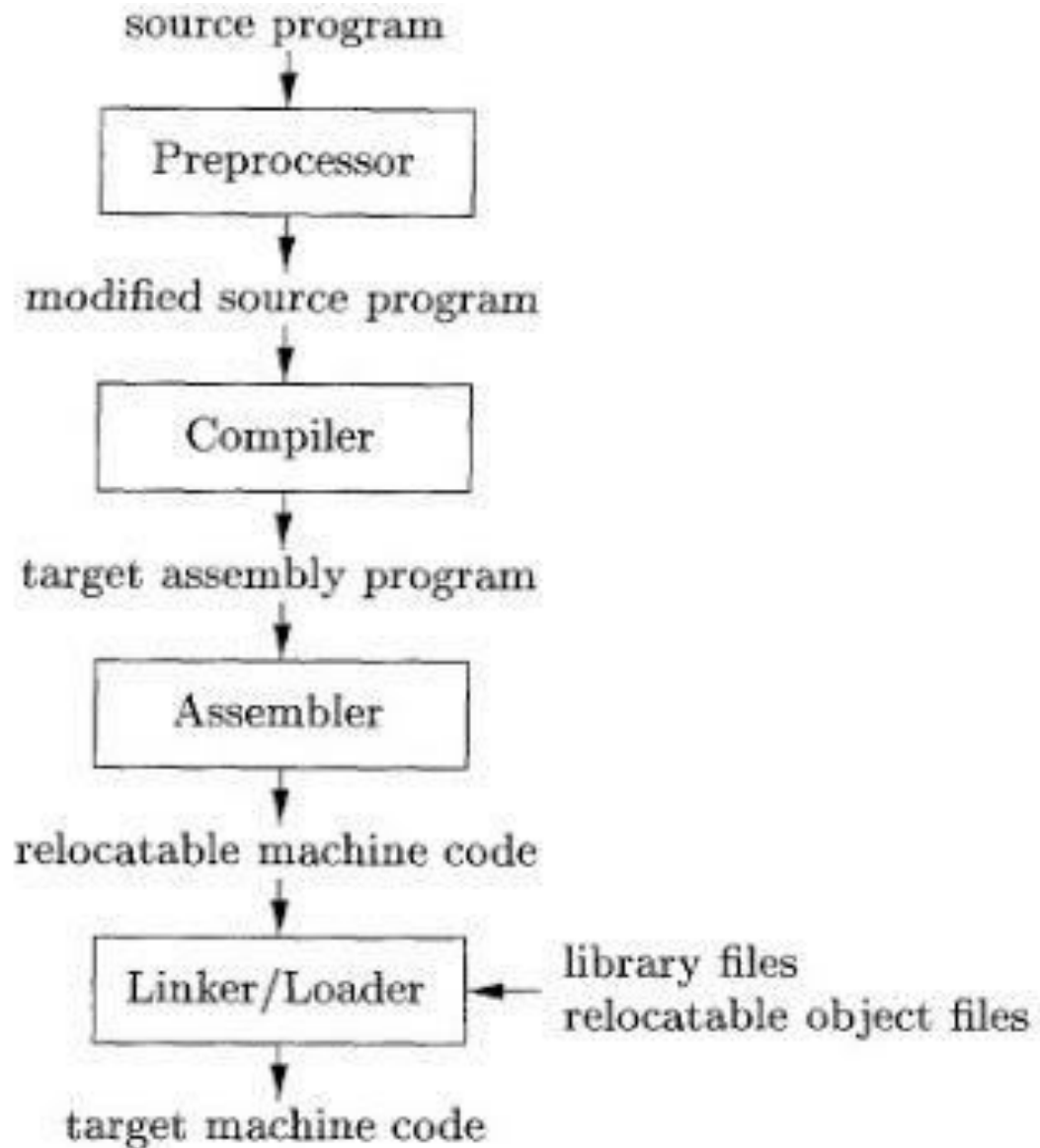
- Languages that use compilers; Pascal, C ++, Ada, Visual Basic, C and many more.



- Languages that use interpreters; Python, HTML, XML, PHP, Script Languages.



Language Processing System





Language Processing System

Preprocessor:

- It replaces macros with their definitions, discovers dependencies and resolves preprocessor directives.

Linker:

- Link and merge various object files to create an executable file.
- Search for called modules in a program and to find out the memory location where all modules are stored.

Loader:

- Performs the tasks of loading executable files into memory and run them.
- Calculates the size of a program which creates additional memory space.



Need for Compiler

- A Computer understands only binary language and executes instructions coded in binary language.
- It is difficult to write every program as a sequence of 0s and 1s?
- Humans are good at giving instructions in English language, whereas computers can only process binary language.
- So, there was a need of a translator that translates the computer instructions given in English language to binary language.

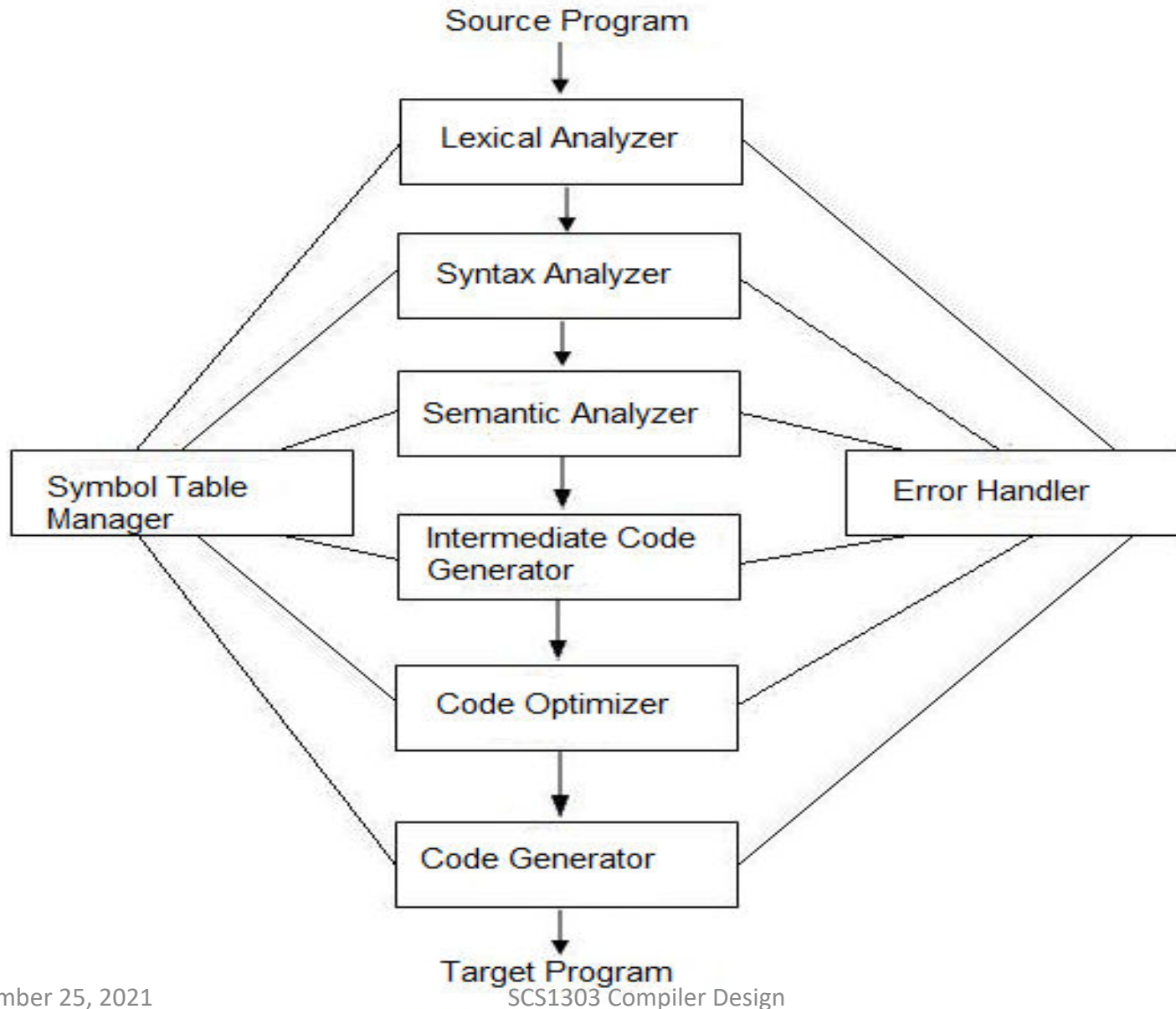


Applications

- The techniques used in compiler design can be applicable to many problems in computer science.
 - Techniques used in a lexical analyzer can be used in **text editors, information retrieval system and pattern recognition programs.**
 - Techniques used in a parser can be used in a **query processing system** such as SQL.
 - Many software having a complex front-end may need techniques used in compiler design.
 - Techniques used in compiler design can be used in **Natural Language Processing(NLP)** systems.



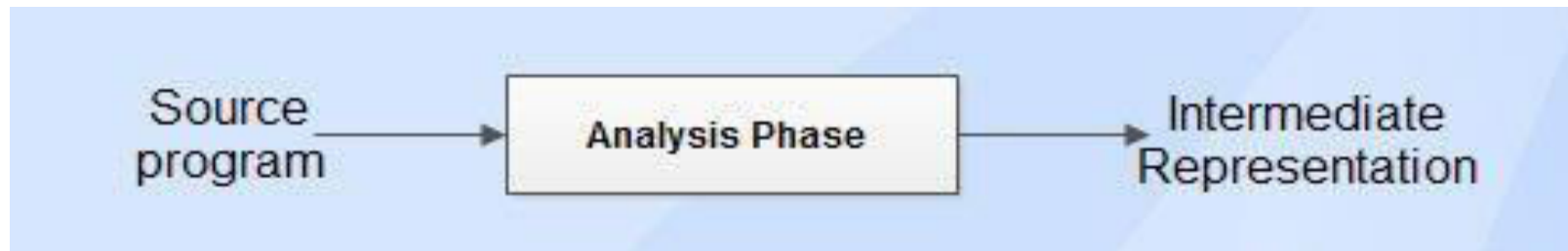
Structure of Compiler





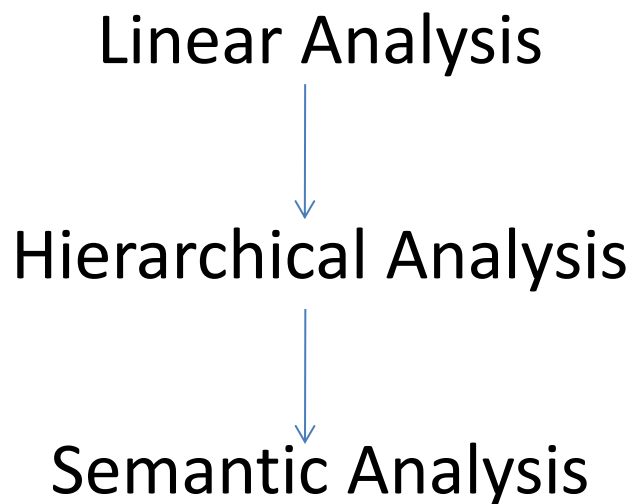
Analysis Part:

- Breaks up the source code into pieces and imposes a grammatical structure.
- Uses this structure to create an intermediate representation of the source code.
- Checks for syntax/semantics and provides messages to user in case of errors.
- Builds a symbol table to collect and store information about source code.
- It is also termed as front end of compiler.





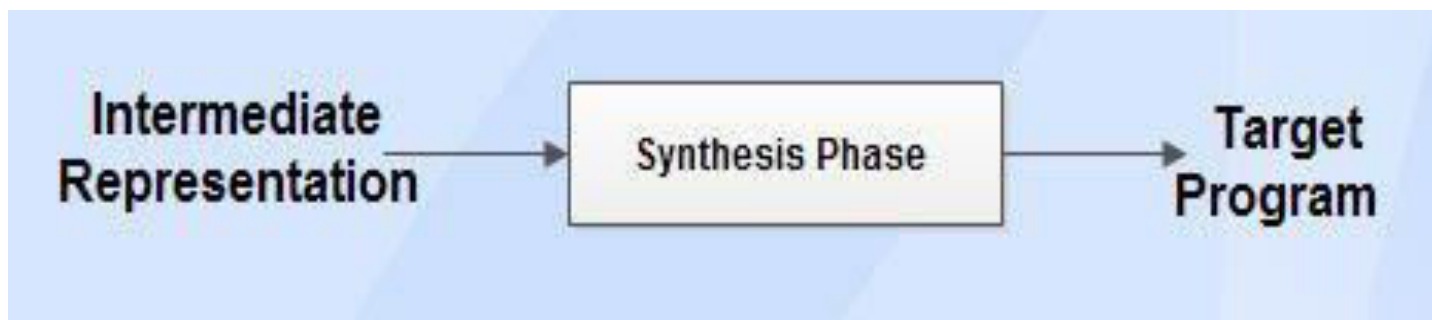
The analysis of a source program is divided into mainly three phases. They are:





Synthesis part:

- Synthesis part takes the intermediate representation as input and transforms it to the target program.
- It is also termed as back end of compiler.



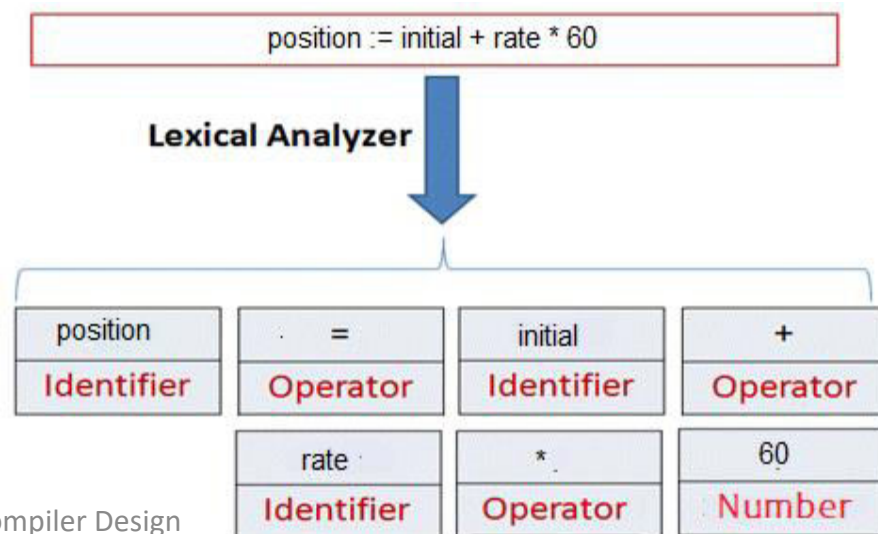


Lexical Analysis or Scanner

- Reads characters in the source code and groups them into meaningful word called **Token**.
- Tokens fall into these categories: Keywords, Operators, Identifiers, Constants, Literals strings and Punctuation.
- **Pattern**: Rule for a set of string in the input for which a token is produced as output.
- A **Lexeme** is a sequence of characters in the source code that is matched by the **Pattern** for a **Token**.

Input: stream of characters

Output: Token





Rules or regular expression

Regular expression for identifier:

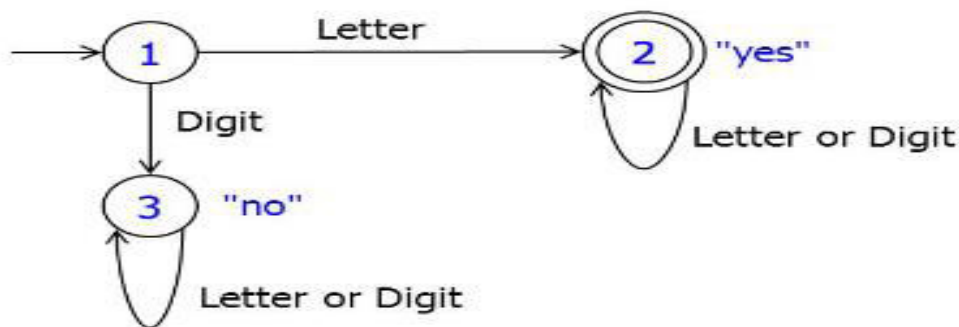
Letter (Letter | Digit)*

→ Kleene's Closure

→ alternate operator

→ concatenation operator

Transition diagram for identifiers



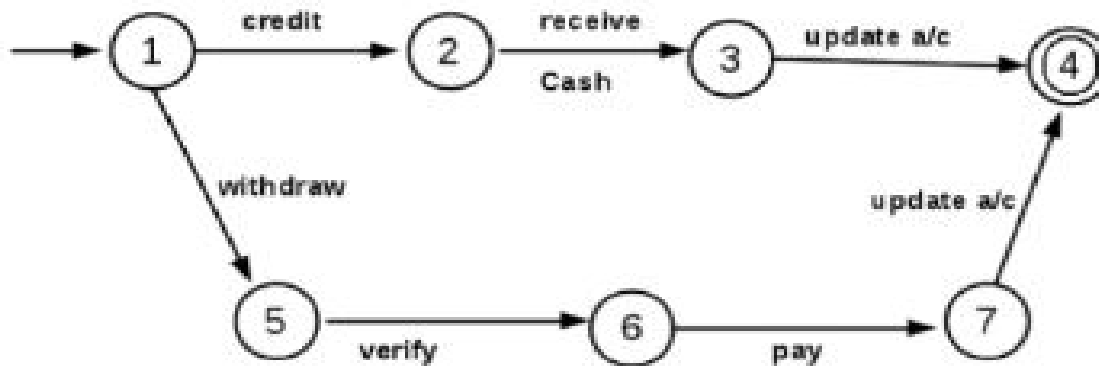
where ,

letter –alphabet set {a-z A-Z}

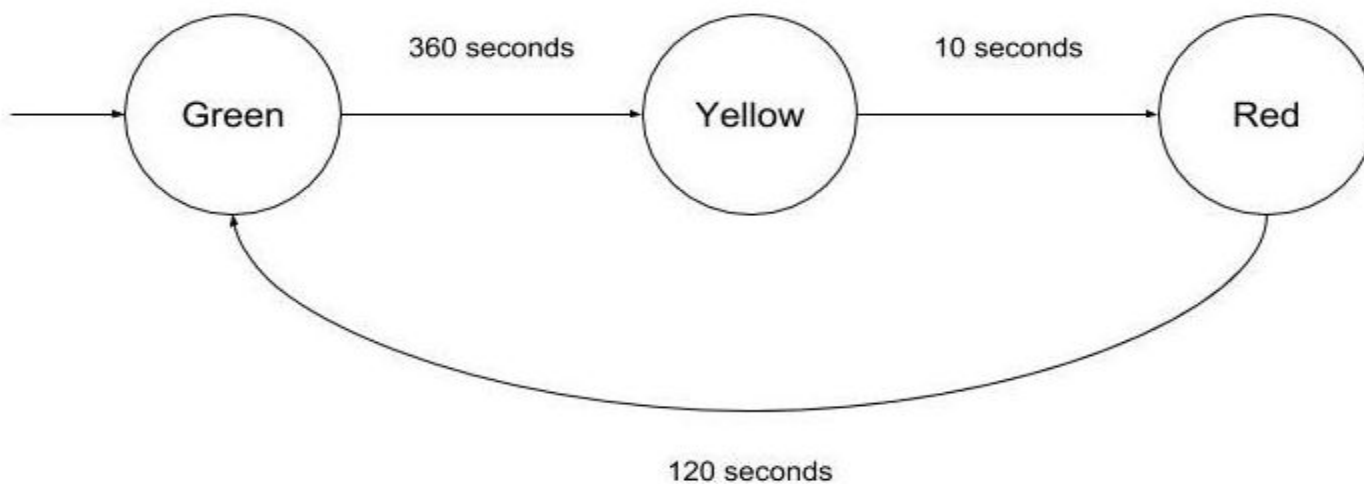
digit –number set {0-9}



APPLICATIONS



Transition diagram for Bank

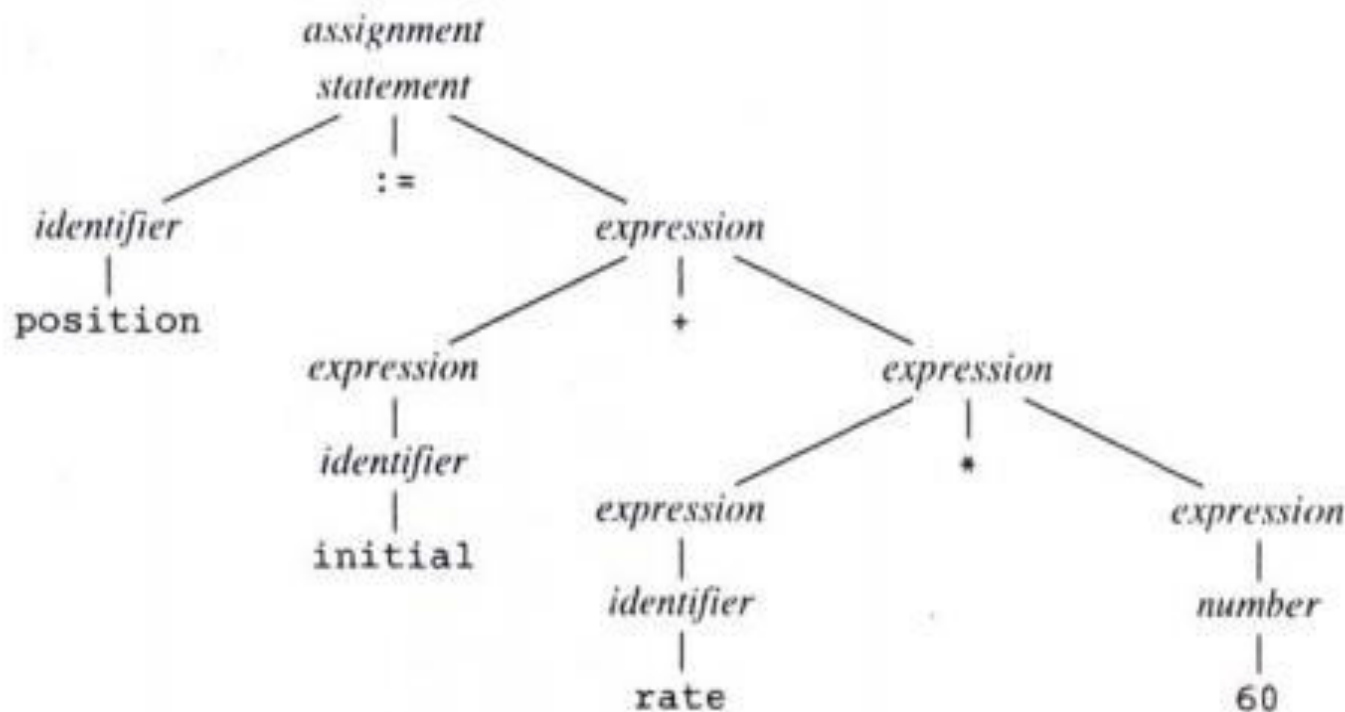


Transition diagram for Traffic signal



Syntax Analysis or Parser

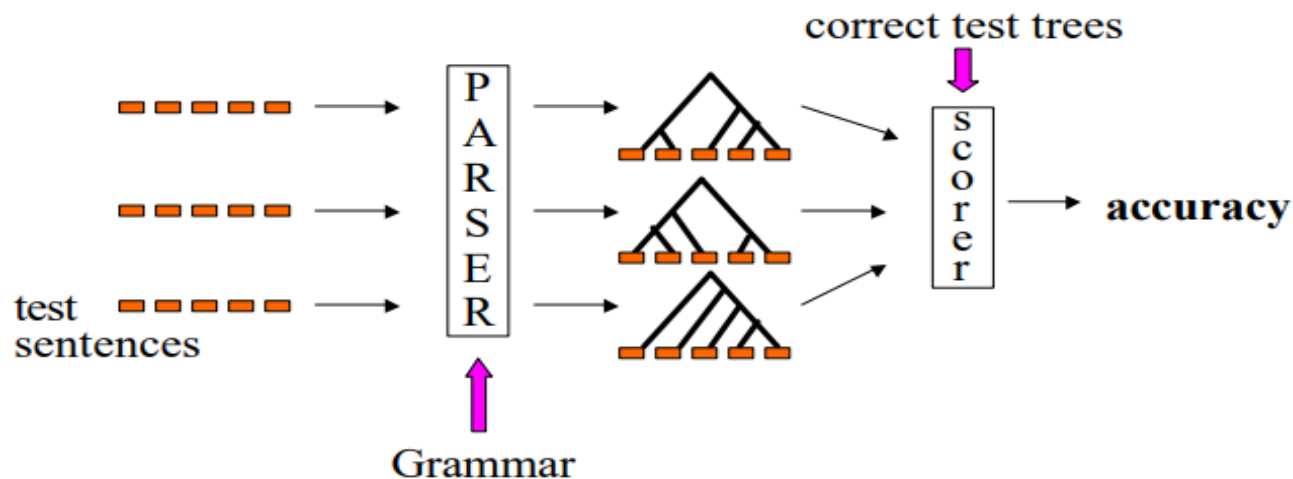
- Parser groups the tokens produced by lexical analyzer into grammatical phrases or syntactic variables.
- Generates a tree like representation called parse tree .
- A parse tree describes the syntactic structure of the input.



Parse tree for **position :=initial + rate * 60**



Applications

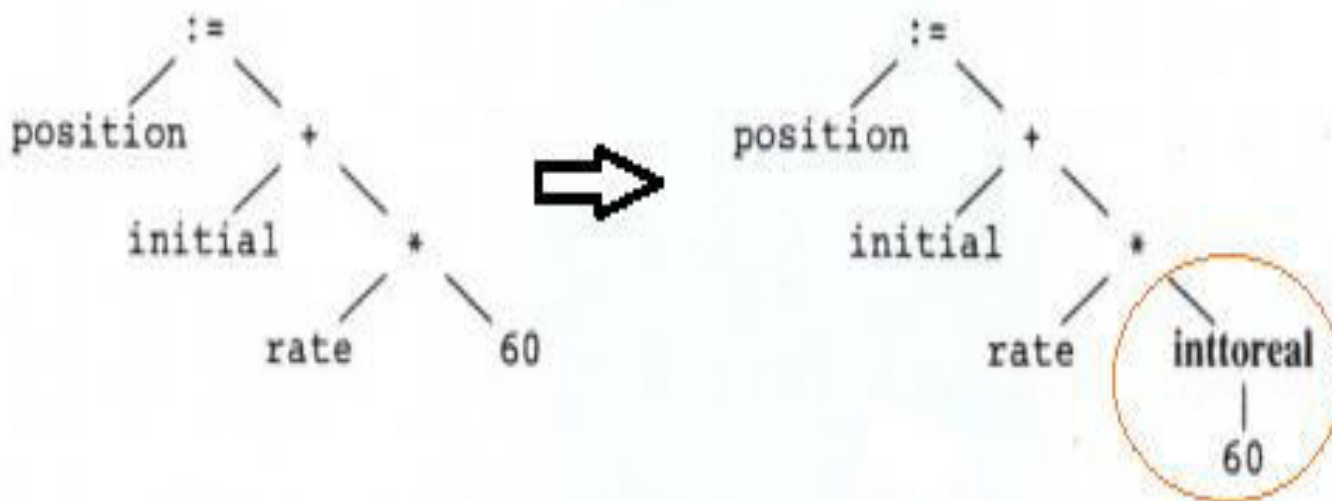


- Grammar Checking
- Indexing for information retrieval
- Information extraction
- Machine translation
- Speech synthesis from parses
- Speech recognition using parsing



Semantic Analysis

- Checks for semantic errors .
- Concentrates on **Type checking** -whether operands are type compatible.
- e.g. real number used to index an array.



Semantic analysis inserts a conversion of integer to real



Intermediate Code Generation

- Produces intermediate representations for the source program which are of the following forms:
 - Postfix notation
 - Three address code
 - Syntax tree
- Commonly used form is the three address code.

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

$t_2 = \text{id}_3 * t_1$

$t_3 = \text{id}_2 + t_2$

$\text{id}_1 = t_3$

where t_1 , t_2 , t_3 are compiler generated temporary variables.

Properties of intermediate code

- It should be easy to produce and easy to translate into target program.



Postfix

Infix Expression	Prefix Expression	Postfix Expression
$A + B * C + D$	$++A * B C D$	$A B C * + D +$
$(A + B) * (C + D)$	$* + A B + C D$	$A B + C D + *$
$A * B + C * D$	$+ * A B * C D$	$A B * C D * +$
$A + B + C + D$	$+++A B C D$	$A B + C + D +$



Code Optimizer

- Improves and produces optimized intermediate code as output.

e.g.

$$t_1 = id_3 * 60.0$$

$$id_1 = id_2 + t_1$$

- Helps in generating fast running machine code.
- Techniques used:
 - Common sub expression elimination
 - Dead code elimination
 - Constant folding
 - Copy propagation
 - Code motion
 - Reduction in strength
 - Induction variable elimination.....etc.



e.g. Constant folding

Before:

```
int f (void)
{
    return 3 + 5;
}
```

After:

```
int f (void)
{
    return 8;
}
```

e.g. Copy Propagation


Before:

```
y = x
z = 3 + y
```

After:

```
z = 3 + x
```

e.g. Code motion

<pre>for l := 1 to 100 do begin z := i; x := 25 * a; y := x + z; end;</pre>		<pre>x := 25 * a; for l := 1 to 100 do begin z := i; y := x + z; end;</pre>
---	---	---



Code Generation

- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- Using registers R1 and R2 the translation of the e.g. is:

MOV id3 , R2

MUL #60.0 , R2

MOV id2 , R1

ADD R2 , R1

MOV R1 , id1



Symbol Table Management

- Stores all the information about identifiers used in the program.
- Data structure containing a record for each identifier, with fields for the attributes of the identifier.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

SYMBOL TABLE		
1	position	...
2	initial	...
3	rate	...
4		



Error Handling

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.

Few errors:

- In lexical analysis: misspelled name of some identifier.
- In syntax analysis: missing semicolon or unbalanced parenthesis.
- In Semantic analysis: Type mismatch.
- In code optimization, errors occur when the result is affected by the optimization.
- In code generation, it shows error when code is missing etc.



Translation of a statement

SYMBOL TABLE		
1	position	---
2	initial	---
3	rate	---
4		

position := initial + rate * 60

lexical analyzer

id₁ := id₂ + id₃ * 60

syntax analyzer

id₁ := id₂ + id₃ * 60

semantic analyzer

id₁ := id₂ + id₃ * inttoreal
60

intermediate code generator

temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3

code optimizer

temp1 := id3 * 60.0
id1 := id2 + temp1

code generator

MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1



References

1. Alfred V. Aho, Jeffery D. Ullman & Ravi Sethi, " Compiler Principles, Techniques & Tools", Addison- Wesley Publishing Company,1986.
2. Alfred V. Aho, Jeffery D. Ullman, "Principles of Compiler Design", Narosa Publishing House, 15th reprint, 1996.
3. D M. Dhamdhere , "System Programming", 2nd Edition, Tata McGraw Hill Publishing, 1999.



THANK YOU



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited with Grade "A" by NAAC | Approved by AICTE



UNIT-I

LEXICAL ANALYSIS



Outline

- Role of Lexical Analyzer
 - Lexical Analysis vs. Parsing
 - Tokens, Patterns and Lexemes
 - Attributes for tokens
 - Lexical Errors
- Input Buffering

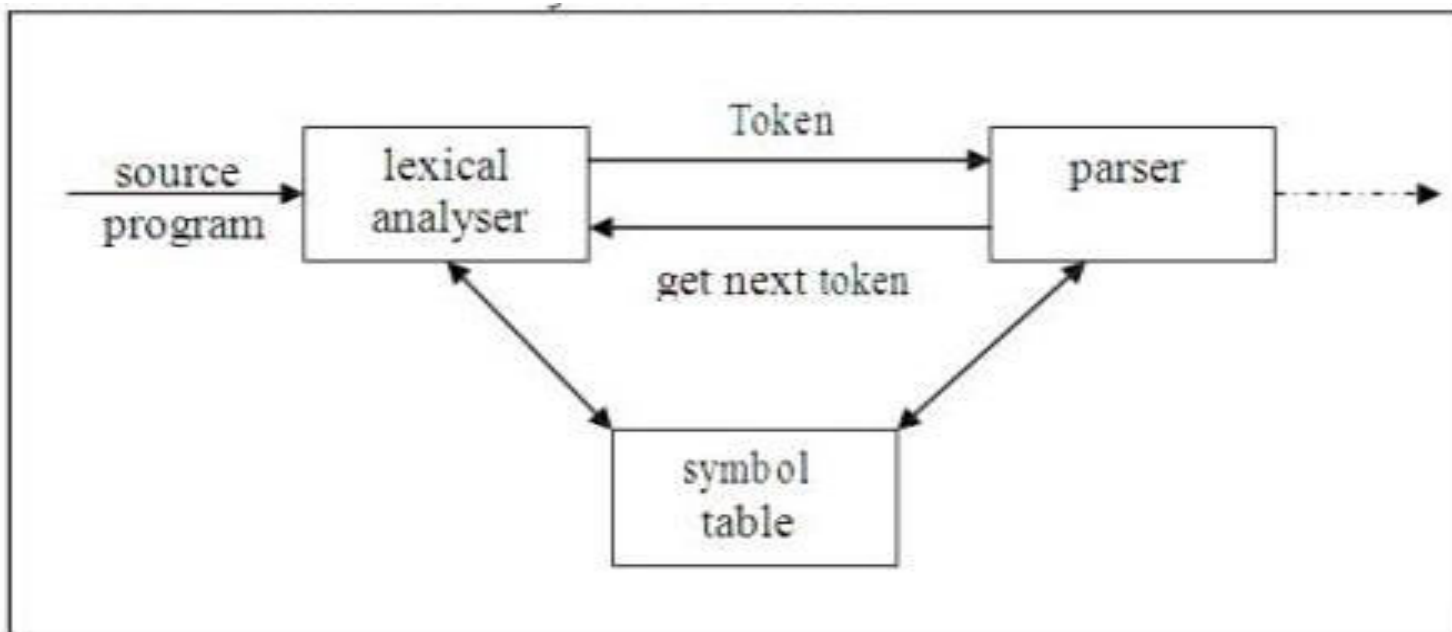


The Role of Lexical Analyzer

- First phase of a compiler

Main Task:

- Reads the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.





The Role of Lexical Analyzer (Contd..)

Secondary tasks:

- Eliminates the following from the source program:

- comments

```
//global variables
```

- whitespaces

- tab

```
printf(    a);
```

- newline characters

```
a=a*2  
printf(a)
```



The Role of Lexical Analyzer (Contd..)

- Correlating the error messages from the compiler with the source program.
 - e.g. keeps track of the number of newline characters seen, so that a line number can be associated with an error message.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int sum=0,n1,n2;
    clrscr();
    printf("enter two number= ");
    scanf ("%d%d",&n1,&n2);
    sum=n1+n2;
    div=n1/0;
    printf("Sum =: %d",sum);
    getch();
}
```

Compile time error
div is not defined but here
it is used in program

Message
Compiling ..\NSS.C:
Error ..\NSS.C 10: Undefined symbol 'div'

F1 Help Space View source Edit source F10 Menu

- Making a copy of source program with error marked(in some compilers).



Lexical analysis vs. Parsing

1. Simplicity of design

- Separation of lexical from syntax phase -> simplify at least one of the tasks e.g. parser dealing with white spaces is complex job
- Unnecessary tokens can be eliminated by scanner

2. Improving compiler efficiency

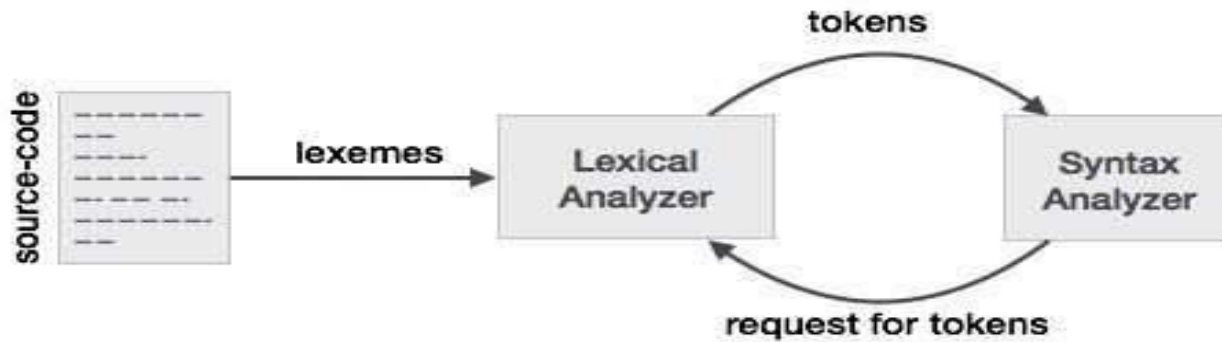
- Liberty to apply specialized techniques that serves only lexical tasks not the whole parsing.
- Speedup reading input characters using specialized buffering techniques.

3. Enhancing compiler portability

- Input device peculiarities are restricted to the lexical analyzer, i.e. specialized symbols and characters (language and machine specific) are isolated during this phase.



Tokens, Patterns and Lexemes



Tokens: Sequence of characters that have a collective meaning.

- **Identifiers:** names the programmer chooses.
- **Keywords:** names already in the programming language.
- **Separators** (also known as punctuators): punctuation characters and special symbols.
- **Operators:** symbols that operate on arguments .
- **Literals:** numeric, logical, textual, reference literals.

A token is a pair a token name and an optional token value



Tokens, Patterns and Lexemes(contd..)

- **Patterns:** There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.
- **Lexeme:** A sequence of characters in the source program that is matched by the pattern for a token.
- **Examples of tokens:**

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but ", surrounded by "'s	"core dumped"



Attributes for tokens

- When more than one lexeme matches a pattern the lexical analyzer must provide additional information about that lexeme matched.

e.g.

E = M * C ** 2;

Lexeme	<token, token attribute>
E	<id, pointer to symbol table entry for E>
=	<assign-op,>
M	<id, pointer to symbol table entry for M>
*	<mult-op,>
C	<id, pointer to symbol table entry for C>
**	<exp-op,>
2	<number, integer value 2>
;	<separator, >



Lexical errors

- Some errors are out of power of lexical analyzer to recognize, e.g. **fi** ($a == f(x)$)
- However it may be able to recognize errors like:
e.g. $d = 2r.....$

such errors are recognized when no pattern for tokens matches a character sequence.

Panic mode Recovery:

- Delete successive characters from the remaining input until the analyzer can find a well-formed token.(May confuse the parser-creating syntax error)

Other Error-recovery actions:

1. Delete an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transposing two adjacent characters



Input Buffering

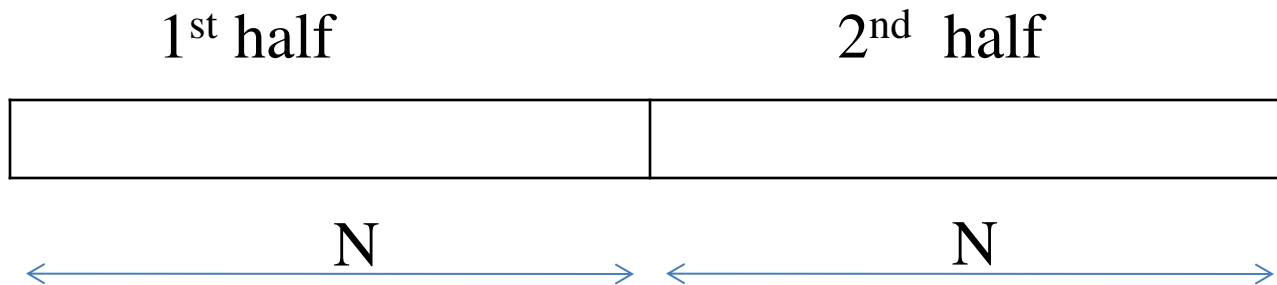
Why we need input buffers?

- Scanner performance is crucial:
 - This is the only part of the compiler that examines the entire source code one character at a time.
 - The scanner accounts for ~25-30% of total compile time.
 - Lexical analyzer needs to lookahead some symbols to decide about the token to return.
- **Speedup** the reading the source program.
- We have two input buffer scheme that is useful when lookahead is necessary,
 - **Buffer pairs**
 - **Sentinels**



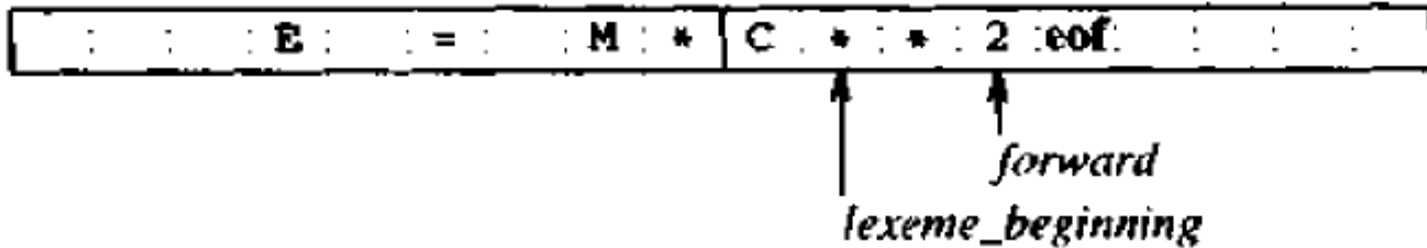
Buffer Pairs

- Buffer is divided into two N-characters halves.
- N is number of characters e.g. 1024 or 4096





Buffer Pairs



An input buffer in two halves.

- **Lexeme_beginning** :marks the beginning of the current lexeme.
- **forward** : scans ahead until a pattern match is found.



Buffer Pairs (contd..)

- Reads N input characters into each half of the buffer with system read command.
- If fewer than N characters remain in the input ,then a special character ***eof*** is read into the buffer after the input characters.
- The string of characters between the two pointers is the current lexeme.
- Initially both pointer points to the first character of the next lexeme to be found.
- With this scheme , comments and white spaces can be treated as pattern that yield no token.



Buffer Pairs(contd..)

- This buffering scheme works well most of the time, but the amount of lookahead is limited.
 - Impossible to recognize token if the distance that the forward pointer must travel is more than the length of the buffer.

e.g. **DECLARE(ARG1,ARG2,ARG3.....ARGn)**

- unable to determine whether **DECLARE** is a keyword or an array name until the character that follows the right parenthesis is reached.
- In C language: we need to lookahead after,
-, = or < to decide what token to return.



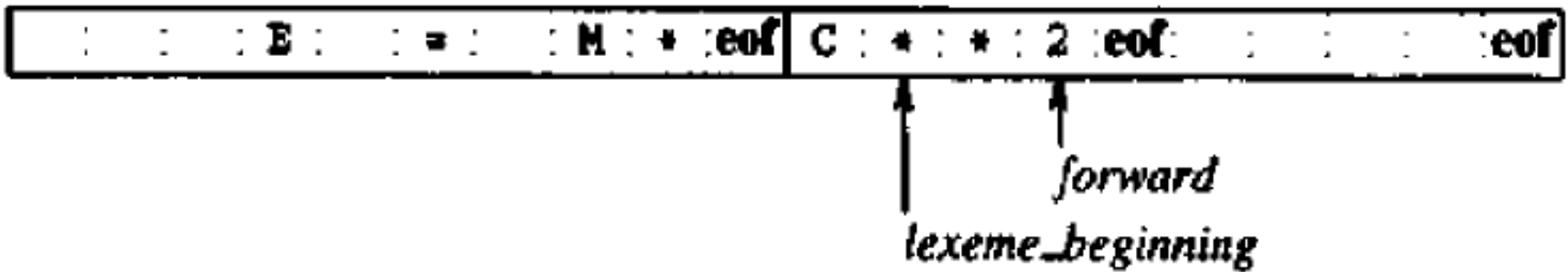
Code to advance forward pointer

```
if forward at end of first half then begin  
    reload second half;  
    forward := forward + 1  
end  
else if forward at end of second half then begin  
    reload first half;  
    move forward to beginning of first half  
end  
else forward := forward + 1;
```




Sentinels

- A special character at the buffer end.



Sentinels at end of each buffer half.



Lookahead code with sentinels

```
forward := forward + 1;  
if forward↑ = eof then begin  
    if forward at end of first half then begin  
        reload second half;  
        forward := forward + 1  
    end  
    else if forward at end of second half then begin  
        reload first half;  
        move forward to beginning of first half  
    end  
    else /* eof within a buffer signifying end of input */  
        terminate lexical analysis  
end
```



THANK YOU



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited with Grade "A" by NAAC | Approved by AICTE



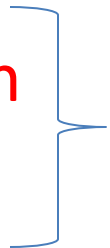
UNIT-I

LEXICAL ANALYSIS



UNIT 1 LEXICAL ANALYSIS

- Structure of compiler
- Functions and Roles of lexical phase
- Input buffering
- Representation of tokens using regular expression
- Properties of regular expression
- Finite Automata
- Regular Expression to Finite Automata
- NFA to Minimized DFA.





Outline

- Specification of tokens
 - Strings and Languages
 - Regular Expressions
- Recognition of tokens
 - Transition Diagrams
- A Language for specifying Lexical Analyzer
 - LEX Tool



Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens.
- While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for token.
- Regular expressions are means for specifying regular languages.
- Example:
letter_(letter | digit)*



Strings and Languages

- **Alphabet:**
 - An alphabet is any finite set of symbols such as letters, digits and punctuation.(denoted by Σ)
 - The set $\{0,1\}$ is binary alphabet
- **String:** (denoted by w or s)
 - Finite sequence of symbol on an alphabet.
 - Sentence and word are also used in terms of string.
 - ϵ is the empty string.
 - $|s|$ is the length of string s .
- **Language:** (denoted by L)
 - Set of strings over some fixed alphabet.
 - \emptyset representing empty set , $\{\epsilon\}$ is a language.
 - Set of all possible identifiers is a language.



Terms for parts of a string

<i>TERM</i>	<i>DEFINITION</i>
prefix of s	A string obtained by removing zero or more trailing symbols of string s ; ban is a prefix of banana.
suffix of s	A string formed by deleting zero or more of the leading symbols of s ; nana is a suffix of banana.
substring of s	A string obtained by deleting a prefix and a suffix from s ; nan is a substring of banana. Every prefix and every suffix of s is a substring of s , but not every substring of s is a prefix or a suffix of s . For every string s , both s and ϵ are prefixes, suffixes, and substrings of s .
proper prefix, suffix, or substring of s	Any nonempty string x that is, respectively, a prefix, suffix, or substring of s such that $s \neq x$.
subsequence of s	Any string formed by deleting zero or more not necessarily contiguous symbols from s ; baaa is a subsequence of banana.



Operations on Languages

- **Concatenation:**

$$L_1 L_2 = \{s_1 s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2\}$$

- **Union:**

$$L_1 \cup L_2 = \{s \mid s \in L_1 \text{ or } s \in L_2\}$$

- **Exponentiation:**

$$L^0 = \{\varepsilon\} \quad L^1 = L \quad L^2 = LL$$

- **Kleene's Closure:**

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

- **Positive Closure:**

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$



Examples

Consider

$$L_1 = \{A, B, \dots, Z, a, b, \dots, z\} \quad L_2 = \{0, 1, 2, \dots, 9\}$$

- $L_1 L_2 = \{a1, a2, b1, b2, c1, c2, d1, d2, \dots\}$ i.e. set of string consisting of a letter followed by digit.
- $L_1 \cup L_2 = \{ab, c, d, 1, 2, \dots\}$ i.e. set of letters and digit.
- $L_1^3 =$ all strings with length three using L_1
- $L_1^* =$ set of all strings of letters ,including ϵ , the empty string.(zero or more occurrences)
- $L_2^+ =$ set of all strings of one or more numbers. (one or more occurrences)
- $L_1 (L_1 \cup L_2)^* = ?$



Regular Expressions(RE)

- RE is used to describe tokens of a programming language.
- Each RE denotes a language.
- A language denoted by a RE is called as a regular set.

Definition of RE:

1. ϵ is a regular expression, $L(\epsilon) = \{\epsilon\}$
2. If ***a*** is a symbol in Σ then ***a*** is a regular expression, $L(a) = \{a\}$
3. Suppose r and s are regular expressions denoting the language $L(r)$ and $L(s)$
 - a. $(r) \mid (s)$ is a regular expression denoting the language $L(r) \cup L(s)$
 - b. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$
 - c. $(r)^*$ is a regular expression denoting $(L(r))^*$
 - d. (r) is a regular expression denoting $L(r)$



- Unnecessary parenthesis can be avoided in regular expression if we adopt the conventions that:
 1. The unary operator $*$ has the higher precedence and is left associative.
 2. Concatenation has the second highest precedence and is left associative.
 3. $|$ has the lowest precedence and is left associative.
- Under these conventions $(a) | ((b)^*(c))$ is equivalent to $a | b^*c$.
 - denote the set of strings that are either a single a or zero or more b 's followed by one c .



Writing Regular Expressions

1. The RE $a|b$ denotes the set $\{a,b\}$
2. The RE $(a|b)(a|b)$ denotes $\{aa,ab,ba,bb\}$ or equivalent RE $aa|ab|ba|bb$
3. The RE a^* denotes the set of all strings of zero or more a's i.e. $\{\epsilon, a, aa, aaa, \dots\}$
4. Write the RE for the language of even no of 1's.
 $\Sigma = \{1\}$
 $w = \{\epsilon, 11, 1111, \dots\}$
 $RE = (11)^*$
5. Write the RE for the language of odd no of 1's.
 $\Sigma = \{1\}$
 $w = \{1, 111, 11111, \dots\}$
 $RE = (11)^*.1$



Writing Regular Expressions

6. Write the RE with a string starting with a.

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

$w = \{a, ab, aba, aab, \dots\}$

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

7. Write the RE with a string starting with either a or ab.

$w = \{a, ab, aba, aab, \dots\}$

RE = $(a+ab)(a+b)^*$

8. Write RE with a string consist of a's and b's any length including ϵ (λ).

$w = \{\epsilon, a, b, ab, ba, aba, aab, \dots\}$

RE = $(a+b)^*$

9. Write RE with a string consists of a's and b's ending with abb.

$w = \{abb, aabb, babb, baabb, \dots\}$

RE = $(a+b)^*abb$



Writing Regular Expressions

1. Write the RE for identifiers in 'C' programming.

RE= (letter | _)(letter | digit | _)*

2. Write the RE for identifiers in 'C' programming.

RE= (digit)+

or

RE= digit (digit)*

– letter -> [A-Za-z]

– digit -> [0-9]



Algebraic properties of RE

Axiom	Description
$r \mid s = s \mid r$	\mid is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	\mid is associative
$(r s) t = r (s t)$	concatenation is associative
$r (s \mid t) = rs \mid rt$ $(s \mid t) r = sr \mid tr$	concatenation distributes over \mid (\mid = alternation)
$r\epsilon = r$ $\epsilon r = r$	ϵ is the identity for concatenation
$(r \mid \epsilon)^* = r^*$	relation between $*$ and ϵ
$r^{**} = r^*$	$*$ is idempotent

where $r, s,$ and t are regular expressions



Regular Definitions

- Giving names to regular expression and define RE using these names.
- Regular definition is a sequence of definitions of the form,

$$d_1 \longrightarrow r_1$$

$$d_2 \longrightarrow r_2$$

...

$$d_n \longrightarrow r_n$$

Where d_i is a each r_i is a regular expression over the symbols in $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$

Examples:

$$\text{letter} \longrightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$$

$$\text{digit} \longrightarrow 0 \mid 1 \mid \dots \mid 9$$

$$\text{identifier} \longrightarrow \text{letter}(\text{letter} \mid \text{digit})^*$$



Recognition of tokens

- Knowing how to specify the tokens for a language ,we need to know how to recognize them.

if \rightarrow if

then \rightarrow then

else \rightarrow else

relop \rightarrow < | <= | = | <> | > | >=

id \rightarrow letter (letter|digit)*

num \rightarrow digit⁺ (.digit⁺)? (E(+|-)?digit⁺)?

- We also need to handle whitespaces:

delim \rightarrow blank | tab | newline

ws \rightarrow delim⁺



Regular Expression patterns for tokens

Regular Expression	Token	Attribute value
<i>ws</i>	-	-
<i>if</i>	if	-
<i>then</i>	then	-
<i>else</i>	else	-
<i>id</i>	id	ptr to sym table entry
<i>num</i>	num	ptr to sym table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE



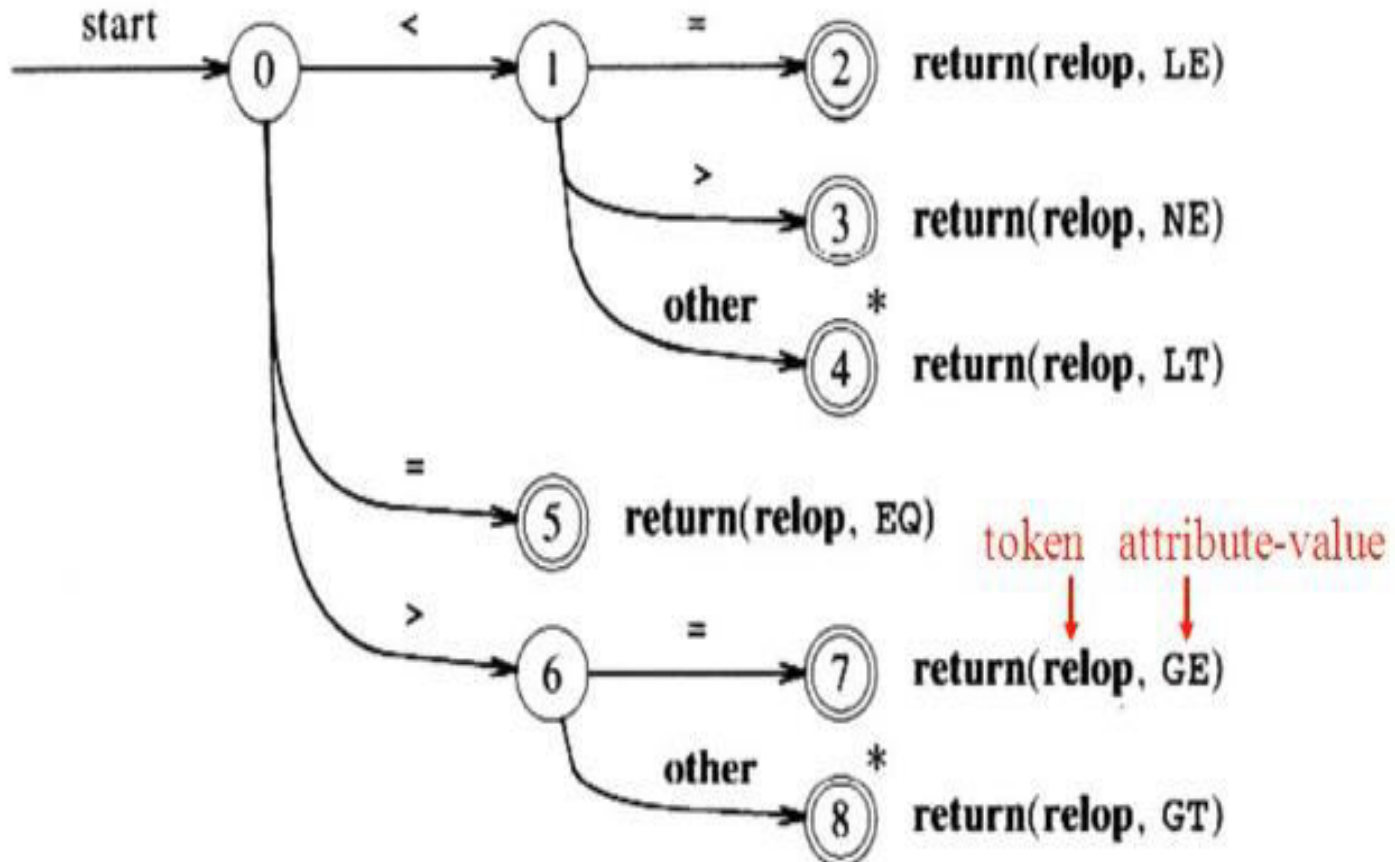
Transition Diagrams

- Lexical Analysis use transition diagram to keep track of information about characters that are seen as the forward pointer scans the input.
- Positions in a transition diagram are drawn as circles and are called **states**.
- The states are connected by arrows called **edges**.
- A double circle indicates an **accepting state** ,in which a token is found.
- * indicates that input retraction must take place.



Transition diagrams

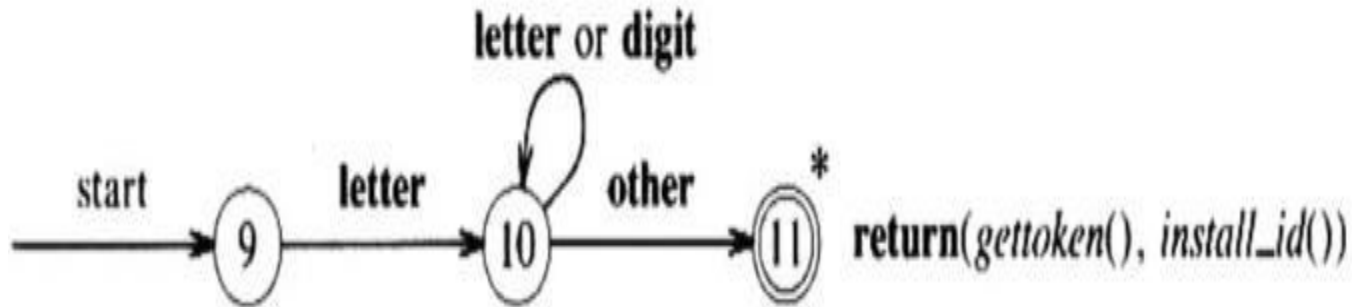
- Transition diagram for *relop*





Transition diagrams (contd.)

- Transition diagram for reserved words and identifiers



•

gettoken(): return token(***id,if.then***,.....)if it looks the symbol table

- Install_id(): return 0 if **keyword** or a pointer to the symbol table entry if ***id***.



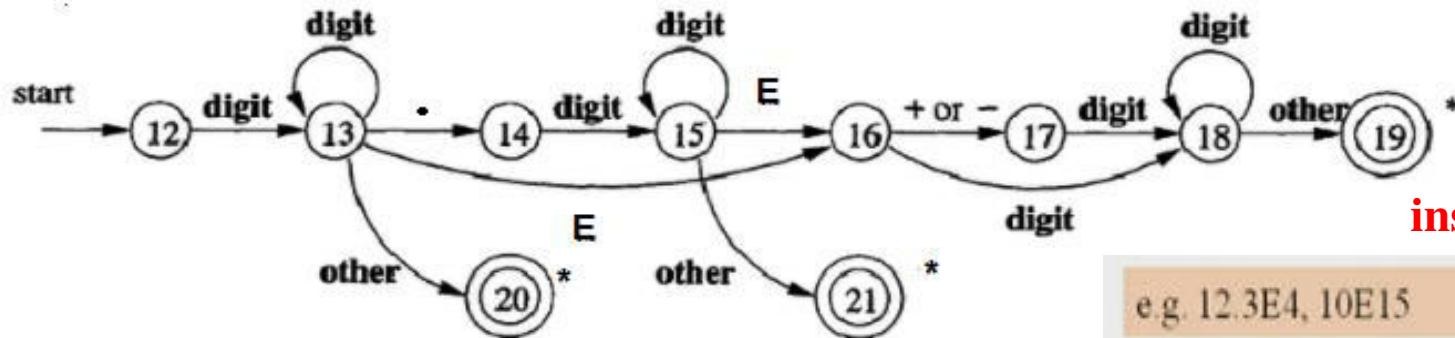
Transition diagrams (contd.)

Transition diagram for unsigned numbers

$\text{num} \rightarrow \text{digit}^+ (. \text{digit}^+)? (E(+|-)? \text{digit}^+)?$

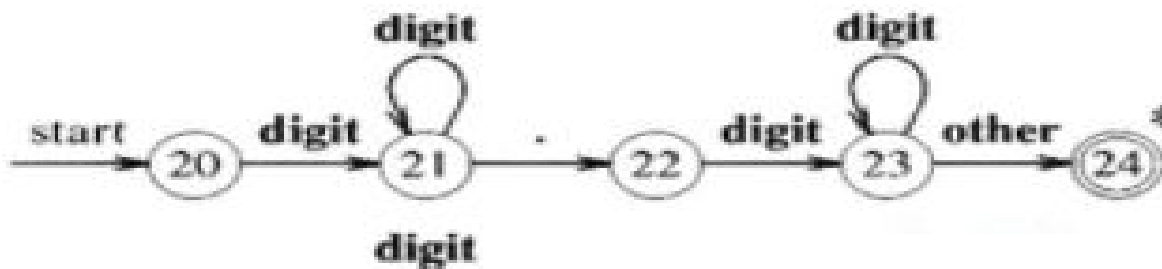
Fractional part

Exponential part



`install_num()`

e.g. 12.3E4, 10E15



`install_num()`



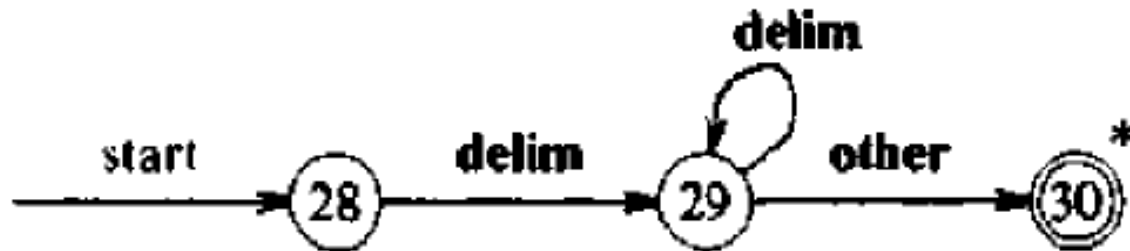
`install_num()`



Transition diagram for whitespace

delim \rightarrow **blank** | **tab** | **newline**

ws \rightarrow **delim**⁺



- Nothing is returned when the accepting state is reached.
- Goes back to first transition diagram to look for another pattern.



Following Transition diagrams

- Transition diagrams are followed one by one trying to determine the next tokens to be returned.
- If failure occurs ,we **retract(*)** the forward pointer to where it was in the start state of the diagram, and activate the next transition diagram.
- If failure occurs in all diagrams then a lexical error has been detected and we invoke an error-recovery routine.

Implement a Transition Diagram

- A sequence of transition diagrams can be converted into a program to look for tokens.
- Each state gets a segment of code.

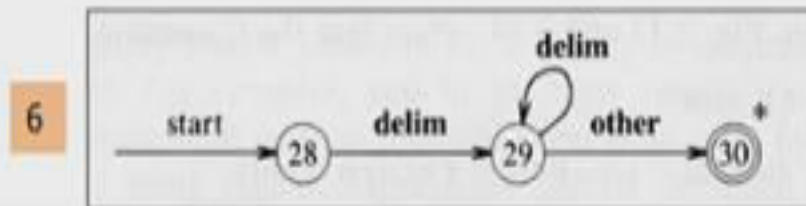
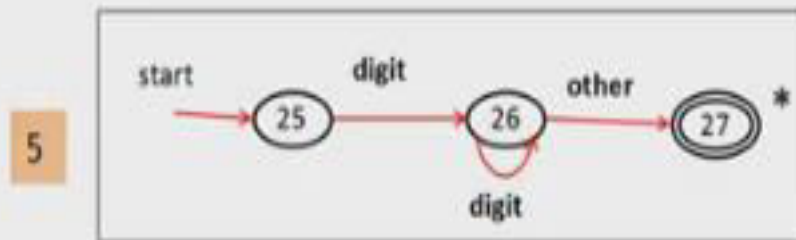
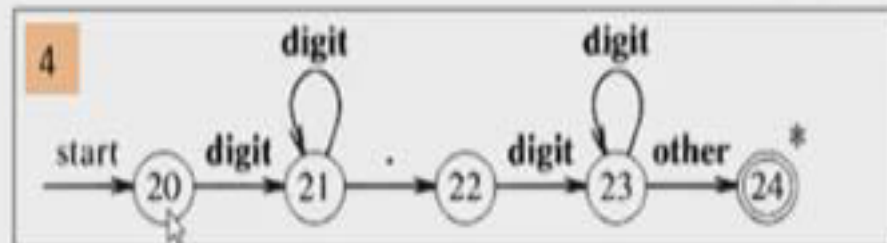
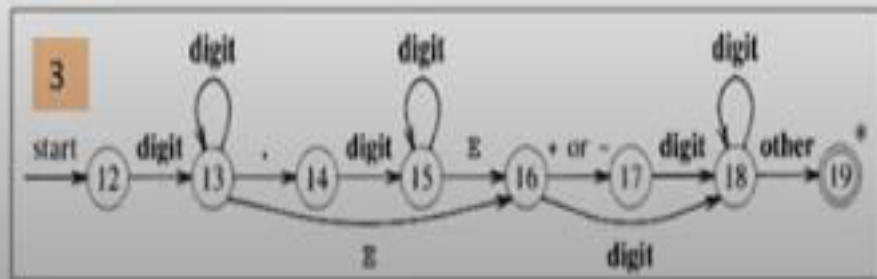
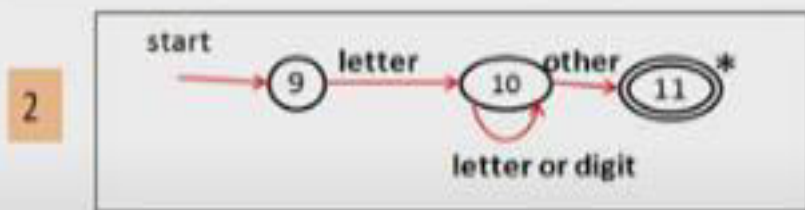
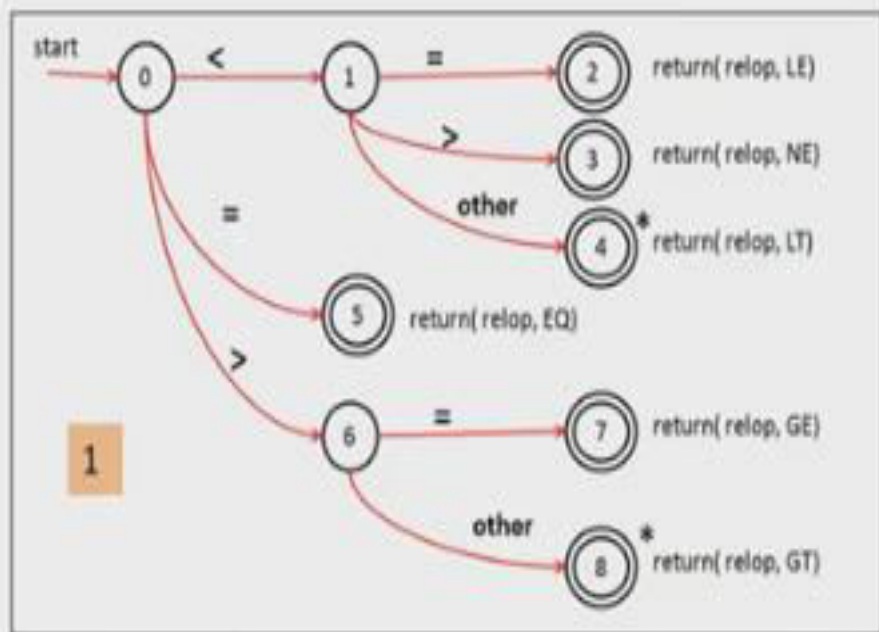


Implementation of Transition Diagram

- **state**- for the current state
- **start**-for the start state of current transition diagram
- **lexical_value**:
 - assigns the pointer returned by `install_id()` and `install_num()` when an identifier or number is found.
- **fail()**-when a transition diagram fails.
 - Retract the forward pointer to the position of the lexeme beginning pointer and to return to the start state of next diagram.
 - If all diagram fails the function `fail()` calls an error recovery routine.



Transition Diagram for all tokens





Implementation of a Transition Diagram

```
int state = 0, start = 0;
int lexical_value;
/* to "return" second component of token */
```

Initializes start, state and lexical_value

```
int fail()
{
    forward = token_beginning;
    switch (start) {
        case 0:    start = 9; break;
        case 9:    start = 12; break;
        case 12:   start = 20; break;
        case 20:   start = 25; break;
        case 25:   recover(); break;
        default:   /* compiler error */
    }
    return start;
}
```

Calls when a transition fails for a given diagram & moves to the next transition diagram



Implementation of a Transition Diagram

```
token nexttoken()
```

Returns the next token

```
{ while(1) {
```

```
    switch (state) {
```

```
    case 0:  c = nextchar();
```

```
        /* c is lookahead character */
```

```
        if (c==blank || c==tab || c==newline) {
```

```
            state = 0;
```

```
            lexeme_beginning++;
```

```
            /* advance beginning of lexeme */
```

```
        }
```

```
        else if (c == '<') state = 1;
```

```
        else if (c == '=') state = 5;
```

```
        else if (c == '>') state = 6;
```

```
        else state = fail();
```

```
        break;
```

```
    ... /* cases 1-8 here */
```

For whitespaces

For relational operators



Implementation of a Transition Diagram

```
case 9:  c = nextchar();
        if (isletter(c)) state = 10;
        else state = fail();
        break;
case 10: c = nextchar();
        if (isletter(c)) state = 10;
        else if (isdigit(c)) state = 10;
        else state = 11;
        break;
case 11: retract(1); install_id();
        return ( gettoken() );

... /* cases 12-24 here */
```

Returns a character
pointed
by forward pointer
and forward pointer ++

**Purpose of nextchar()
function: Reading next
character in a buffer**



Implementation of a Transition Diagram

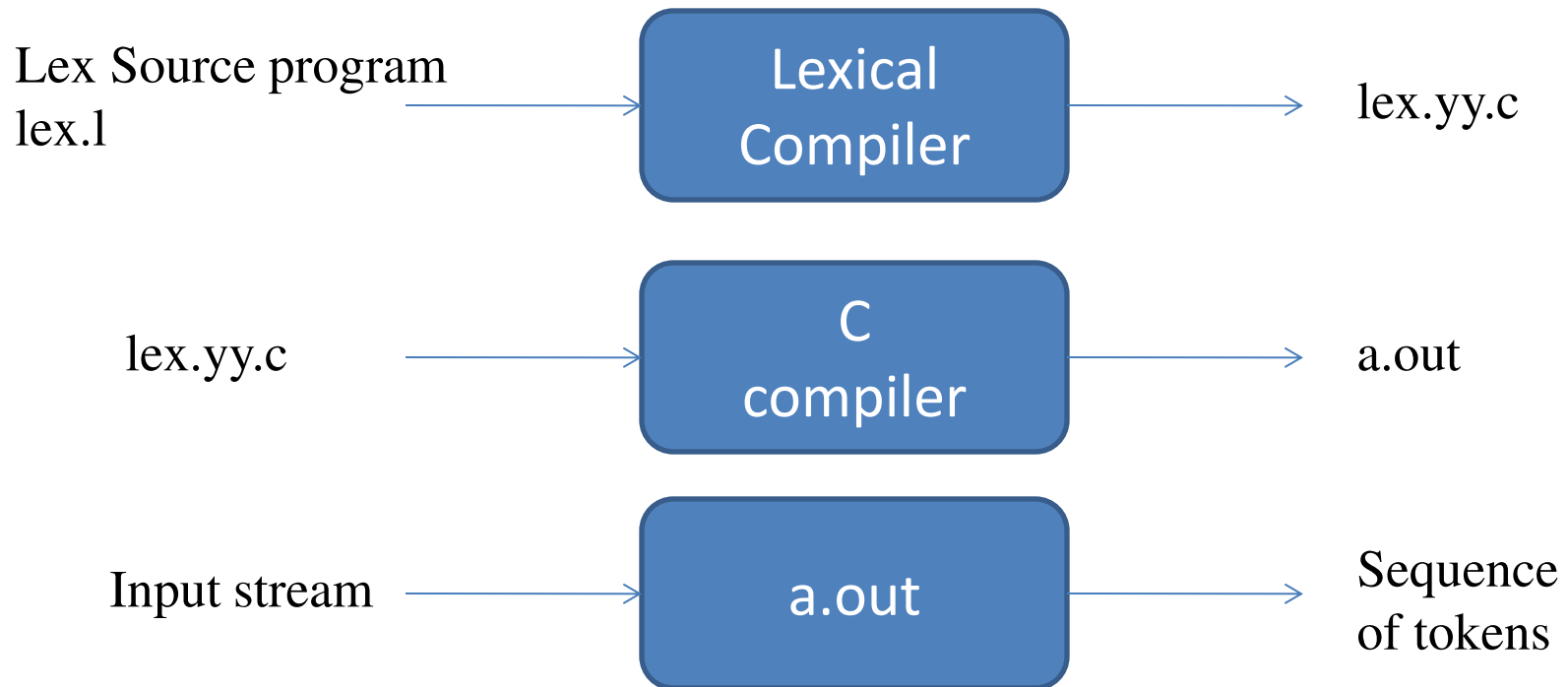
```
case 25: c = nextchar();
        if (isdigit(c)) state = 26;
        else state = fail();
        break;
case 26: c = nextchar();
        if (isdigit(c)) state = 27;
        else state = 27;
        break;
case 27: retract(1); install_num();
        return ( NUM );
    }
}
```

Retracts the forward
pointer by one character

Enters the number in a
literal table and returns a
pointer to that entry



Lexical Analyzer Generator - Lex





Structure of Lex programs

declarations

%%

translation rules \longrightarrow Pattern {Action}

%%

auxiliary functions



LEX program

```
%{  
    /* 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-Za-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) install_id(); return(ID); }  
{number} {yylval = (int) install_num(); return(NUMBER);}
```

```
%%  
Int install_id() { /* funtion 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 install_num() { /* similar to  
    installID, but puts numerical  
    constants into a separate table  
    */  
}
```

...



THANK YOU



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited with Grade "A" by NAAC | Approved by AICTE



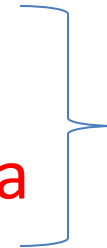
UNIT-I

LEXICAL ANALYSIS



UNIT 1 LEXICAL ANALYSIS

- Structure of compiler
- Functions and Roles of lexical phase
- Input buffering
- Representation of tokens using regular expression
- Properties of regular expression
- Finite Automata
- Regular Expression to Finite Automata
- NFA to Minimized DFA.





Outline

- Finite Automata
 - Non-Deterministic Finite Automata(NFA)
 - Deterministic Finite Automata(DFA)
 - Conversion of Regular Expression to NFA(Thompson's Construction)
 - Conversion of NFA to DFA(Subset Construction Method)
 - Conversion of DFA to Minimized DFA
- Assignments



What is Finite Automata?

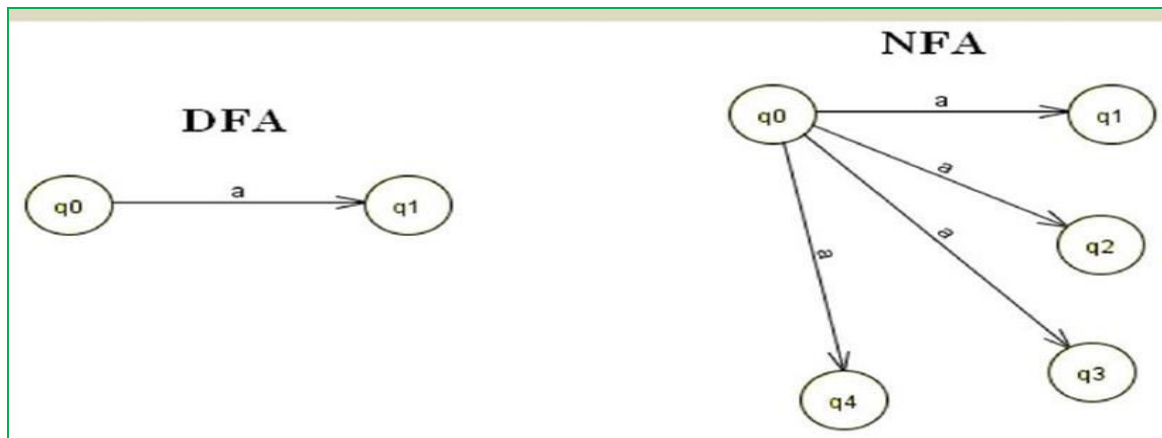
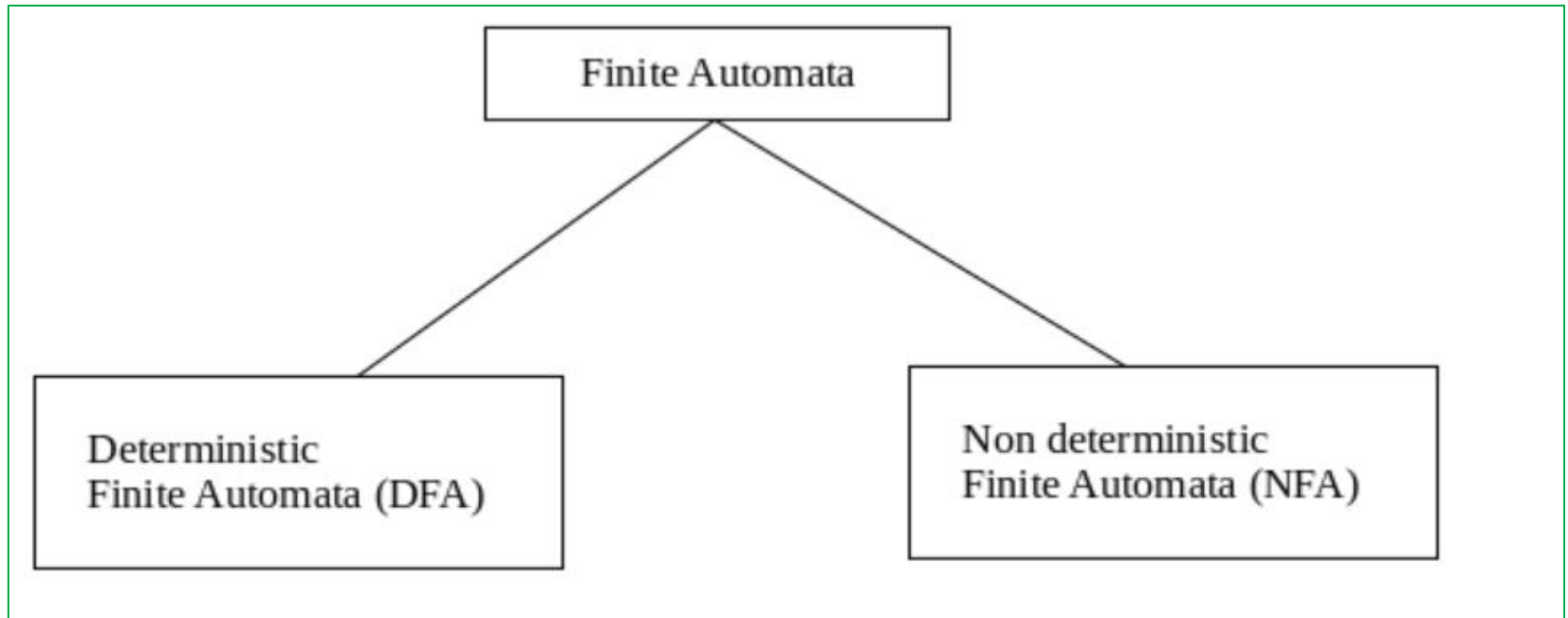
- A recognizer for a language is a program that takes as input a string x and answers “**yes**” if x is a sentence of the language and “**no**” otherwise.
- **Finite automata:** a generalized transition diagram to compile regular expression into a recognizer.
- A machine that accepts Regular Language.

Applications:

- Compilers
- Text processing
- Hardware design



Types of Automata





Finite Automata

- Both DFA and NFA are capable of recognizing the regular sets.(what a regular expression denote.)
- Time-space Trade-off:
 - DFA can lead to fast recognizers than NFA.
 - DFA can be much larger than an equivalent NFA.



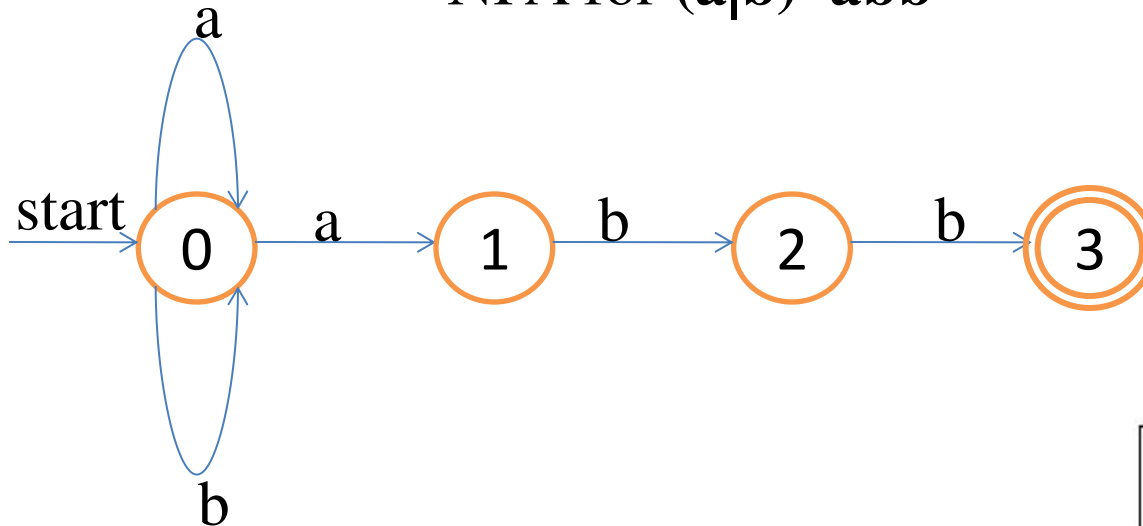
Non-Deterministic Finite Automata

- NFA is a mathematical model that consists of:
 - a set of states S
 - a set of input symbols Σ (input symbol alphabet)
 - a transition function **move** that maps **state-symbol pairs** to set of states
 - a state s_0 that is distinguished as the **start or initial** state
 - a set of states F distinguished as **accepting or final** states.
- An NFA can be represented diagrammatically by a labelled directed graph called a transition graph.
- The nodes are states and the labelled edges represent the transition function.
- This graph looks like a transition diagram, but the same character can label two or more transitions out of one state.
- Edges can be labelled by the special symbol ϵ .



NFA-Example

NFA for $(a|b)^*abb$



Transition Table

State	input symbol	
	a	b
0	{0,1}	{0}
1	—	{2}
2	—	{3}
3	—	—

- the set of states of the NFA is $\{0,1,2,3\}$
- the input symbol alphabet is $\{a,b\}$
- state 0 is start state
- The accepting state 3 is indicated by double circle.



Formal definition of NFA

- NFA also has five tuples same as DFA, but with different transition function, as shown follows:
- $\delta: Q \times \Sigma \rightarrow 2^Q$

NFA is a 5 tuple $M=(Q, \Sigma, q_0, \delta, A)$ machine, where,

Q -is a finite set of states

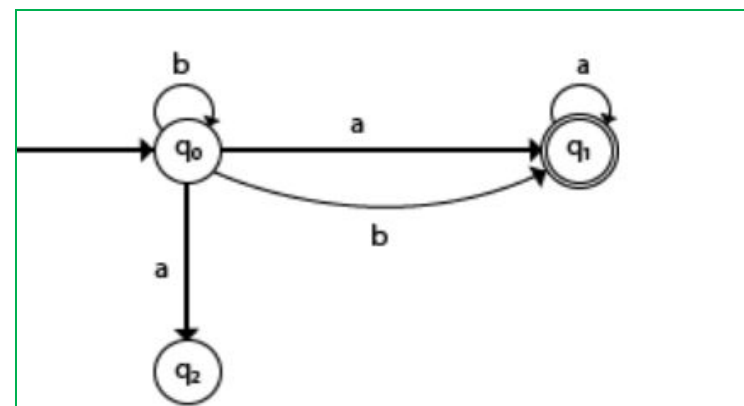
Σ - finite set of input symbols

$q_0 \in Q$ - Initial states

$A \subseteq Q$ - Set of accepting states

δ - transition function

$\delta: Q \times \Sigma \rightarrow 2^Q$





DFA (Deterministic Finite Automata)

- DFA is a special case of NFA in which:
 - for each state **s** and input symbol **a** ,there is at most one edge labeled **a** leaving **s**.
i.e. only one path for specific input symbol from the current state to the next state.
 - no state has an **ϵ -transition** i.e. a transition on input **a** .
- Each entry in the transition table of a DFA is a single state.



Formal Definition of DFA

A DFA or finite state machine is a 5 tuple
 $M = (Q, \Sigma, q_0, \delta, A)$ machine,
where,

Q - is a finite set of states

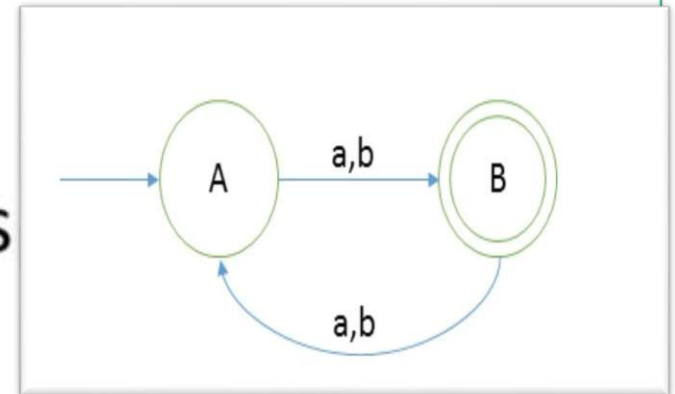
Σ - finite set of input symbols

$q_0 \in Q$ - Initial states

$A \subseteq Q$ - Set of accepting states

δ - transition function

$\delta : Q \times \Sigma \rightarrow Q$



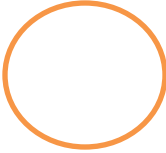
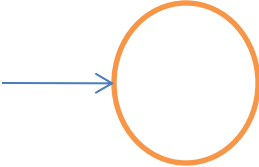
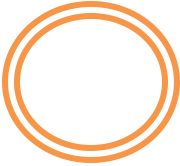



DFA vs. NFA

DFA	NFA
DFA can be understood as one machine.	Multiple small machines computing at the same time.
Difficult to construct - Complex structure.	Easier to construct.
Rejects the string if not terminated at the accepting state.	Rejects only after multiple checks.
Less execution time on input string.	More execution time.
All DFA are derived from NFA.	Not all NFA are DFA.
DFA requires more space.	Requires less Space.
The next possible state is clearly set.	Ambiguity occurs.



Finite Automata Notations

- A state 
- Start State 
- Accepting State 
- A transition 



Conversion Process

Regular Expression



NFA



DFA



Minimized DFA

**Thompson's
Construction
Method**



Regular Expressions to NFA

(Thompson's Construction Method)

- For each kind of RE, define an NFA.
- **Input:** A Regular Expression r over an alphabet Σ
- **Output:** An **NFA** N accepting $L(r)$
- **Method:**

Step 1: For ε



The NFA recognizes $\{\varepsilon\}$

Step 2: For a in Σ

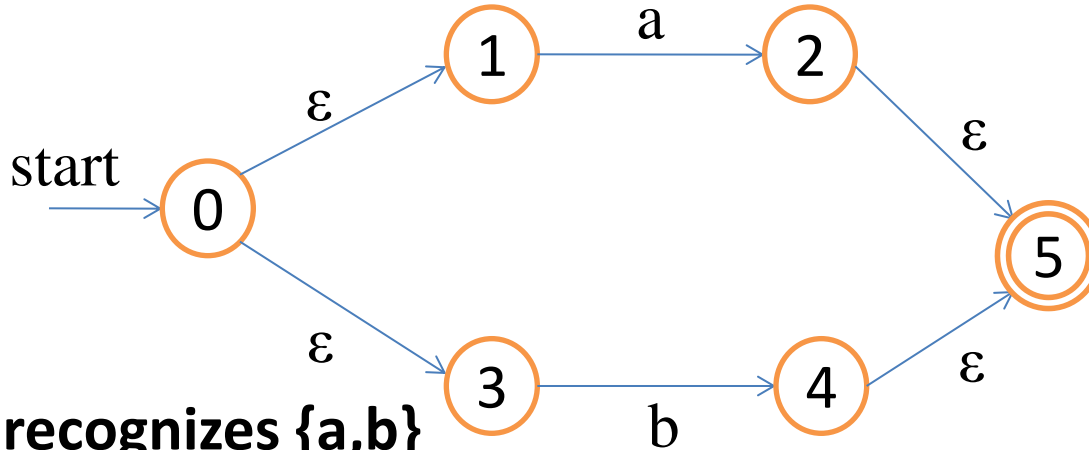


The NFA recognizes $\{a\}$



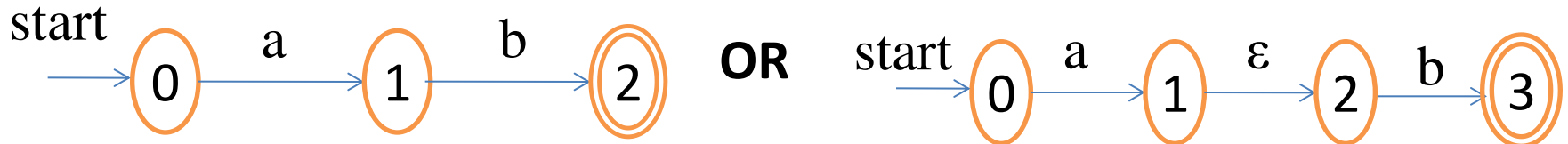
Thompson's Construction Method

Step 3: RE=a | b



The NFA recognizes {a,b}

Step 4: RE=ab

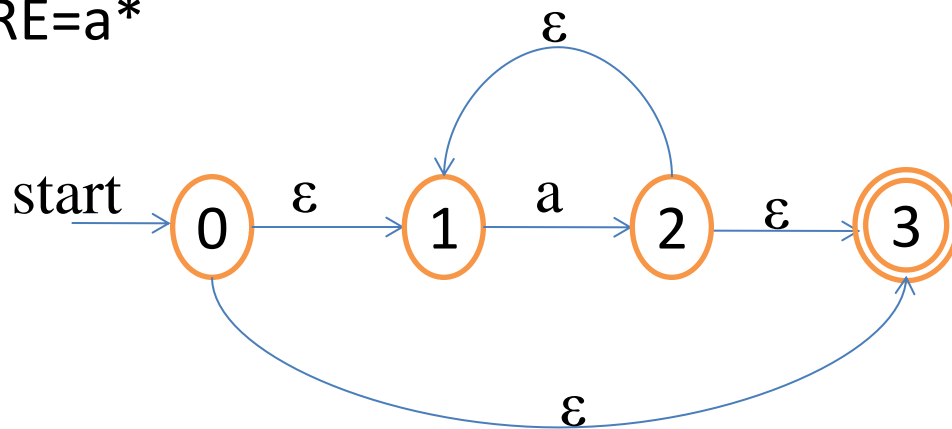


The NFA recognizes {ab}



Thompson's Construction Method

Step 5: $RE = a^*$



The NFA recognizes $\{\epsilon, a, aa, aaa, \dots\}$

Step 6: $RE = a^+$

$= a.a^*$

Follow step 4 for construction.

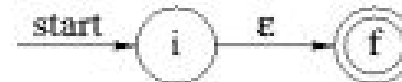


Converting a Regular Expression to an NFA

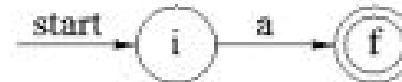
Regular Expression

NFA

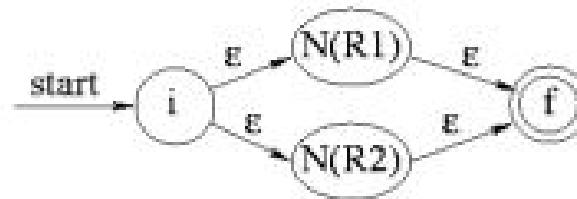
ϵ



a



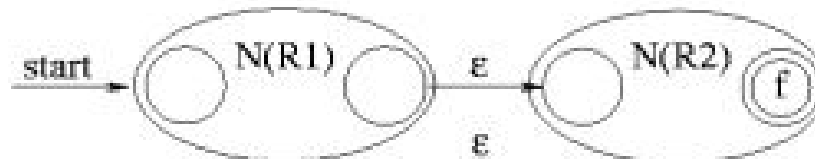
$R1|R2$



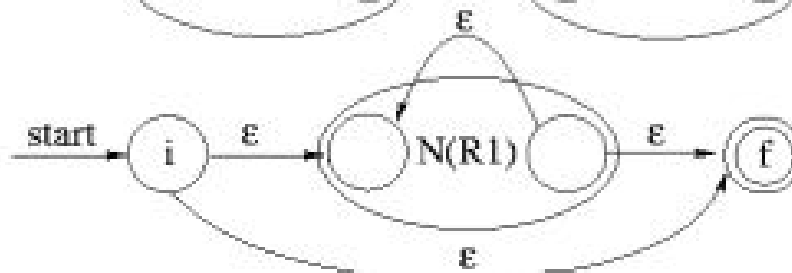
$R1 R2$



or



$R1^*$





Problem 1

- Construct Non deterministic Finite Automata for the following regular expression using Thompson's Construction method.

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

b. $(a|b)^*$

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

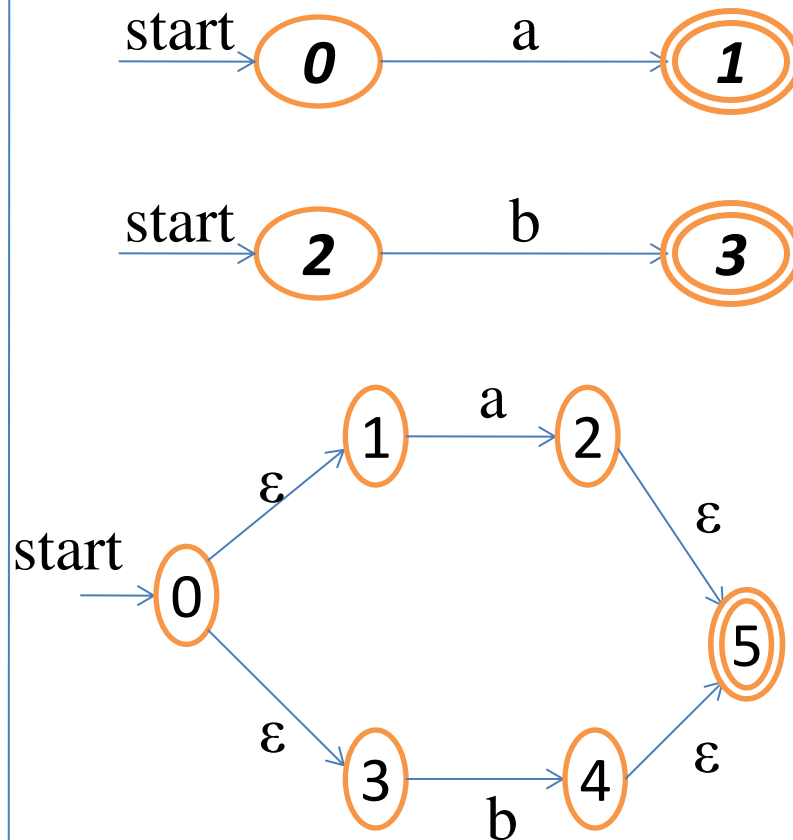
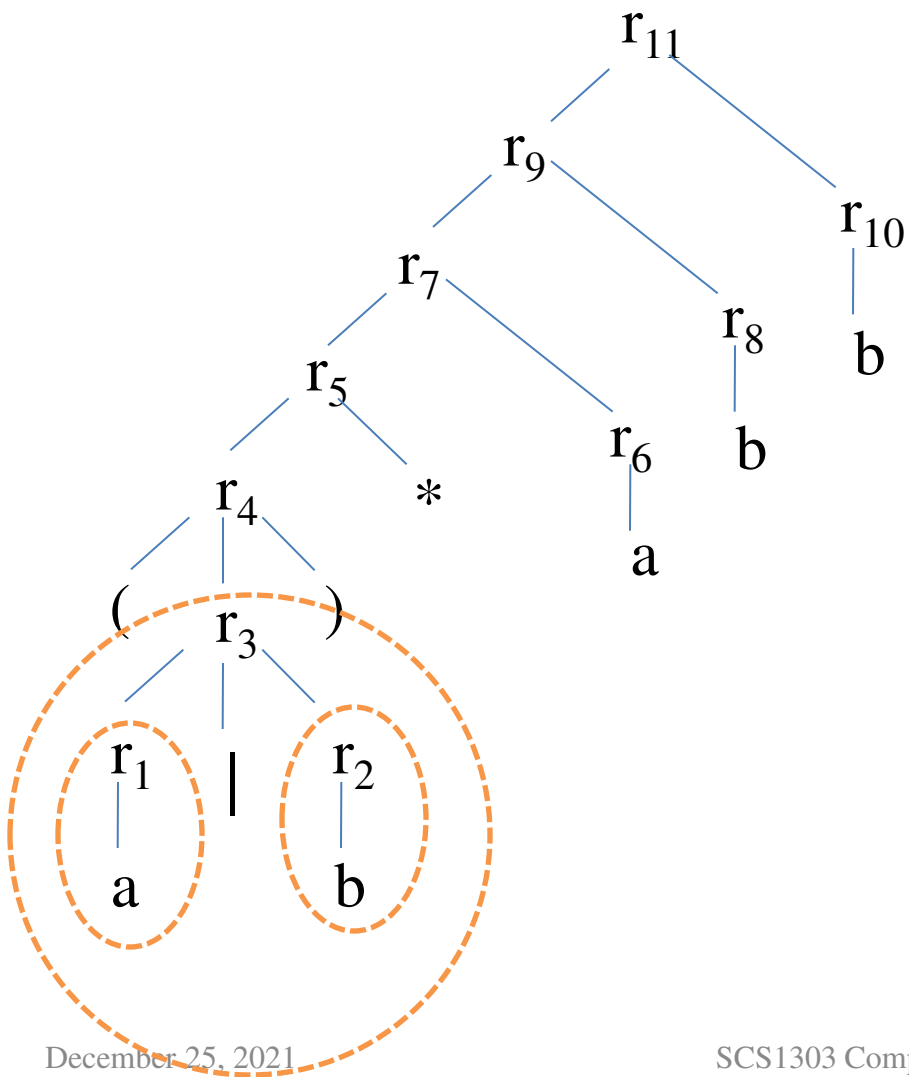
d. $((\epsilon|a)b^*)^*$

e. $(0|123)^*$



Example 1

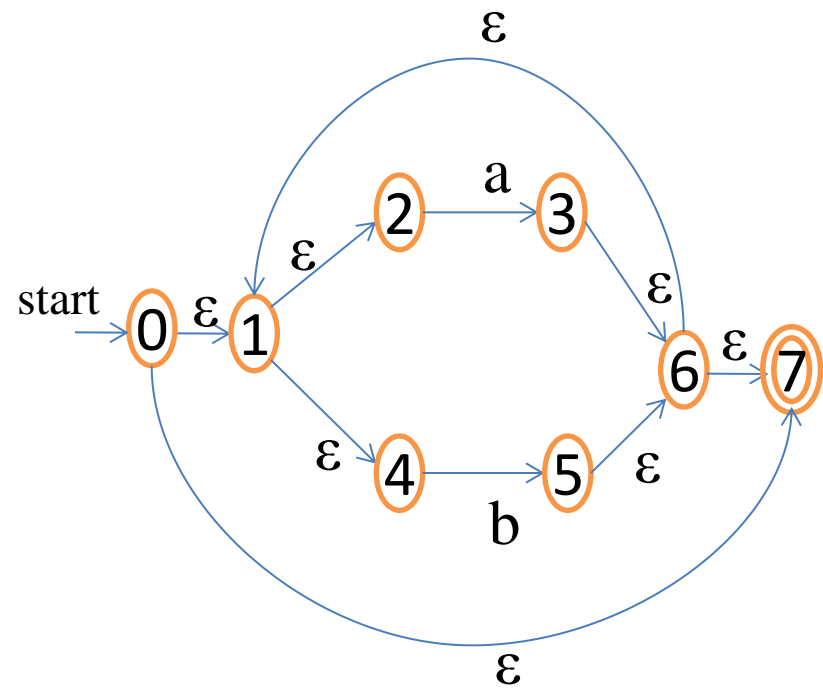
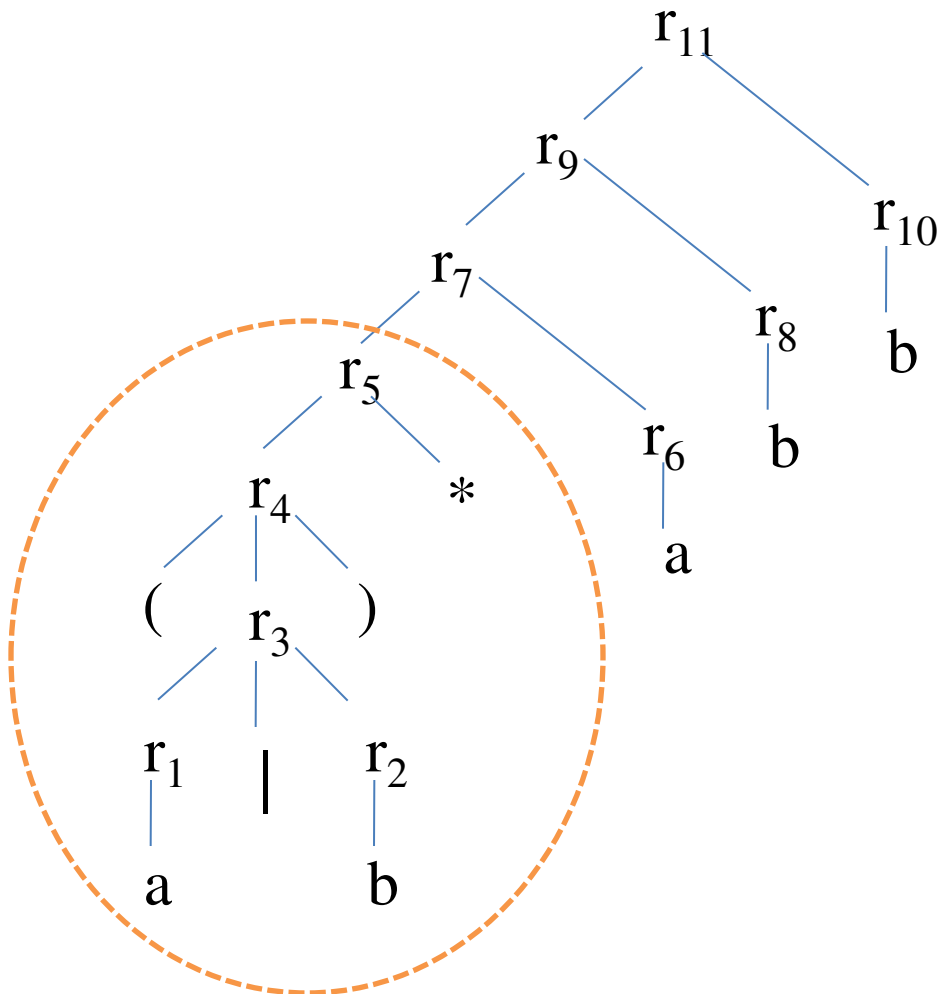
- Decomposition of $(a|b)^*abb$





Example 1(contd.)

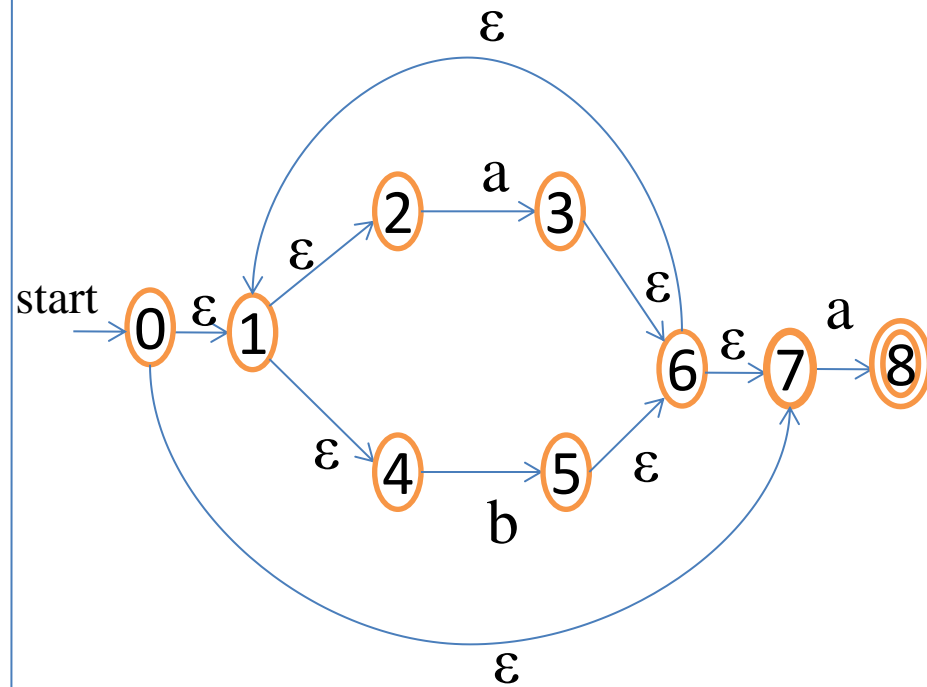
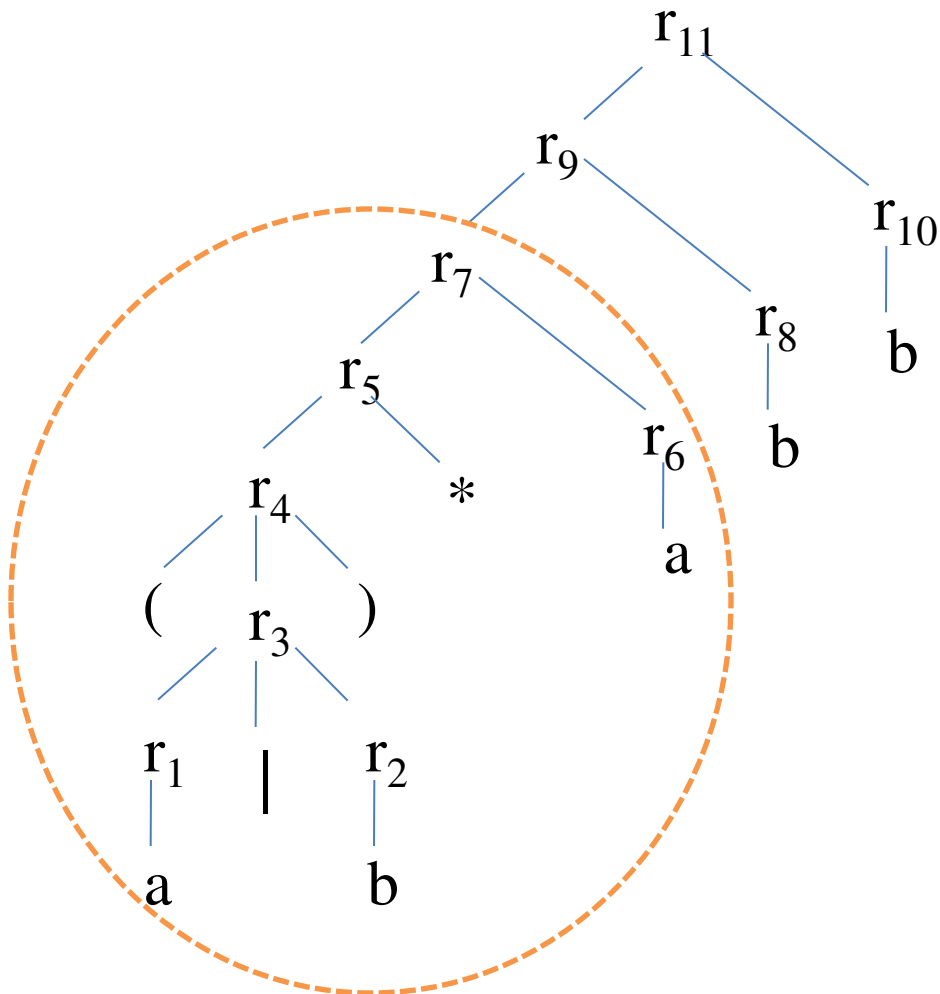
- Decomposition of $(a|b)^*abb$





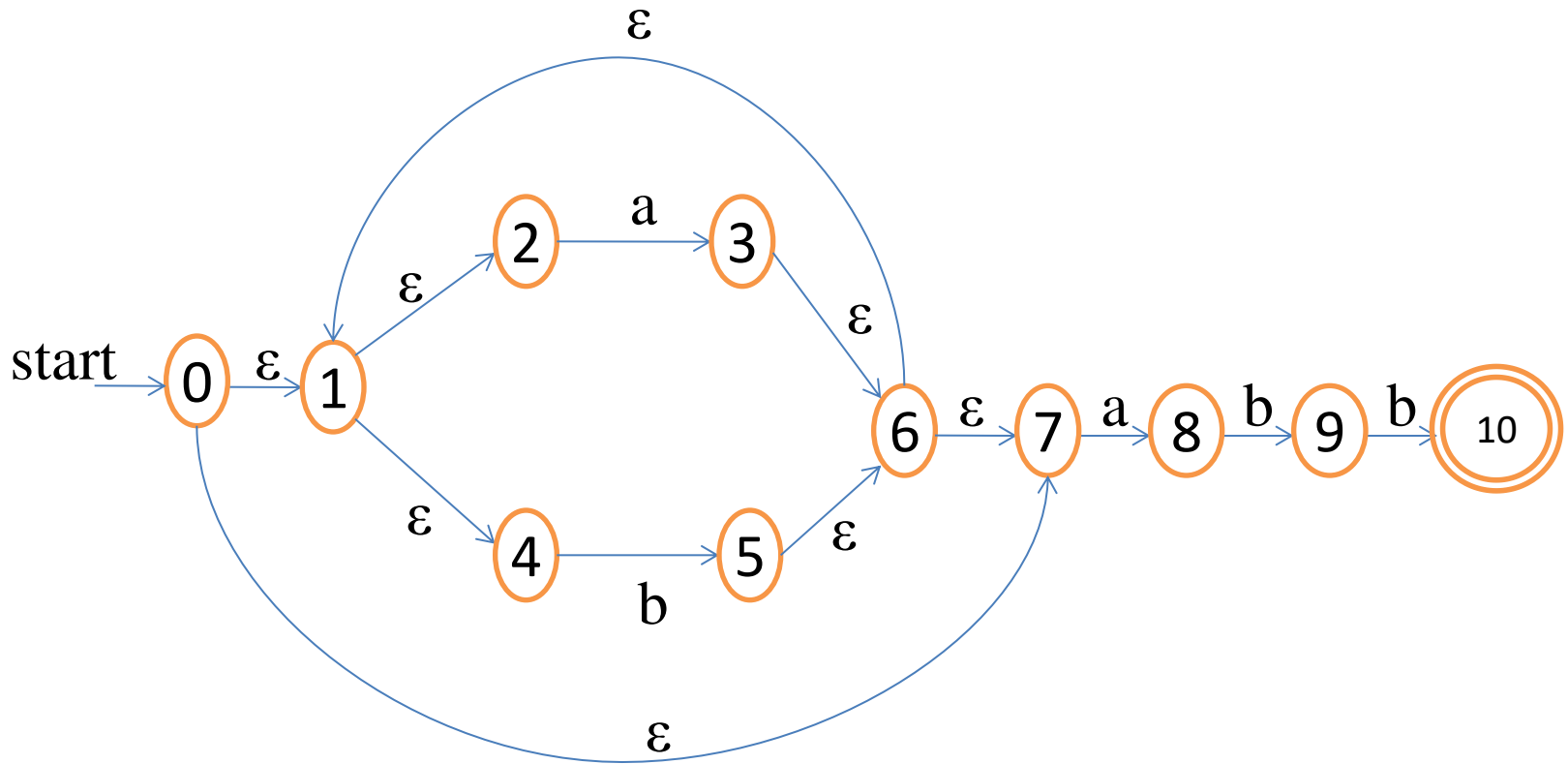
Example 1(contd.)

- Decomposition of $(a|b)^*abb$





NFA for $(a | b)^*abb$





Assignment

- Construct Non deterministic Finite Automata for the following regular expression using Thompson's Construction method.

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

b. $(a|b)^*$

c. $aa^*|bb^*$

d. $((\epsilon|a)b^*)^*$

e. $(0|123)^*$



THANK YOU



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited with Grade "A" by NAAC | Approved by AICTE



UNIT-I

LEXICAL ANALYSIS



UNIT 1 LEXICAL ANALYSIS

- Structure of compiler
- Functions and Roles of lexical phase
- Input buffering
- Representation of tokens using regular expression
- Properties of regular expression
- Finite Automata
- Regular Expression to Finite Automata
- NFA to Minimized DFA.





Outline

- Finite Automata
 - Non-Deterministic Finite Automata(NFA)
 - Deterministic Finite Automata(DFA)
 - Conversion of Regular Expression to NFA(Thompson's Construction)
 - Conversion of NFA to DFA(Subset Construction Method)
 - Conversion of DFA to Minimized DFA
- Assignments



Conversion Process

Regular Expression



NFA



DFA



Minimized DFA

**Subset
Construction
Method**



Subset Construction Method (NFA to DFA)

- **Input:** An **NFA N**
- **Output:** A **DFA D** accepting the same language
- **Method:**
 - Constructs a Transition table **$Dtran$** for **D** .
 - Each DFA state is a set of NFA states.
 - Constructs a transition table **$Dtran$** so that **DFA D** will simulate **“in parallel”** all possible moves **NFA N** can make on a given input string.



Subset Construction Method

- Use the **operations** given in the table to keep track of sets of NFA states.

OPERATION	DESCRIPTION
$\epsilon\text{-closure}(s)$	Set of NFA states reachable from NFA state s on ϵ -transitions alone.
$\epsilon\text{-closure}(T)$	Set of NFA states reachable from some NFA state s in T on ϵ -transitions alone.
$\text{move}(T, a)$	Set of NFA states to which there is a transition on input symbol a from some NFA state s in T .

where,

$s \rightarrow$ an NFA state

$T \rightarrow$ a set of NFA states

$a \rightarrow$ input symbol



Computation of ϵ -closure

```
push all states of  $T$  onto stack;  
initialize  $\epsilon$ -closure( $T$ ) to  $T$ ;  
while ( stack is not empty ) {  
    pop  $t$ , the top element, off stack;  
    for ( each state  $u$  with an edge from  $t$  to  $u$  labeled  $\epsilon$  )  
        if (  $u$  is not in  $\epsilon$ -closure( $T$ ) ) {  
            add  $u$  to  $\epsilon$ -closure( $T$ );  
            push  $u$  onto stack;  
        }  
}
```



Subset Construction algorithm

Initially, $\epsilon\text{-closure}(s_0)$ is the only state in $Dstates$ and it is unmarked

while there is an unmarked state T in $Dstates$ **do**

 mark T

for each input symbol $a \in \Sigma$ **do**

$U := \epsilon\text{-closure}(\text{move}(T, a))$

if U is not in $Dstates$ **then**

 add U as an unmarked state to $Dstates$

end if

$Dtran[T, a] := U$

end do

end do

s_0 represents the **start state**.

$Dstates$ denotes the set of states of **DFA D** (each state in D corresponds to a set of NFA states)

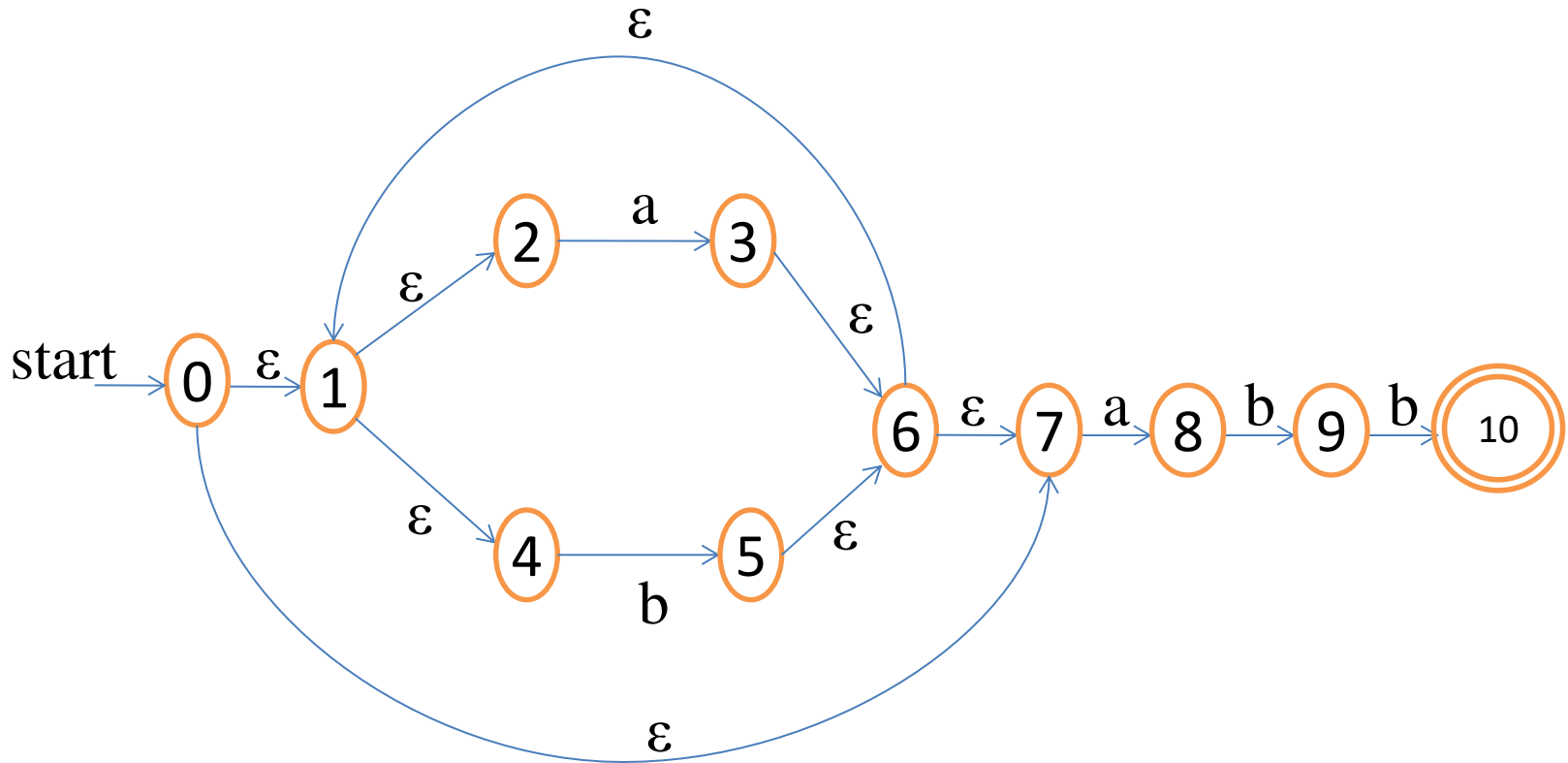
$Dtran$ represents the **transition table** for **DFA D**



Conversion Process

Step 1:

NFA **N** for the Regular Expression **(a|b)*abb**



Step 2: Start state of equivalent DFA is **ϵ -closure(0)**

ϵ -closure(0) = { 0,1,2,4,7 } \longrightarrow **A** New DFA State

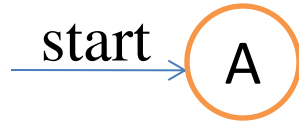


Conversion Process

Step 2: Start state of equivalent DFA is $\epsilon\text{-closure}(0)$

$\epsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\}$ \longrightarrow **A** New DFA State

Start state of DFA:



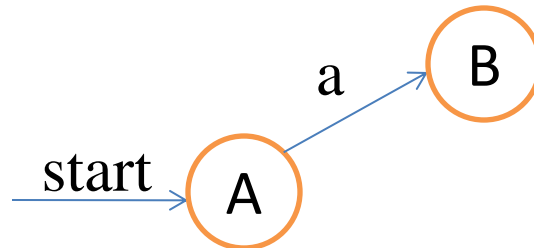
Step 2.1: Compute $\epsilon\text{-closure}(\text{move}(A, a))$

$\text{move}(A, a) = \{3, 8\}$

$\epsilon\text{-closure}(\text{move}(A, a)) = \epsilon\text{-closure}(3, 8) = \{3, 8, 6, 7, 1, 2, 4\}$

$\epsilon\text{-closure}(\text{move}(A, a)) = \{1, 2, 3, 4, 6, 7, 8\}$ \longrightarrow **B** New DFA State

$\text{Dtran}[A, a] = B$





Conversion Process

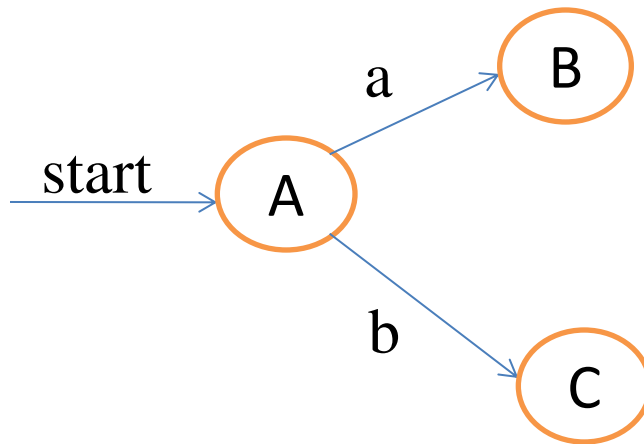
Step 2.2: Compute ϵ -closure(move(A,b))

$$\text{move}(A,b) = \{5\}$$

$$\epsilon\text{-closure}(\text{move}(A,b)) = \epsilon\text{-closure}(5) = \{5,6,7,1,2,4\}$$

$$\epsilon\text{-closure}(\text{move}(A,a)) = \{1,2,4,5,6,7\} \longrightarrow \text{C} \quad \text{New DFA State}$$

$$\text{Dtran}[A,b] = \text{C}$$



Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B		
C		



Conversion Process

Step 3: Compute Transition from **state B** on input symbol {a,b}

$$B = \{1,2,3,4,6,7,8\}$$

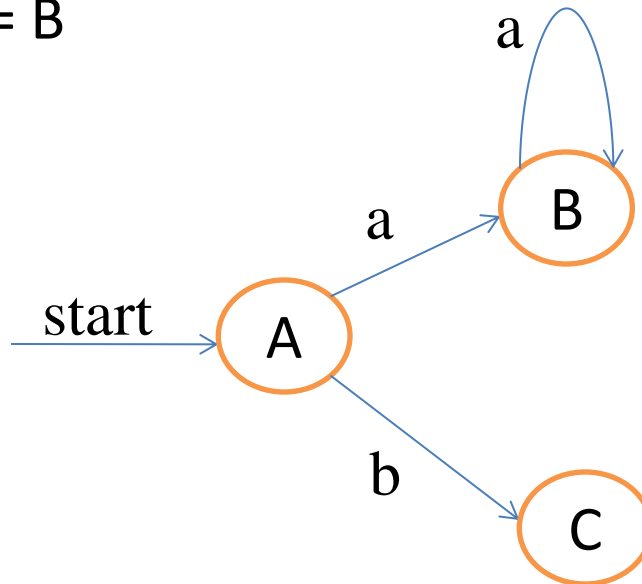
Step 3.1: Compute ϵ -closure(move(B,a))

$$\text{move}(B,a) = \{3,8\}$$

$$\epsilon\text{-closure}(\text{move}(B,a)) = \epsilon\text{-closure}(3,8) = \{3,8,6,7,1,2,4\}$$

$$\epsilon\text{-closure}(\text{move}(B,a)) = \{1,2,3,4,6,7,8\} \longrightarrow \text{B existing state}$$

$$D\text{tran}[B,a] = B$$



Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B	B	
C		



Conversion Process

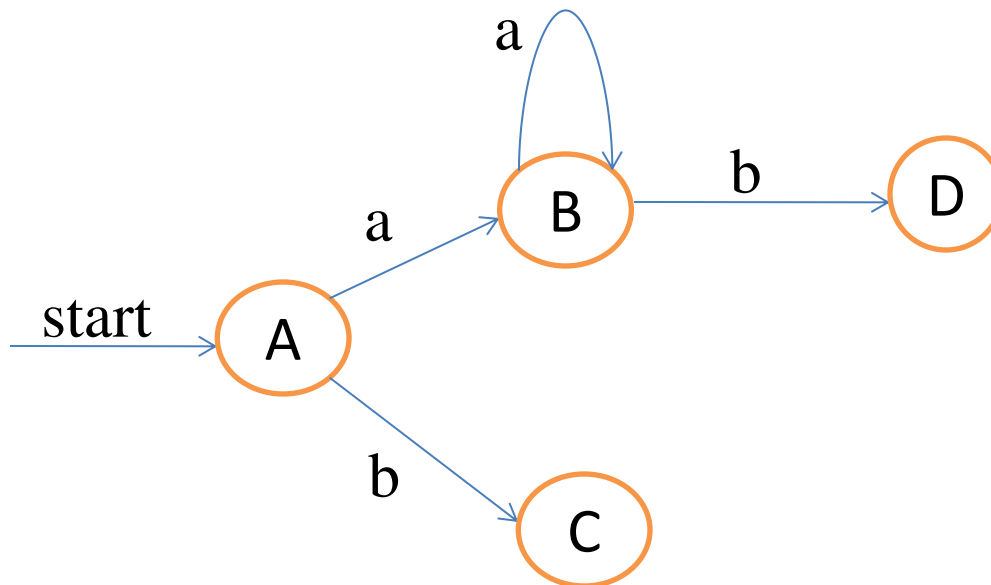
Step 3.2: Compute ϵ -closure(move(B,b))

move(B,b) = {5,9}

ϵ -closure(move(B,b)) = ϵ -closure(5,9) = {5,9,6,7,1,2,4}

ϵ -closure(move(B,b)) = {1,2,4,5,6,7,9} \longrightarrow **D** New DFA State

Dtran[B,b] = D



Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C		
D		



Conversion Process

Step 4: Compute Transition from **state C** on input symbol {a,b}

$$C = \{1,2,,4,5,6,7\}$$

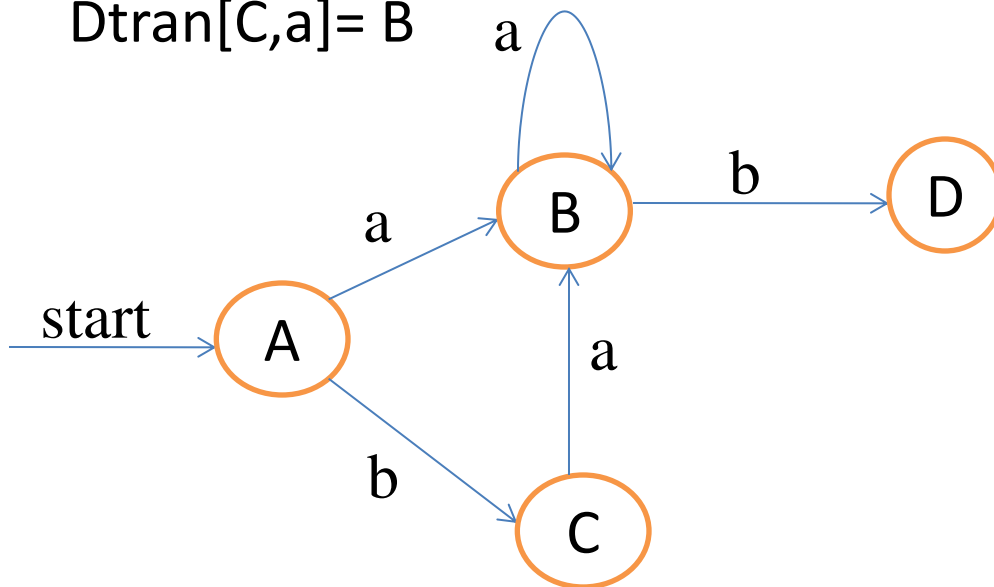
Step 4.1: Compute ϵ -closure(move(C,a))

$$\text{move}(C,a) = \{3,8\}$$

$$\epsilon\text{-closure}(\text{move}(C,a)) = \epsilon\text{-closure}(3,8) = \{3,8,6,7,1,2,4\}$$

$$\epsilon\text{-closure}(\text{move}(C,a)) = \{1,2,3,4,6,7,8\} \longrightarrow \text{B existing state}$$

$$\text{Dtran}[C,a] = B$$



Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	



Conversion Process

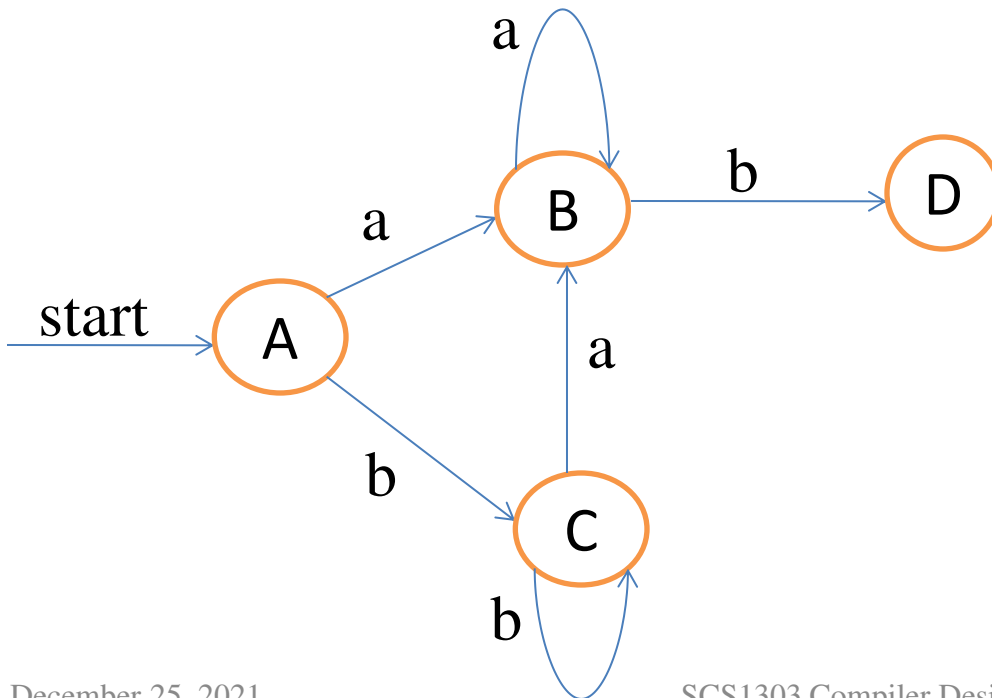
Step 4.2: Compute ϵ -closure(move(C,b))

$$\text{move}(C,b) = \{5\}$$

$$\epsilon\text{-closure}(\text{move}(C,b)) = \epsilon\text{-closure}(5) = \{5,6,7,1,2,4\}$$

$$\epsilon\text{-closure}(\text{move}(C,b)) = \{1,2,4,5,6,7\} \longrightarrow \text{C existing DFA State}$$

$$\text{Dtran}[C,b] = C$$



Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D		



Conversion Process

Step 5: Compute Transition from **state D** on input symbol {a,b}

$$D = \{1,2,,4,5,6,7,9\}$$

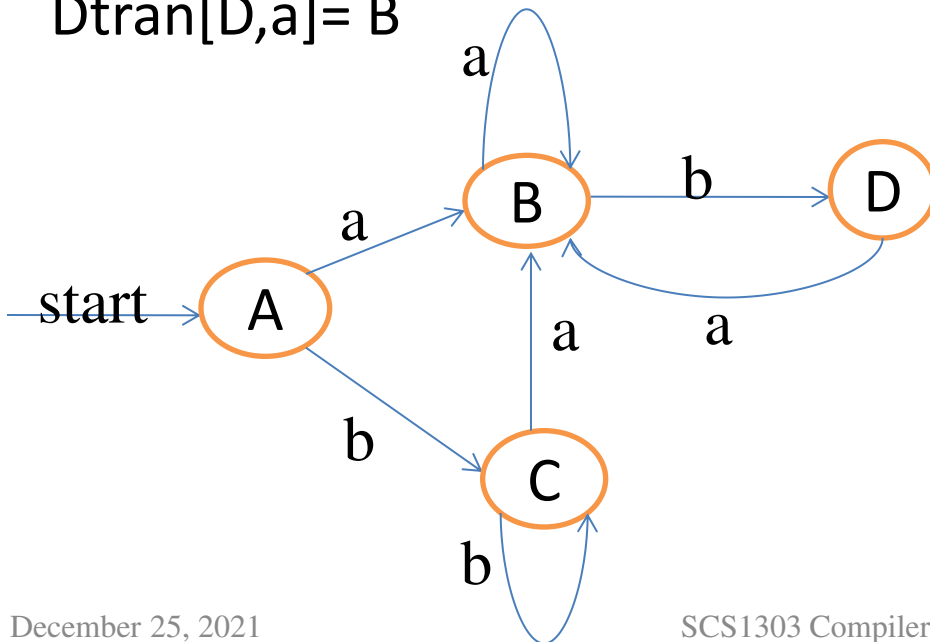
Step 5.1: Compute ϵ -closure(move(D,a))

$$\text{move}(D,a) = \{3,8\}$$

$$\epsilon\text{-closure}(\text{move}(D,a)) = \epsilon\text{-closure}(3,8) = \{3,8,6,7,1,2,4\}$$

$$\epsilon\text{-closure}(\text{move}(D,a)) = \{1,2,3,4,6,7,8\} \longrightarrow \text{B existing state}$$

$$D\text{tran}[D,a] = B$$



DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D	B	



Conversion Process

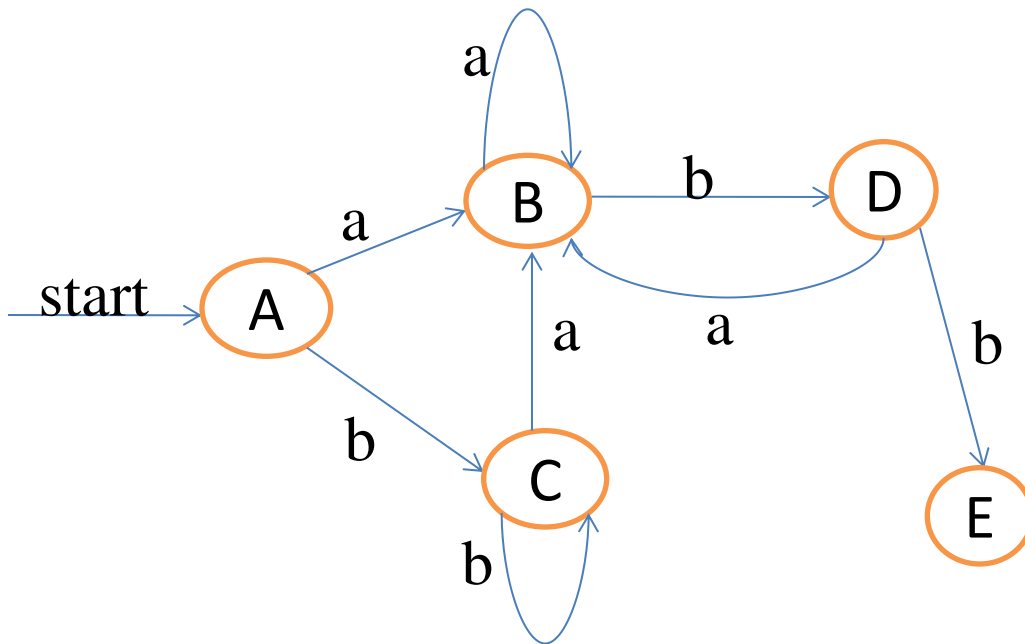
Step 4.2: Compute ϵ -closure(move(D,b))

move(D,b) = {5,10}

ϵ -closure(move(D,b)) = ϵ -closure(5,10) = {5,10,6,7,1,2,4}

ϵ -closure(move(C,b)) = {1,2,4,5,6,7,10} \longrightarrow **E** New DFA State

Dtran[D,b] = E



DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E		



Step 6: Compute Transition from **state E** on input symbol {a,b}

$$E = \{1,2,4,5,6,7,10\}$$

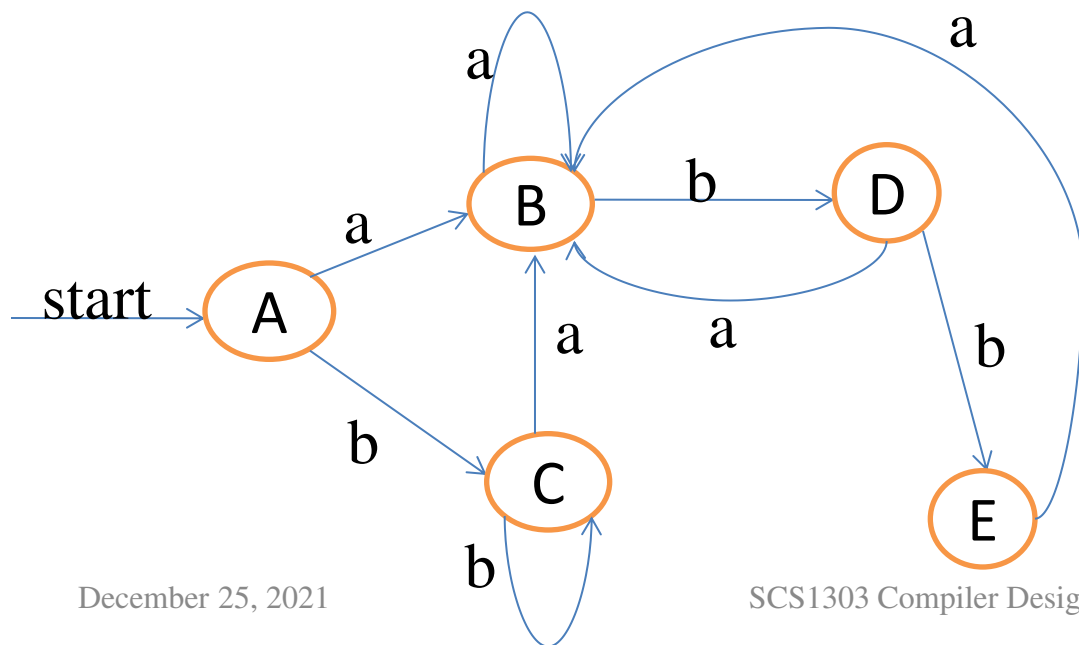
Step 6.1: Compute ϵ -closure(move(E,a))

$$\text{move}(E,a) = \{3,8\}$$

$$\epsilon\text{-closure}(\text{move}(E,a)) = \epsilon\text{-closure}(3,8) = \{3,8,6,7,1,2,4\}$$

$$\epsilon\text{-closure}(\text{move}(E,a)) = \{1,2,3,4,6,7,8\} \longrightarrow \text{B existing state}$$

$$\text{Dtran}[E,a] = B$$



DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	



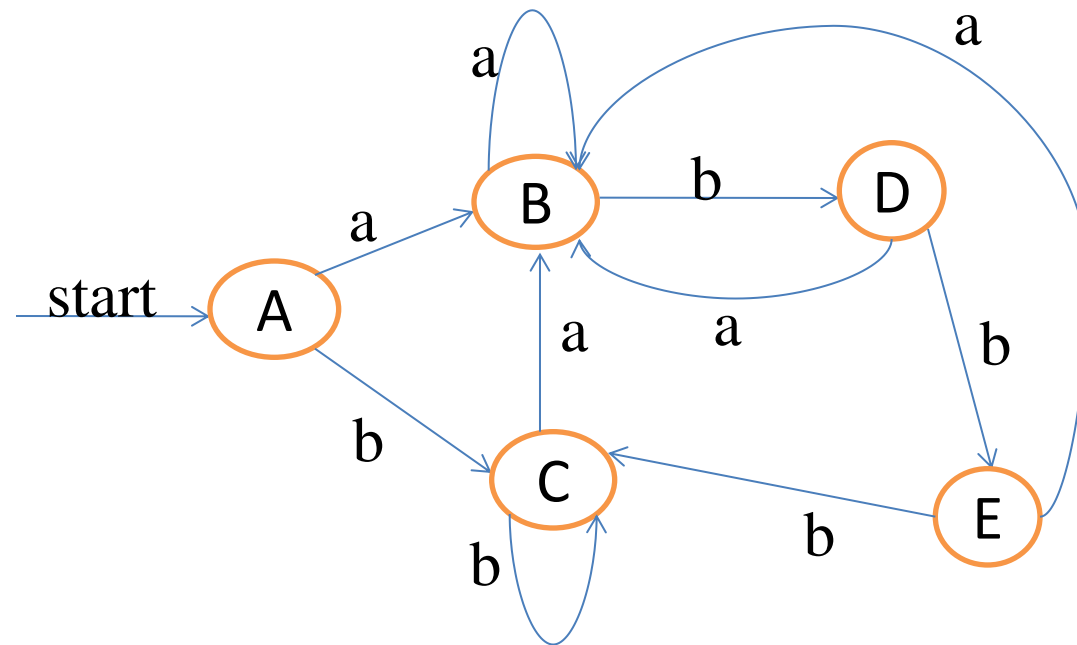
Step 6.2: Compute ϵ -closure(move(E,b))

move(E,b) = {5}

ϵ -closure(move(E,b)) = ϵ -closure(5) = {5,6,7,1,2,4}

ϵ -closure(move(E,b)) = {1,2,4,5,6,7} \longrightarrow **C** existing state

Dtran[E,b] = C

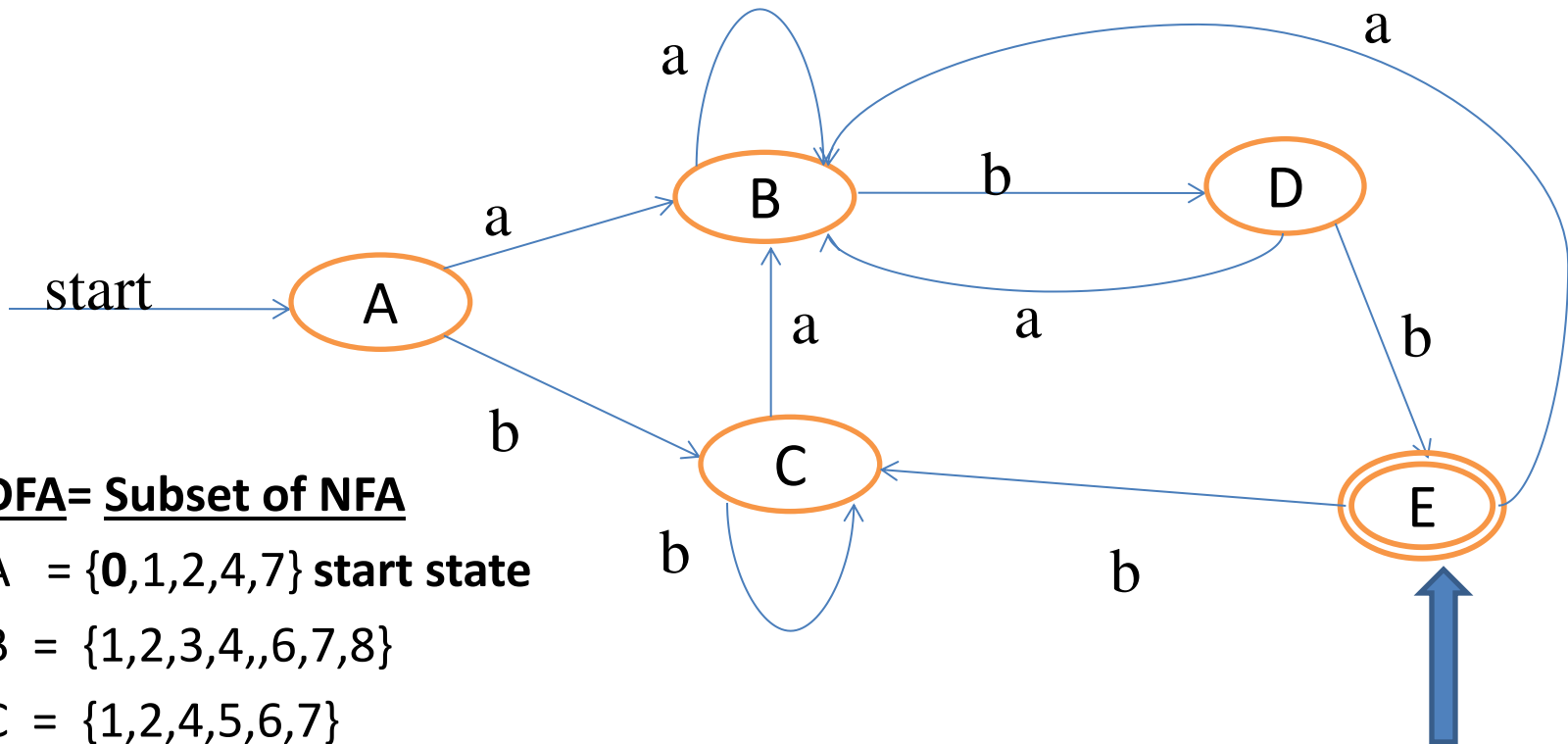


DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



Step 7: No more new DFA states are formed.
Stop the subset construction method.

DFA for $(a|b)^*abb$



DFA= Subset of NFA

A = {0,1,2,4,7} start state

B = {1,2,3,4,,6,7,8}

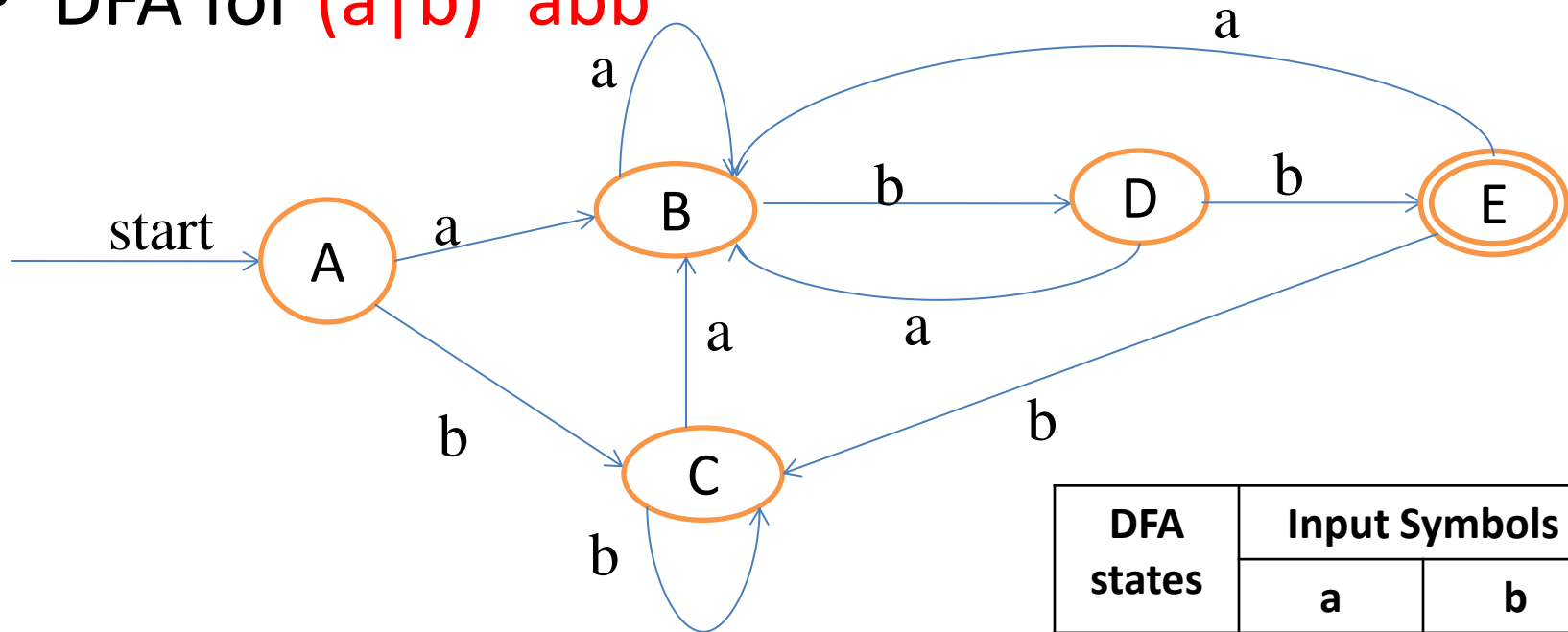
C = {1,2,4,5,6,7}

D = {1,2,4,5,6,7,9}

E = {1,2,4,5,6,7,10} ← State E contains the NFA accepting state(state 10)



- DFA for $(a|b)^*abb$



Transition Table **Dtran**



DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



Next Topic: Minimization of DFA

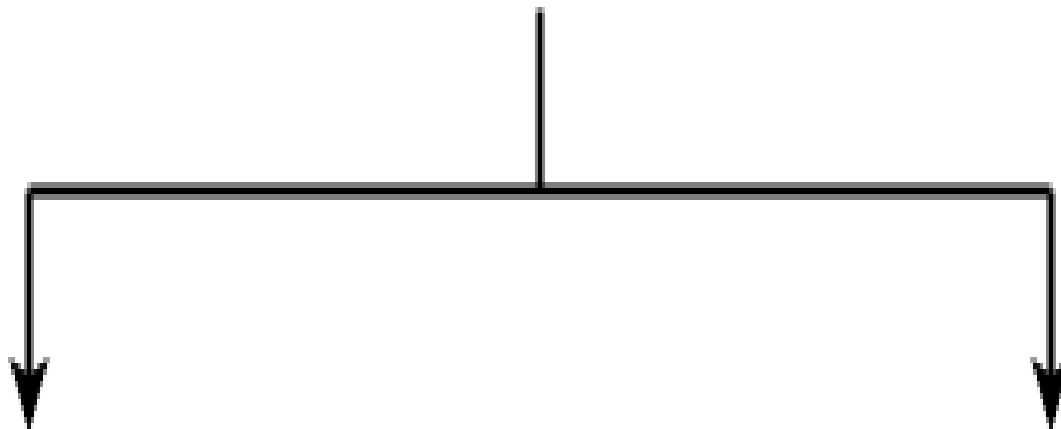


Minimization of DFA

- The process of reducing a given DFA to its minimal form is called as minimization of DFA.
- It contains the minimum number of states.
- The DFA in its minimal form is called as a **Minimal DFA**.



Minimization of DFA (Methods)



Equivalence Theorem



**Myhill Nerode Theorem
(Table Filling Method)**



Minimization of DFA Using Equivalence Theorem

Eliminate all the dead states and inaccessible states from the given DFA (if any).

Dead State

All those non-final states which transit to itself for all input symbols in Σ are called as dead states.

Inaccessible State

All those states which can never be reached from the initial state are called as inaccessible states.



Minimized DFA

- **Input:** A DFA D with set of states S , set of inputs Σ , transitions defined for all states and inputs, start state s_0 , and set of accepting states F .
- **Output:** A minimized DFA M' accepting the same language as D and having as few states as possible.
- **Method:**
 1. Construct an initial partition Π of the set of states with two groups: the **accepting states F** and the **non-accepting states $S-F$** .
 2. Applying the algorithm to construct new partition.
 3. If M' has a **dead state**, it can be removed.

Dead state: a state that is not an accepting state and that has transition to itself on all input symbols .

 - Also, remove any states not reachable from the start state.
 - Any transition to a dead state become undefined.



Partition Algorithm

for each group G of Π **do**

begin

partition G into subgroups such that two states s and t

of G are in the same subgroup if and only if for all

input symbols a , states s and t have transition on a to states
in the same group of Π ;

replace G in Π^{new} by the set of all subgroups formed

end



1. Initial partition Π consists of two groups.

- set of non-accepting states and set of accepting states
- (ABCD)(E)

2. As per the algorithm construct Π^{new}

- first consider the group (E) – set of accepting states
- this group consists of a single state, so it cannot be split further.
- Consider the group (ABCD) – set of non-accepting states.

- On input a, each state has a transition to B and hence remain in same group.
- On input b, ABC remain in same group, while D goes to E, a member of another group.

$$\Pi^{new} \quad (ABC)(D)(E)$$

$$\Pi^{new} \quad (AC)(B)(D)(E)$$

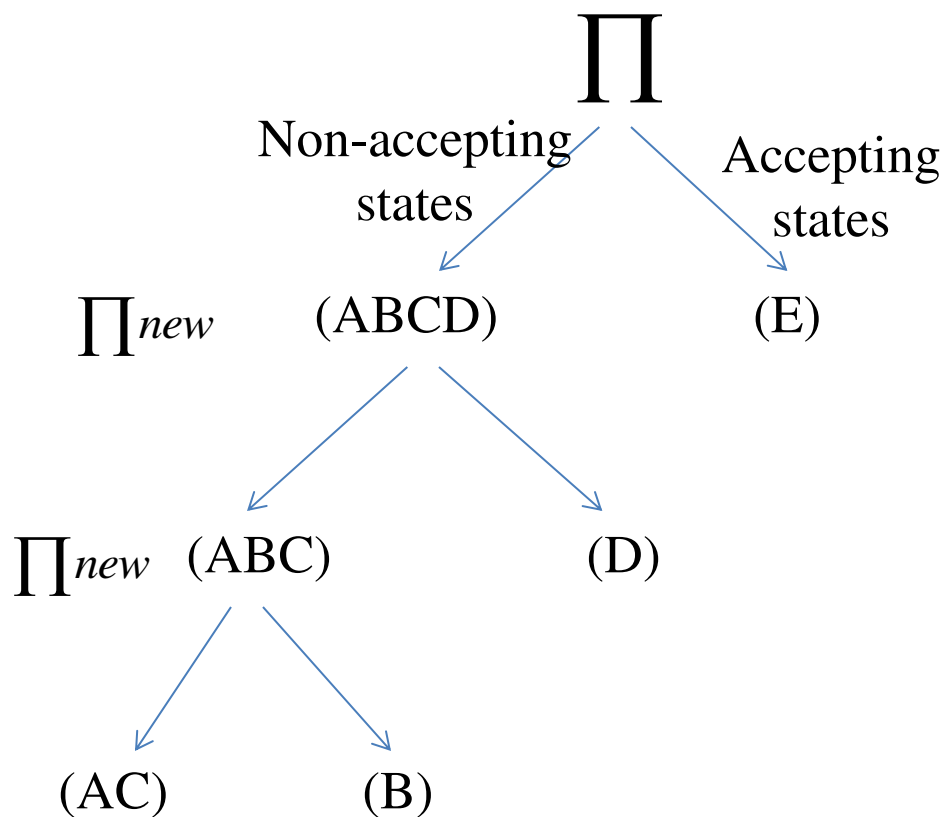
DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Non-accepting states

accepting states



Steps for Partition



DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

accepting states



Choose **A** as
representative for
group (**AC**)

DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



DFA states	Input Symbols	
	a	b
A	B	A
B	B	D
D	B	E
E	B	A

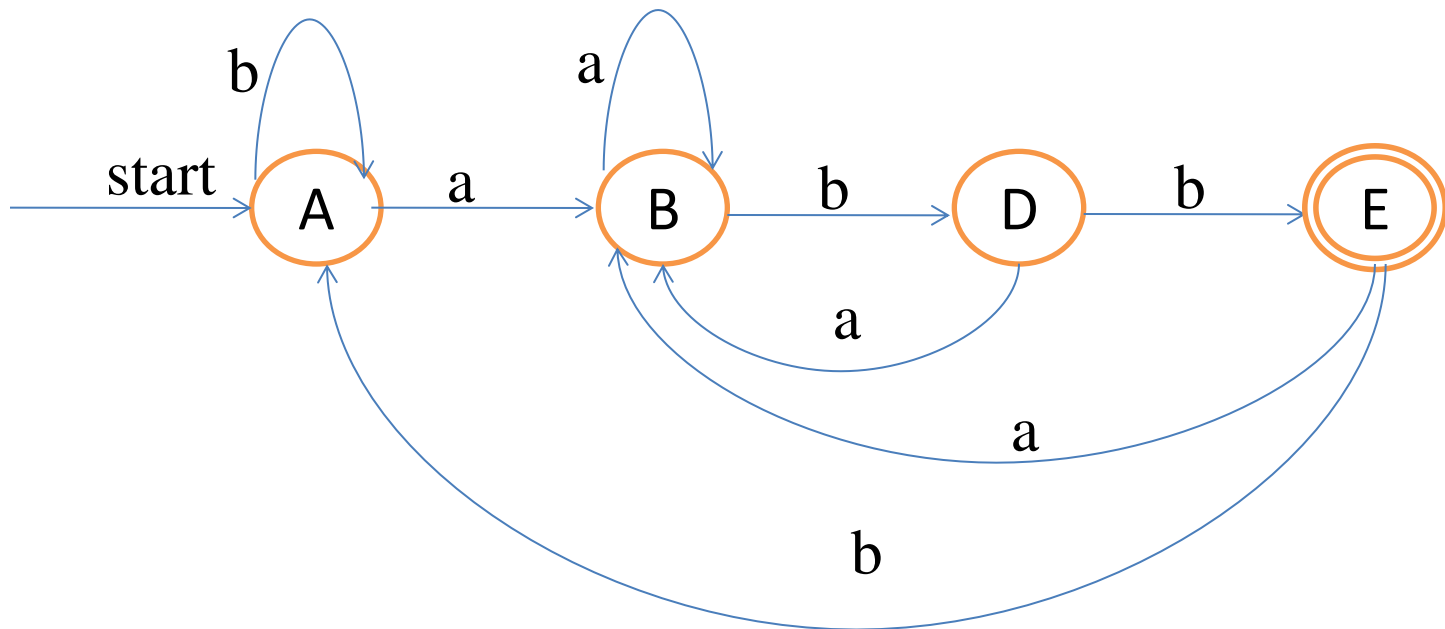
**Transition Table
for reduced DFA**



Minimized DFA

$(a|b)^*abb$

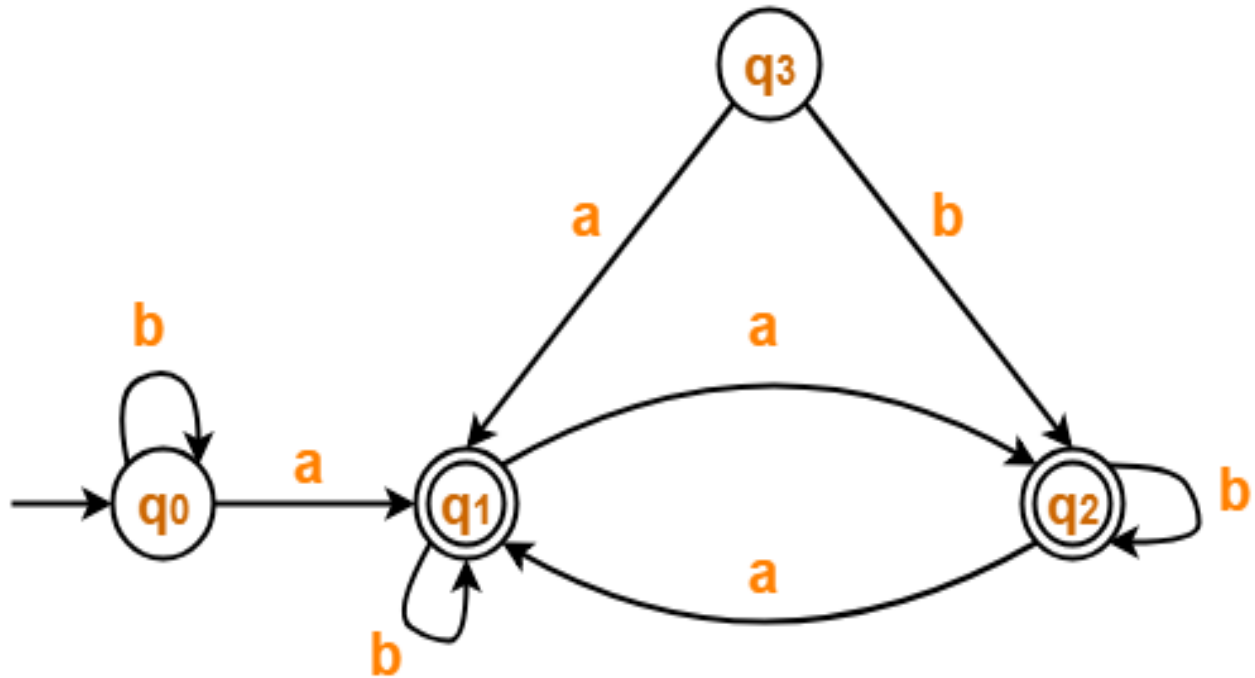
DFA states	Input Symbols	
	a	b
A	B	A
B	B	D
D	B	E
E	B	A





Example 1

Minimize the given DFA-

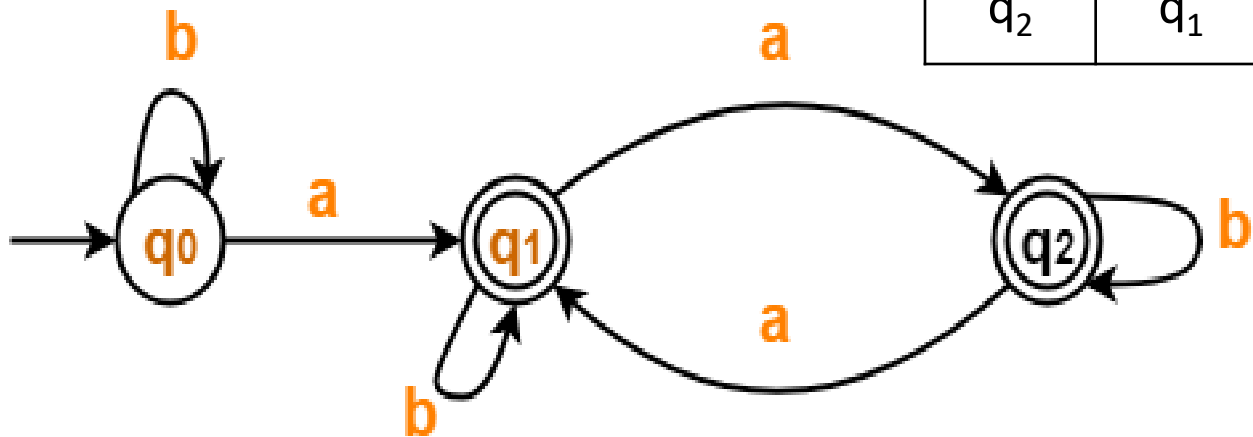




Solution 1

- State q_3 is inaccessible from the initial state.
- So, we eliminate it and its associated edges from the DFA.

The resulting DFA is-



DFA states	Input Symbols	
	a	b
q_0	q_1	q_0
q_1	q_2	q_1
q_2	q_1	q_2



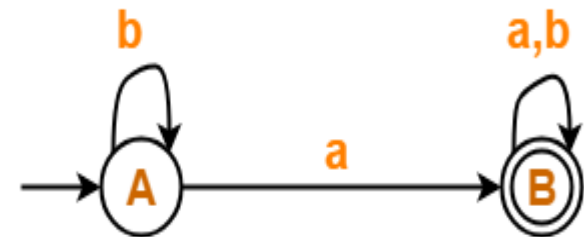
DFA states	Input Symbols	
	a	b
q_0	q_1	q_0
q_1	q_2	q_1
q_2	q_1	q_2



Partition= (q_0) (q_1, q_2)



DFA states	Input Symbols	
	a	b
q_0	q_1	q_0
q_1	q_1	q_1



Minimal DFA



Equivalent REs

3.22 We can prove that two regular expressions are equivalent by showing that their minimum-state DFA's are the same, except for state names.

Using this technique, show that the following regular expressions are all equivalent.

a) $(a|b)^*$

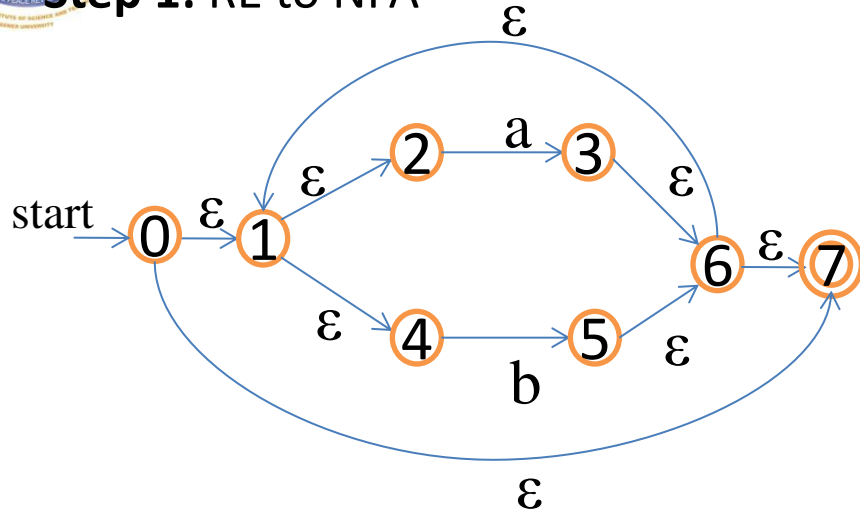
b) $(a^*|b^*)^*$

c) $((\epsilon|a)b^*)^*$



1. $(a|b)^*$

Step 1: RE to NFA

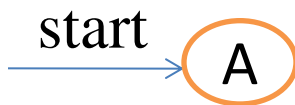


Step 2: NFA to DFA

Step 2.1: Start state of equivalent DFA is $\epsilon\text{-closure}(0)$

$\epsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\} \longrightarrow \text{A}$
New DFA State

Start state of DFA:



Step 2.2:

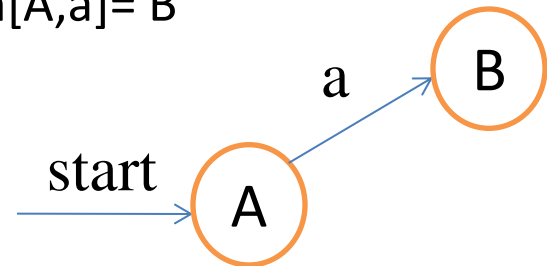
Compute $\epsilon\text{-closure}(\text{move}(A, a))$

$\text{move}(A, a) = \{3\}$

$\epsilon\text{-closure}(\text{move}(A, a)) = \epsilon\text{-closure}(3)$
 $= \{3, 6, 7, 1, 2, 4\}$
 $= \{1, 2, 3, 4, 6, 7\} \longrightarrow \text{B}$

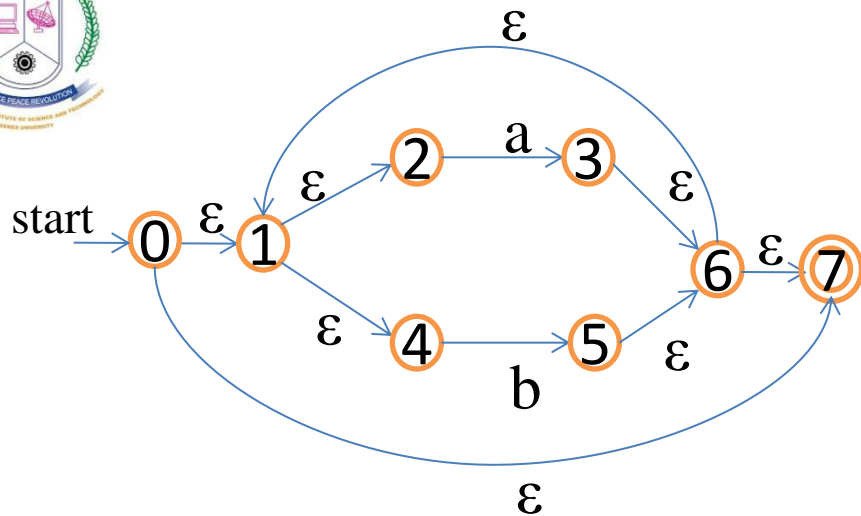
New DFA State

$D\text{tran}[A, a] = B$



Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	
B		



$A = \{0, 1, 2, 4, 7\}$

Step 2.3: Compute ϵ -closure(move(A,b))

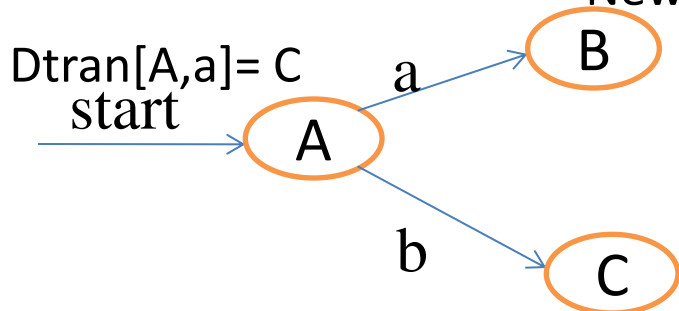
move(A,b) = {5}

ϵ -closure(move(A,b)) =

ϵ -closure(5) = {5, 6, 7, 1, 2, 4}

= {1, 2, 4, 5, 6, 7} \longrightarrow C

New DFA State



Dtran[A,a] = C
start

Step 2.4: Transition from B={1,2,3,4,6,7}

Compute ϵ -closure(move(B,a))

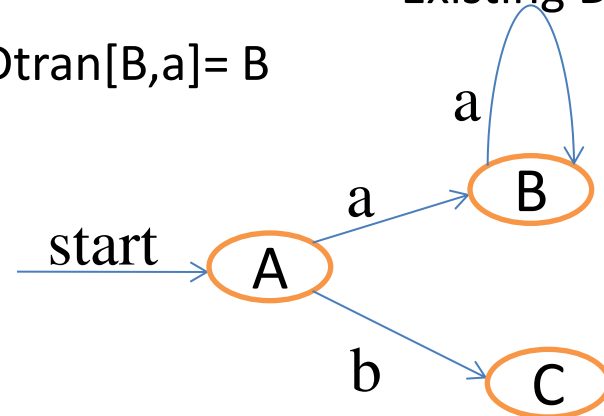
move(B,a) = {3}

ϵ -closure(move(B,a)) = ϵ -closure(3)

= {1, 2, 3, 4, 6, 7} \longrightarrow B

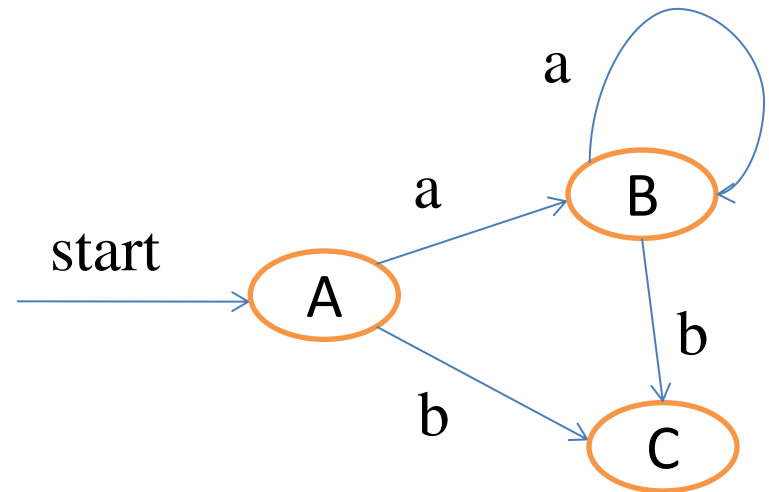
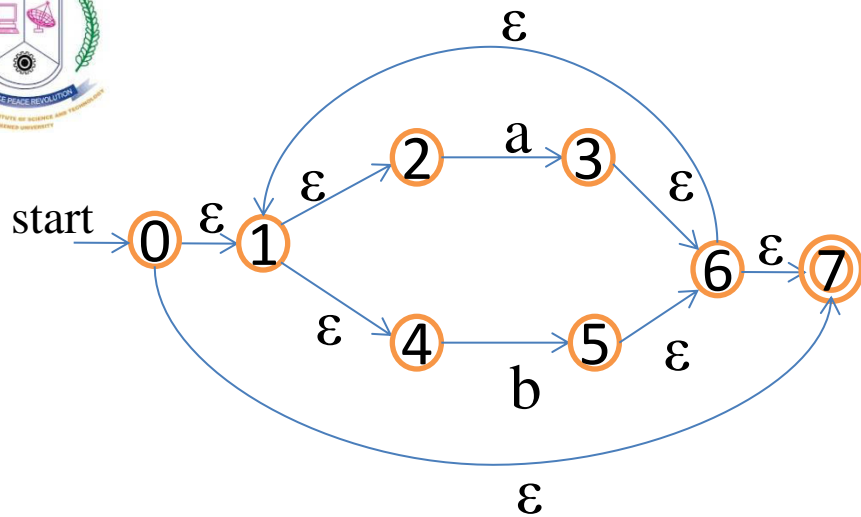
Existing DFA State

Dtran[B,a] = B



Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B	B	
C		



Step 2.5: Transition from $B = \{1, 2, 3, 4, 6, 7\}$

Compute $\epsilon\text{-closure}(\text{move}(B, b))$

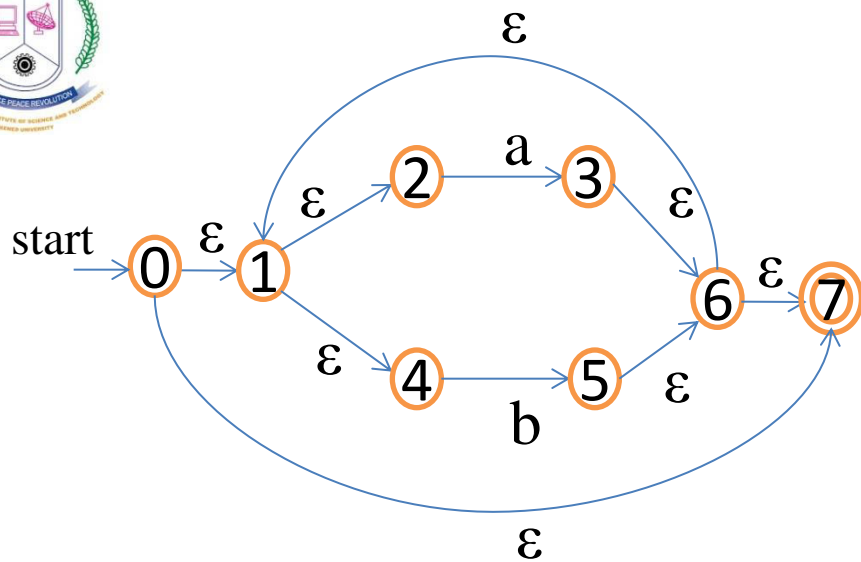
$\text{move}(B, b) = \{5\}$

$\epsilon\text{-closure}(\text{move}(B, b)) = \epsilon\text{-closure}(5) = C$

$D\text{tran}[B, b] = C$

Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B	B	C
C		



Step 2.6: Transition from $C=\{1,2,4,5,6,7\}$
 Compute $\epsilon\text{-closure}(\text{move}(C,a))$

$\text{move}(C,a) = \{3\}$

$\epsilon\text{-closure}(\text{move}(C,a)) =$

$\epsilon\text{-closure}(3) = \{3,6,7,1,2,4\} = \{1,2,3,4,6,7\}$
 Existing DFA State $\rightarrow \text{B}$

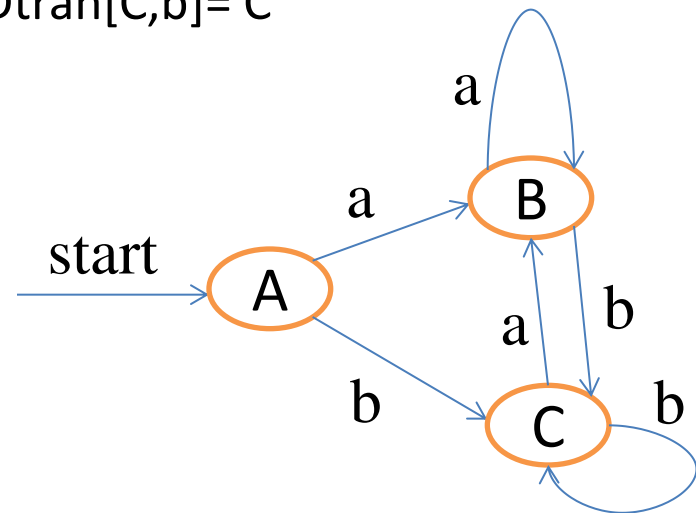
$\text{Dtran}[C,a] = B$

Step 2.7: Compute $\epsilon\text{-closure}(\text{move}(C,b))$

$\text{move}(C,b) = \{5\}$

$\epsilon\text{-closure}(\text{move}(C,b)) = C$

$\text{Dtran}[C,b] = C$



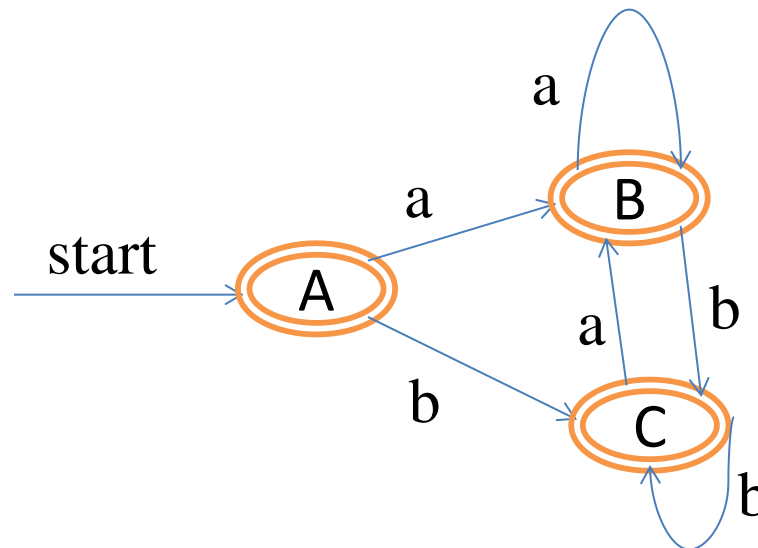
Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B	B	C
C	B	C



Step 3: No more new DFA states are formed.
Stop the subset construction method.

DFA for $(a|b)^*$



DFA= Subset of NFA

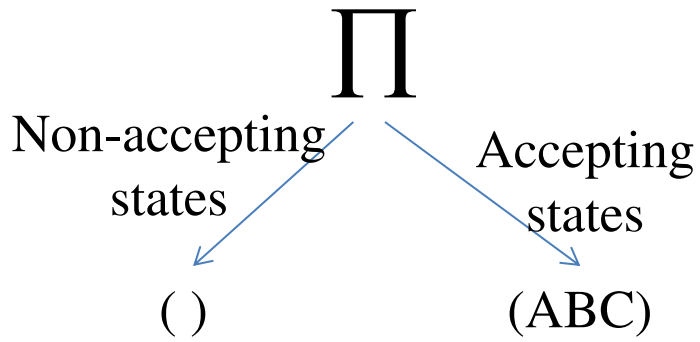
$A = \{0,1,2,4,7\}$  **start state and accepting state**(state 0 and 7)

$B = \{1,2,3,4,6,7\}$  **State B** contains the NFA accepting state(state 7)

$C = \{1,2,4,5,6,7\}$  **State C** contains the NFA accepting state(state 7)



Minimized DFA



All states remain in same group

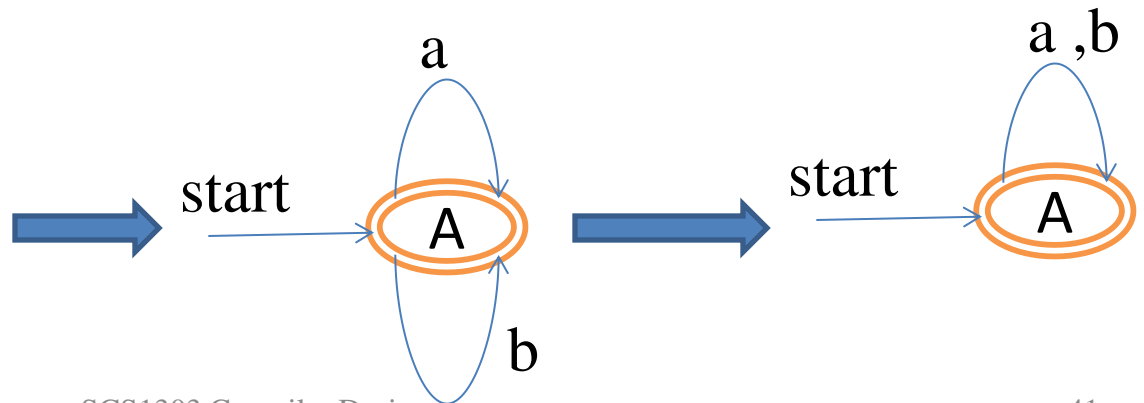
DFA states	Input Symbols	
	a	b
A	B	C
B	B	C
C	B	C

accepting states

Therefore, consider one representative state.

A=B=C

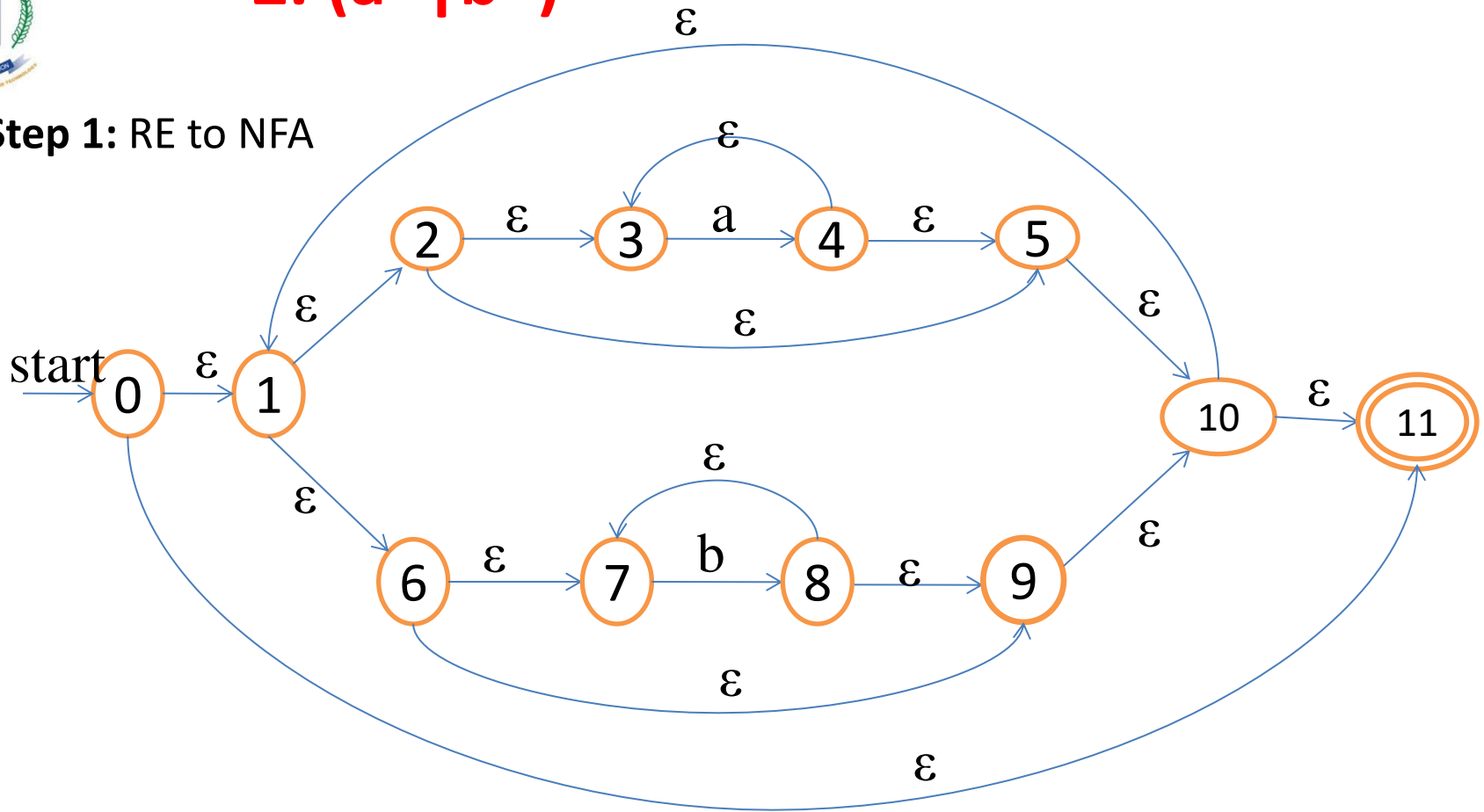
DFA states	Input Symbols	
	a	b
A	A	A





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

Step 1: RE to NFA



Step 2: Start state of equivalent DFA is $\epsilon\text{-closure}(0)$

$$\epsilon\text{-closure}(0) = \{0, 1, 11, 2, 6, 3, 5, 10, 7, 9\} = \{0, 1, 2, 3, 5, 6, 7, 9, 10, 11\}$$



New DFA State

Start state of DFA:



Step 2.1: Transition from

$A = \{0, 1, 2, 3, 5, 6, 7, 9, 10, 11\}$

Compute $\epsilon\text{-closure}(\text{move}(A, a))$

$\text{move}(A, a) = \{4\}$

$\epsilon\text{-closure}(\text{move}(A, a)) =$

$\epsilon\text{-closure}(4) = \{4, 5, 3, 10, 11, 1, 2, 6, 7, 9\}$

$= \{1, 2, 3, 4, 5, 6, 7, 9, 10, 11\} \longrightarrow \textcircled{B}$

New DFA State

$\text{Dtran}[A, a] = B$

Step 2.2: Compute $\epsilon\text{-closure}(\text{move}(A, b))$

$\text{move}(A, b) = \{8\}$

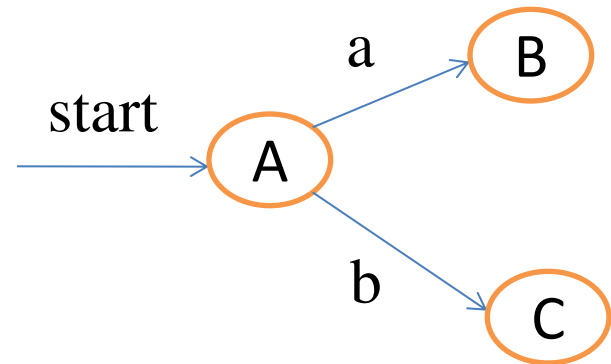
$\epsilon\text{-closure}(\text{move}(A, b)) =$

$\epsilon\text{-closure}(8) = \{8, 9, 7, 10, 11, 1, 2, 6, 3, 5\}$

$= \{1, 2, 3, 5, 6, 7, 8, 9, 10, 11\} \longrightarrow \textcircled{C}$

New DFA State

$\text{Dtran}[A, b] = C$



Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B		
C		



Step 3.1: Transition from

$B = \{1, 2, 3, 4, 5, 6, 7, 9, 10, 11\}$

Compute $\epsilon\text{-closure}(\text{move}(B, a))$

$\text{move}(B, a) = \{4\}$

$\epsilon\text{-closure}(\text{move}(B, a)) =$

$\epsilon\text{-closure}(4) = \{4, 5, 3, 10, 11, 1, 2, 6, 7, 9\}$

$= \{1, 2, 3, 4, 5, 6, 7, 9, 10, 11\} \longrightarrow \textcircled{B}$

Existing DFA State

$\text{Dtran}[B, a] = B$

Step 3.2: Compute $\epsilon\text{-closure}(\text{move}(B, b))$

$\text{move}(B, b) = \{8\}$

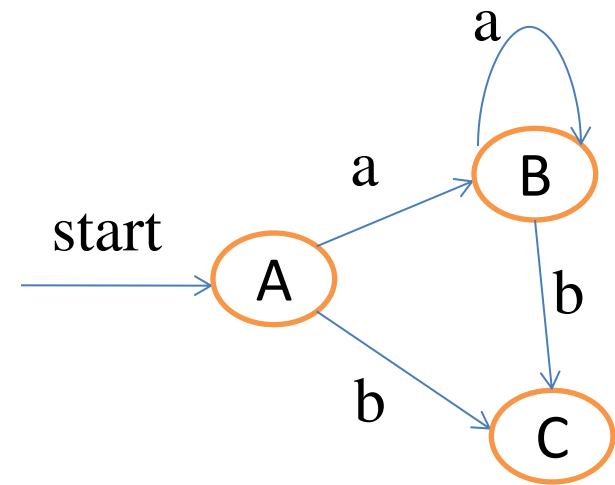
$\epsilon\text{-closure}(\text{move}(B, b)) =$

$\epsilon\text{-closure}(8) = \{8, 9, 7, 10, 11, 1, 2, 6, 3, 5\}$

$= \{1, 2, 3, 5, 6, 7, 8, 9, 10, 11\} \longrightarrow \textcircled{C}$

Existing DFA State

$\text{Dtran}[B, b] = C$



Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B	B	C
C		



Step 4.1: Transition from

$C = \{1, 2, 3, 5, 6, 7, 8, 9, 10, 11\}$

Compute $\epsilon\text{-closure}(\text{move}(C, a))$

$\text{move}(C, a) = \{4\}$

$\epsilon\text{-closure}(\text{move}(C, a)) =$

$\epsilon\text{-closure}(4) = \{4, 5, 3, 10, 11, 1, 2, 6, 7, 9\}$

$= \{1, 2, 3, 4, 5, 6, 7, 9, 10, 11\} \longrightarrow \textcircled{B}$

Existing DFA State

$\text{Dtran}[C, a] = B$

Step 4.2: Compute $\epsilon\text{-closure}(\text{move}(C, b))$

$\text{move}(C, b) = \{8\}$

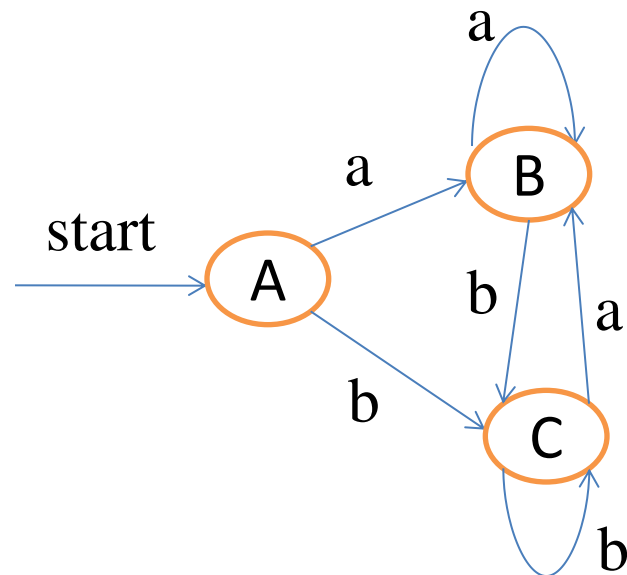
$\epsilon\text{-closure}(\text{move}(C, b)) =$

$\epsilon\text{-closure}(8) = \{8, 9, 7, 10, 11, 1, 2, 6, 3, 5\}$

$= \{1, 2, 3, 5, 6, 7, 8, 9, 10, 11\} \longrightarrow \textcircled{C}$

Existing DFA State

$\text{Dtran}[C, b] = C$



Transition Table **Dtran**

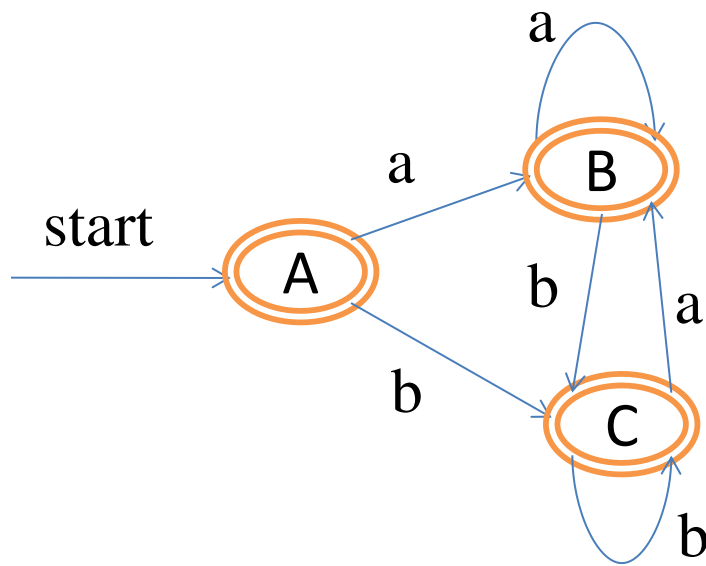
DFA states	Input Symbols	
	a	b
A	B	C
B	B	C
C	B	C



Step 5: No more new DFA states are formed.

Stop the subset construction method.

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



DFA= Subset of NFA

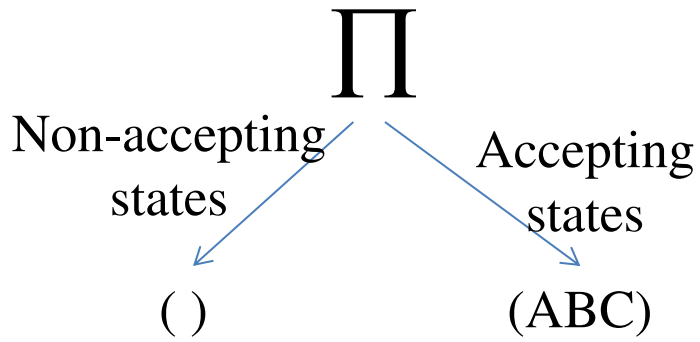
A = {0,1,2,3,5,6,7,9,10,11} start state and accepting state(state 0 and 11)

B = {1,2,3,4,5,6,7,9,10,11} B contains the NFA accepting state(state 11)

C = {1,2,3,5,6,7,8,9,10,11} C contains the NFA accepting state(state 11)



Minimized DFA



All states remain in same group

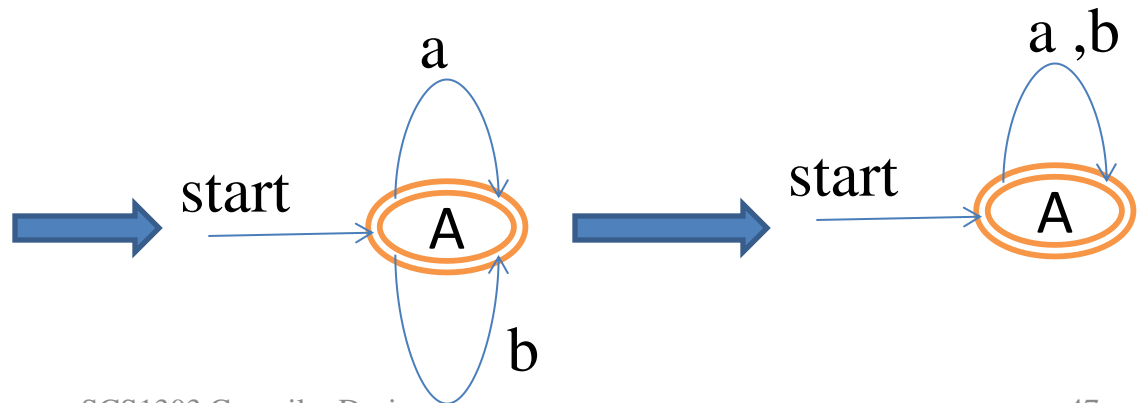
DFA states	Input Symbols	
	a	b
A	B	C
B	B	C
C	B	C

accepting states

Therefore, consider one representative state.

$A=B=C$

DFA states	Input Symbols	
	a	b
A	A	A





$((\epsilon | a)b^*)^* ?$

**Will this Regular Expression produce
an equivalent minimized DFA?**

Try it.....



THANK YOU