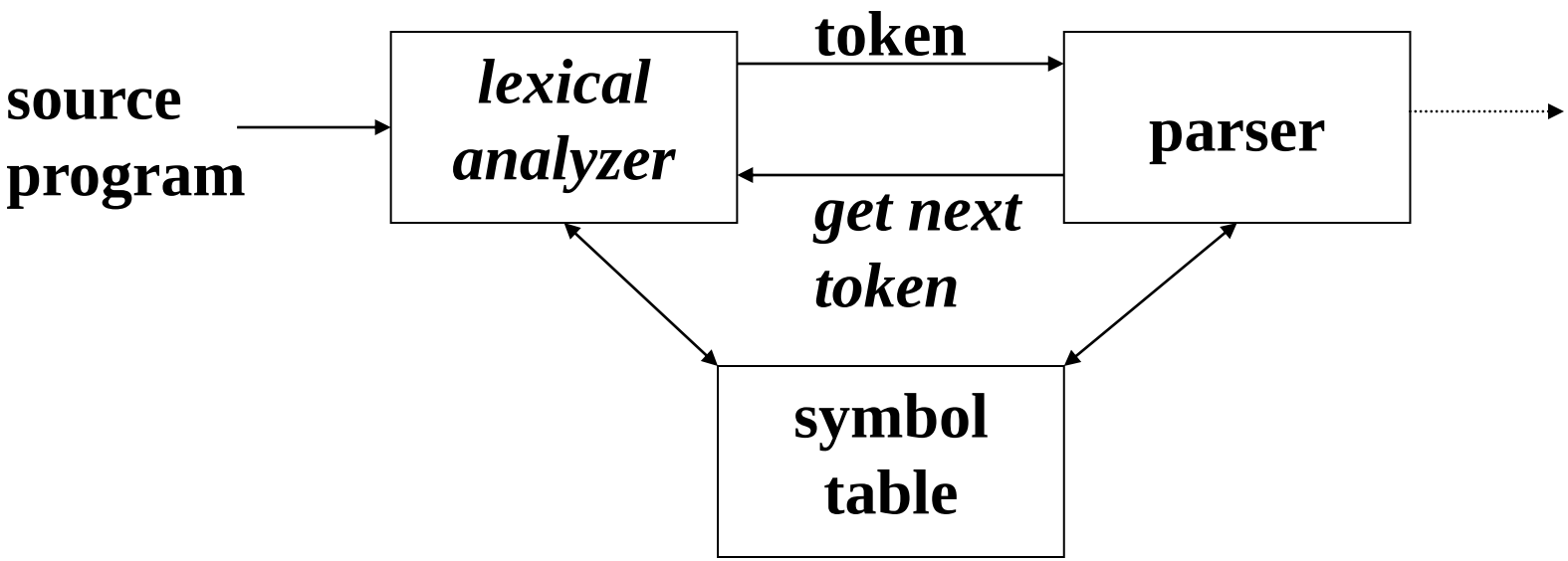
Chapter 3: Lexical Analysis

Lexical Analysis

- Basic Concepts & Regular Expressions
  - What does a Lexical Analyzer do?
  - How does it Work?
  - Formalizing Token Definition & Recognition
- LEX - A Lexical Analyzer Generator (Defer)
- Reviewing Finite Automata Concepts
  - Non-Deterministic and Deterministic FA
  - Conversion Process
    - Regular Expressions to NFA
    - NFA to DFA
- Relating NFAs/DFAs /Conversion to Lexical Analysis
- Concluding Remarks /Looking Ahead

Lexical Analyzer in Perspective



Important Issue:

What are Responsibilities of each Box ?

Focus on Lexical Analyzer and Parser.

Lexical Analyzer in Perspective

- **LEXICAL ANALYZER**
  - Scan Input
  - Remove WS, NL, …
  - Identify Tokens
  - Create Symbol Table
  - Insert Tokens into ST
  - Generate Errors
  - Send Tokens to Parser
- **PARSER**
  - Perform Syntax Analysis
  - Actions Dictated by Token Order
  - Update Symbol Table Entries
  - Create Abstract Rep. of Source
  - Generate Errors
  - And More…. (We'll see later)

What Factors Have Influenced the Functional Division of Labor ?

- Separation of Lexical Analysis From Parsing Presents a Simpler Conceptual Model
  - A parser embodying the conventions for comments and white space is significantly more complex that one that can assume comments and white space have already been removed by lexical analyzer.
- Separation Increases Compiler Efficiency
  - Specialized buffering techniques for reading input characters and processing tokens…
- Separation Promotes Portability.
  - Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.

Introducing Basic Terminology

- What are Major Terms for Lexical Analysis?
  - **TOKEN**
    - A classification for a common set of strings
    - Examples Include <Identifier>, <number>, etc.
  - **PATTERN**
    - The rules which characterize the set of strings for a token
    - Recall File and OS Wildcards ([A-Z]*.*)
  - **LEXEME**
    - Actual sequence of characters that matches pattern and is classified by a token
    - Identifiers: x, count, name, etc…

Introducing Basic Terminology

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	if
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except "

Classifies Pattern

Actual values are critical. Info is :

1. Stored in symbol table
2. Returned to parser

Attributes for Tokens

Tokens influence parsing decision; the attributes influence the translation of tokens.

Example: E = M * C ** 2

<**id,** pointer to symbol-table entry for R>

<**assign_op,** >

<**id,** pointer to symbol-table entry for M>

<**mult_op,** >

<**id,** pointer to symbol-table entry for C>

<**exp_op,** >

<**num,** integer value 2>

Handling Lexical Errors

- Error Handling is very localized, with Respect to Input Source
- For example: whil (x = 0) do
  generates **no** lexical errors in PASCAL
- **In what Situations do Errors Occur?**
  - Lexical analyzer is unable to proceed because none of the patterns for tokens matches a prefix of remaining input.
- **Panic mode Recovery**
  - Delete successive characters from the remaining input until the analyzer can find a well-formed token.
  - May confuse the parser
- **Possible error recovery actions:**
  - Deleting or Inserting Input Characters
  - Replacing or Transposing Characters

Buffer Pairs

- Lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced.
- Use a function **ungetc** to push lookahead characters back into the input stream.
- Large amount of time can be consumed moving characters.

Special Buffering Technique

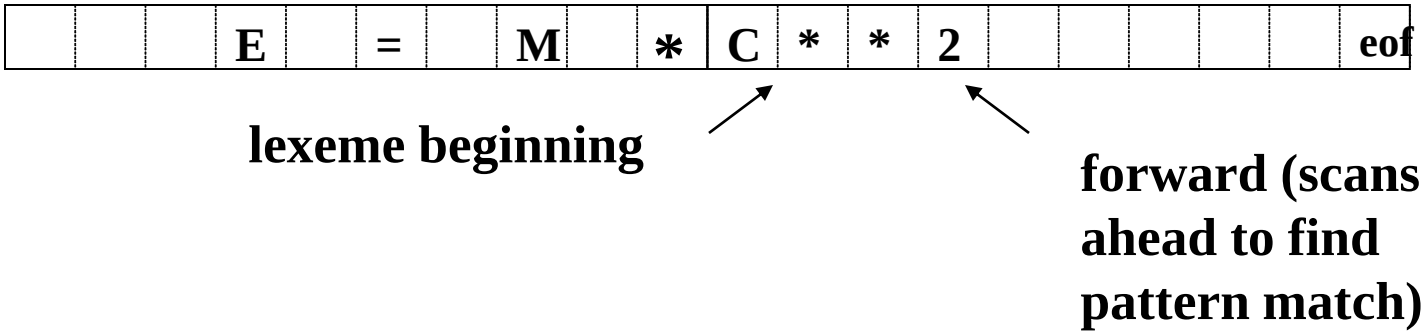
- ✓Use a buffer divided into two N-character halves
- ✓N = Number of characters on one disk block
- ✓One system command read N characters
- ✓Fewer than N character => eof

Buffer Pairs (2)

✓Two pointers to the input buffer are maintained

✓The string of characters between the pointers is the current lexeme

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



Comments and white space can be treated as patterns that yield no token

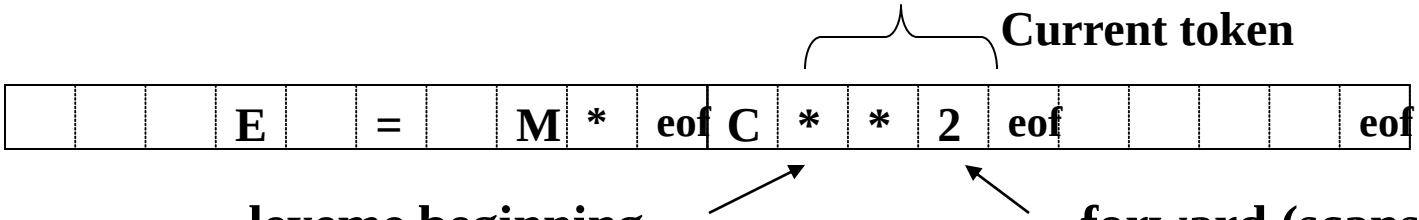
Code to advance forward pointer

```
if  forward at the 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 biginning of first half
end
else  forward  : =  forward  + 1;
```

Pitfalls

1. This buffering scheme works quite well most of the time but with it amount of lookahead is limited.
2. Limited lookahead makes it impossible to recognize tokens in situations where the distance, forward pointer must travel is more than the length of buffer.

Algorithm: Buffered I/O with Sentinels




```
forward : = forward + 1 ;
if forward ↑ = eof then begin
   if forward at end of first half then begin
        reload second half ;        ← Block I/O
        forward : = forward + 1
   end
   else if forward at end of second half then begin
        reload first half ;         ← Block I/O
        move forward to biginning of first half
   end
   else / * eof within buffer signifying end of input * /
        terminate lexical analysis
end                  2nd eof ⇒ no more input !
```

Algorithm performs I/O's. We can still have get & ungetchar Now these work on real memory buffers !

Formalizing Token Definition

EXAMPLES AND OTHER CONCEPTS:

Suppose: S is the string banana

Prefix : ban, banana

Suffix : ana, banana

Substring : nan, ban, ana, banana

Subsequence: bnan, nn

Proper prefix, subfix, or substring *cannot* be all of S

Language Concepts

A language, L, is simply any set of strings over a fixed alphabet.

Alphabet	Languages
{0,1}	{0,10,100,1000,100000…}
	{0,1,00,11,000,111,…}
{a,b,c}	{abc,aabbcc,aaabbbccc,…}
{A, … ,Z}	{TEE,FORE,BALL,…}
	{FOR,WHILE,GOTO,…}
{A,…,Z,a,…,z,0,…9, +,-,…,<,>,…}	{ All legal PASCAL progs}

Special Languages: ∅ - EMPTY LANGUAGE

∈ - contains ∈ string only

Formal Language Operations

OPERATION	DEFINITION
union **of L and M written** $L \cup M$	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
concatenation of **L and M written LM**	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure **of L written L***	$L^* = \bigcup_{i=0}^{\infty} L^i$ **L* denotes "zero or more concatenations of " L**
positive closure **of L written** L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ **denotes "one or more concatenations of " L**

Formal Language Operations Examples

$L = \{A, B, C, D\}$ $D = \{1, 2, 3\}$

$L \cup D = \{A, B, C, D, 1, 2, 3\}$

$LD = \{A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, D3\}$

$L^2 = \{ AA, AB, AC, AD, BA, BB, BC, BD, CA, \ldots DD\}$

$L^4 = L^2 \; L^2 = ??$

$L^* = \{$ All possible strings of L plus $\in \}$

$L^+ = L^* - \in$

$L (L \cup D) = ??$

$L (L \cup D)^* = ??$

Language & Regular Expressions

- A Regular Expression is a Set of Rules / Techniques for Constructing Sequences of Symbols (Strings) From an Alphabet.
- Let Σ Be an Alphabet, r a Regular Expression Then L(r) is the Language That is Characterized by the Rules of r

Rules for Specifying Regular Expressions:

Regular expressions over alphabet Σ

- $\in$ is a regular expression denoting $\{\in\}$, set containing the empty string
- If **a** is in Σ, **a** is a regular expression that denotes {**a**}
- Let r and s be regular expressions with languages L(r) and L(s). Then

precedence ↑

(a) (r) | (s) is a regular expression $\Rightarrow$ L(r) $\cup$ L(s)

(b) (r)(s) is a regular expression $\Rightarrow$ L(r) L(s)

(c) (r)* is a regular expression $\Rightarrow$ (L(r))*

(d) (r) is a regular expression $\Rightarrow$ L(r)

All are Left-Associative. Parentheses are dropped as allowed by precedence rules.

EXAMPLES of Regular Expressions

$L = \{A, B, C, D\}$ $D = \{1, 2, 3\}$

A | B | C | D = L

(A | B | C | D) (A | B | C | D) = L^2

(A | B | C | D)* = L*

(A | B | C | D) ((A | B | C | D) | (1 | 2 | 3)) = L (L ∪ D)

Algebraic Properties of Regular Expressions

AXIOM	DESCRIPTION
r | s = s | r	| is commutative
r | (s | t) = (r | s) | t	| is associative
(r s) t = r (s t)	concatenation is associative
r (s | t) = r s | r t (s | t) r = s r | t r	concatenation distributes over |
∈r = r r∈ = r	∈ Is the identity element for concatenation
r* = (r | ∈)*	relation between * and ∈
r** = r*	* is idempotent

Regular Expression Examples

1. **All Strings that start with “tab” or end with “bat”:**

   tab{A,…,Z,a,...,z}*|{A,…,Z,a,....,z}*bat

2. **All Strings in Which Digits 1,2,3 exist in ascending numerical order:**

   {A,…,Z}*1 {A,…,Z}*2 {A,…,Z}*3 {A,…,Z}*

Regular Definitions

We may give names to regular expressions and to define regular expression using these names as if they were symbols.

Let Σ is an alphabet of basic symbols. The regular definition is a sequence of definitions of the form

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

$$\ldots$$

$$d_n \rightarrow r_n$$

Where,

each d_i is a distinct name, and

each r_i is a regular expression over the symbols in

$$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$$

Towards Token Definition

Regular Definitions: Associate names with Regular Expressions

For Example : PASCAL IDs

letter → A | B | C | … | Z | a | b | … | z

digit → 0 | 1 | 2 | … | 9

id → letter (letter | digit)*

Shorthand Notation:

"+" : one or more $r^* = r^+ \mid \in$ (Kleene) & $r^+ = r\, r^*$ (Positive)

"?" : zero or one r?=r | ∈

[range] : set range of characters (replaces "|")

[A-Z] = A | B | C | … | Z

Example Using Shorthand : PASCAL IDs

id → [A-Za-z][A-Za-z0-9]*

Unsigned Number

1240, 39.45, 6.33E15, or 1.578E-41

digit → 0 | 1 | 2 | … | 9

digits → digit digit*

optional_fraction → . digits | ∈

optional_exponent → (E (+ | -| ∈) digits) | ∈

num → digits optional_fraction optional_exponent

Shorthand

digit → 0 | 1 | 2 | … | 9

digits → digit^{+}

optional_fraction → (. digits) ?

optional_exponent → (E (+ | -) ? digits) ?

num → digits optional_fraction optional_exponent

Token Recognition

How can we use concepts developed so far to assist in recognizing tokens of a source language ?

Assume Following Tokens:

if, then, else, relop, id, num

What language construct are they used for ?

Given Tokens, What are Patterns ?

if → **if**

then → **then**

else → **else**

relop → < | <= | > | >= | = | <>

id → **letter (letter | digit)***

num → **digit** $^{+}$ **(. digit** $^{+}$ **) ? (E(+ | -) ? digit** $^{+}$ **) ?**

What does this represent ?

Grammar:

stmt → |**if** *expr* **then** *stmt*
|**if** *expr* **then** *stmt* **else** *stmt*
|∈

expr → *term* **relop** *term* | *term*

term → **id** | **num**

What Else Does Lexical Analyzer Do?

Scan away *blanks*, new lines, tabs

Can we Define Tokens For These?

$$\begin{aligned}
\textbf{blank} &\rightarrow \textbf{blank} \\
\textbf{tab} &\rightarrow \textbf{tab} \\
\textbf{newline} &\rightarrow \textbf{newline} \\
\textbf{delim} &\rightarrow \textbf{blank} \mid \textbf{tab} \mid \textbf{newline} \\
\textbf{ws} &\rightarrow \textbf{delim}^{+}
\end{aligned}$$

Ans: No token is returned to parser

Overall

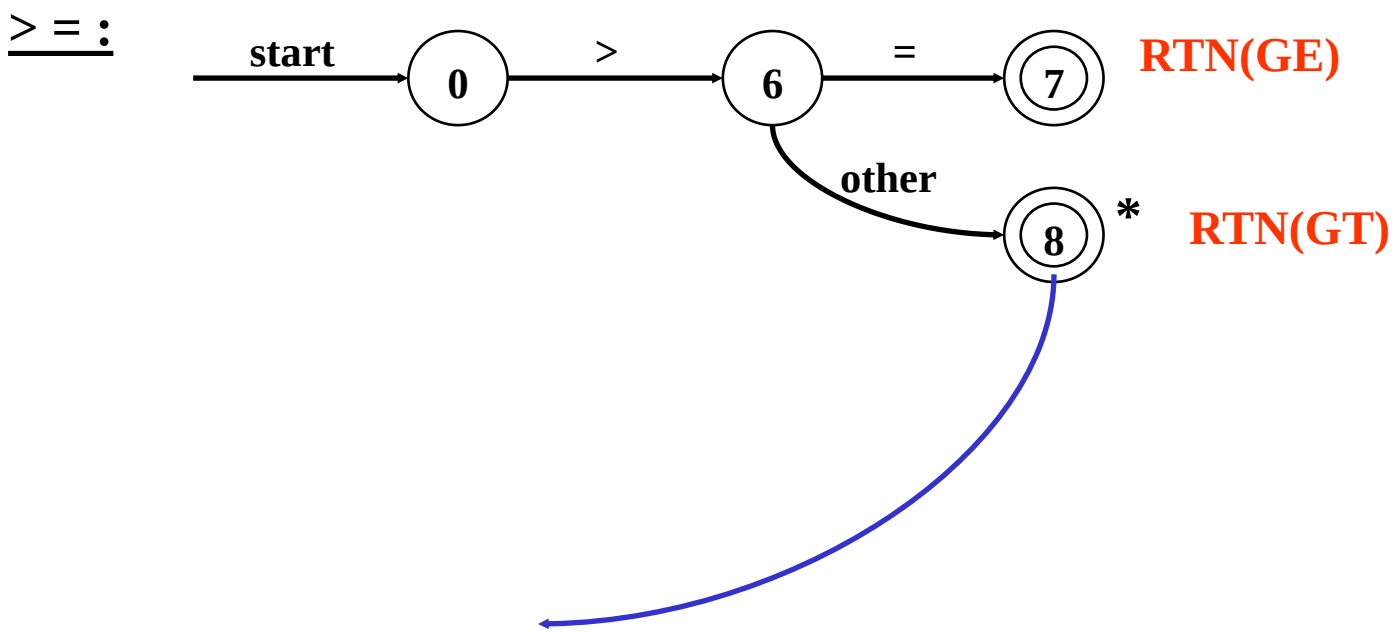
Regular Expression	Token	Attribute-Value
ws	-	-
if	**if**	-
then	**then**	-
else	**else**	-
id	**id**	**pointer to table entry**
num	**num**	**Exact value**
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
< >	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Note: Each token has a unique token identifier to define category of lexemes

Constructing Transition Diagrams for Tokens

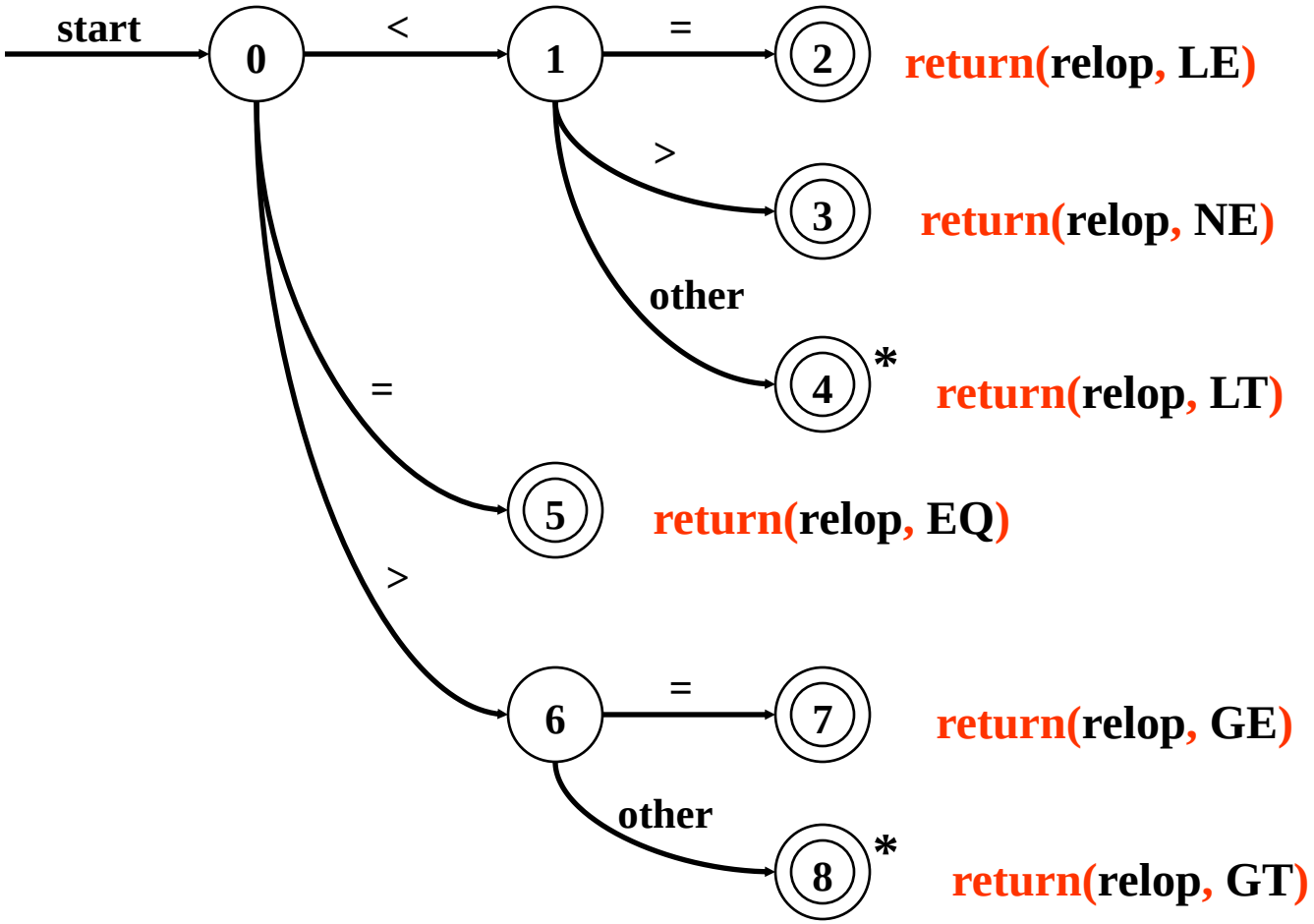
- **Transition Diagrams (TD) are used to represent the tokens**
- **As characters are read, the relevant TDs are used to attempt to match lexeme to a pattern**
- **Each TD has:**
  - **States : Represented by Circles**
  - **Actions : Represented by Arrows between states**
  - **Start State : Beginning of a pattern (Arrowhead)**
  - **Final State(s) : End of pattern (Concentric Circles)**
- **Each TD is Deterministic (assume) - No need to choose between 2 different actions !**

Example TDs



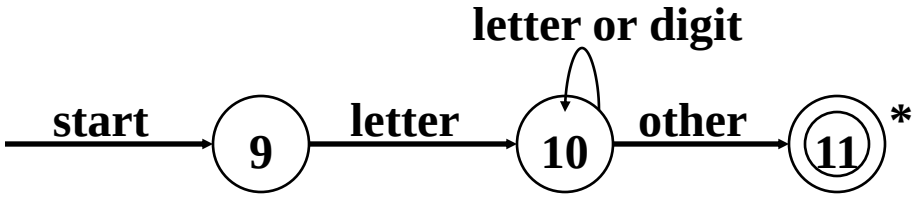
We've accepted ">" and have read other char that must be unread.

Example : All RELOPs



Example TDs : id and delim

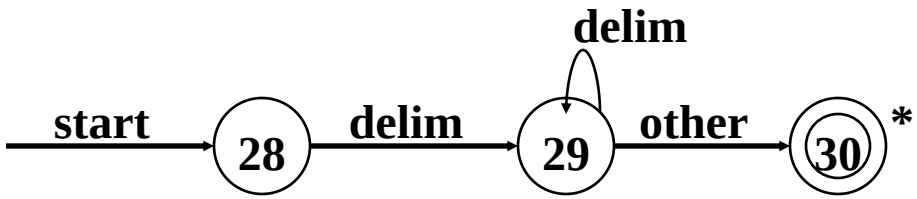
id :



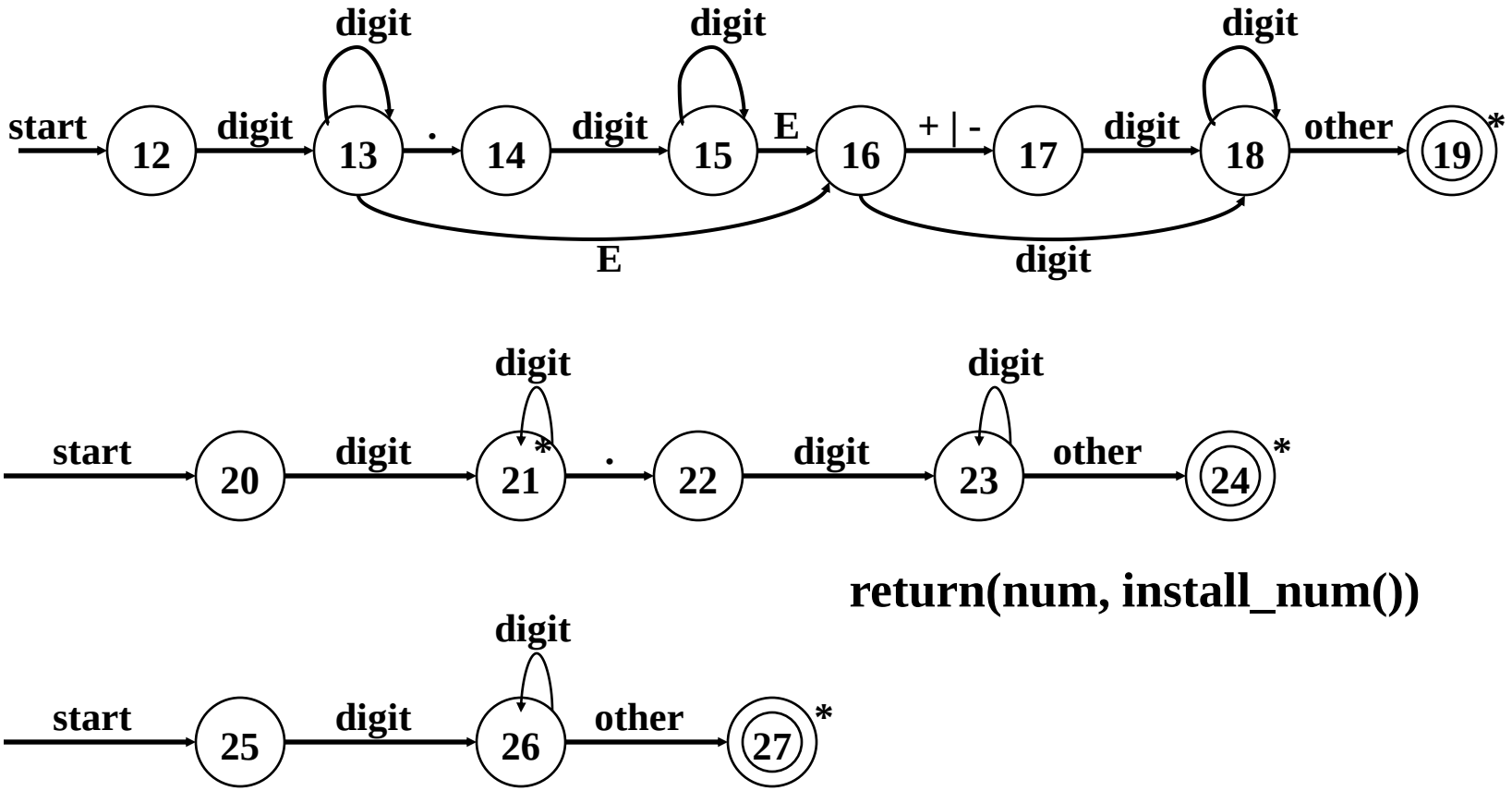
return(get_token(), install_id())

Either returns ptr or "0" if reserved

delim :



Example TDs : Unsigned #s



Questions: Is ordering important for unsigned #s ?

Why are there no TDs for `then, else, if` ?

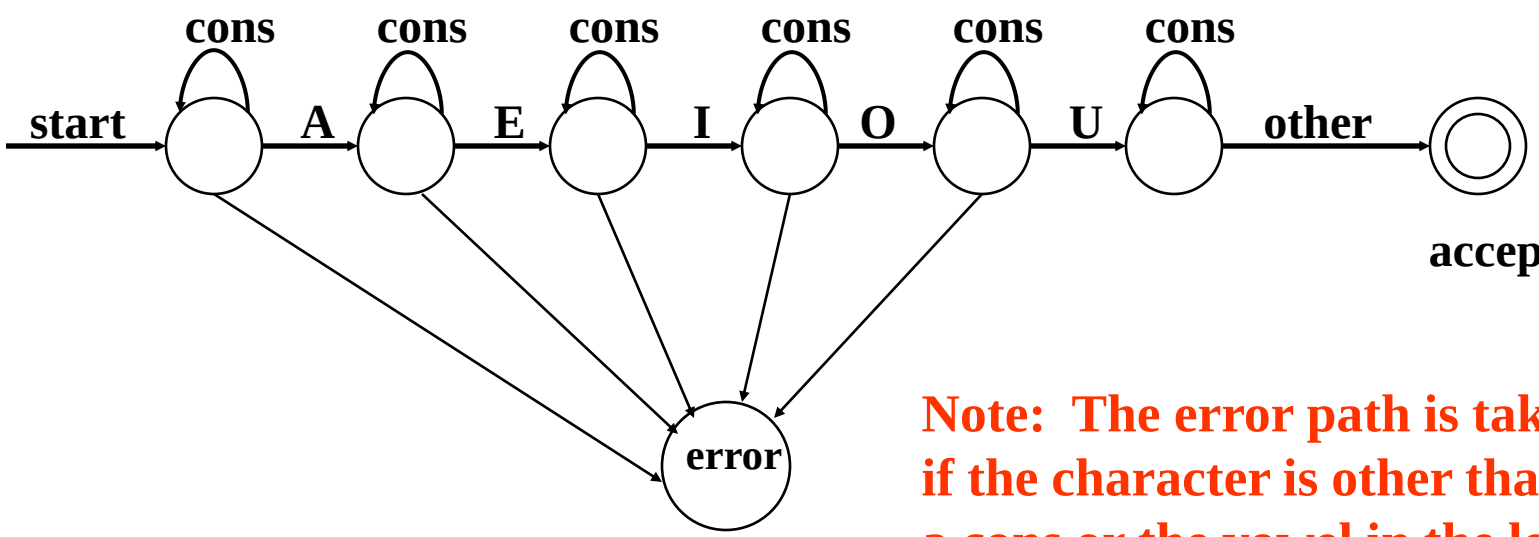
QUESTION :

What would the transition diagram (TD) for strings containing each vowel, in their strict lexicographical order, look like ?

Answer

cons → B | C | D | F | … | Z

string → cons* A cons* E cons* I cons* O cons* U cons*



Note: The error path is taken if the character is other than a cons or the vowel in the lex order.

What Else Does Lexical Analyzer Do?

All Keywords / Reserved words are matched as ids

- After the match, the symbol table or a special keyword table is consulted
- Keyword table contains string versions of all keywords and associated token values

if	15
then	16
begin	17
...	...

- When a match is found, the token is returned, along with its symbolic value, i.e., "then", 16
- If a match is not found, then it is assumed that an **id** has been discovered

Implementing Transition Diagrams

A sequence of transition diagrams can be converted into a program to look for the tokens specified by the grammar

Each state gets a segment of code

FUNCTIONS USED

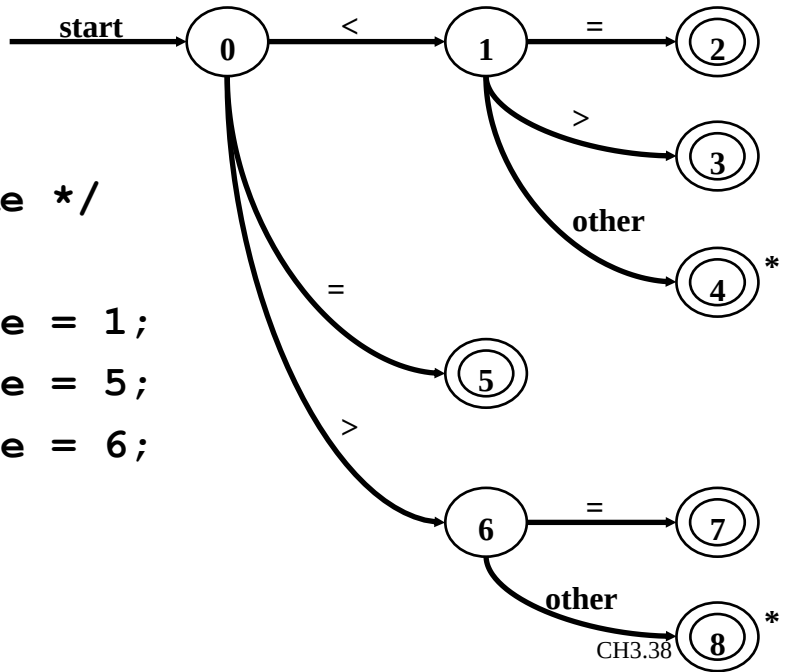
- nextchar(),
- retract(),
- install_num(),
- install_id(),
- gettoken(),
- isdigit(),
- isletter(),
- recover()

Implementing Transition Diagrams

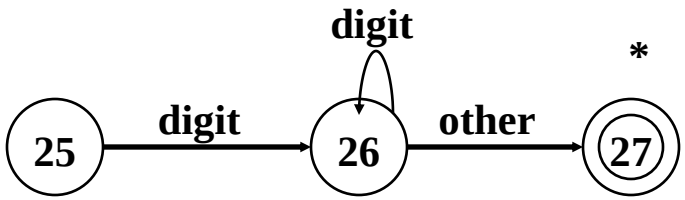
```
int state = 0, start = 0

lexeme_beginning = forward;
token nexttoken()
{   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 */
```

repeat until a "return" occurs



Implementing Transition Diagrams, II




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

advances `forward`

Case numbers correspond to transition diagram states !

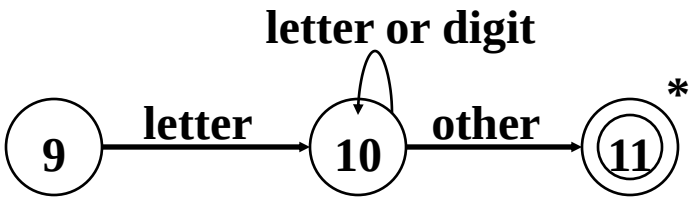
looks at the region `lexeme_beginning ... forward`

retracts `forward`

Implementing Transition Diagrams, III

```
.............
 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); lexical_value = install_id();
           return ( gettoken(lexical_value) );
.............
```

reads token name from ST



When Failures Occur:

```
int fail()
{
     forward = lexeme 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:   /* lex error */
     }
     return start;
}
```

Switch to next transition diagram

Finite Automata & Language Theory

Finite Automata : **A recognizer that takes an input string & determines whether it's a valid sentence of the language**

Non-Deterministic : **Has more than one alternative action for the same input symbol.**

Deterministic : **Has at most one action for a given input symbol.**

Both types are used to recognize regular expressions.

NFAs & DFAs

Non-Deterministic Finite Automata (NFAs) easily represent regular expression, but are somewhat less precise.

Deterministic Finite Automata (DFAs) require more complexity to represent regular expressions, but offer more precision.

We'll review both plus conversion algorithms, i.e., NFA → DFA and DFA → NFA

Non-Deterministic Finite Automata

An NFA is a mathematical model that consists of :

- **S, a set of states**
- **Σ, the symbols of the input alphabet**
- ***move*, a transition function.**
  - ***move*(state, symbol) → set of states**
  - ***move*** **: S × Σ∪{∈} → Pow(S)**
- **A state, $s_0 \in S$, the start state**
- **$F \subseteq S$, a set of final or accepting states.**

Representing NFAs

Transition Diagrams : **Number states (circles), arcs, final states, …**

Transition Tables: **More suitable to representation within a computer**

We'll see examples of both !

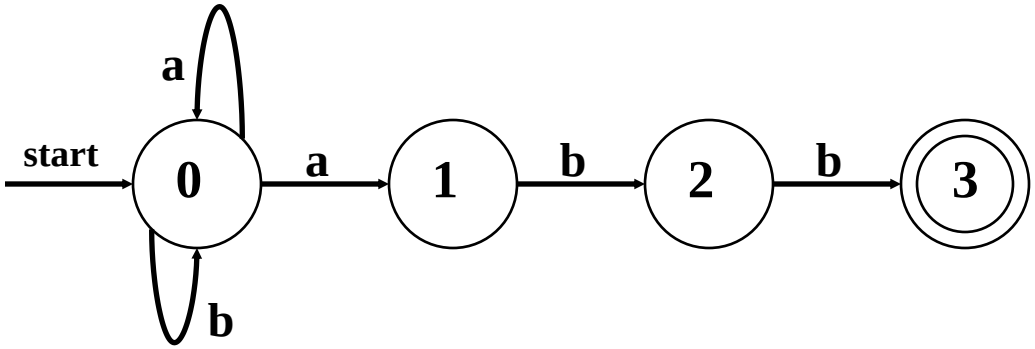
Example NFA

$S = \{ 0, 1, 2, 3 \}$

$s_0 = 0$

$F = \{ 3 \}$

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

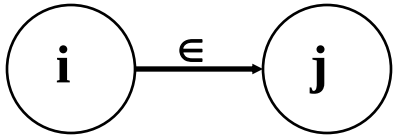


What Language is defined ?

What is the Transition Table ?

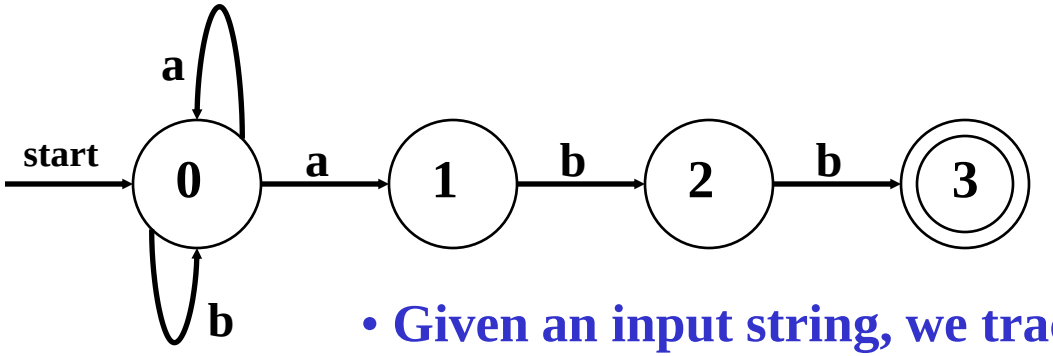
state \ input	a	b
0	{ 0, 1 }	{ 0 }
1	--	{ 2 }
2	--	{ 3 }

∈ (null) moves possible



Switch state but do not use any input symbol

How Does An NFA Work ?



- Given an input string, we trace moves
- If no more input & in final state, ACCEPT

EXAMPLE:

Input: ababb

move(0, a) = 1
move(1, b) = 2
move(2, a) = ? (undefined)

REJECT !

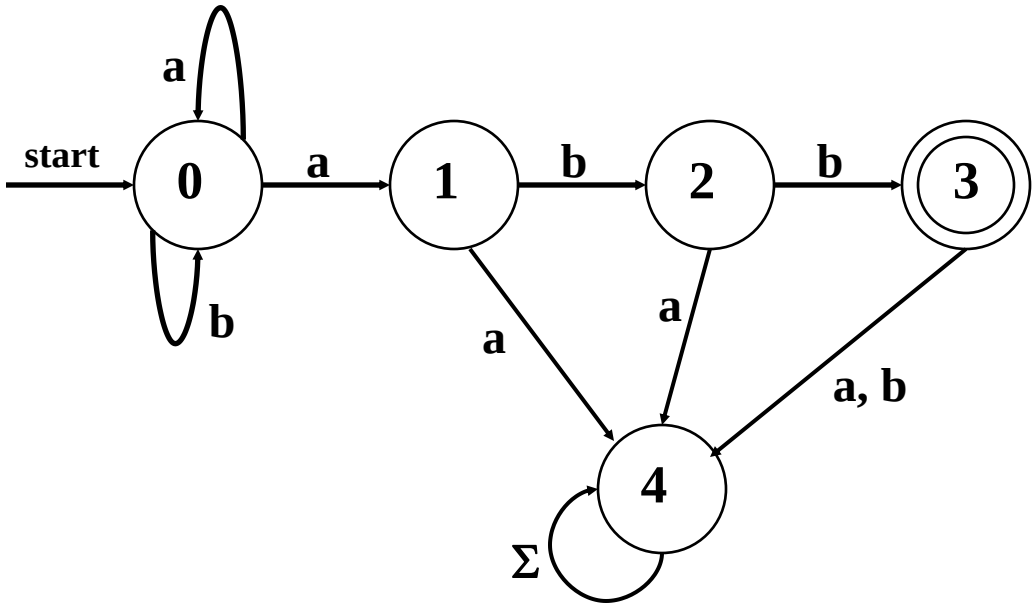
-OR-

move(0, a) = 0
move(0, b) = 0
move(0, a) = 1
move(1, b) = 2
move(2, b) = 3
ACCEPT !

Conversion of NFA to DFA

Handling Undefined Transitions

We can handle undefined transitions by defining one more state, a "death" state, and transitioning all previously undefined transition to this death state.



NFA- Regular Expressions & Compilation

Problems with NFAs for Regular Expressions:

1. Valid input might not be accepted
2. NFA may behave differently on the same input

Relationship of NFAs to Compilation:

1. Regular expression "recognized" by NFA
2. Regular expression is "pattern" for a "token"
3. Tokens are building blocks for lexical analysis
4. Lexical analyzer can be described by a collection of NFAs. Each NFA is for a language token.

Second NFA Example

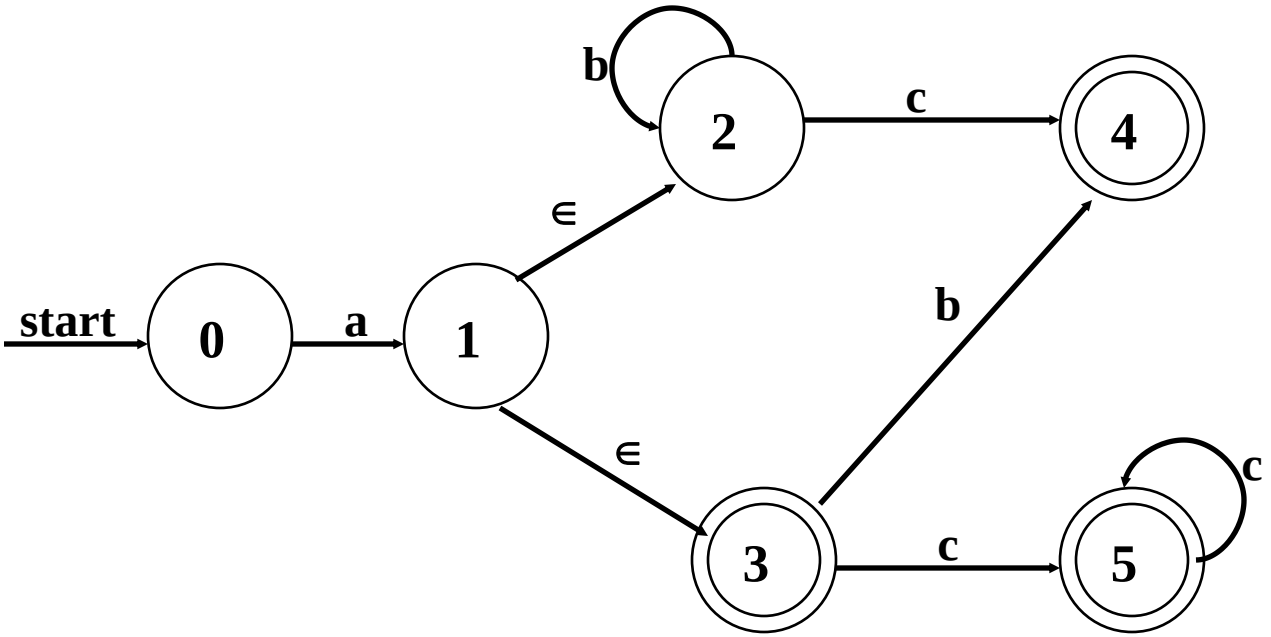
Given the regular expression : (a (b*c)) | (a (b | c^+)?)

Find a transition diagram NFA that recognizes it.

Second NFA Example - Solution

Given the regular expression : (a (b*c)) | (a (b | c^+)?)

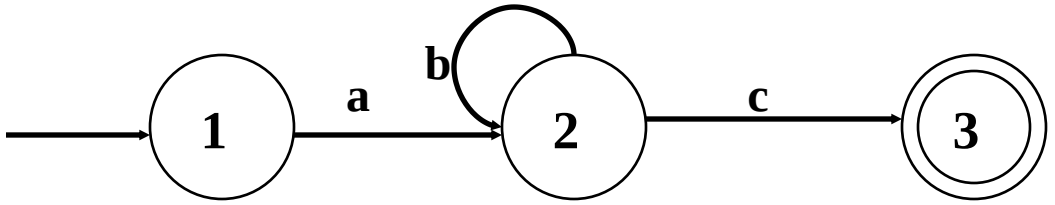
Find a transition diagram NFA that recognizes it.



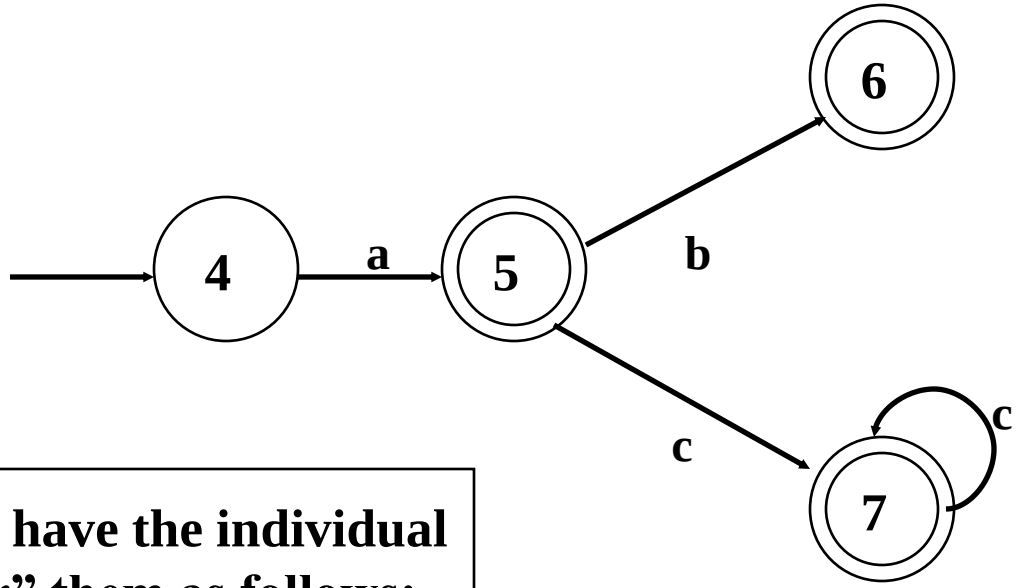
String abbc can be accepted.

Alternative Solution Strategy

a (b*c)

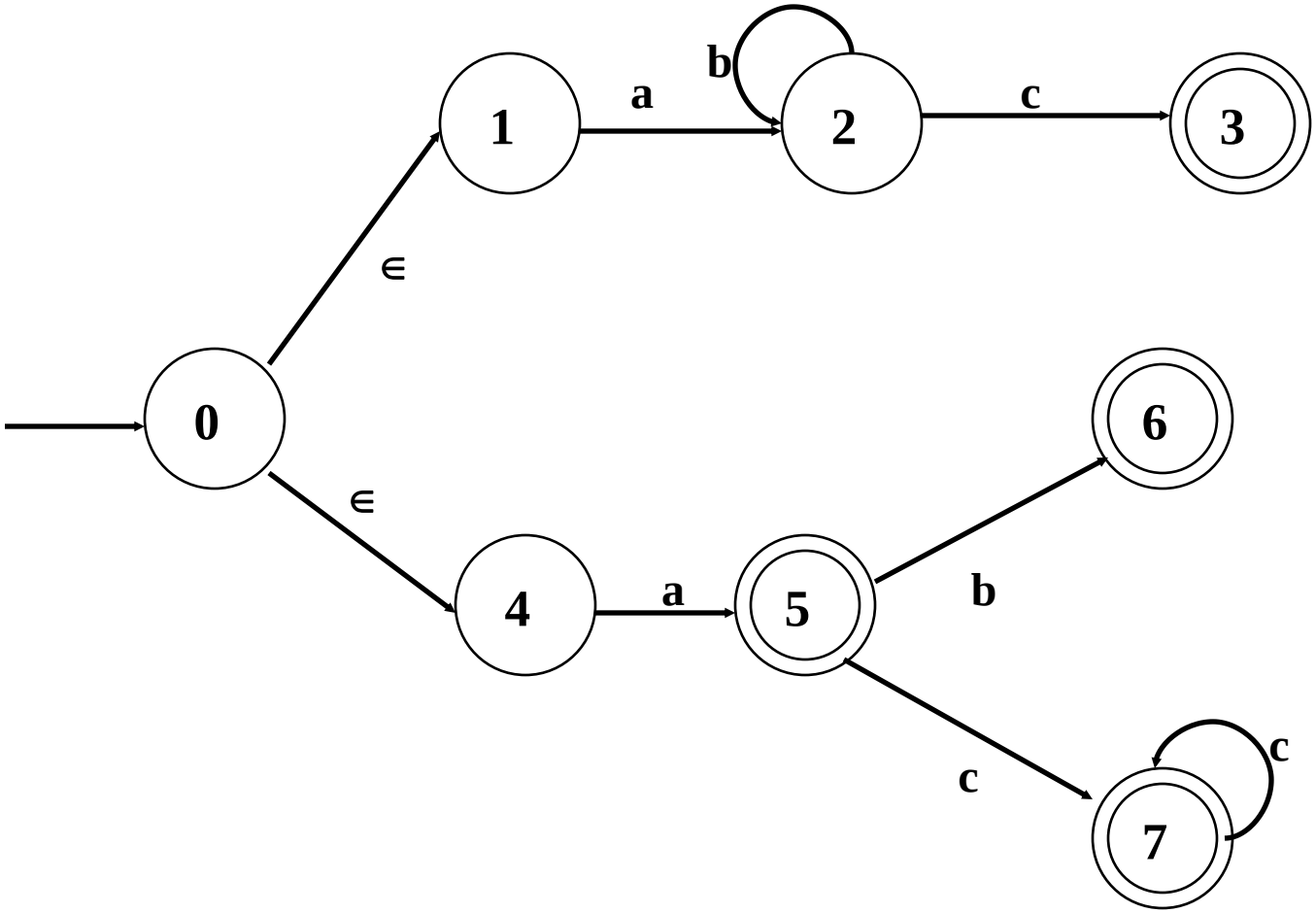


a (b | c+)?



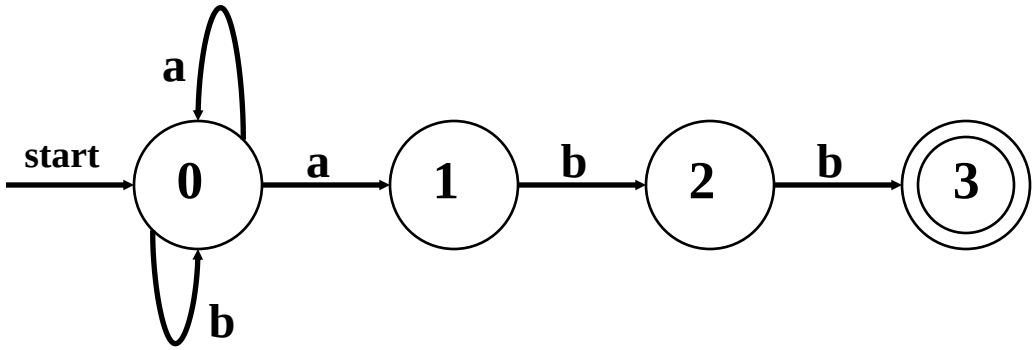
Now that you have the individual diagrams, "or" them as follows:

Using Null Transitions to "OR" NFAs



Other Concepts

Not all paths may result in acceptance.



aabb is accepted along path : $0 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3$

BUT… it is <u>not accepted</u> along the valid path:

$$0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0$$

Deterministic Finite Automata

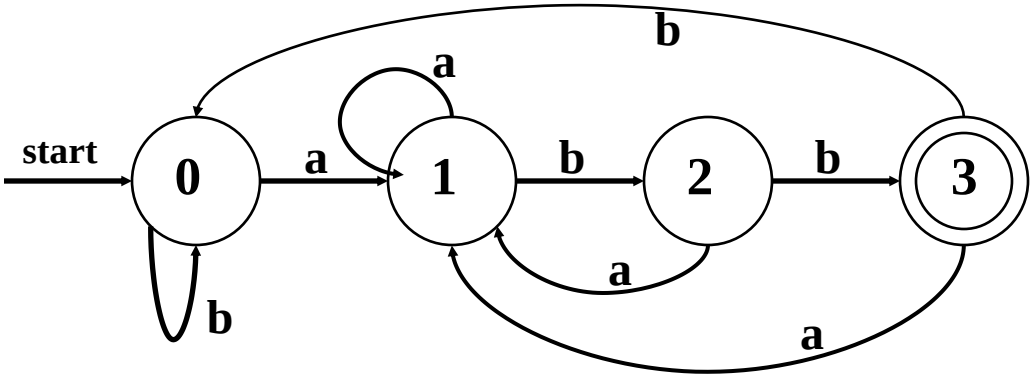
A DFA is an NFA with the following restrictions:

- **$\in$ moves are <u>not</u> allowed**
- **For every state s $\in$S, there is one and only one path from s for every input symbol a $\in \Sigma$.**

Since transition tables don't have any alternative options, DFAs are easily simulated via an algorithm.

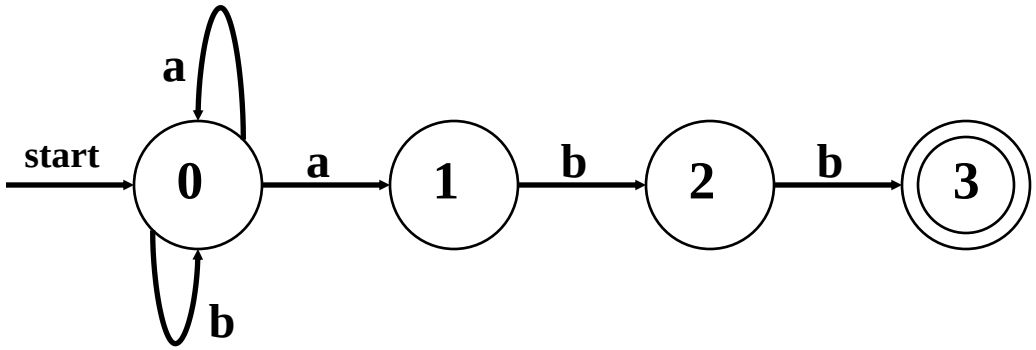
```
s ← s0
c ← nextchar;
while c ≠ eof do
  s ← move(s,c);
  c ← nextchar;
end;
if s is in F then return "yes"
             else return "no"
```

Example - DFA



What Language is Accepted?

Recall the original NFA:



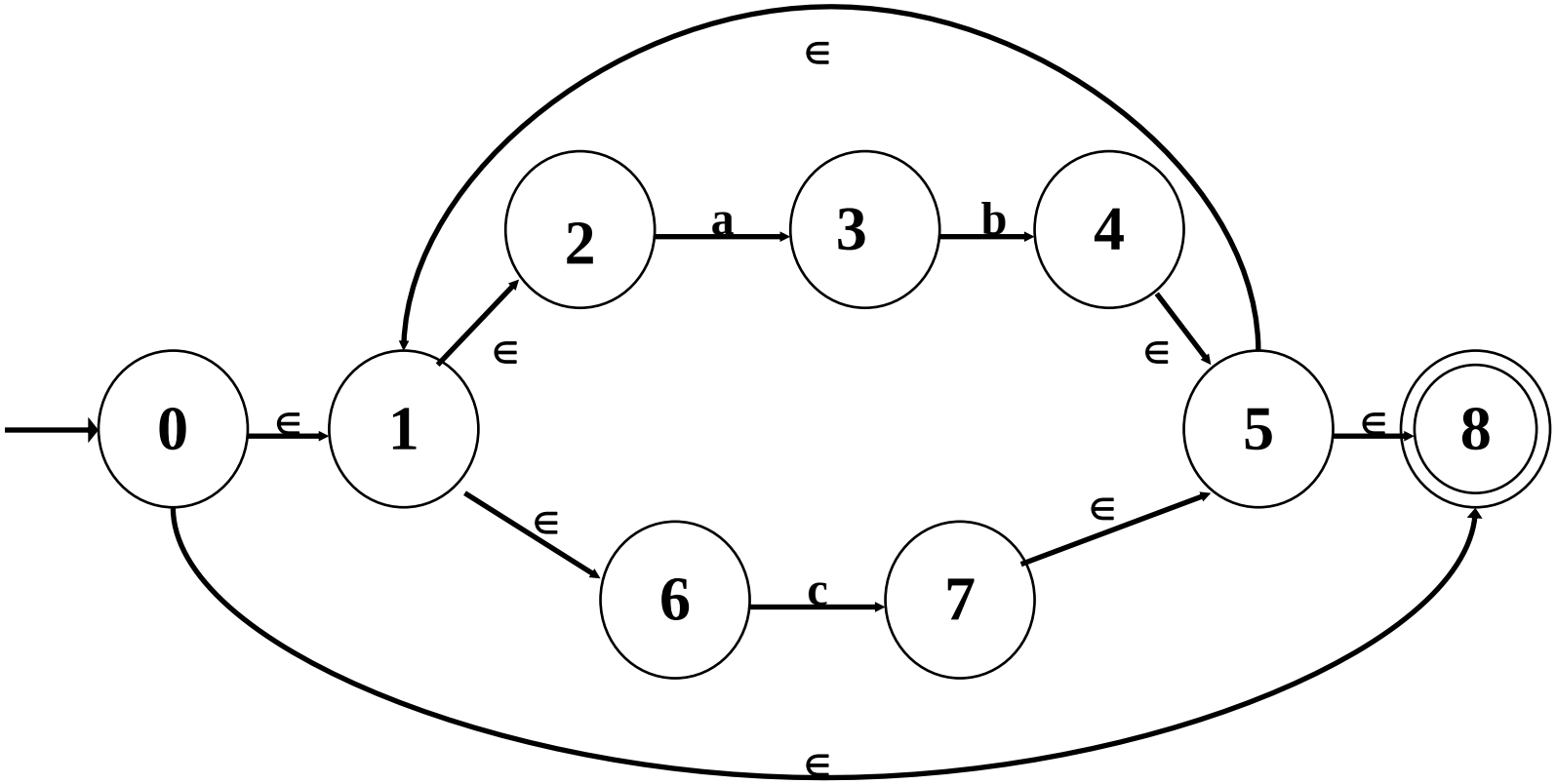
Conversion : NFA → DFA Algorithm

- **Algorithm Constructs a Transition Table for DFA from NFA**
- **Each state in DFA corresponds to a SET of states of the NFA**
- **Why does this occur ?**
  - **$\in$ moves**
  - **non-determinism**

  Both require us to characterize multiple situations that occur for accepting the same string.

  (Recall : Same input can have multiple paths in NFA)
- **Key Issue : Reconciling AMBIGUITY !**

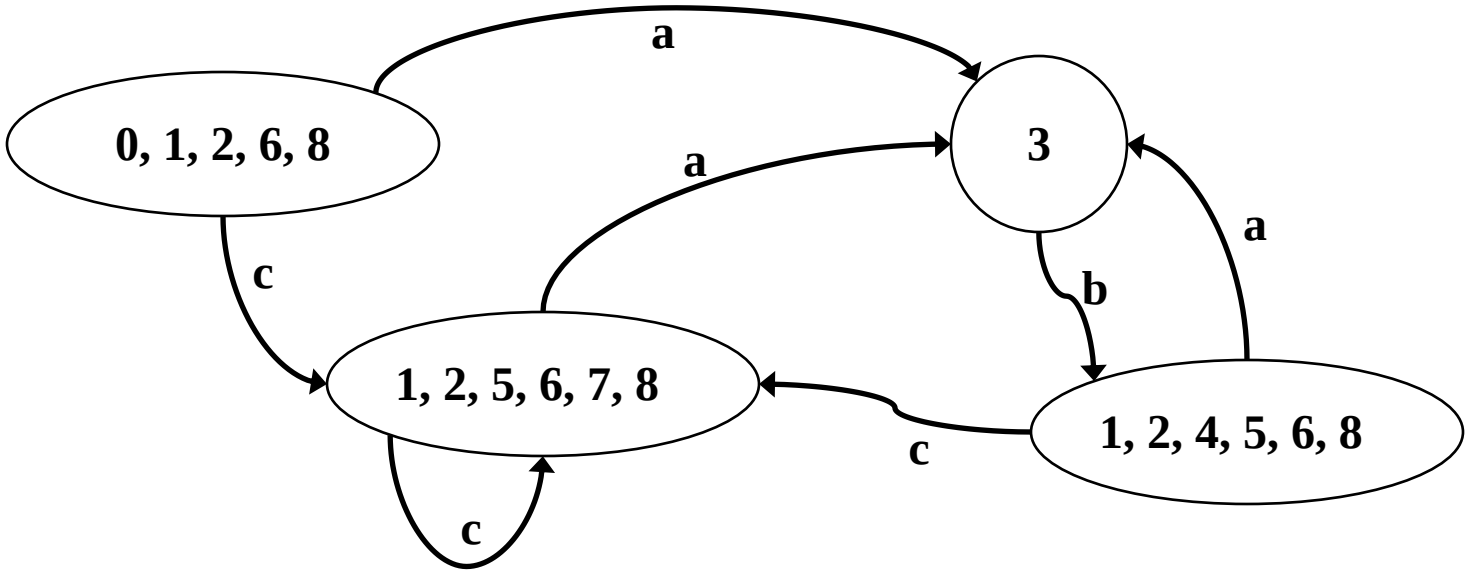
Converting NFA to DFA – 1st Look



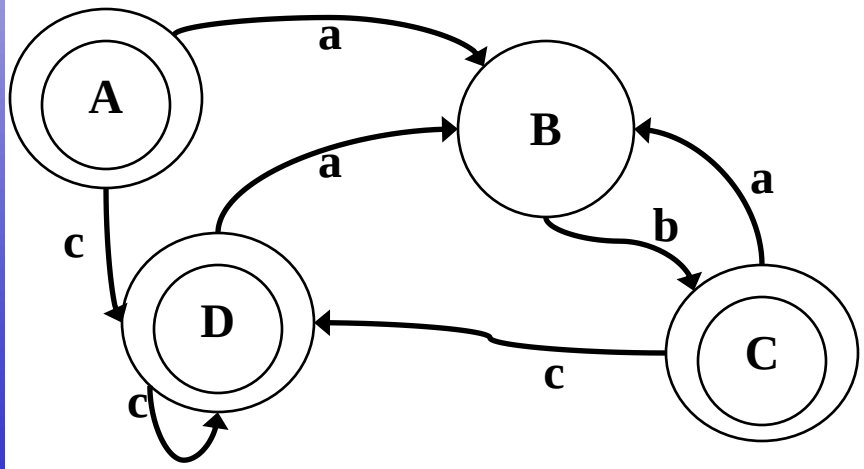
From State 0, Where can we move without consuming any input ?

This forms a new state: 0,1,2,6,8 What transitions are defined for this new state ?

The Resulting DFA



Which States are FINAL States ?



How do we handle alphabet symbols not defined for A, B, C, D ?

Algorithm Concepts

NFA N = (S, Σ, s_0, F, MOVE)

∈-Closure(s) : s ∈S

: set of states in S that are reachable from s via ∈-moves of N that originate from s.

No input is consumed

∈-Closure(T) : T ⊆ S

: NFA states reachable from all t ∈ T on ∈-moves only.

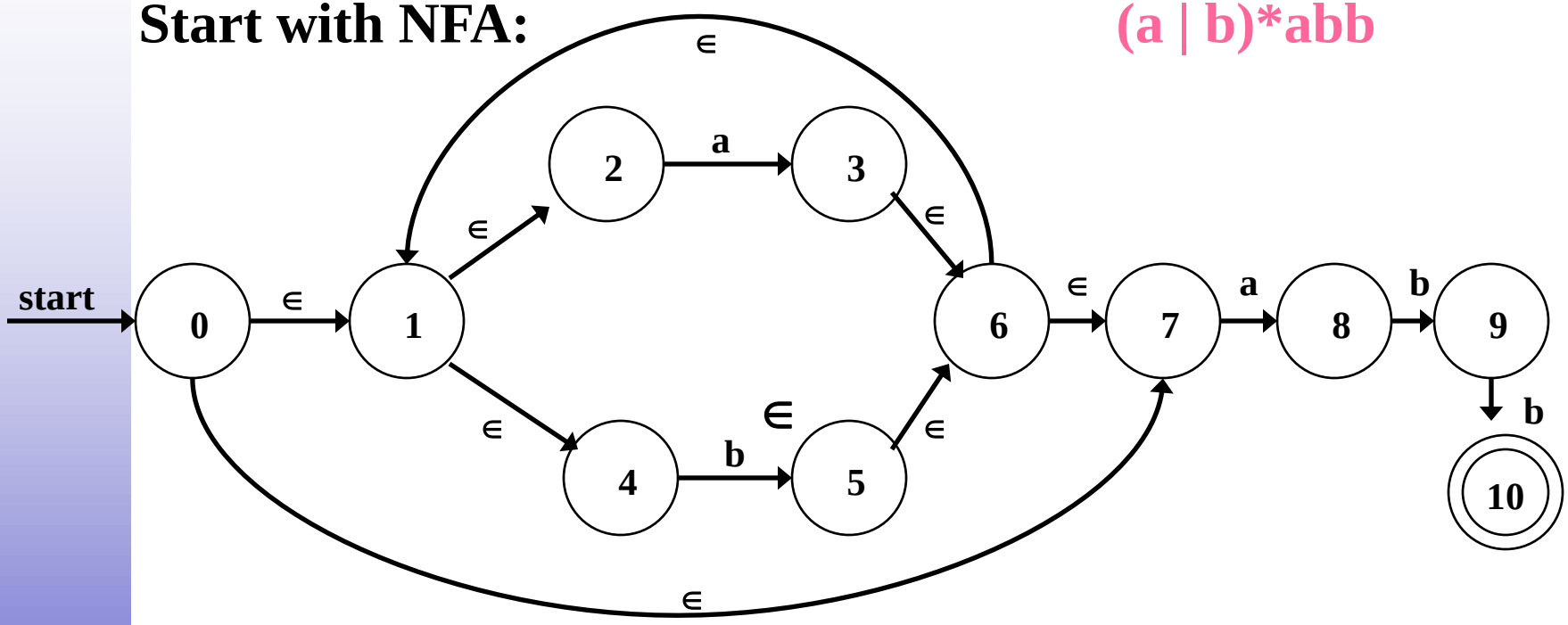
***move*(T,a) : T ⊆ S, a∈Σ**

: Set of states to which there is a transition on input a from some t ∈ T

These 3 operations are utilized by algorithms / techniques to facilitate the conversion process.

Illustrating Conversion – An Example

Start with NFA: **(a | b)*abb**



First we calculate: ε-closure(0) **(i.e., state 0)**

ε-closure(0) = {0, 1, 2, 4, 7} **(all states reachable from 0 on ε-moves)**

Let A={0, 1, 2, 4, 7} be a state of new DFA, D.

Conversion Example – continued (1)

2nd , we calculate : a : ∈-closure(*move*(A,a)) and
b : ∈-closure(*move*(A,b))

a : ∈-closure(*move*(A,a)) = ∈-closure(*move*({0,1,2,4,7},a))}
adds {3,8} (since *move*(2,a)=3 and *move*(7,a)=8)

From this we have : ∈-closure({3,8}) = {1,2,3,4,6,7,8}
(since 3→6 →1 →4, 6 →7, and 1 →2 all by ∈-moves)

Let B={1,2,3,4,6,7,8} be a new state. Define Dtran[A,a] = B.

b : ∈-closure(*move*(A,b)) = ∈-closure(*move*({0,1,2,4,7},b))

adds {5} (since *move*(4,b)=5)

From this we have : ∈-closure({5}) = {1,2,4,5,6,7}
(since 5→6 →1 →4, 6 →7, and 1 →2 all by ∈-moves)

Let C={1,2,4,5,6,7} be a new state. Define Dtran[A,b] = C.

Conversion Example – continued (2)

3rd , we calculate for state B on {a,b}

a : ∈-closure(*move*(B,a)) = ∈-closure(*move*({1,2,3,4,6,7,8},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[B,a] = B.

b : ∈-closure(*move*(B,b)) = ∈-closure(*move*({1,2,3,4,6,7,8},b))}
= {1,2,4,5,6,7,9} = D

Define Dtran[B,b] = D.

4th , we calculate for state C on {a,b}

a : ∈-closure(*move*(C,a)) = ∈-closure(*move*({1,2,4,5,6,7},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[C,a] = B.

b : ∈-closure(*move*(C,b)) = ∈-closure(*move*({1,2,4,5,6,7},b))}
= {1,2,4,5,6,7} = C

Define Dtran[C,b] = C.

Conversion Example – continued (3)

5th , we calculate for state D on {a,b}

a : ∈-closure(*move*(D,a)) = ∈-closure(*move*({1,2,4,5,6,7,9},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[D,a] = B.

b : ∈-closure(*move*(D,b)) = ∈-closure(*move*({1,2,4,5,6,7,9},b))}
= {1,2,4,5,6,7,10} = E

Define Dtran[D,b] = E.

Finally, we calculate for state E on {a,b}

a : ∈-closure(*move*(E,a)) = ∈-closure(*move*({1,2,4,5,6,7,10},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[E,a] = B.

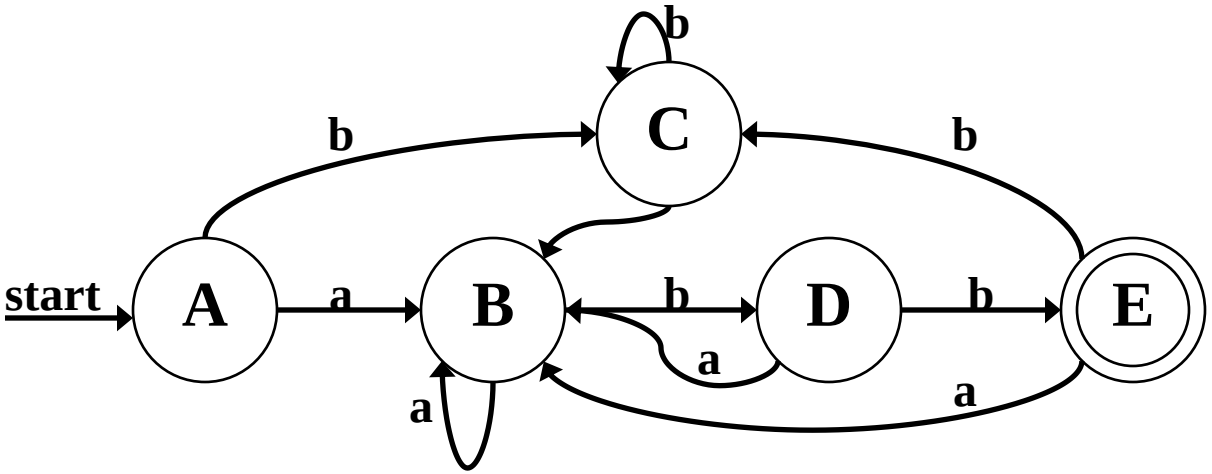
b : ∈-closure(*move*(*E*,b)) = ∈-closure(*move*({1,2,4,5,6,7,10},b))}
= {1,2,4,5,6,7} = C

Define Dtran[E,b] = C.

Conversion Example – continued (4)

This gives the transition table Dtran for the DFA of:

Dstates	Input Symbol a	Input Symbol b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



Algorithm For Subset Construction

computing the ∈-closure

```
push all states in T onto stack;
initialize ∈-closure(T) to T;
while stack is not empty do begin
    pop t, the top element, off the stack;
    for each state u with edge from t to u labeled ∈ do
        if u is not in ∈-closure(T) do begin
            add u to ∈-closure(T) ;
            push u onto stack
    end
end
```

Algorithm For Subset Construction – (2)

initially, $\in$-closure(s_0) is only (unmarked) state in Dstates;

`while` there is unmarked state T in Dstates `do begin`

    mark T;

    `for` each input symbol *a* `do begin`

        U := $\in$-closure(*move*(T,*a*));

        `if` U is not in Dstates `then`

            add U as an unmarked state to Dstates;

        Dtran[T,*a*] := U

    `end`

`end`

Regular Expression to NFA Construction

We now focus on transforming a Reg. Expr. to an NFA

This construction allows us to take:

- **Regular Expressions (which describe tokens)**
- **To an NFA (to characterize language)**
- **To a DFA (which can be "computerized")**

The construction process is component-wise

Builds NFA from components of the regular expression in a special order with particular techniques.

NOTE: Construction is "syntax-directed" translation, i.e., syntax of regular expression is determining factor for NFA construction and structure.

Motivation: Construct NFA For:

∈:

a :

b:

ab:

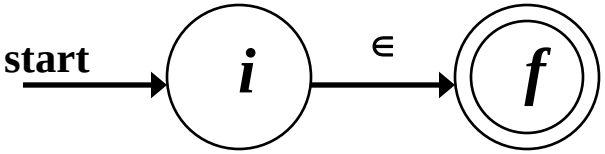
∈ | ab :

a*

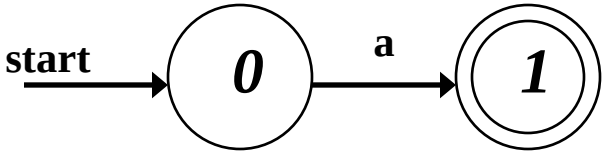
(∈ | ab)* :

Motivation: Construct NFA For:

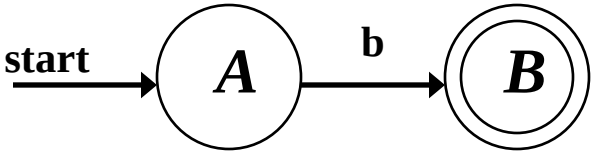
∈ :



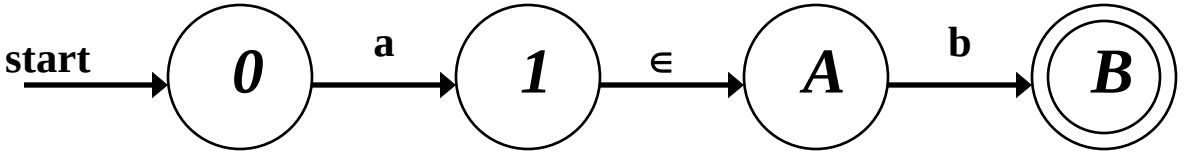
a :



b:



ab:



∈ | ab :

a*

(∈ | ab)* :

Construction Algorithm : R.E. → NFA

Construction Process :

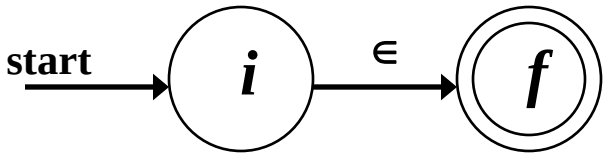
1st : Identify subexpressions of the regular expression

- **∈**
- **Σ symbols**
- **r | s**
- **rs**
- **r***

2nd : Characterize "pieces" of NFA for each subexpression

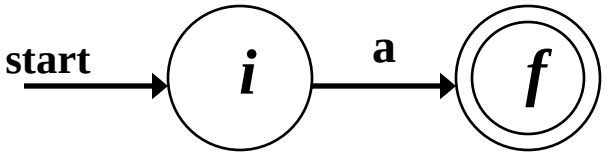
Piecing Together NFAs
Algorithm: Thompson's Construction

1. For ∈ in the regular expression, construct NFA



L(∈)

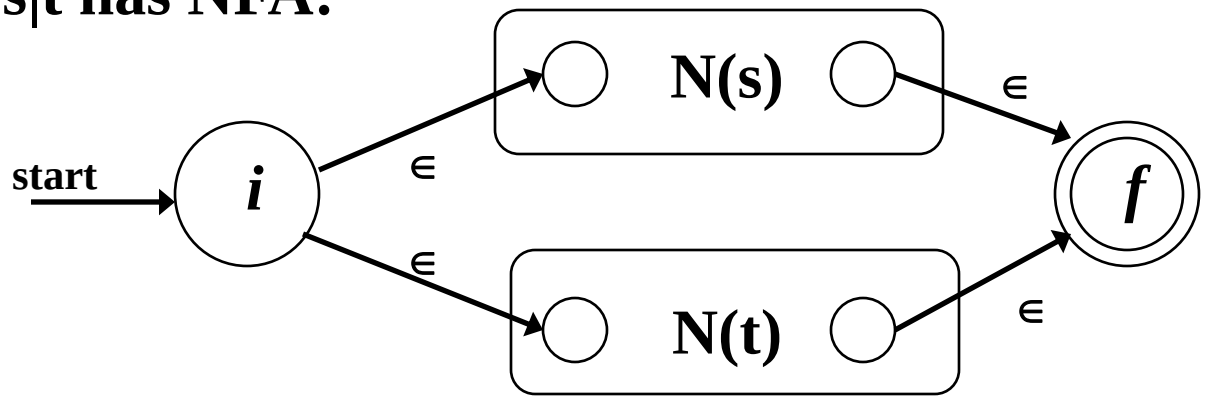
2. For a ∈ Σ in the regular expression, construct NFA



L(a)

Piecing Together NFAs – continued(1)

3.(a) If s, t are regular expressions, N(s), N(t) their NFAs s|t has NFA:

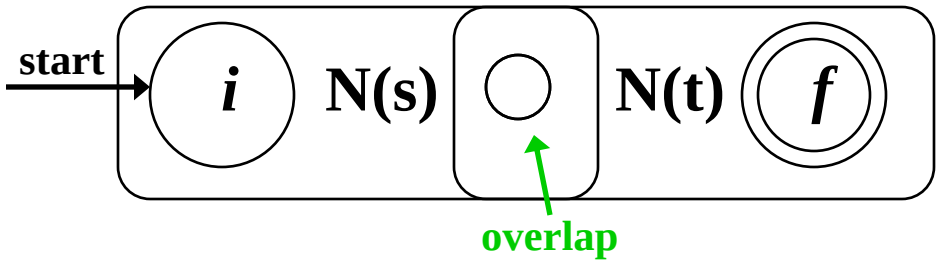


$L(s) \cup L(t)$

where *i* and *f* are new start / final states, and ∈-moves are introduced from *i* to the old start states of N(s) and N(t) as well as from all of their final states to *f*.

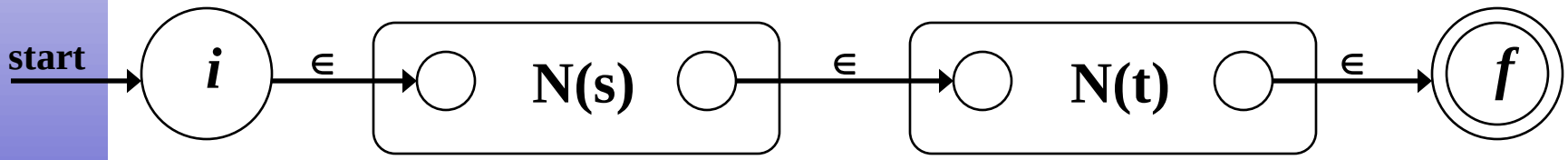
Piecing Together NFAs – continued(2)

3.(b) If s, t are regular expressions, N(s), N(t) their NFAs st (concatenation) has NFA:



L(s) L(t)

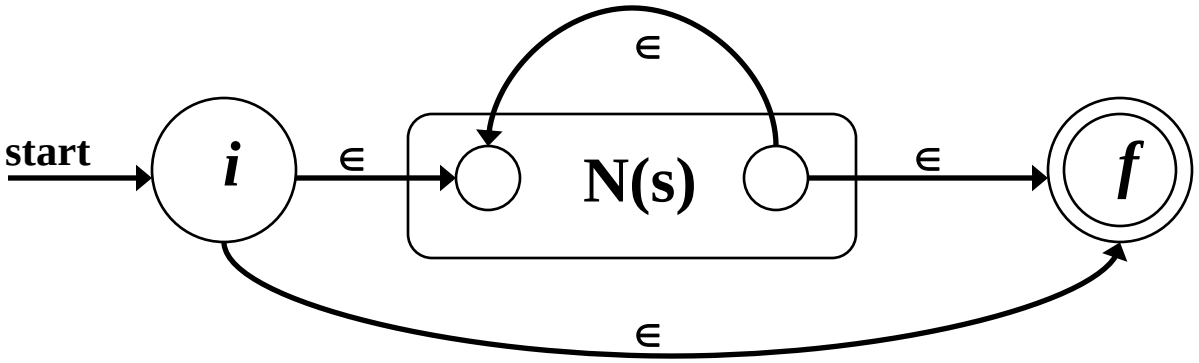
Alternative:



where *i* is the start state of N(s) (or new under the alternative) and *f* is the final state of N(t) (or new). Overlap maps final states of N(s) to start state of N(t).

Piecing Together NFAs – continued(3)

3.(c) If s is a regular expressions, N(s) its NFA, s* (Kleene star) has NFA:



where : *i* is new start state and *f* is new final state

∈-move *i* to *f* (to accept null string)

∈-moves *i* to old start, old final(s) to *f*

∈-move old final to old start (WHY?)

Properties of Construction

Let r be a regular expression, with NFA N(r), then

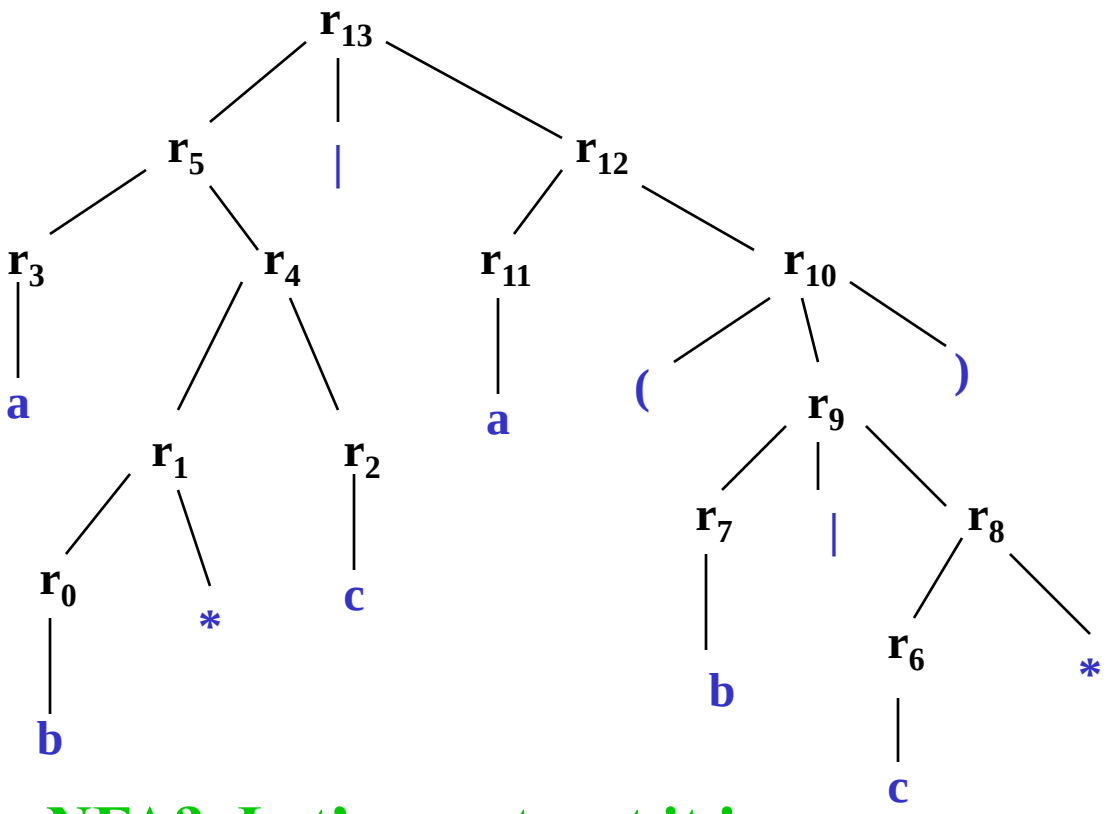
1. **N(r) has #of states $\leq$ 2*(#symbols + #operators) of r**
2. **N(r) has exactly one start and one accepting state**
3. **Each state of N(r) has at most one outgoing edge $a \in \Sigma$ or at most two outgoing $\in$'s**
4. **BE CAREFUL to assign unique names to all states !**

Detailed Example

See example in textbook for (a | b)*abb

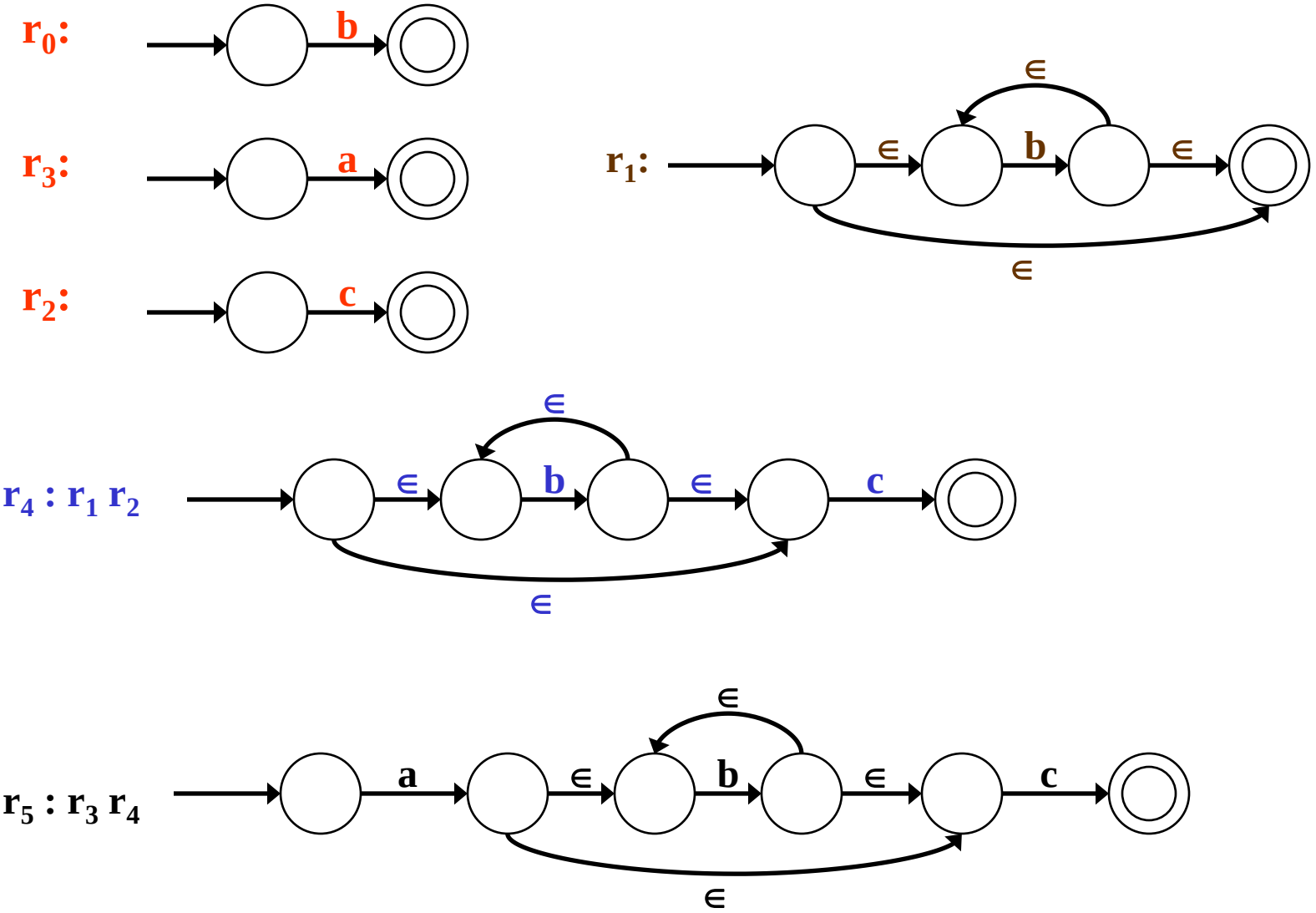
2nd Example - (ab*c) | (a(b|c*))

Parse Tree for this regular expression:

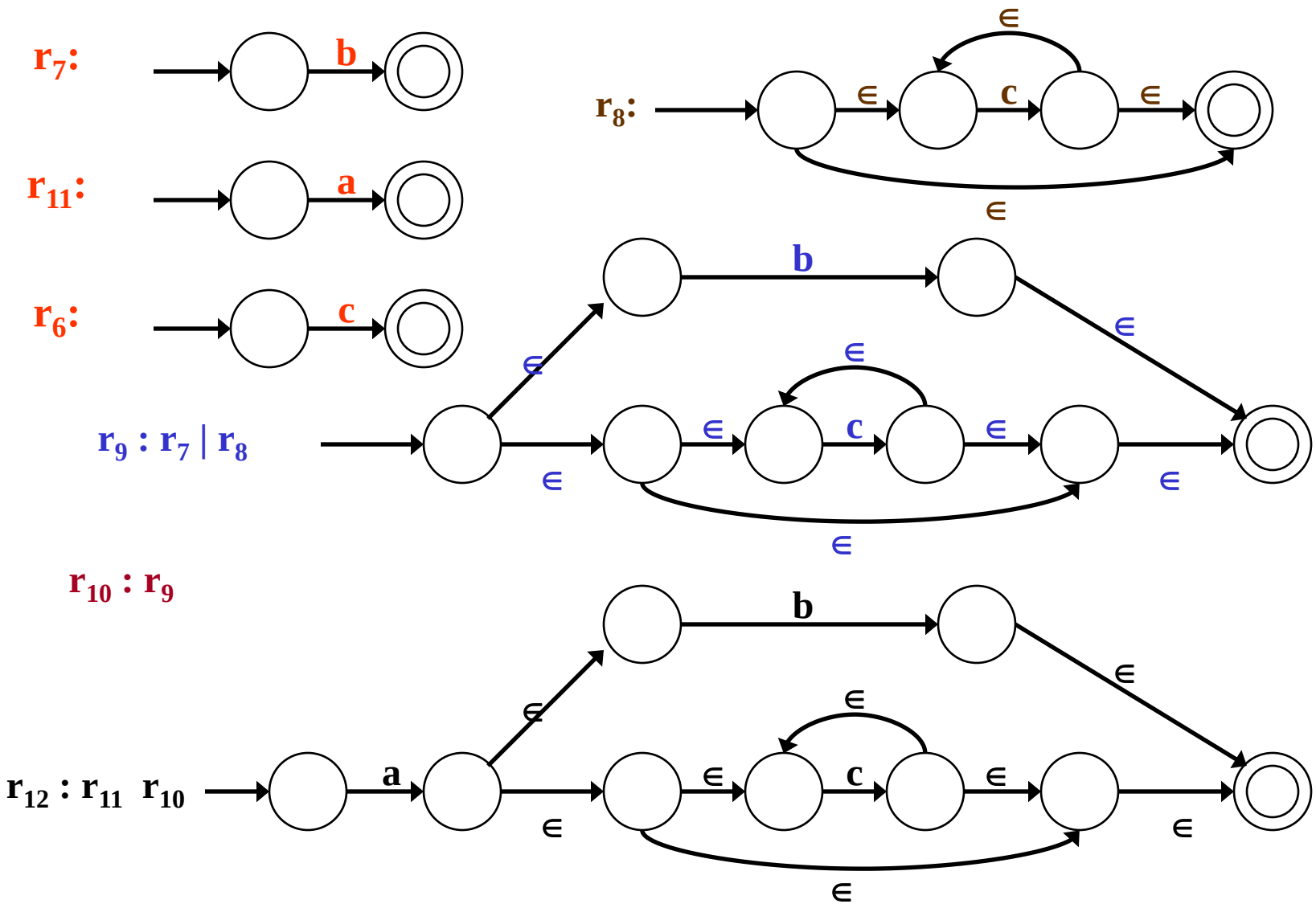


What is the NFA? Let's construct it !

Detailed Example – Construction(1)

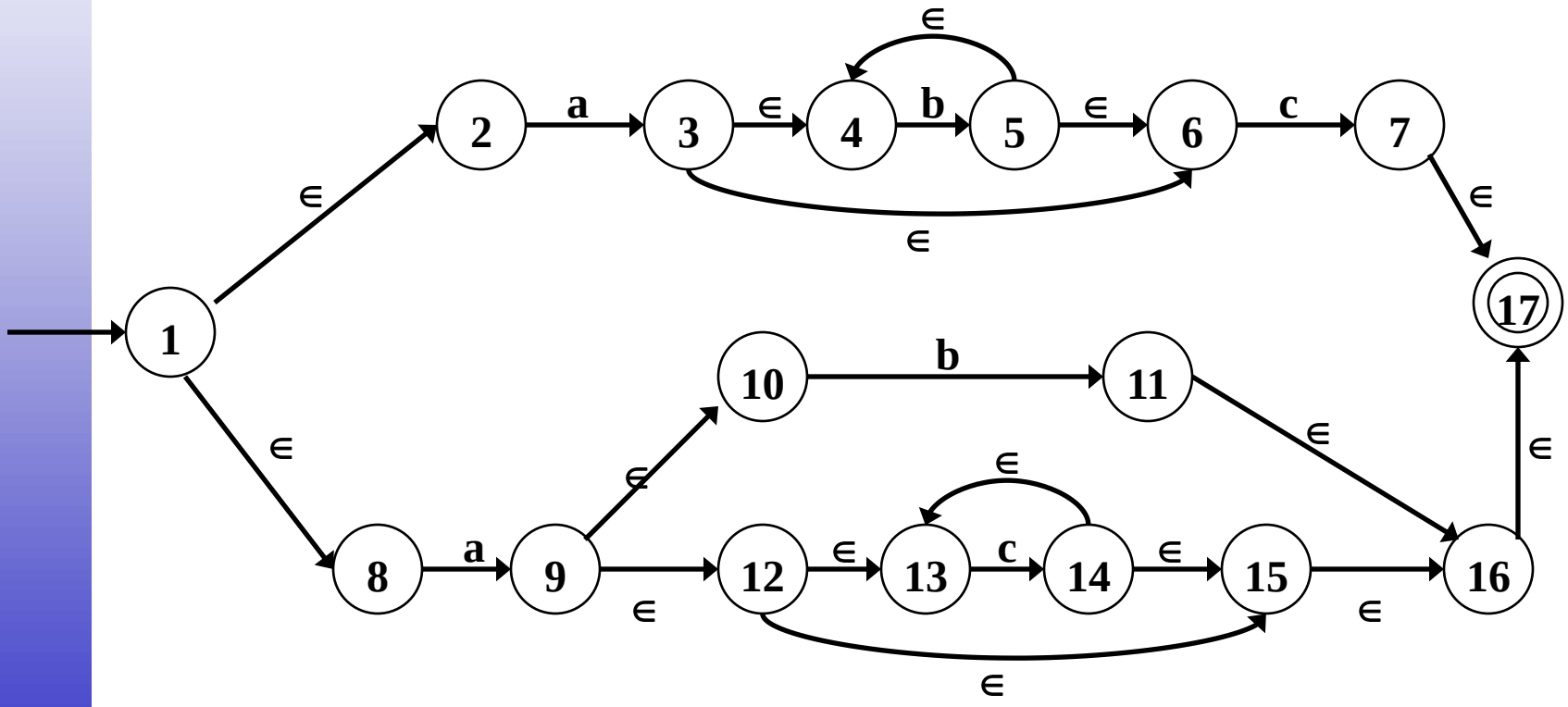


Detailed Example – Construction(2)



Detailed Example – Final Step

$$r_{13} : r_5 \mid r_{12}$$



Direct Simulation of an NFA

```
s ← s0
c ← nextchar;
while c ≠ eof do
   s ← move(s,c);
   c ← nextchar;
end;
if s is in F then return "yes"
             else return "no"
```

DFA simulation

```
S ← ∈-closure({s0})
c ← nextchar;
while c ≠ eof do
   S ← ∈-closure(move(S,c));
   c ← nextchar;
end;
if S∩F≠∅ then return "yes"
         else return "no"
```

NFA simulation

Final Notes : R.E. to NFA Construction

✓So, an NFA may be simulated by algorithm, when NFA is constructed using Previous techniques

✓Algorithm run time is proportional to |N| * |x| where |N| is the number of states and |x| is the length of input

✓Alternatively, we can construct DFA from NFA and use the resulting Dtran to recognize input:

Assignment

No marks will be awarded (Part of the Syllabus)

	space required	time to simulate
NFA	O(|r|)	O(|r|*|x|)
DFA	$O(2^{|r|})$	O(|x|)

where |r| is the length of the regular expression.

Pulling Together Concepts

- **Designing Lexical Analyzer Generator**

  Reg. Expr. → NFA construction

  NFA → DFA conversion

  DFA simulation for lexical analyzer

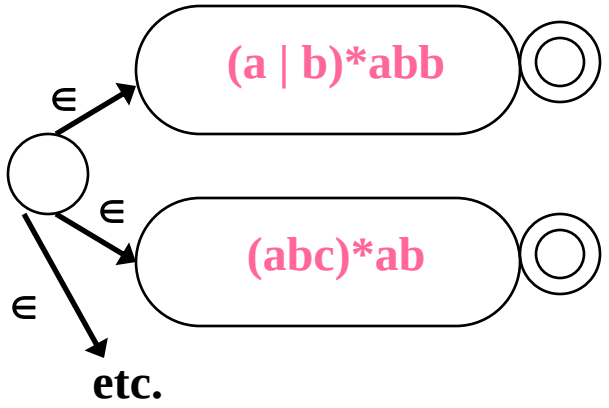
- **Recall Lex Structure**

  Pattern Action

  Pattern Action

  … …

e.g.



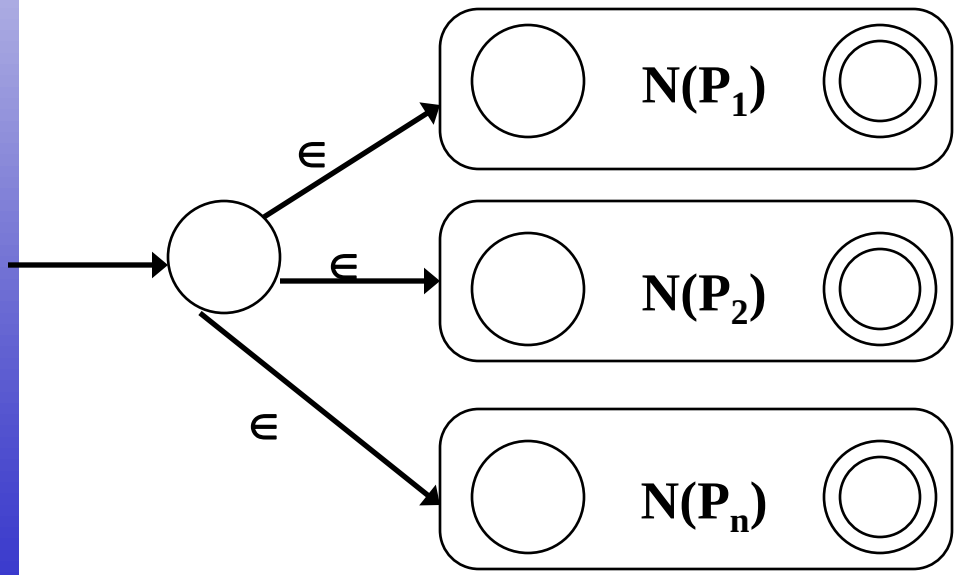
Recognizer!

- Each pattern recognizes lexemes

- Each pattern described by regular expression

Lex Specification → Lexical Analyzer

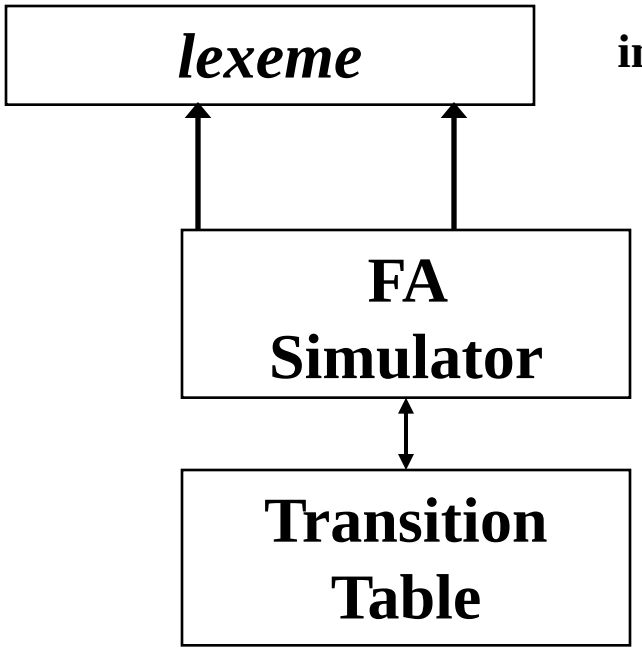
- **Let $P_1, P_2, \ldots, P_n$ be Lex patterns**
  (regular expressions for valid tokens in prog. lang.)
- **Construct $N(P_1), N(P_2), \ldots N(P_n)$**
- **Note: accepting state of $N(P_i)$ will be marked by P_i**
- **Construct NFA:**



- **Lex applies conversion algorithm to construct DFA that is equivalent!**

Pictorially

(a) Lex Compiler



(b) Schematic lexical analyzer

Pattern Matching Based on NFA

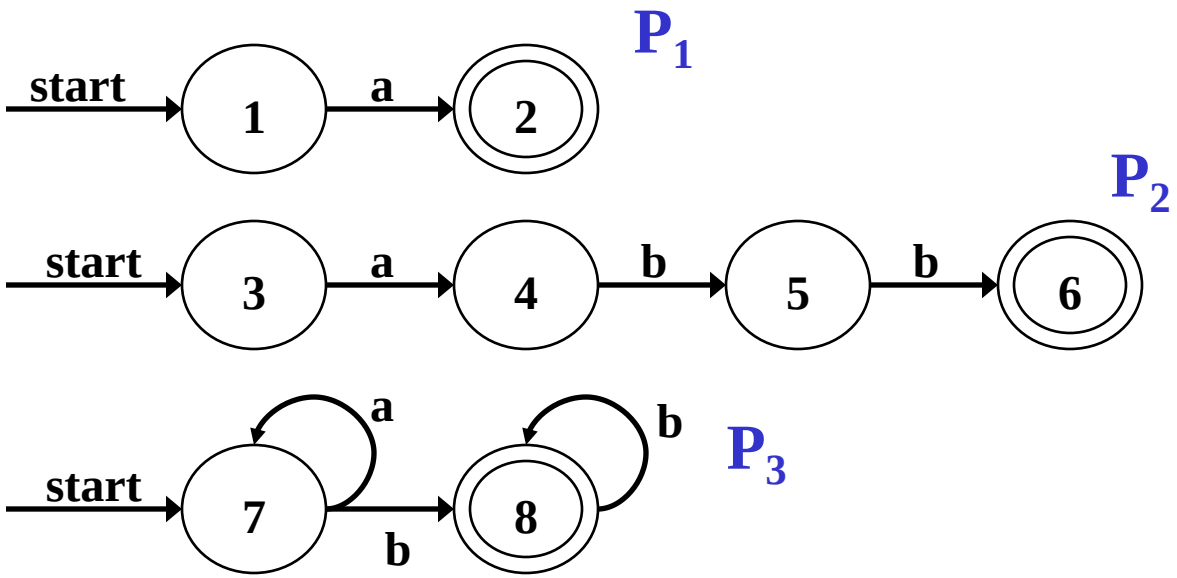
P_1 : a {action}

P_2 : abb {action}

P_3 : a*b^+ {action}

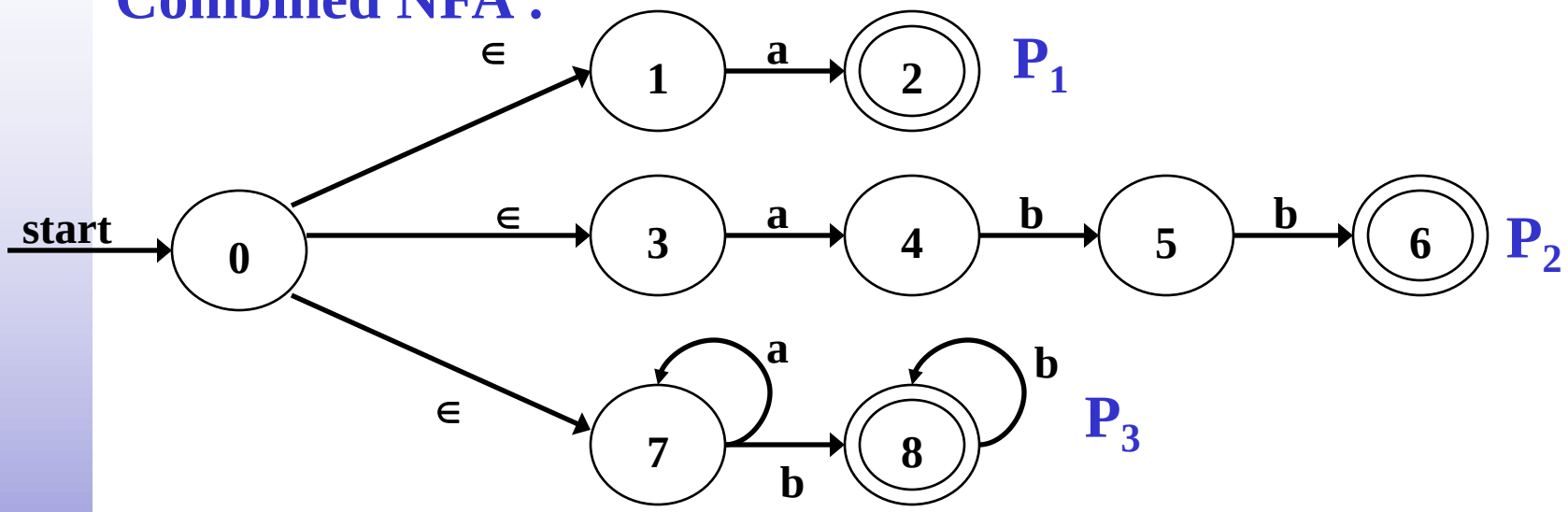
3 patterns

NFA's :



Pattern Matching Based on NFA continued (2)

Combined NFA :



Examples

	a	a	b	a	
	{0,1,3,7}	{2,4,7}	{7}	{8}	death
pattern matched:	-	P_1	-	P_3	-

	a	b	b	
	{0,1,3,7}	{2,4,7}	{5,8}	{6,8}
pattern matched:	-	P_1	P_3	P_2,P_3 ⇐ break tie in favor of P_2

DFA for Lexical Analyzers

Alternatively Construct DFA:

keep track of correspondence between patterns and new accepting states

	Input Symbol			
STATE	**a**	**b**	**Pattern**	
{0,1,3,7}	{2,4,7}	{8}	none	
{2,4,7}	{7}	{5,8}	P_1	
{8}	-	{8}	P_3	
{7}	{7}	{8}	none	
{5,8}	-	{6,8}	P_3	
{6,8}	-	{8}	P_2	

break tie in favor of P_2

Example

Input: aaba

Input: aba

	Input Symbol		
STATE	a	b	Pattern
{0,1,3,7}	{2,4,7}	{8}	none
{2,4,7}	{7}	{5,8}	P_1
{8}	-	{8}	P_3
{7}	{7}	{8}	none
{5,8}	-	{6,8}	P_3
{6,8}	-	{8}	P_2

Optimization of DFA based Pattern Matching

Our Target:

1. **Construct DFA directly from Regular Expression**
2. **Minimizes no of states of DFA**
3. **Produce fast but more compact representations for transition table of DFA than a straightforward two dimensional table**

Important States of an NFA

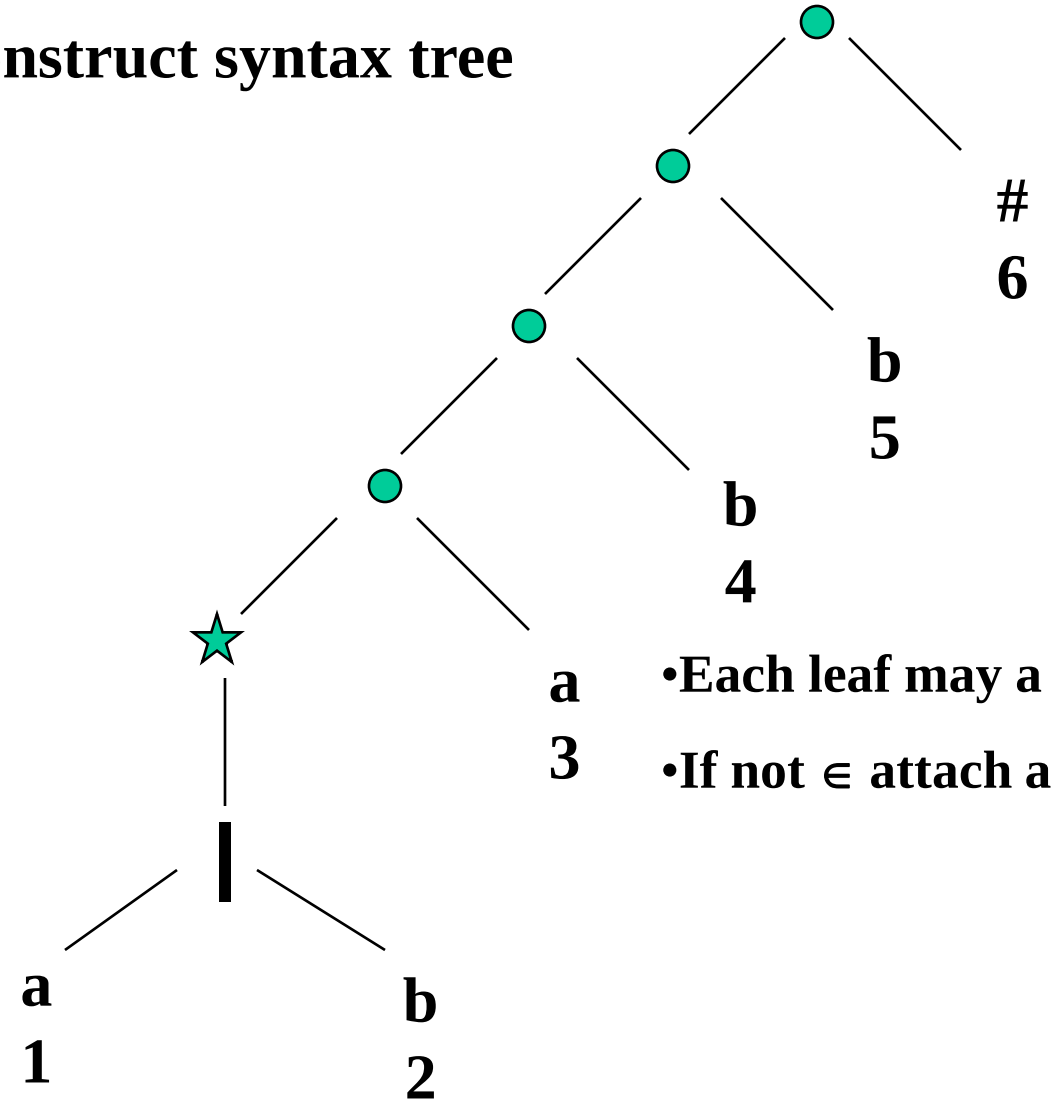
Important States of NFA:

if it has a non-$\in$ out-transition

- The subset construction algorithm uses only the important states in a subset T when it determines $\in$-closure(move(T,a))
- The resulting NFA has exactly one accepting state, but the accepting state is not important because it has no transition leaving it.
- We give the accepting state a transition on #, making it important state of NFA.
- By using augmented regular expression (r)#, we can forget about accepting states as the subset construction proceeds.
- When the construction is complete, any DFA state with a transition on # must be an accepting state.

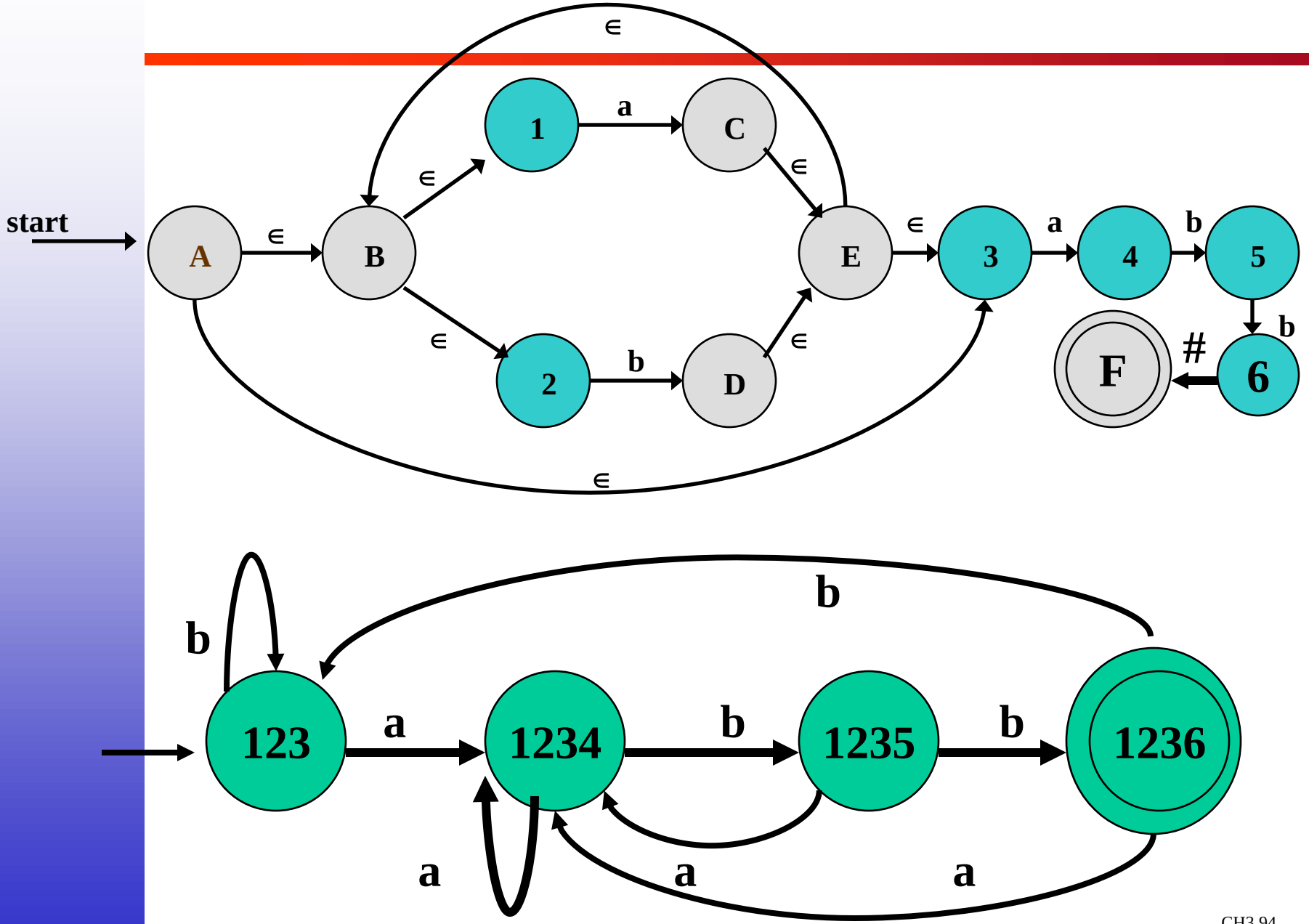
Example: (a | b)*abb

Construct syntax tree



- **Each leaf may a symbol / ∈**
- **If not ∈ attach a unique integer**

NFA Vs DFA

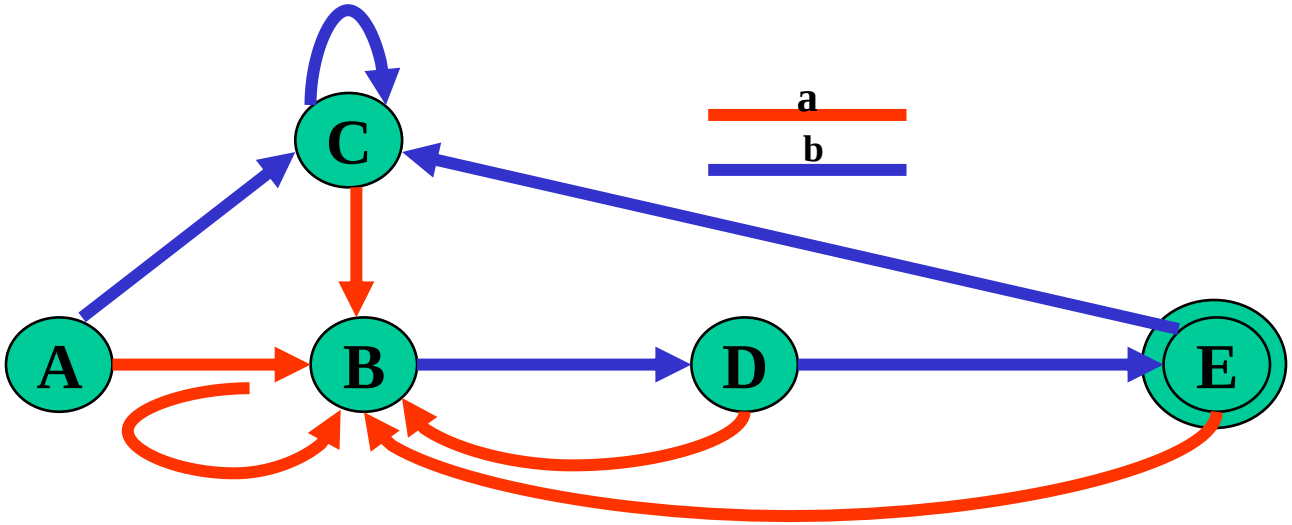


Minimizing the Number of States of DFA

1. Construct initial partition Π of S with two groups: accepting/ non-accepting.
2. (Construct Π_{new})For each group *G* of Π **do begin**
    1. Partition *G* into subgroups such that two states s,t of *G* are in the same subgroup iff for all symbols a states s,t have transitions on a to states of the same group of Π.
    2. Replace *G* in Π_{new} by the set of all these subgroups.
3. Compare Π_{new} and Π. If equal, Π_{final}:= Π then proceed to 4,

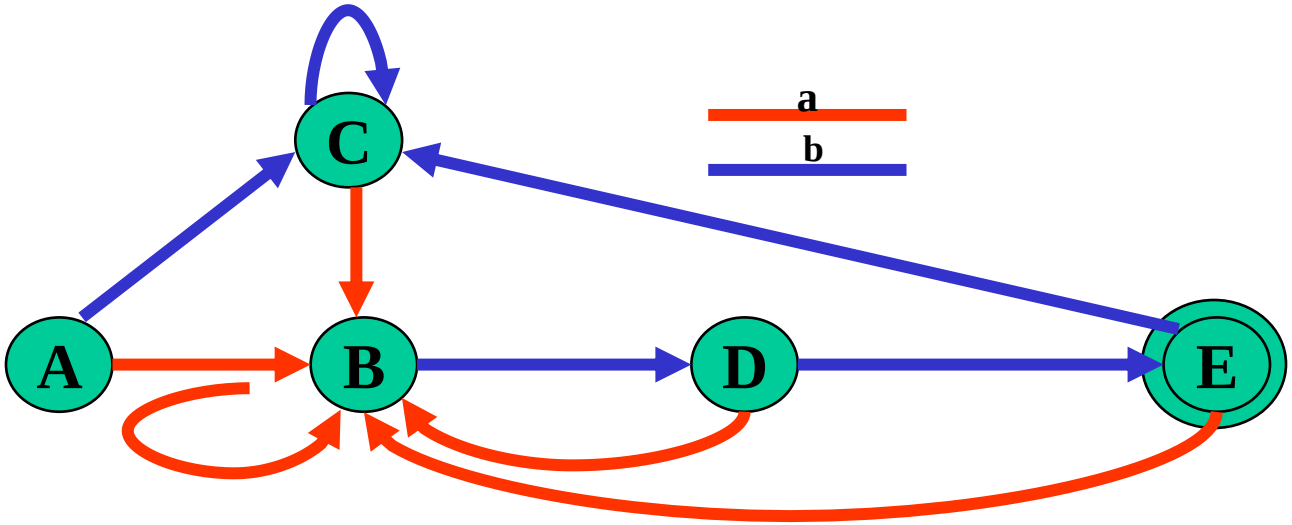
   else set Π := Π_{new} and goto 2.
4. Aggregate states belonging in the groups of Π_{final}
5. If M' has a dead state, that is, a state d that is not a accepting and that has transitions to itself on all input symbols, then remove d from M'. Also remove any states not reachable from the start state. Any transitions to d from other states become undefined.

Example of minimization of DFA States



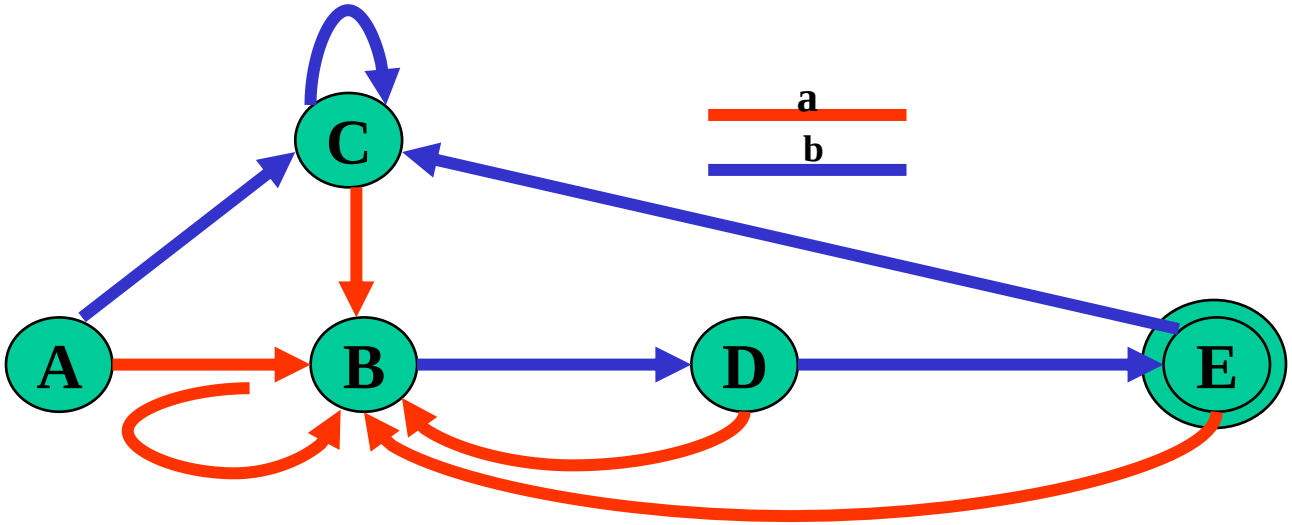
G1	G2
E	ABCD
No Change	a→ G2 b→ G2 G1 (ABC) (D)

Example of minimization of DFA States



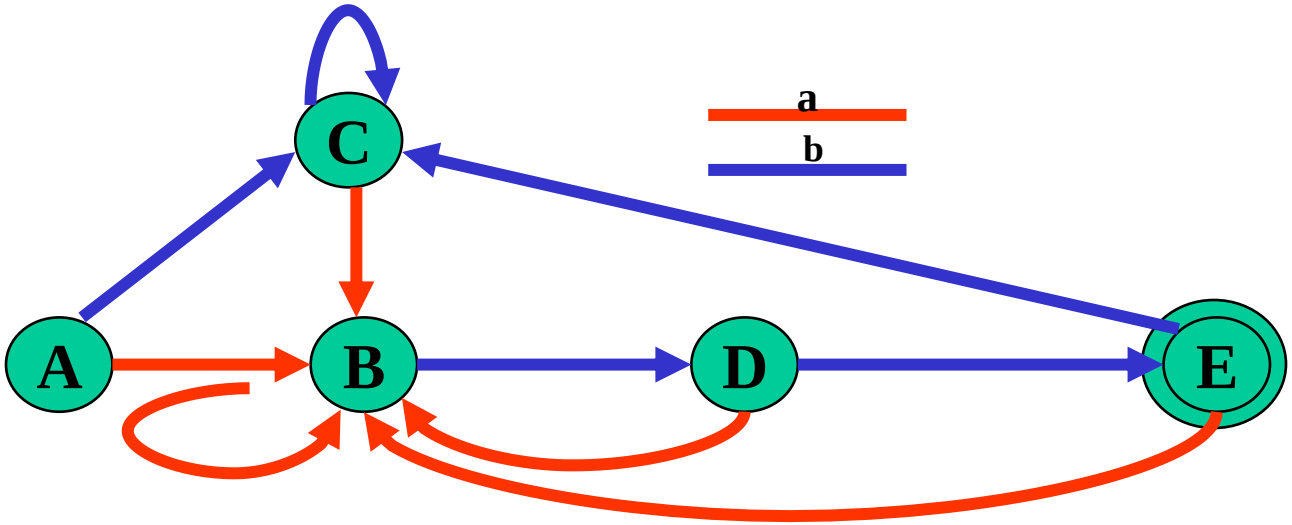
G1	G2	G3
ABC	D	E
a→ G1 b→ G1 G2 (AC) (B)	No Change	No change

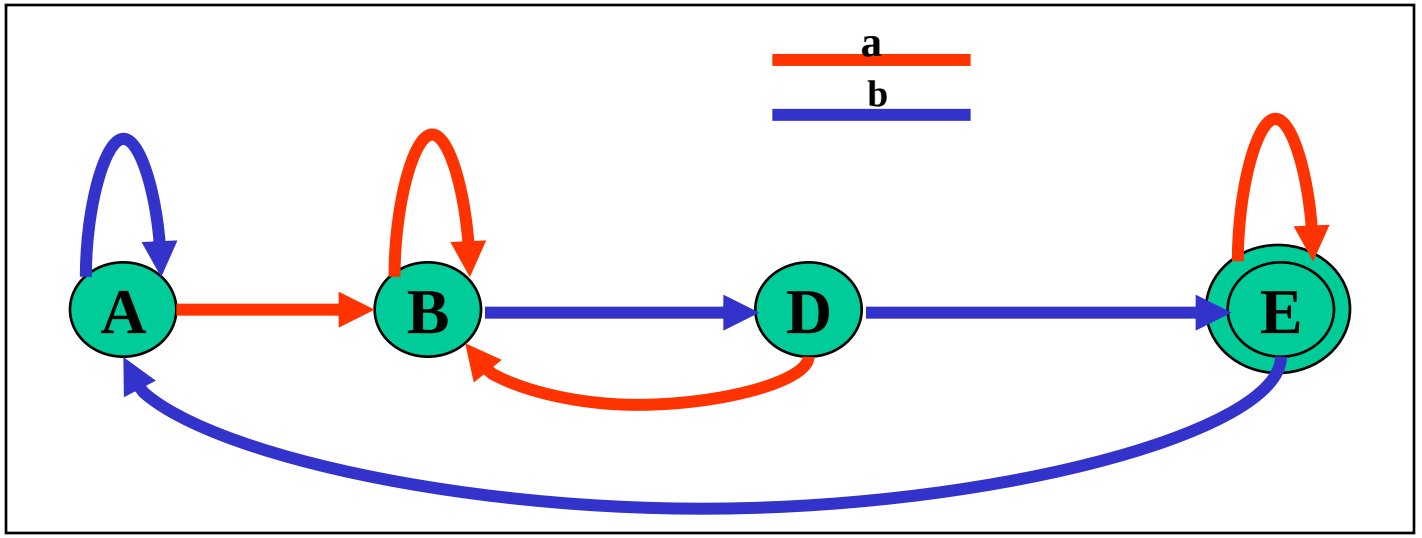
Example of minimization of DFA States



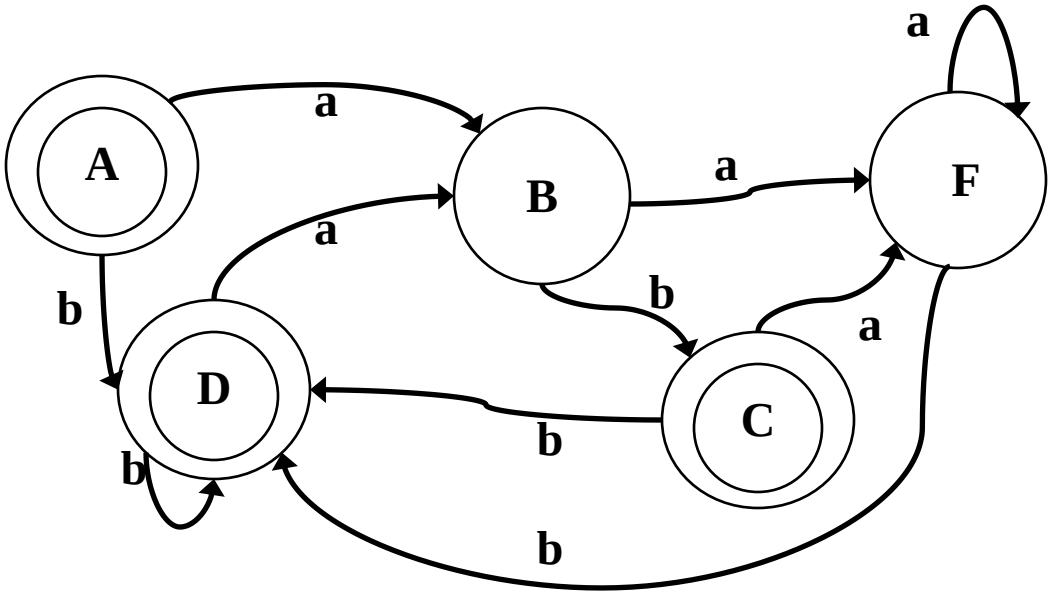
G1	G2	G3	G4
AC	B	D	E
No Change	No Change	No Change	No change

Example of minimization of DFA States

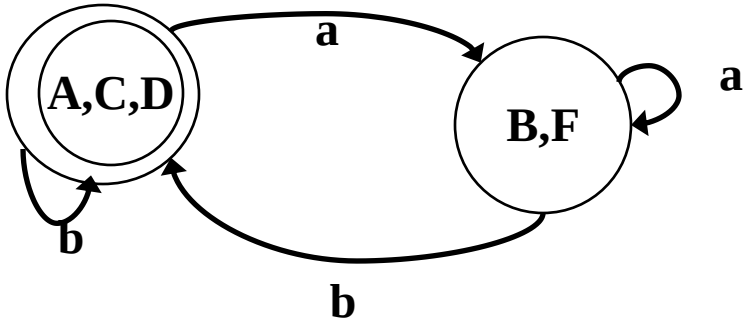




example



Minimized DFA:



Concluding Remarks

Focused on Lexical Analysis Process, Including

- **Regular Expressions**
- **Finite Automaton**
- **Conversion**
- **Lex**
- **Interplay among all these various aspects of lexical analysis**

Looking Ahead:

The next step in the compilation process is Parsing:

- **Top-down vs. Bottom-up**

-- **Relationship to Language Theory**

The End