

Lexical Analysis

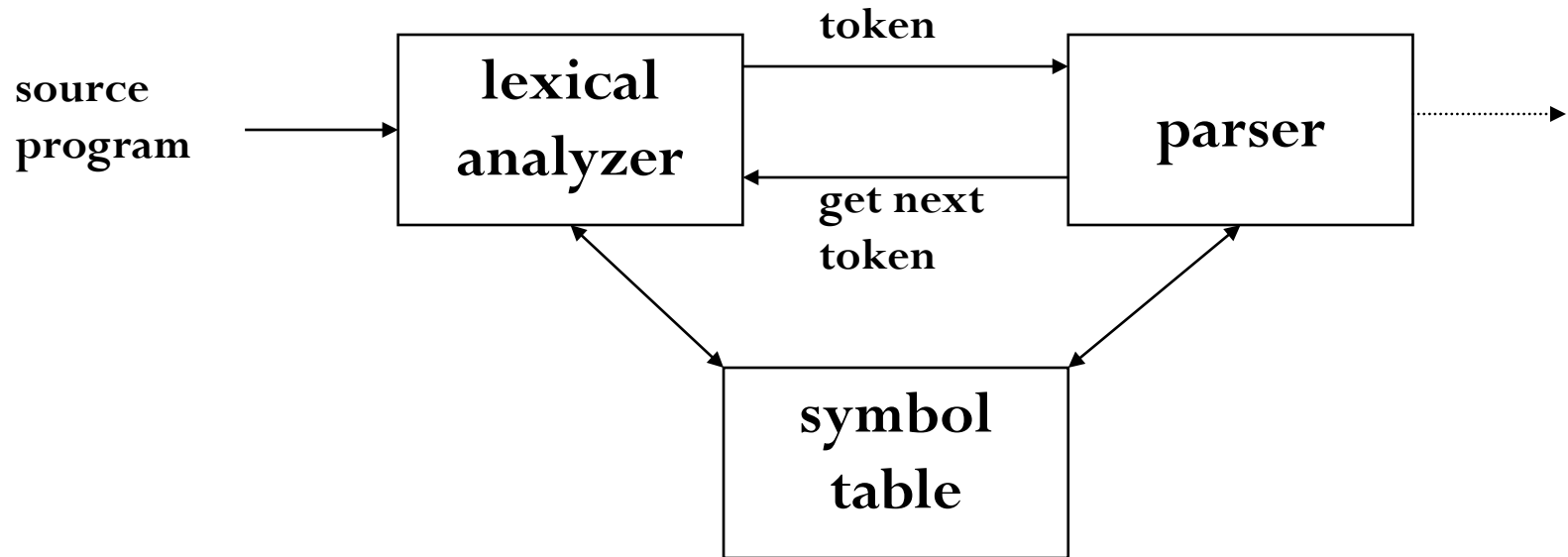
CSE 4102

Lecture 02

Lexical Analysis

- Basic Concepts & Regular Expressions
 - What does a Lexical Analyzer do?
 - How does it Work?
 - Formalizing Token Definition & Recognition
- 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

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 than 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 pair consisting of a token name and an optional attribute value.
- A particular keyword, or a sequence of input characters denoting identifier.
- Examples: ID, NUM, IF, EQUALS, ...

- **PATTERN**

- A description of a form that the lexemes of a token may take.
- For keywords, the pattern is just a sequence of characters that form keywords.

- **LEXEME**

- Actual sequence of characters that matches pattern and is classified by a token
- Example:

if x == -12.30 then ...

Introducing Basic Terminology

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

Actual values are critical. Info is :

1. Stored in symbol table
2. Returned to parser

Classifies
Pattern

Attributes for Tokens

- When more than one lexeme can match a pattern, a lexical analyzer must provide the compiler **additional information** about that lexeme matched.
- Information about identifiers, its lexeme, type and location at which it was first found is kept in **symbol table**.
- The appropriate attribute value for an identifier is **a pointer to the symbol table entry for that identifier**.

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 E>

<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

- Most error tend to be “typos”
- Its **hard** for lexical analyzer without the aid of other components, that there is a **source-code error**.
 - If the statement **fi** is encountered for the first time in a C program it can not tell whether **fi** is **misspelling** of **if** statement or a **undeclared literal**.
 - Probably the parser in this case will be able to handle this.
- Error Handling is very **localized**, with Respect to Input Source
- For example: `whil (x = 0) do`
generates **no** lexical errors in **PASCAL**

Handling Lexical Errors

- EOF within a String / missing “
- Invalid ASCII character in file
- String/ID exceeds maximum length
- Numerical overflow
- etc ...

Handling Lexical Errors

- **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 – creating syntax error
- **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 look-ahead 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

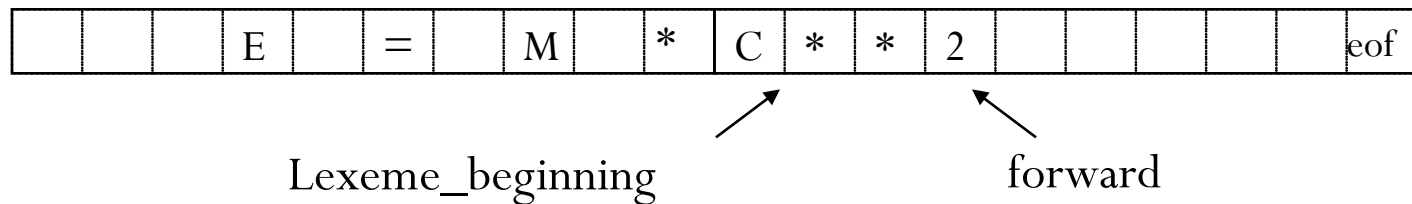
N = Number of characters on one disk block

One system command read N characters

Fewer than N character => **eof**

Buffer Pairs (2)

- **Two pointers** lexeme beginning and forward to the input buffer are maintained.
- The string of characters between the pointers is the current lexeme.
- Initially both pointers point to first character of the next lexeme to be found.
Forward pointer scans ahead until a match for a pattern is found
- Once the next lexeme is determined, the forward pointer is set to the character at its right end.
- After the lexeme is processed both pointers are set to the character **immediately past the lexeme.**



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 beginning 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 look-ahead is limited.
2. Limited look-ahead makes it impossible to recognize tokens in situations where the distance, forward pointer must travel is more than the length of buffer.

Specification of Tokens

Regular expressions are an important notation for specifying lexeme patterns

An **alphabet** is a finite set of symbols.

- Typical example of symbols are letters, digits and punctuation etc.
- The set $\{0, 1\}$ is the binary alphabet.

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

- The length of string s is denoted as $|s|$
- Empty string is denoted by ϵ

Prefix: ban, banana, ϵ , etc are the prefixes of banana

Suffix: nana, banana, ϵ , etc are suffixes of banana

Kleene or closure of a language L , denoted by L^* .

- L^* : concatenation of L zero or more times
- L^0 : concatenation of L zero times
- L^+ : concatenation of L one or more times

Kleene closure

Let: $L = \{ a, bc \}$

L^* denotes “zero or more concatenations of” L

Example: $L^0 = \{ \varepsilon \}$

$L^1 = L = \{ a, bc \}$

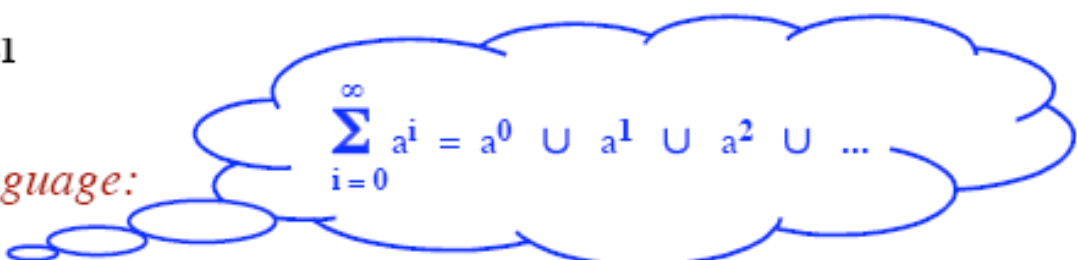
$L^2 = LL = \{ aa, abc, bca, bc bc \}$

$L^3 = LLL = \{ aaa, aabc, abca, abc bc, bcaa, bcabc, bcbca, bcb bc \}$

...etc...

$L^N = L^{N-1}L = LL^{N-1}$

The “Kleene Closure” of a language:


$$\sum_{i=0}^{\infty} a^i = a^0 \cup a^1 \cup a^2 \cup \dots$$

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots$$

Example:

$$L^* = \{ \underbrace{\varepsilon}_{L^0}, \underbrace{a, bc}_{L^1}, \underbrace{aa, abc, bca, bc bc}_{L^2}, \underbrace{aaa, aabc, abca, abc bc, \dots}_{L^3} \}$$

Example

Let: $L = \{ \mathbf{a}, \mathbf{b}, \mathbf{c}, \dots, \mathbf{z} \}$

$D = \{ \mathbf{0}, \mathbf{1}, \mathbf{2}, \dots, \mathbf{9} \}$

$D^+ =$ “The set of strings with one or more digits”

$L \cup D =$ “The set of all letters and digits (alphanumeric characters)”

$LD =$ “The set of strings consisting of a letter followed by a digit”

$L^* =$ “The set of all strings of letters, including ϵ , the empty string”

$(L \cup D)^* =$ “Sequences of zero or more letters and digits”

$L((L \cup D)^*) =$ “Set of strings that start with a letter, followed by zero or more letters and digits.”

Rules for specifying Regular Expressions

Regular expressions over alphabet Σ

1. ϵ is a regular expression that denotes $\{\epsilon\}$.
2. If \mathbf{a} is a symbol (i.e., if $\mathbf{a} \in \Sigma$), then \mathbf{a} is a regular expression that denotes $\{\mathbf{a}\}$.
3. Suppose r and s are regular expressions denoting the languages $L(r)$ and $L(s)$. Then
 - a) $(r) \mid (s)$ is a regular expression denoting $L(r) \cup L(s)$.
 - b) $(r)(s)$ is a regular expression denoting $L(r)L(s)$.
 - c) $(r)^*$ is a regular expression denoting $(L(r))^*$.
 - d) (r) is a regular expression denoting $L(r)$.

Regular Expressions Construction

abc Concatenation; Most characters stand for themselves

Meta Characters:

| Usual meanings

* Example: **(a|b)*c***

()

+ One or more, e.g., **ab+c**

? Optional, e.g., **ab?c**

[**x-y**] Character classes, e.g., [**a-z**] [**a-zA-Z0-9**]*

[**^x-y**] Anything but [**x-y**]

\b The usual escape sequences, e.g., \n

.

^ Beginning of line

\$ End of line

"..." To use the meta characters literally,

Example: PCAT comments: "**(*,***)**"

{...} Defined names, e.g., {**letter**}

/ Look-ahead

Example: **ab/cd**

(Matches **ab**, but only when followed by **cd**)

• **Precedence:**

- * has highest precedence.
- Concatenation as middle precedence.
- | has lowest precedence.
- Use parentheses to override these rules.

• **Examples:**

- **a b* = a (b*)**
 - If you want **(a b)*** you must use parentheses.
- **a | b c = a | (b c)**
 - If you want **(a | b) c** you must use parentheses.

Example

- Let $\Sigma = \{a, b\}$
 - The regular expression $a \mid b$ denotes the set $\{a, b\}$
 - The regular expression $(a \mid b)(a \mid b)$ denotes $\{aa, ab, ba, bb\}$
 - The regular expression a^* denotes the set of all strings of zero or more a's. i.e., $\{\epsilon, a, aa, aaa, \dots\}$
 - The regular expression $(a \mid b)^*$ denotes the set containing zero or more instances of an a or b.
 - The regular expression $a \mid a^*b$ denotes the set containing the string a and all strings consisting of zero or more a's followed by one b.

Regular Definition

- If Σ is an alphabet of basic symbols then a regular definition is a sequence of the following form:

$$\begin{array}{l} d_1 \rightarrow r_1 \\ d_2 \rightarrow r_2 \\ \dots\dots\dots \\ d_n \rightarrow r_n \end{array}$$

where

- Each d_i is a new symbol such that $d_i \notin \Sigma$ and $d_i \neq d_j$ where $j < i$
- Each r_i is a regular expression over $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$

Regular Definition

```
Letter    = a | b | c | ... | z
Digit    = 0 | 1 | 2 | ... | 9
ID       = Letter ( Letter | Digit )*
```

Names (e.g., Letter) are underlined to distinguish from a sequence of symbols.

```
Letter ( Letter | Digit )*
= { "Letter", "LetterLetter", "LetterDigit", ... }
```

Each definition may only use names *previously* defined.

⇒ No recursion

Regular Sets = no recursion

CFG = recursion

Addition Notation / Shorthand

One-or-more: $^+$

$$X^+ = X(X^*)$$

$$\underline{\text{Digit}}^+ = \underline{\text{Digit}} \underline{\text{Digit}}^* = \underline{\text{Digits}}$$

Optional (zero-or-one): $?$

$$X? = (X \mid \epsilon)$$

$$\underline{\text{Num}} = \underline{\text{Digit}}^+ (\cdot \underline{\text{Digit}}^+) ?$$

Character Classes: $[\text{FirstChar} - \text{LastChar}]$

Assumption: The underlying alphabet is known ...and is ordered.

$$\underline{\text{Digit}} = [0-9]$$

$$\underline{\text{Letter}} = [a-zA-Z] = [A-Za-z]$$

Variations:

$$\text{Zero-or-more: } ab^*c = a\{b\}c = a\{b\}^*c$$

$$\text{One-or-more: } ab^+c = a\{b\}^+c$$

$$\text{Optional: } ab?c = a[b]c$$

What does
 $ab \dots bc$
mean?

Unsigned Number

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

digit $\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

digits $\rightarrow \text{digit digit}^*$

optional_fraction $\rightarrow . \text{digits} \mid \epsilon$

optional_exponent $\rightarrow (E (+ \mid - \mid \epsilon) \text{ digits}) \mid \epsilon$

num $\rightarrow \text{digits optional_fraction optional_exponent}$

Shorthand

digit $\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

digits $\rightarrow \text{digit}^+$

optional_fraction $\rightarrow (. \text{digits}) ?$

optional_exponent $\rightarrow (E (+ \mid -) ? \text{digits}) ?$

num $\rightarrow \text{digits optional_fraction optional_exponent}$

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

Given Tokens, What are Patterns ?

if → if

then → then

else → else

relop → < | <= | > | >= | = | <>

id → letter (letter | digit)*

num → digit ⁺ (. digit ⁺) ? (E (+ | -) ? digit ⁺) ?

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?

blank \rightarrow blank

tab \rightarrow tab

newline \rightarrow newline

delim \rightarrow blank | tab | newline

ws \rightarrow delim⁺

In these cases no token is returned to parser

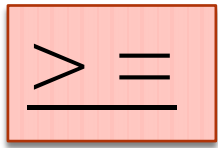
Overall

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

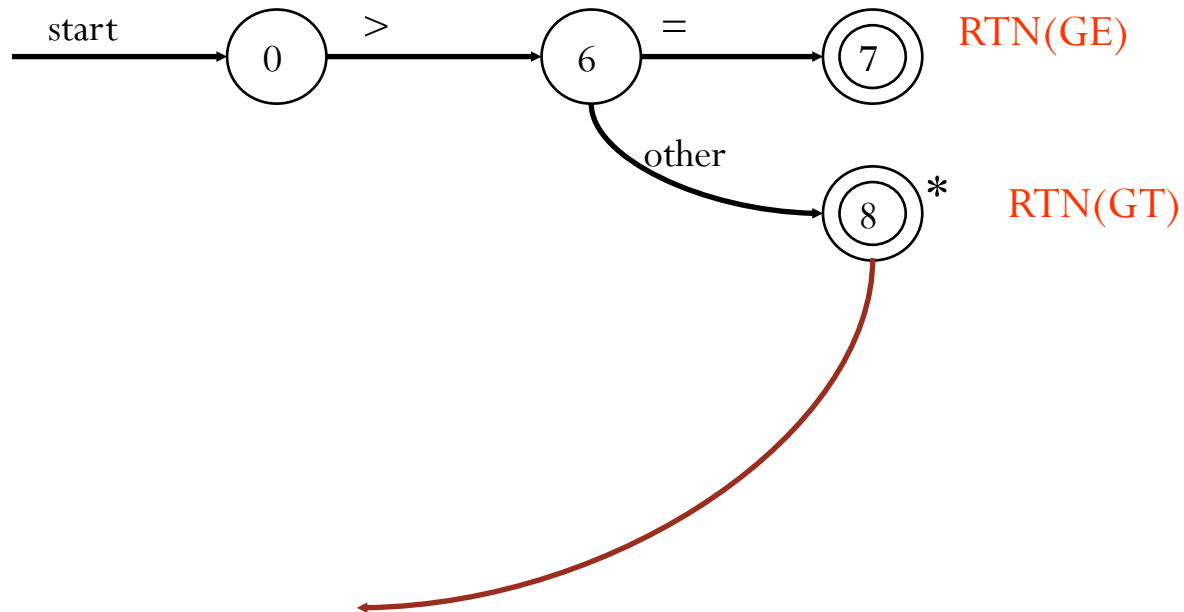
Constructing Transition Diagrams for Tokens

- **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**)
 - **Edges**: arrows connecting the states
- Each TD is **Deterministic (assume)** - No need to choose between 2 different actions !

Example TDs

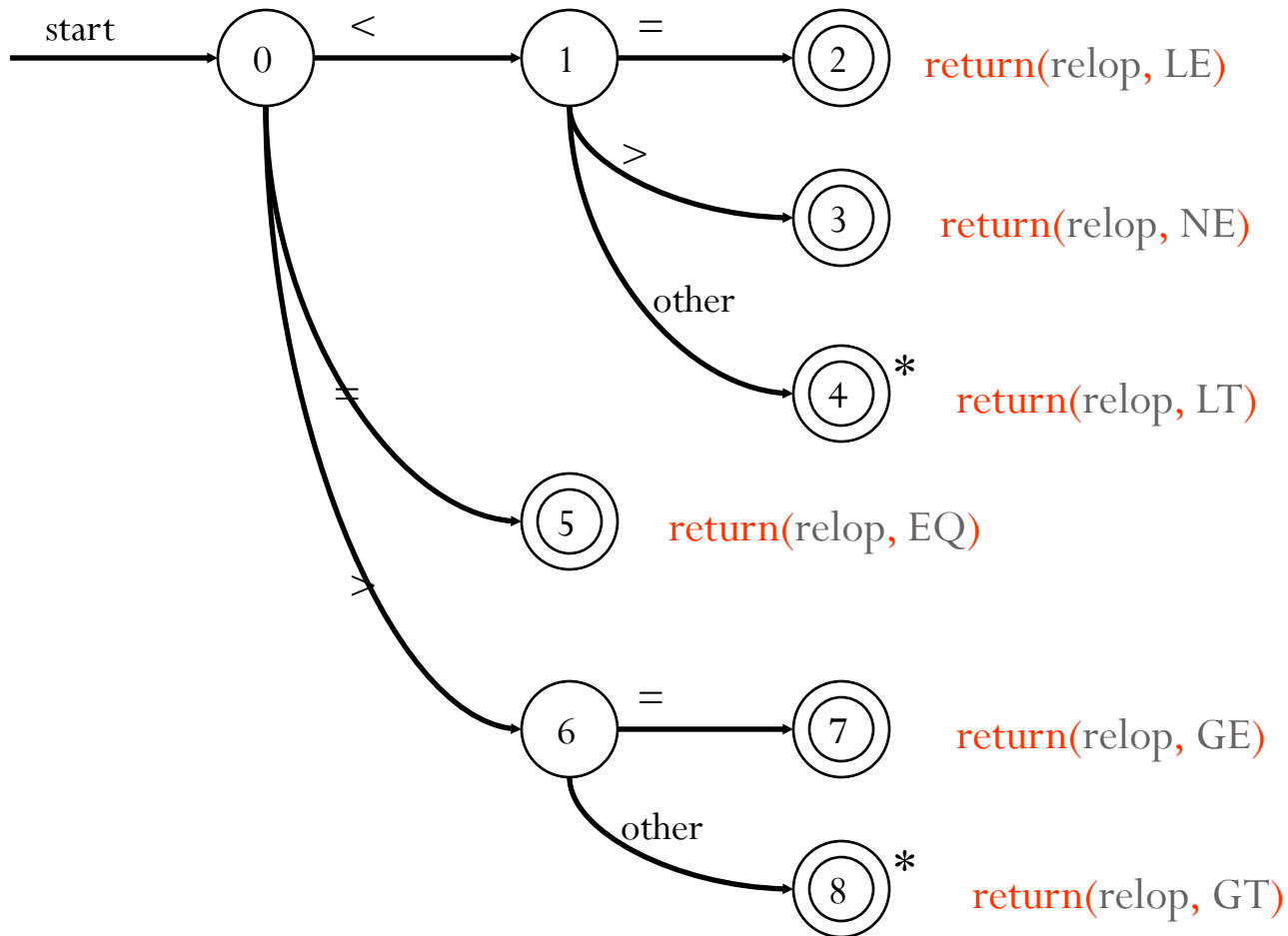


⋮



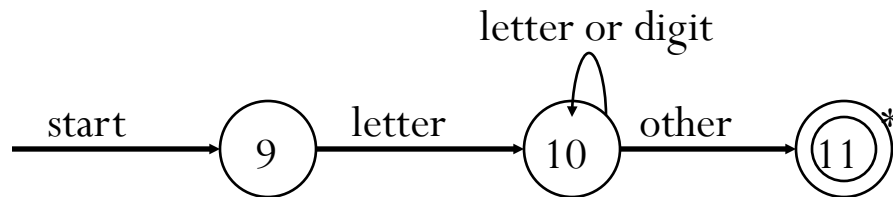
We've accepted ">" and have read one extra char that must be unread.

Example : All RELOPs



Example TDs : id and delim

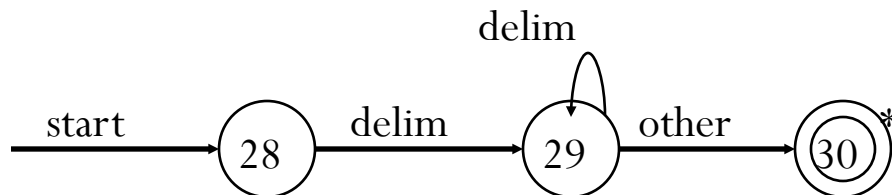
id :



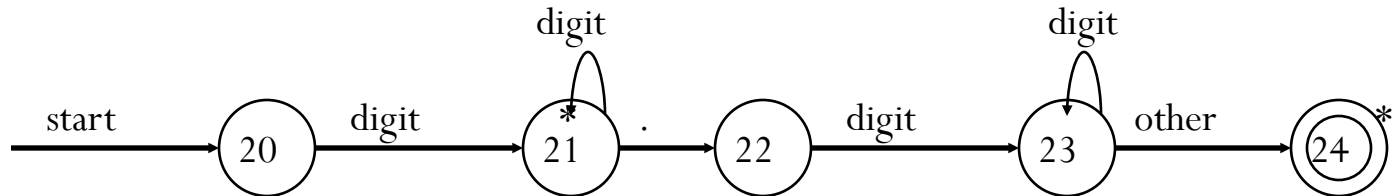
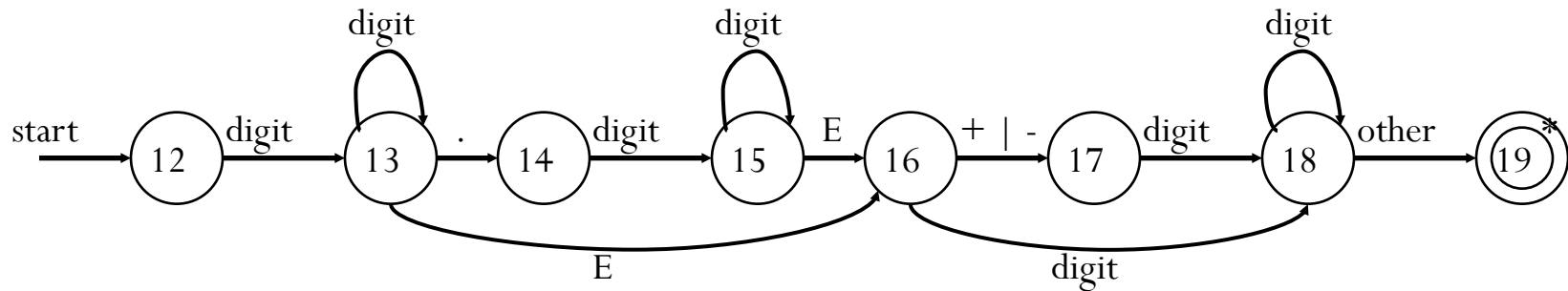
return(get_token(), install_id())

Either returns ptr or "0" if reserved

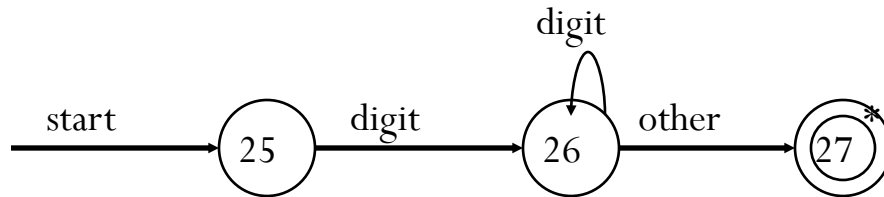
delim :



Example TDs : Unsigned #s



return(num, install_num())



Questions: Is ordering important for unsigned #s ?

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

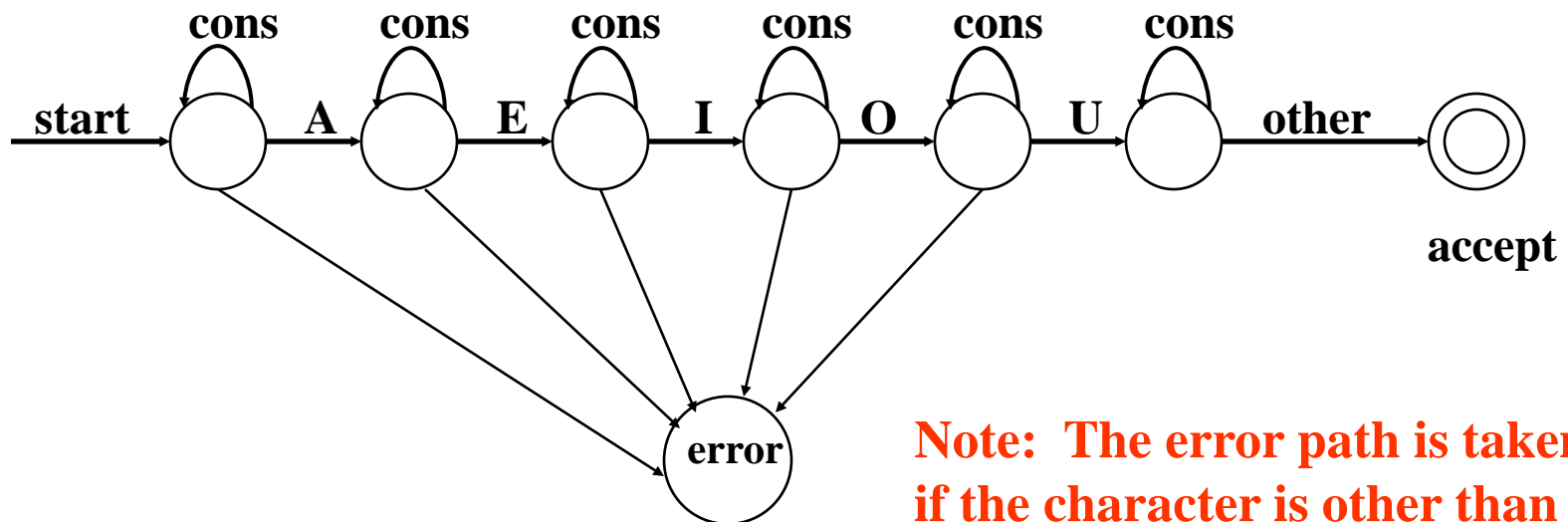
QUESTION :

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

Answer

cons \rightarrow **B | C | D | F | G | H | J | ... | N | P | ... | T | V | .. | Z**

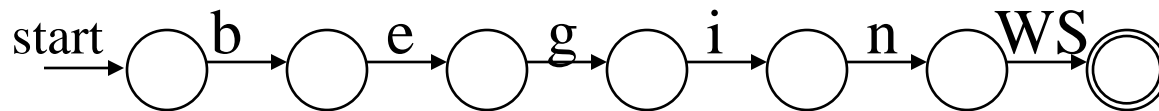
string \rightarrow **cons* A cons* E cons* I cons* O cons* U cons***



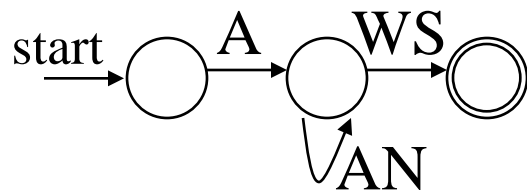
Note: The error path is taken if the character is other than a cons or the vowel in the lex order.

Capturing Multiple Tokens

Capturing keyword “begin”



Capturing variable names



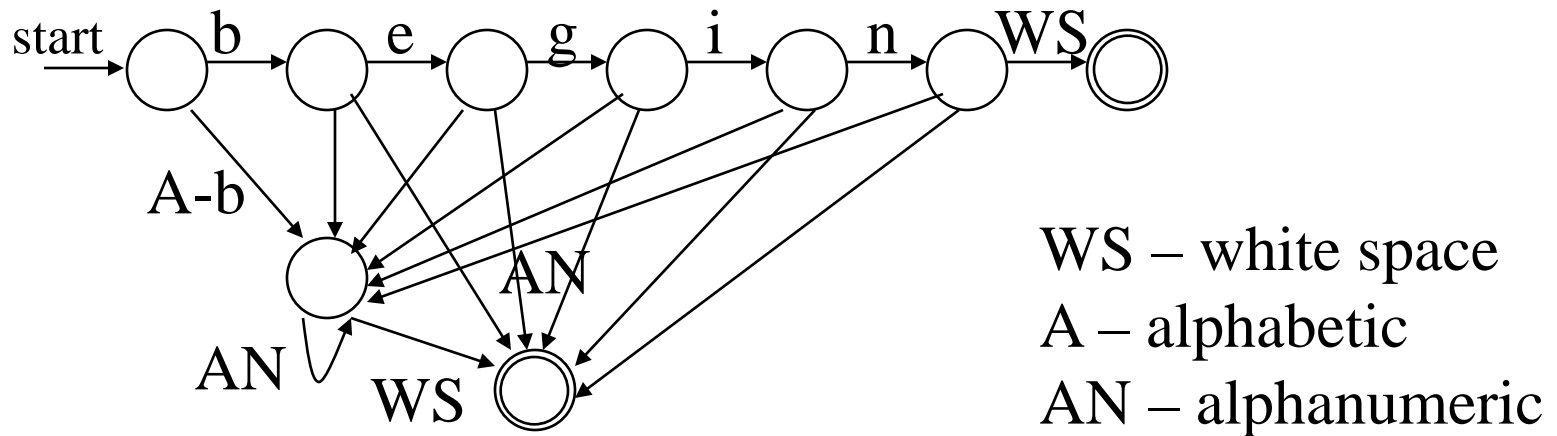
WS – white space

A – alphabetic

AN – alphanumeric

What if both need to happen at the same time?

Capturing Multiple Tokens



Machine is much more complicated – just for these two tokens!

Finite State Automata (FSAs)

- “Finite State Machines”, “Finite Automata”, “FA”
- 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.
 - The regular expression is compiled into a recognizer by constructing a **generalized transition diagram** called a **finite automaton**.
- Each state is labeled with a state name
- Directed edges, labeled with symbols
- Two types
 - Deterministic (DFA)
 - Non-deterministic (NFA)

Nondeterministic Finite Automata

A **nondeterministic finite automaton** (NFA) is a mathematical model that consists of

1. A set of **states** S
2. A set of **input symbols** Σ
3. A **transition function** that maps state/symbol pairs to a set of states
4. A special state s_0 called the **start state**
5. A set of states F (subset of S) of **final states**

INPUT: string

OUTPUT: yes or no

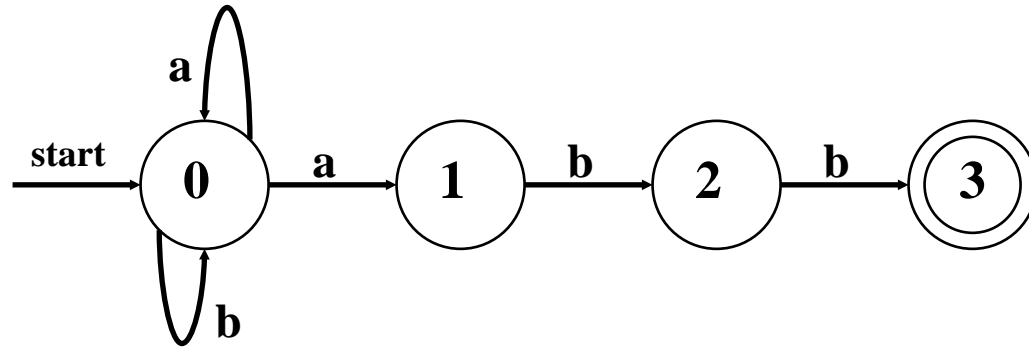
Example – NFA : $(a | b)^*abb$

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

$s_0 = 0$

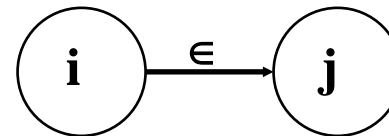
$F = \{ 3 \}$

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



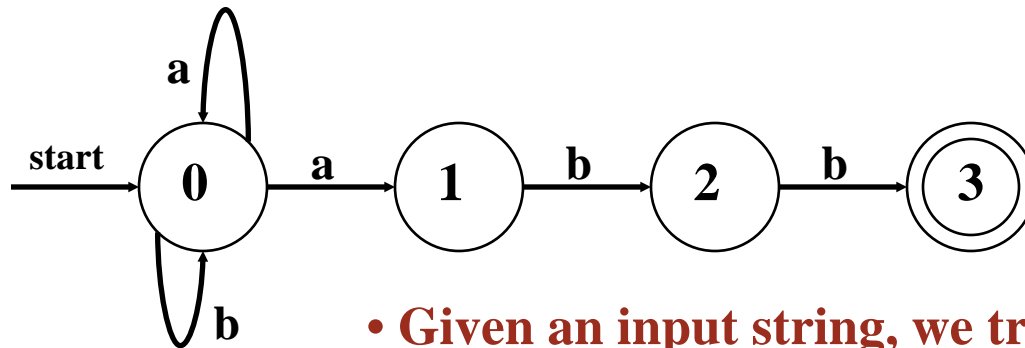
	input	
	a	b
s		
t	0	{ 0, 1 }
a	1	--
t	2	{ 2 }
e		--
		{ 3 }

ϵ (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 !

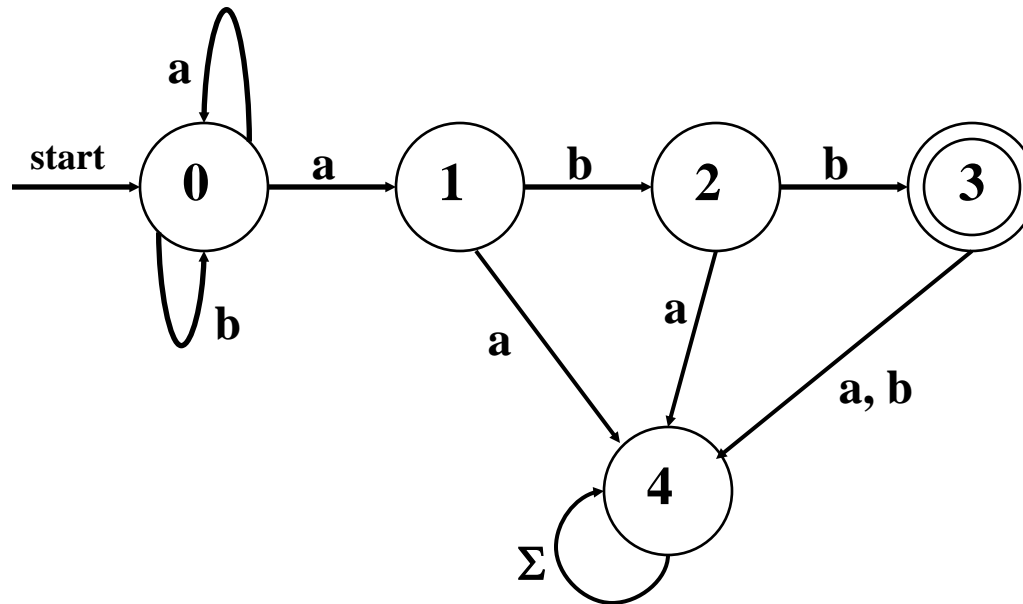
-OR-

$move(0, a) = 0$
 $move(0, b) = 0$
 $move(0, a) = 1$
 $move(1, b) = 2$
 $move(2, b) = 3$

ACCEPT !

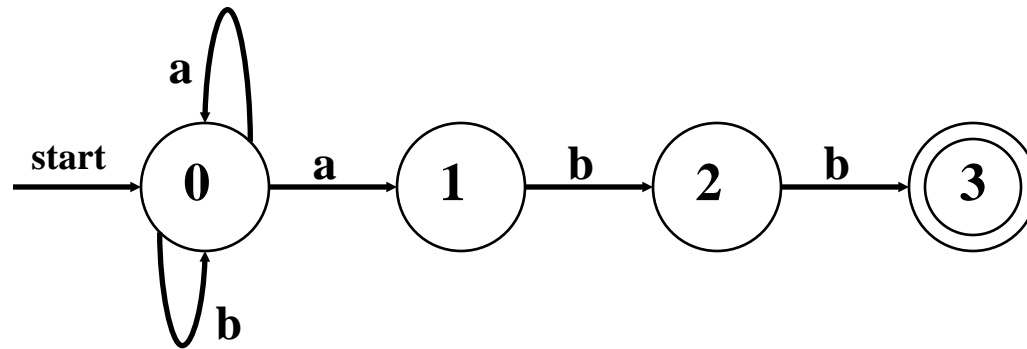
Handling Undefined Transitions

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



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 not accepted along the valid path:

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

Deterministic Finite Automata

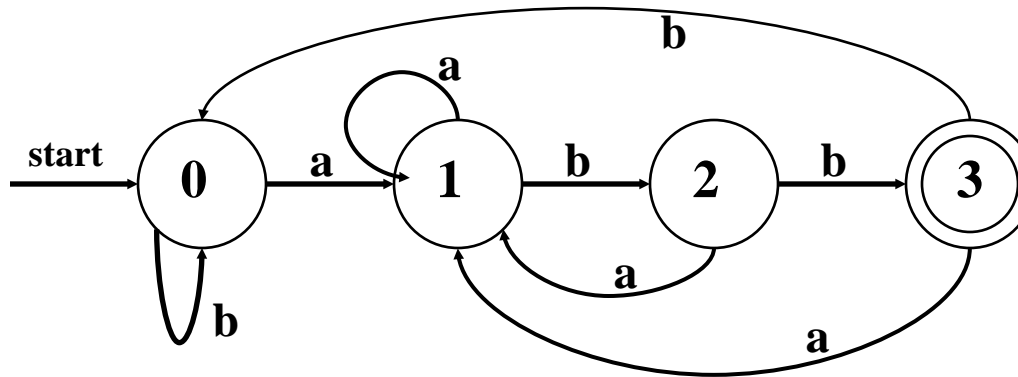
A DFA is an NFA with the following restrictions:

- ϵ moves are not 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.

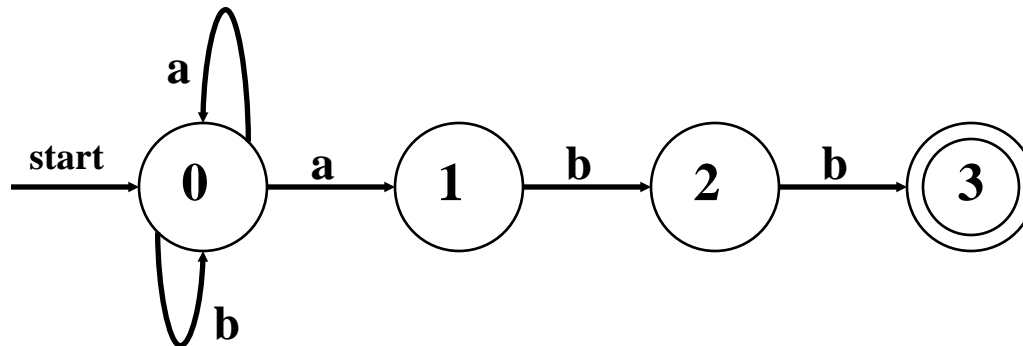
```
s ← s0
c ← nextchar;
while c ≠ eof do
    s ← move(s,c);
    c ← nextchar;
end;
if s is in F then return "yes"
else return "no"
```

Example – DFA : $(a | b)^*abb$



What Language is Accepted?

Recall the original NFA:



Relation between RE, NFA and DFA

1. There is an algorithm for converting any RE into an NFA.
2. There is an algorithm for converting any NFA to a DFA.
3. There is an algorithm for converting any DFA to a RE.

These facts tell us that REs, NFAs and DFAs have equivalent expressive power.

All three describe the class of regular languages.

NFA vs DFA

- An NFA may be **simulated by algorithm**, when NFA is **constructed from the R.E**
 - 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 uses it to recognize input
 - The space requirement of a DFA can be large. The RE $(a+b)^*a(a+b)(a+b)\dots(a+b)$ [n-1 (a+b) at the end] has no DFA with less than 2^n states. Fortunately, such RE in practice does not occur often

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

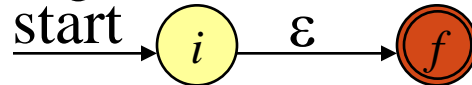
where $|r|$ is the length of the regular expression.

Converting Regular Expressions to NFAs

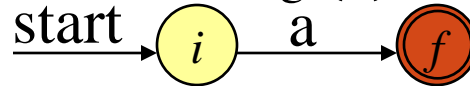
Converting Regular Expressions to NFAs

Thompson's Construction

- Empty string ε is a regular expression denoting $\{ \varepsilon \}$



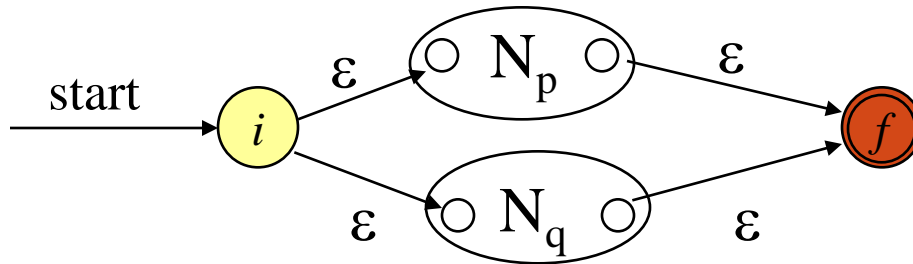
- a is a regular expression denoting $\{a\}$ for any a in Σ



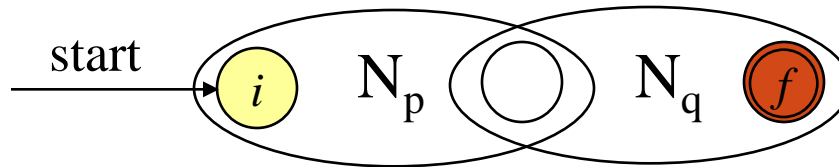
Converting Regular Expressions to NFAs

If P and Q are regular expressions with NFAs N_p, N_q :

$P \mid Q$ (union)



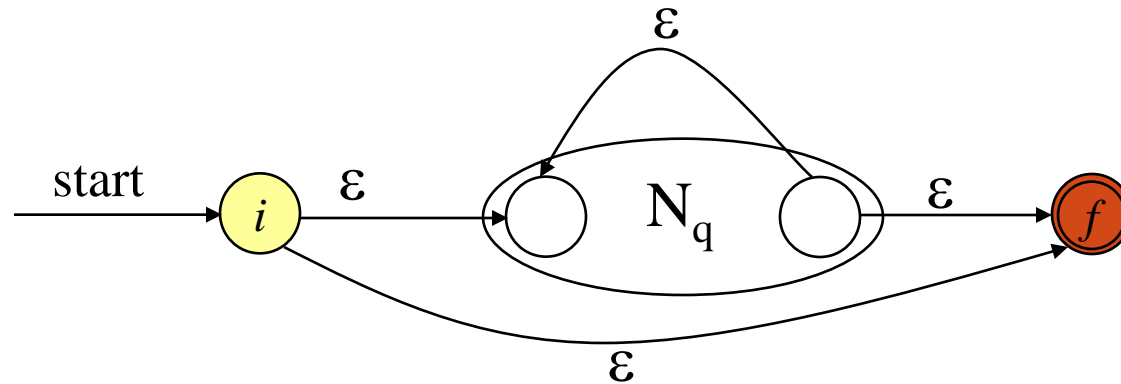
PQ (concatenation)



Converting Regular Expressions to NFAs

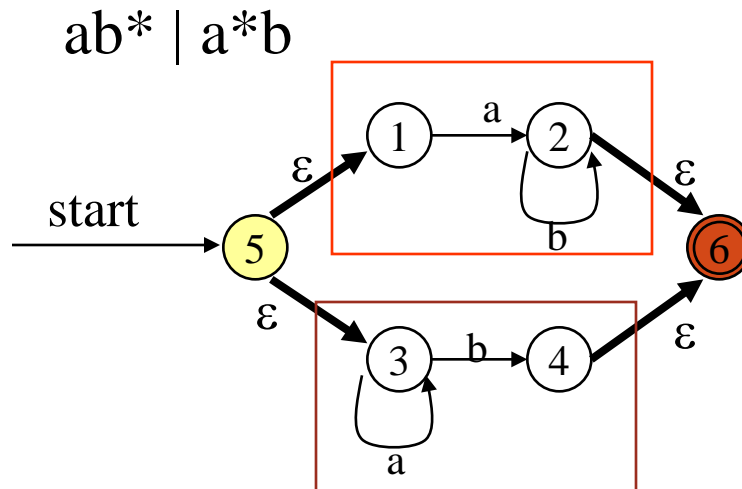
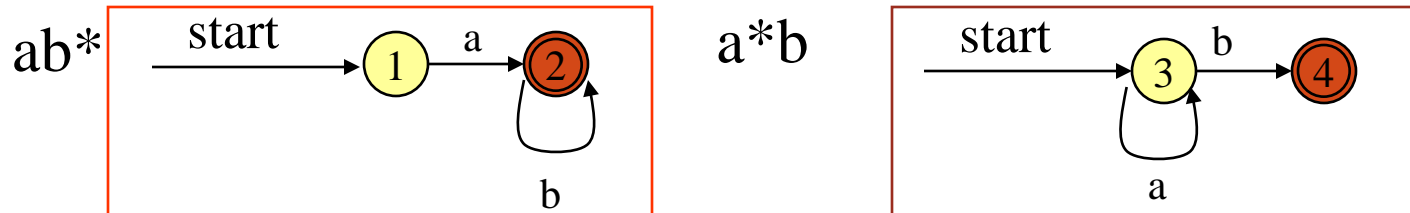
If Q is a regular expression with NFA N_q :

Q^* (closure)



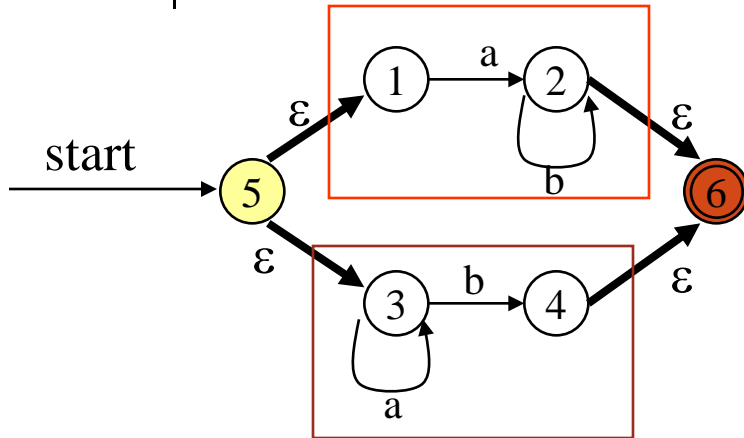
Example $(ab^* \mid a^*b)^*$

Starting with:

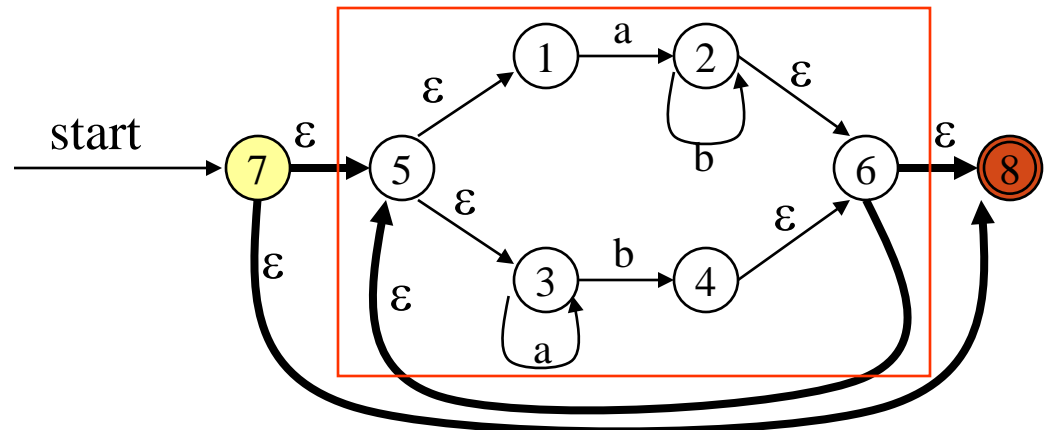


Example $(ab^* \mid a^*b)^*$

$ab^* \mid a^*b$



$(ab^* \mid a^*b)^*$



Properties of Construction

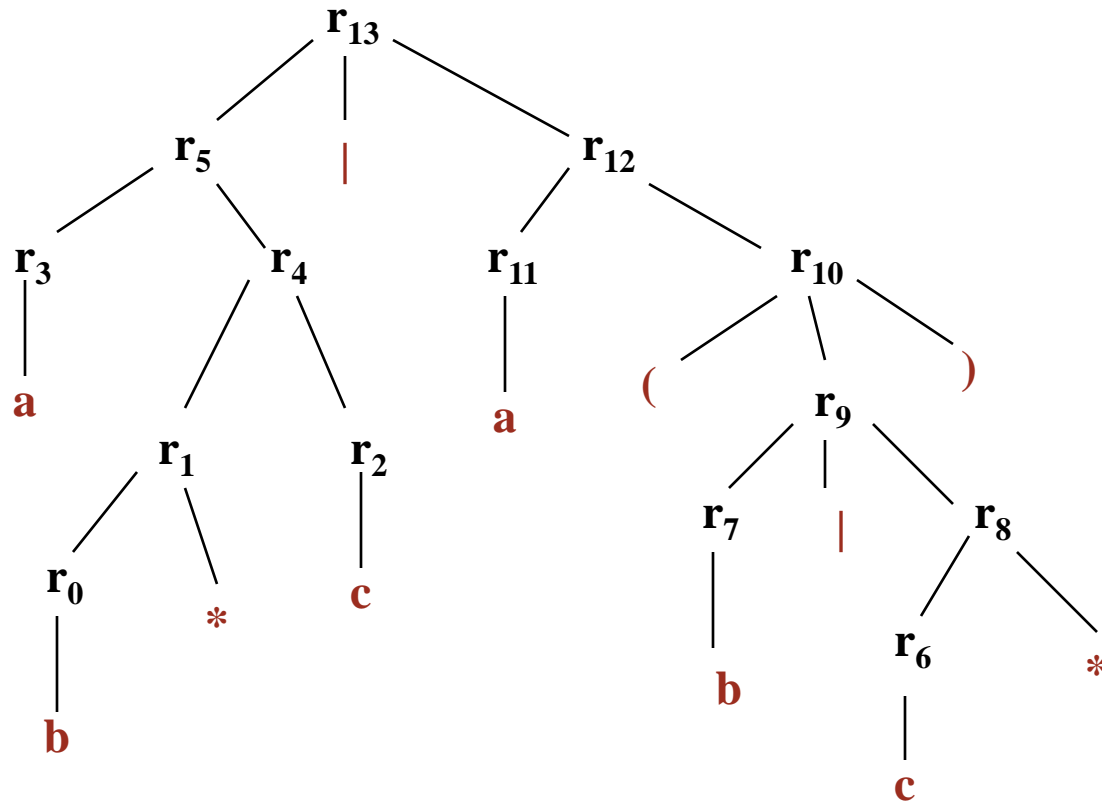
Let r be a regular expression, with NFA $N(r)$, then

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 ϵ -transition

Detailed Example

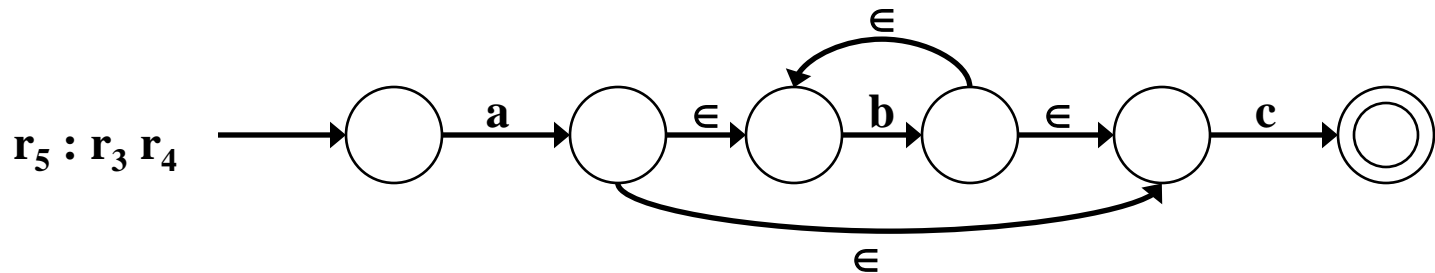
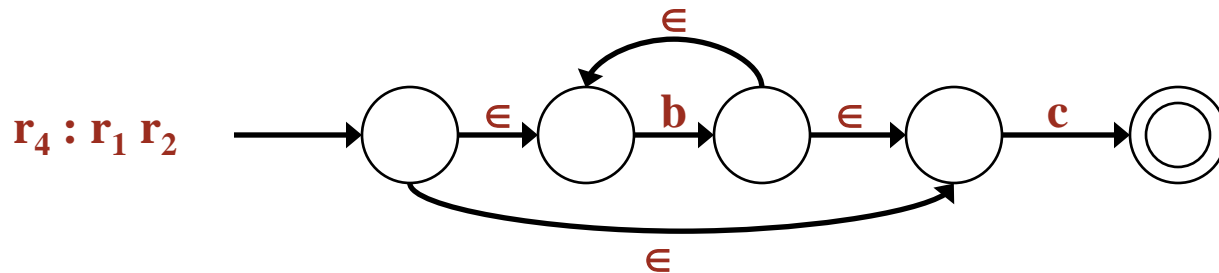
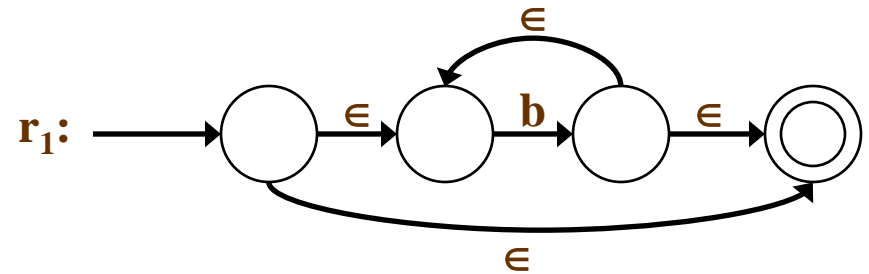
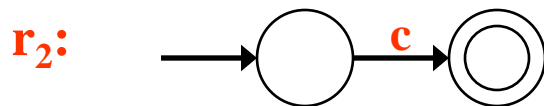
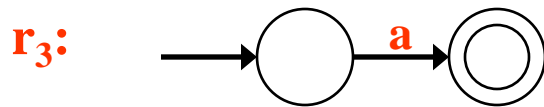
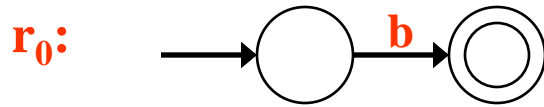
$(ab^*c) \mid (a(b \mid c^*))$

Parse Tree for this regular expression:



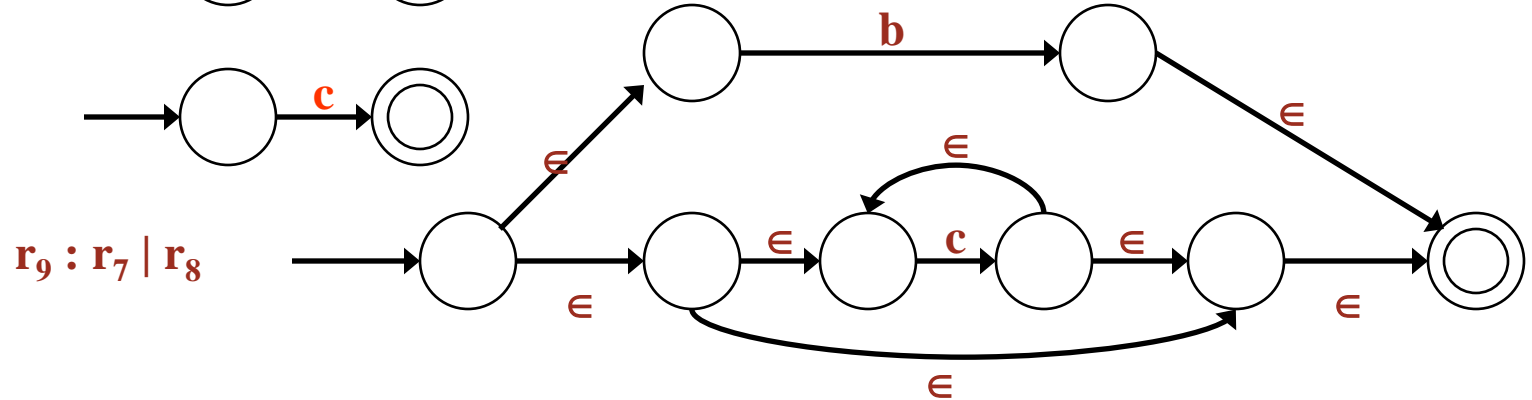
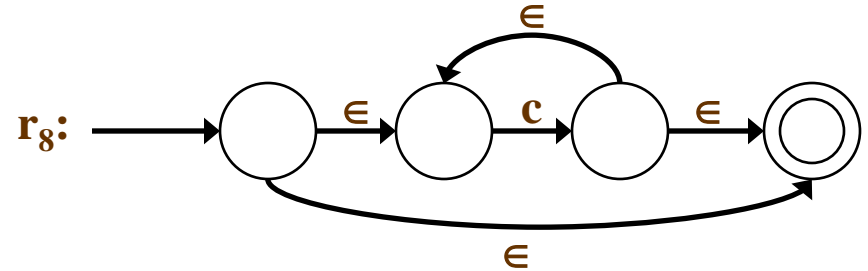
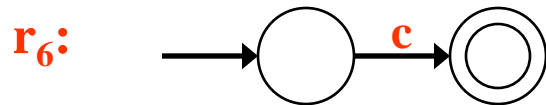
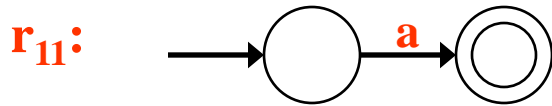
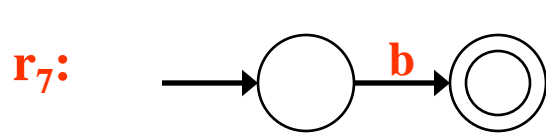
Detailed Example – Construction(1)

$(ab^*c) \mid (a(b \mid c^*))$

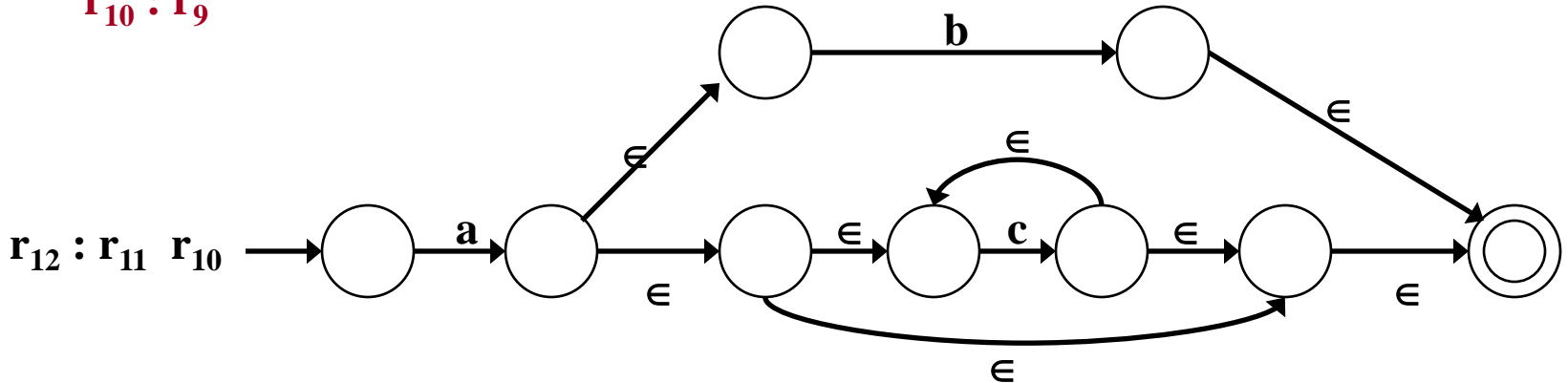


Detailed Example – Construction(2)

$(ab^*c) \mid (a(b \mid c^*))$

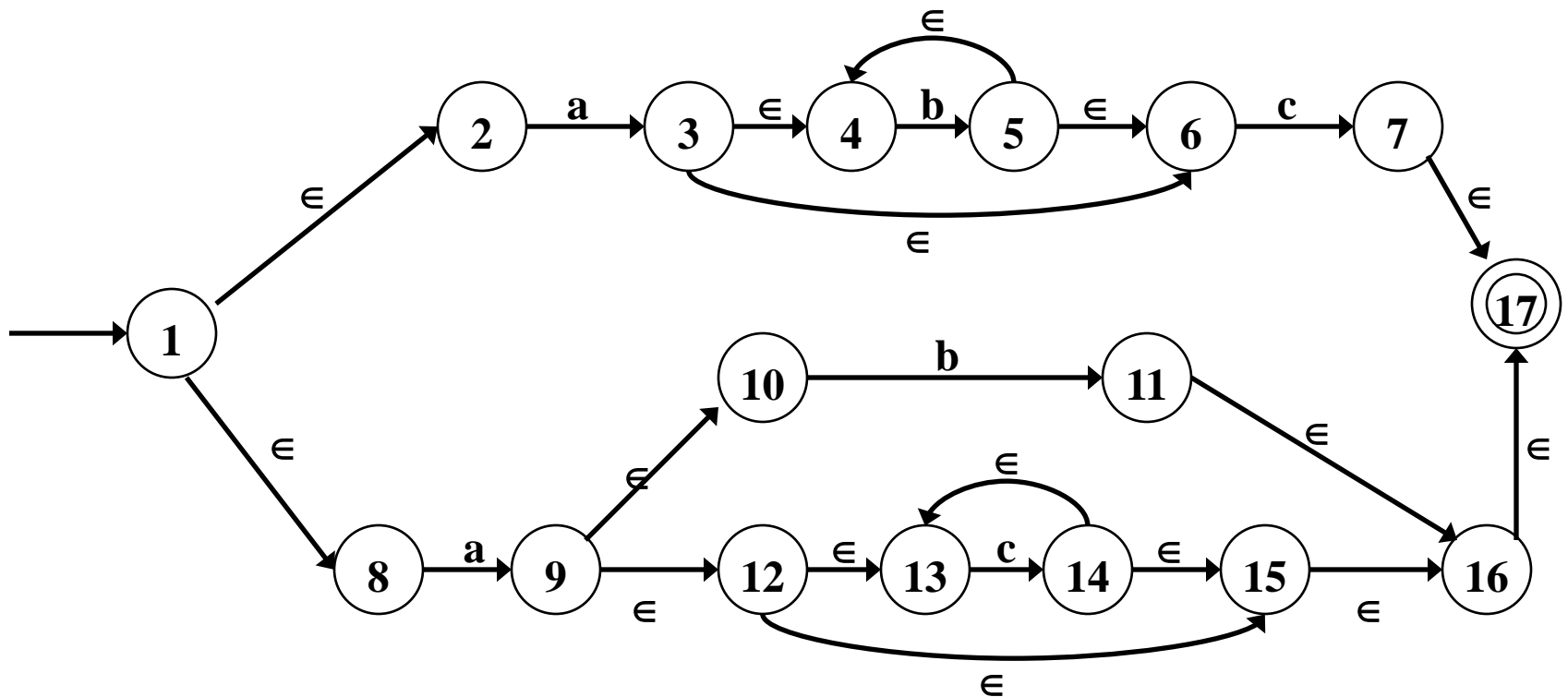


$r_{10} : r_9$



Detailed Example – Final Step

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



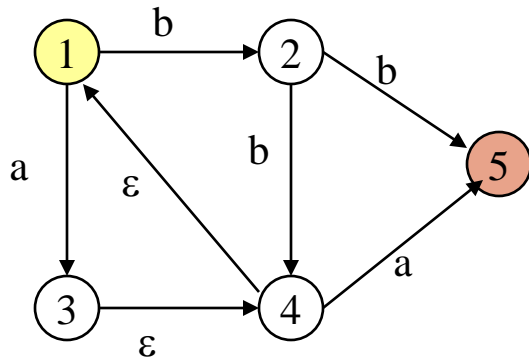
Converting NFAs to DFAs

Converting NFAs to DFAs (subset construction)

- **Idea:** Each state in the new DFA will correspond to some set of states from the NFA. The DFA will be in state $\{s_0, s_1, \dots\}$ after input if the NFA could be in *any* of these states for the same input.
- **Input:** NFA N with state set S_N , alphabet Σ , start state s_N , final states F_N , transition function $T_N: S_N \times \{\Sigma \cup \epsilon\} \rightarrow S_N$
- **Output:** DFA D with state set S_D , alphabet Σ , start state $s_D = \epsilon\text{-closure}(s_N)$, final states F_D , transition function $T_D: S_D \times \Sigma \rightarrow S_D$

Terminology: ϵ -closure

ϵ -closure(T) = T + all NFA states reachable from any state in T using only ϵ transitions.



$$\epsilon\text{-closure}(\{1,2,5\}) = \{1,2,5\}$$

$$\epsilon\text{-closure}(\{4\}) = \{1,4\}$$

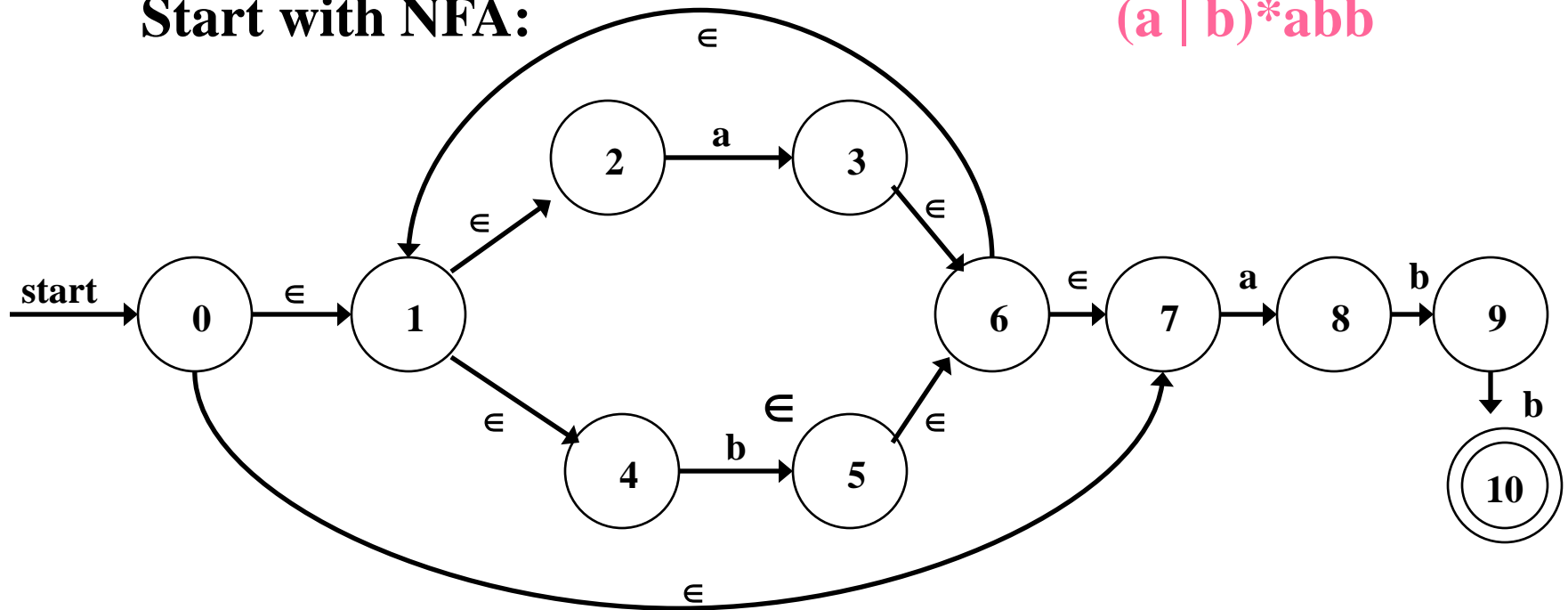
$$\epsilon\text{-closure}(\{3\}) = \{1,3,4\}$$

$$\epsilon\text{-closure}(\{3,5\}) = \{1,3,4,5\}$$

Illustrating Conversion – An Example

Start with NFA:

$(a \mid b)^*abb$



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 : $a : \epsilon\text{-closure}(\text{move}(A,a))$ and
 $b : \epsilon\text{-closure}(\text{move}(A,b))$

a : $\epsilon\text{-closure}(\text{move}(A,a)) = \epsilon\text{-closure}(\text{move}(\{0,1,2,4,7\},a))$
adds $\{3,8\}$ (since $\text{move}(2,a)=3$ and $\text{move}(7,a)=8$)

From this we have : $\epsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\}$
(since $3 \rightarrow 6 \rightarrow 1 \rightarrow 4$, $6 \rightarrow 7$, and $1 \rightarrow 2$ all by ϵ -moves)

Let $B = \{1,2,3,4,6,7,8\}$ be a new state. Define **$D\text{tran}[A,a] = B$** .

b : $\epsilon\text{-closure}(\text{move}(A,b)) = \epsilon\text{-closure}(\text{move}(\{0,1,2,4,7\},b))$

adds $\{5\}$ (since $\text{move}(4,b)=5$)

From this we have : $\epsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\}$
(since $5 \rightarrow 6 \rightarrow 1 \rightarrow 4$, $6 \rightarrow 7$, and $1 \rightarrow 2$ all by ϵ -moves)

Let $C = \{1,2,4,5,6,7\}$ be a new state. Define **$D\text{tran}[A,b] = C$** .

Conversion Example – continued (2)

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

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

Define $\mathbf{Dtran}[\mathbf{B},a] = \mathbf{B}$.

$$\underline{b} : \epsilon\text{-closure}(\text{move}(\mathbf{B},b)) = \epsilon\text{-closure}(\text{move}(\{1,2,3,4,6,7,8\},b)) \\ = \{1,2,4,5,6,7,9\} = \mathbf{D}$$

Define $\mathbf{Dtran}[\mathbf{B},b] = \mathbf{D}$.

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

$$\underline{a} : \epsilon\text{-closure}(\text{move}(\mathbf{C},a)) = \epsilon\text{-closure}(\text{move}(\{1,2,4,5,6,7\},a)) \\ = \{1,2,3,4,6,7,8\} = \mathbf{B}$$

Define $\mathbf{Dtran}[\mathbf{C},a] = \mathbf{B}$.

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

Define $\mathbf{Dtran}[\mathbf{C},b] = \mathbf{C}$.

Conversion Example – continued (3)

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

$$\underline{a} : \epsilon\text{-closure}(\text{move}(\text{D},a)) = \epsilon\text{-closure}(\text{move}(\{1,2,4,5,6,7,9\},a)) \\ = \{1,2,3,4,6,7,8\} = \text{B}$$

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

$$\underline{b} : \epsilon\text{-closure}(\text{move}(\text{D},b)) = \epsilon\text{-closure}(\text{move}(\{1,2,4,5,6,7,9\},b)) \\ = \{1,2,4,5,6,7,10\} = \text{E}$$

Define $\text{Dtran}[\text{D},b] = \text{E}$.

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

$$\underline{a} : \epsilon\text{-closure}(\text{move}(\text{E},a)) = \epsilon\text{-closure}(\text{move}(\{1,2,4,5,6,7,10\},a)) \\ = \{1,2,3,4,6,7,8\} = \text{B}$$

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

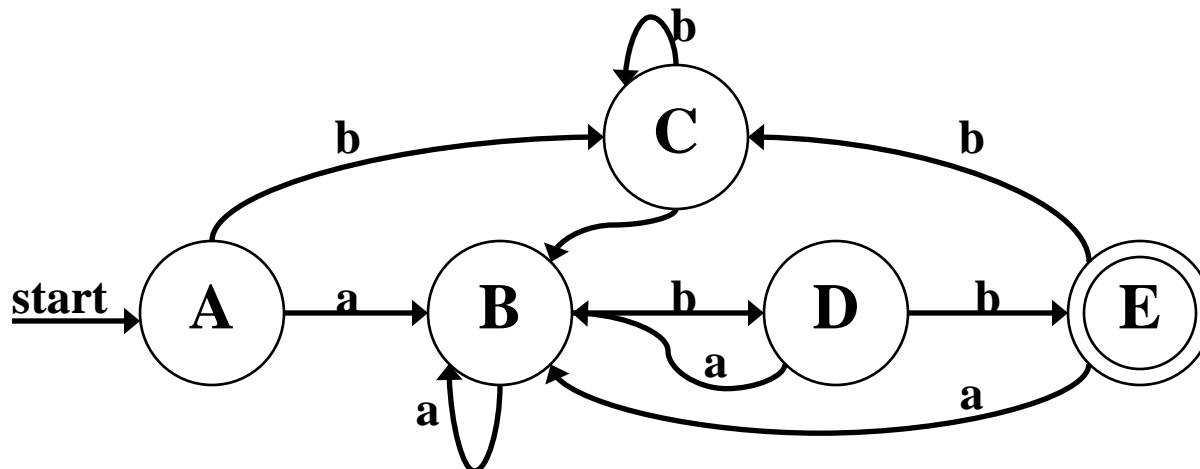
$$\underline{b} : \epsilon\text{-closure}(\text{move}(\text{E},b)) = \epsilon\text{-closure}(\text{move}(\{1,2,4,5,6,7,10\},b)) \\ = \{1,2,4,5,6,7\} = \text{C}$$

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

Conversion Example – continued (4)

This gives the transition table **Dtran** for the DFA of:

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



Algorithm For Subset Construction

push all states in T onto stack;

initialize ϵ -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 ϵ do

 if u is not in ϵ -closure(T) do begin

 add u to ϵ -closure(T) ;

 push u onto stack

 end

 end

**computing the
 ϵ -closure**

Algorithm For Subset Construction – (2)

initially, ϵ -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

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

 if U is not in **Dstates** then

 add U as an unmarked state to **Dstates**;

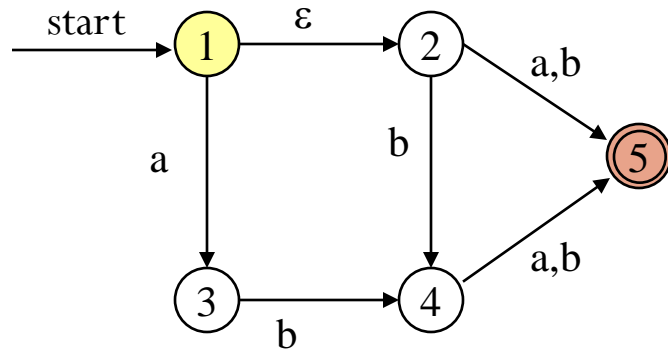
Dtran[T, a] := U

 end

end

Example 2: Subset Construction

NFA



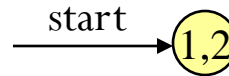
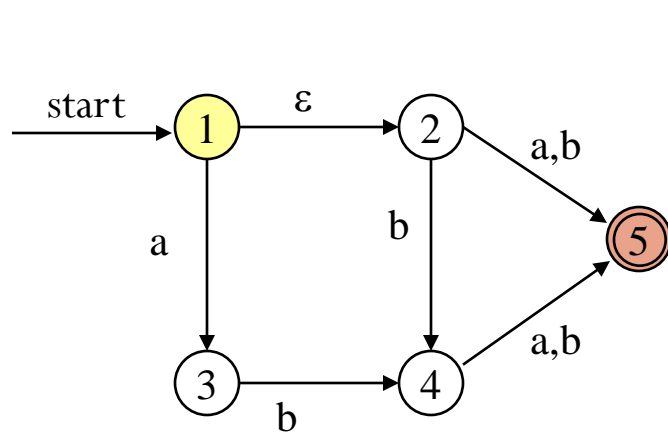
NFA N with

- State set $S_N = \{1,2,3,4,5\}$,
- Alphabet $\Sigma = \{a,b\}$
- Start state $s_N=1$,
- Final states $F_N=\{5\}$,
- Transition function $T_N: S_N \times \{\Sigma \cup \varepsilon\} \rightarrow S_N$

	a	b	ε
1	3	-	2
2	5	5, 4	-
3	-	4	-
4	5	5	-
5	-	-	-

Example 2: Subset Construction

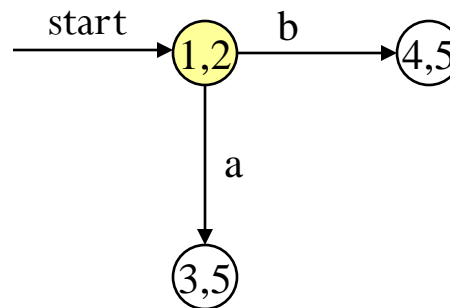
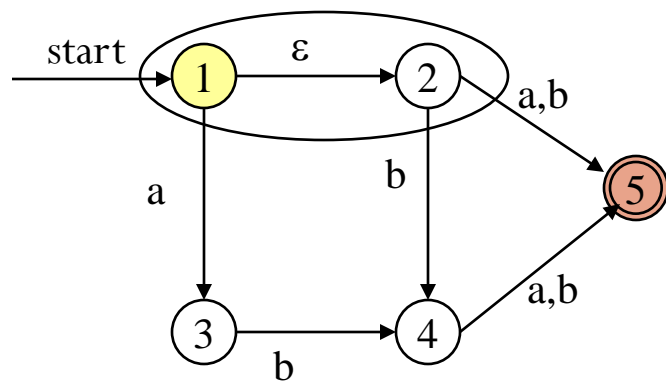
NFA



T	ϵ -closure(move(T, a))	ϵ -closure(move(T, b))
{1,2}		

Example 2: Subset Construction

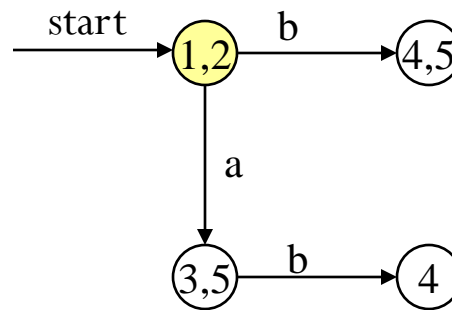
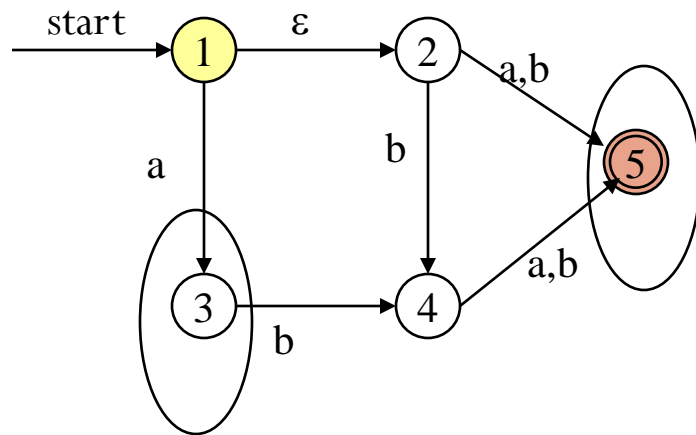
NFA



T	ϵ -closure(move(T, a))	ϵ -closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}		
{4,5}		

Example 2: Subset Construction

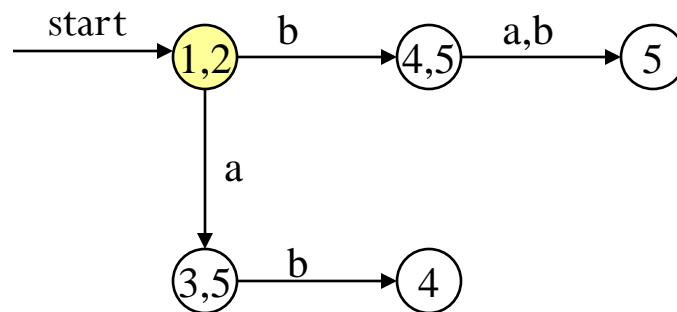
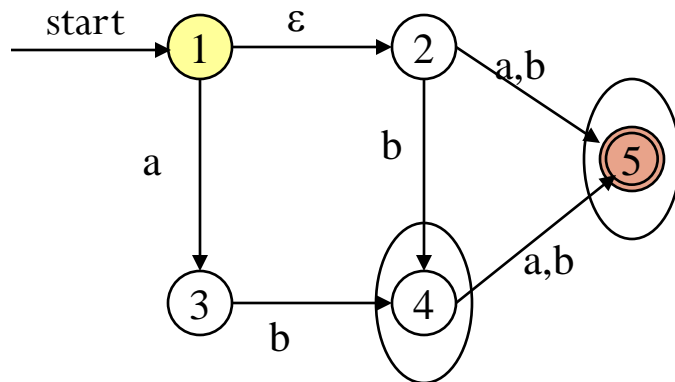
NFA



T	ϵ -closure(move(T, a))	ϵ -closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}		
{4}		

Example 2: Subset Construction

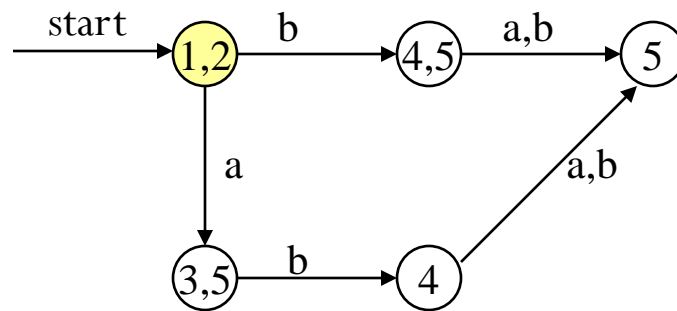
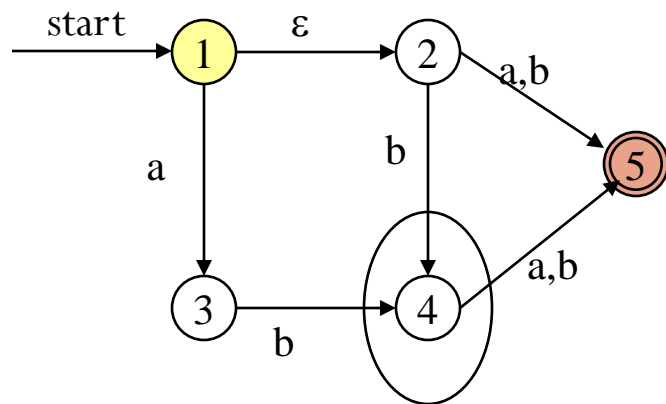
NFA



T	ϵ -closure(move(T, a))	ϵ -closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}		
{5}		

Example 2: Subset Construction

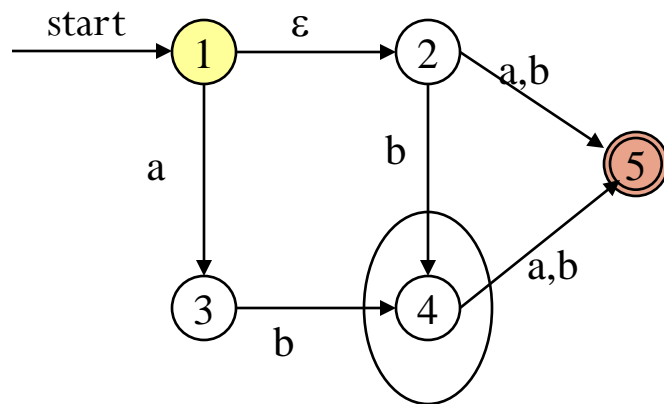
NFA



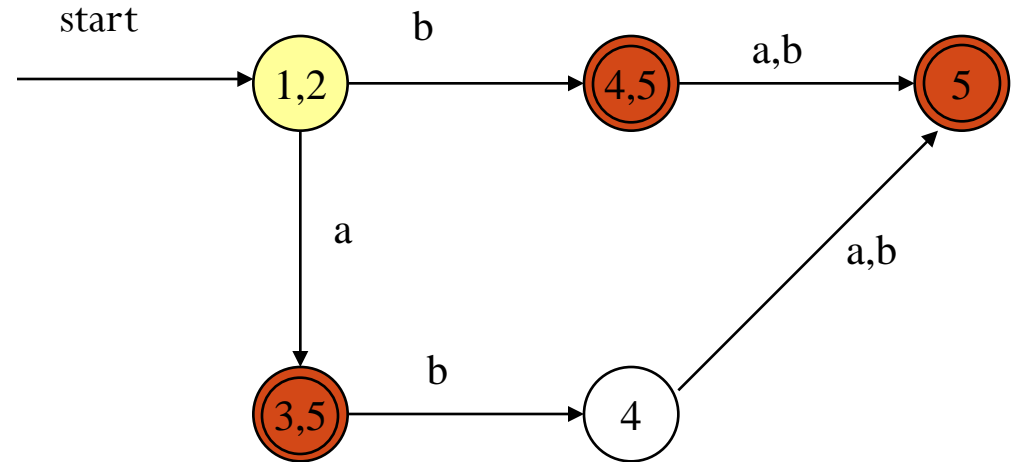
T	ϵ -closure(move(T, a))	ϵ -closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}	{5}	{5}
{5}	-	-

Example 2: Subset Construction

NFA



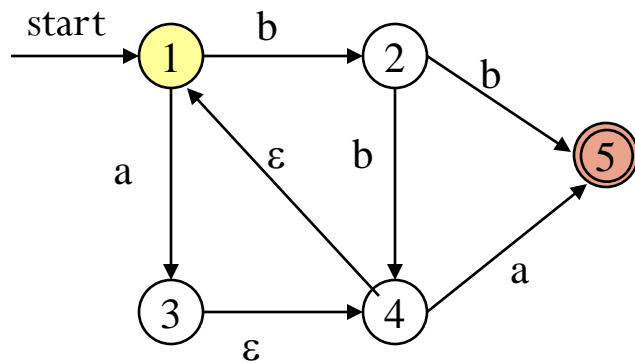
All final states since the NFA final state is included



T	ϵ -closure(move(T, a))	ϵ -closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}	{5}	{5}
{5}	-	-

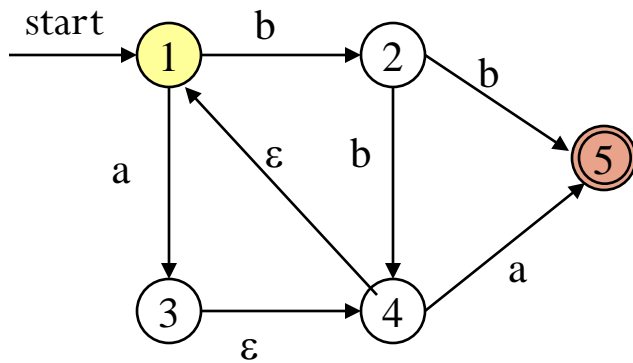
Example 3: Subset Construction

NFA

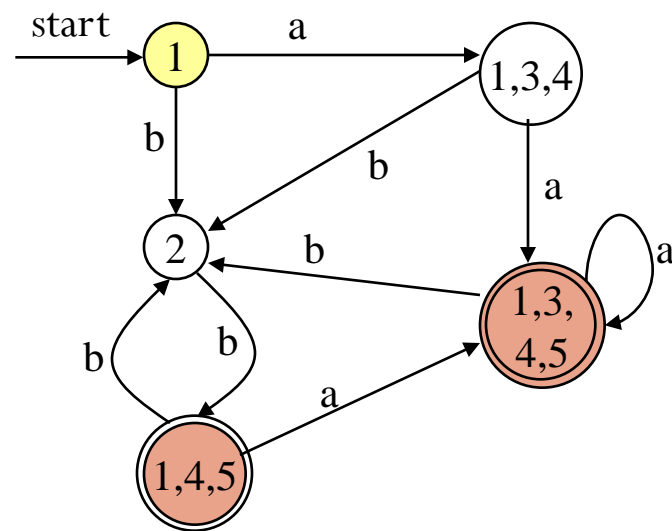


Example 3: Subset Construction

NFA

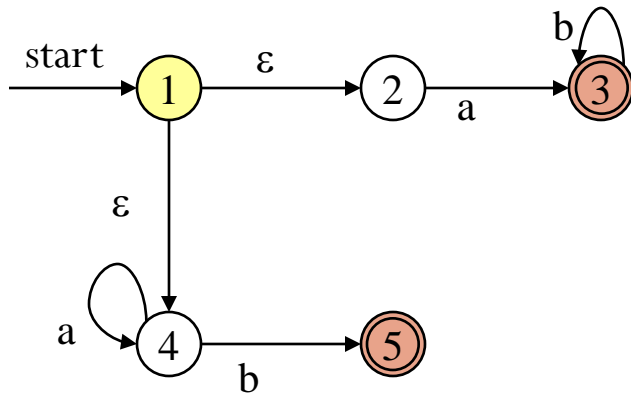


DFA

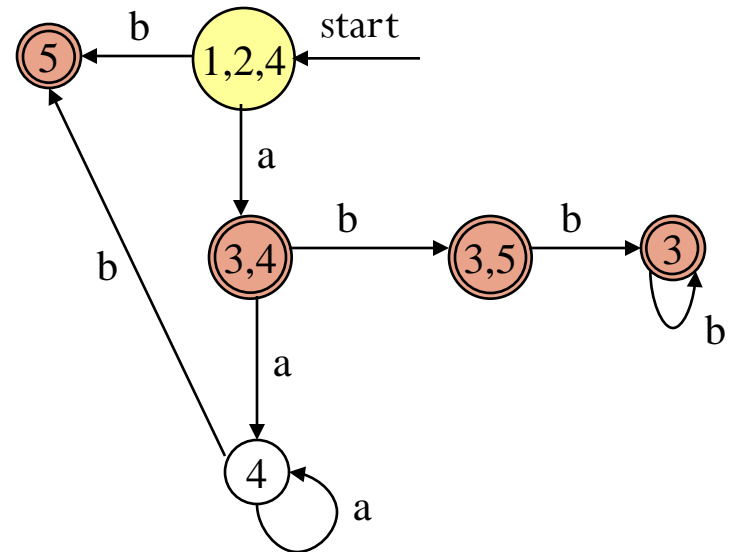


Example 4: Subset Construction

NFA



DFA



Thank You

Any Questions?