# Lexical Analysis

CSE 4102
Lecture 02

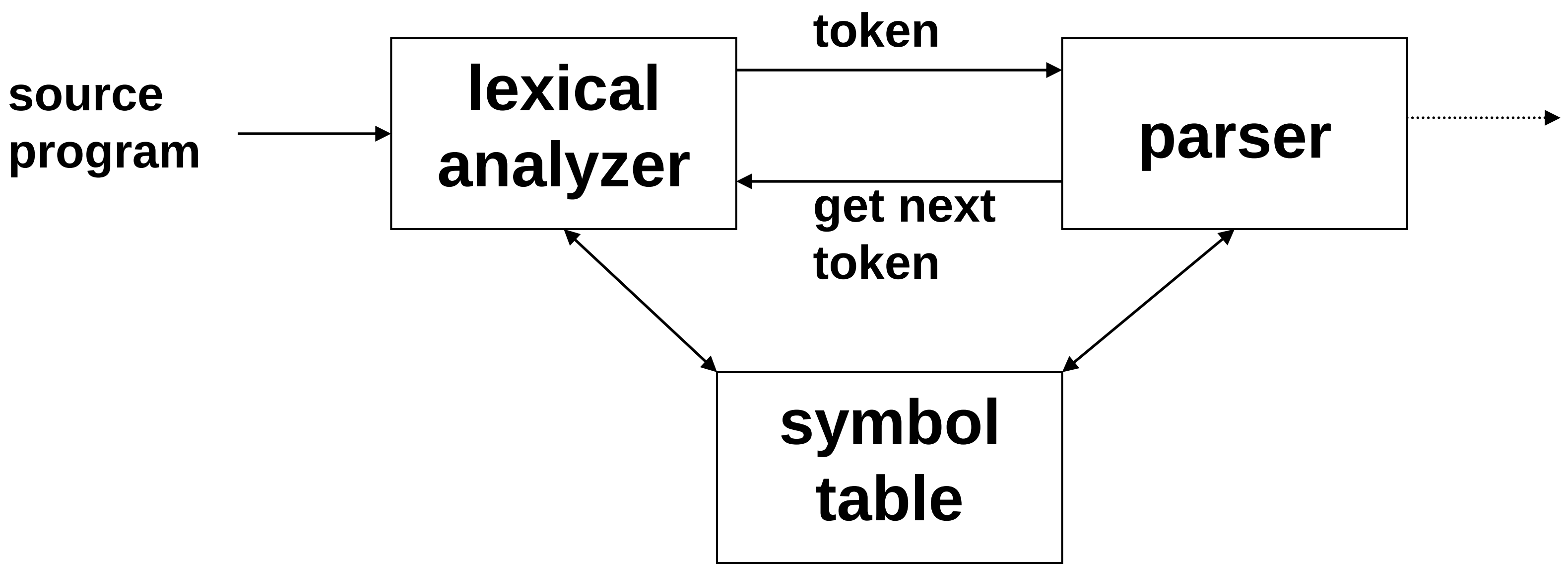
# Lexical Analysis

- 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 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 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:

      <u>if</u> x == -12.30 <u>then</u> …

# 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, 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

- When more than one lexeme can match a pattern, a lexical analyzer must provide the compiler **additional information** about that lexeme matched.

- In formation 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 <u>beginning</u> and <u>forward</u> 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.

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | E | | = | | M | | * | C | * | * | 2 | | | | | | eof |

Lexeme_beginning forward

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

# 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 is 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:*
$L^0 = \{ \varepsilon \}$
$L^1 = L = \{ a, bc \}$
$L^2 = LL = \{ aa, abc, bca, bcbc \}$
$L^3 = LLL = \{ aaa, aabc, abca, abcbc, bcaa, bcabc, bcbca, bcbcbc \}$
...etc...
$L^N = L^{N-1}L = LL^{N-1}$

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

*The "Kleene Closure" of a language:*

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

*Example:*

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

# Example

*Let:* L = { **a**, **b**, **c**, ..., **z** }
D = { **0**, **1**, **2**, ..., **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 $\varepsilon$, the empty string"*

$( L \cup D )^{\mathbf{*}}$ = *"Sequences of zero or more letters and digits"*

$L ( ( L \cup D )^{\mathbf{*}} )$ = *"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 $\Sigma$

1. $\varepsilon$ is a regular expression that denotes $\{\varepsilon\}$.
2. If **a** is a symbol (i.e., if **a**$\in \Sigma$), then **a** is a regular expression that denotes {a}.
3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then
   a) (r) | (s) is a regular expression denoting L(r) U 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 Charaters:*

| | |
|---|---|
| `\|` | 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]` |
| `\x` | The usual escape sequences, e.g., `\n` |
| `.` | Any character except '\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 | b denotes the set {a, b}
  - The regular expression (a|b)(a|b) denotes {aa, ab, ba, bb}
  - The regular expression $a^*$ denotes the set of all strings of zero or more a's. i.e., {ε, a, aa, aaa, ….. }
  - The regular expression $(a|b)^*$ denotes the set containing zero or more instances of an a or b.
  - The regular expression a|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:

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

where

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

# Regular Definition

<u>Letter</u> = `a | b | c | ... | z`

<u>Digit</u> = `0 | 1 | 2 | ... | 9`

<u>ID</u> = <u>Letter</u> `(` <u>Letter</u> `|` <u>Digit</u> `)*`

Names (e.g., <u>Letter</u>) 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:* $^{+}$

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

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

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

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

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

*Character Classes:* [*FirstChar*–*LastChar*]

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

Digit = `[0-9]`

Letter = `[a-zA-Z]` = `[A-Za-z]`

*Variations:*

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

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

*Optional:* `ab?c` = `a[b]c`

***What does*** `ab...bc` ***mean?***

# 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 ?

<u>Assume Following Tokens:</u>

if, then, else, relop, id, num

<u>Given Tokens, What are Patterns ?</u>

if $\rightarrow$ if

then $\rightarrow$ then

else $\rightarrow$ else

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

id $\rightarrow$ letter ( letter | digit )*

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

Grammar:

*stmt* $\rightarrow$ |if *expr* then *stmt*

|if *expr* then *stmt* else *stmt*

|$\epsilon$

*expr* $\rightarrow$ *term* relop *term* | *term*

*term* $\rightarrow$ 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 |

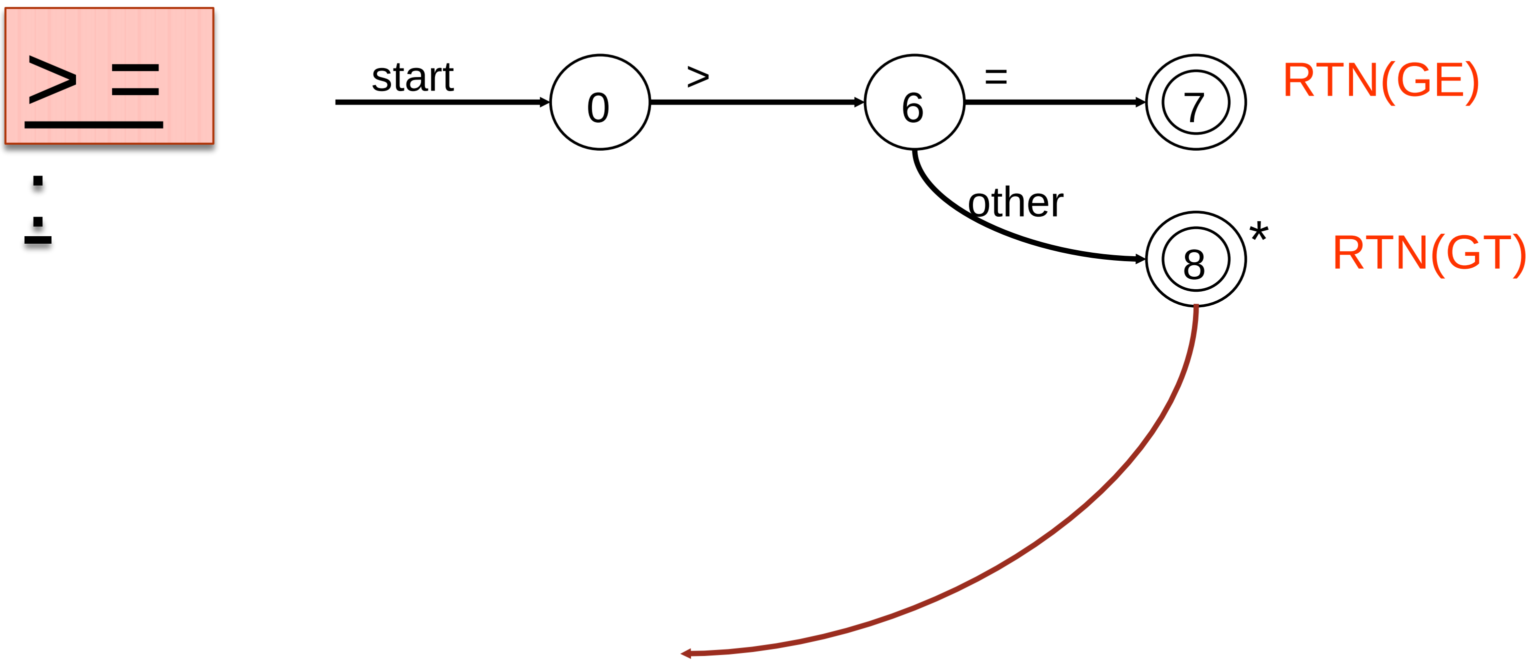


Note: Each token has a unique token identifier to define category of lexemes

# Constructing Transition Diagrams for Tokens

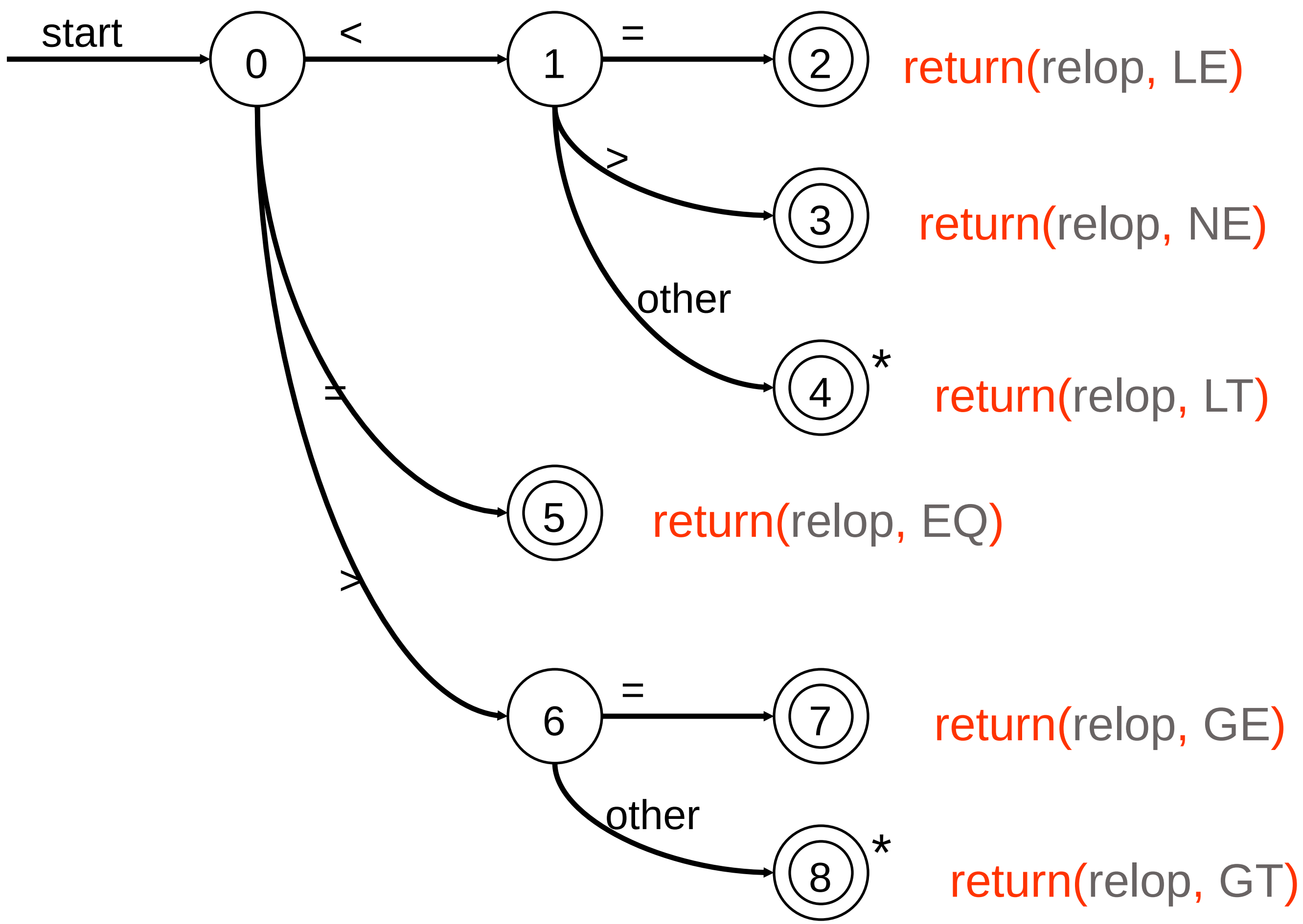
- Transition Diagrams (TD) are used to represent the tokens
- As characters are read, the relevant TDs are used to attempt to match lexeme to a pattern
- Each TD has:
  - States : Represented by Circles
  - Actions : Represented by Arrows between states
  - Start State : Beginning of a pattern (Arrowhead)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

# Example TDs



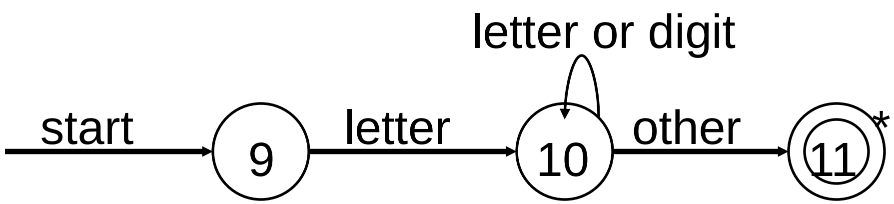

We've accepted ">" and have read one extra char that must be unread.

# Example : All RELOPs



# Example TDs : id and delim

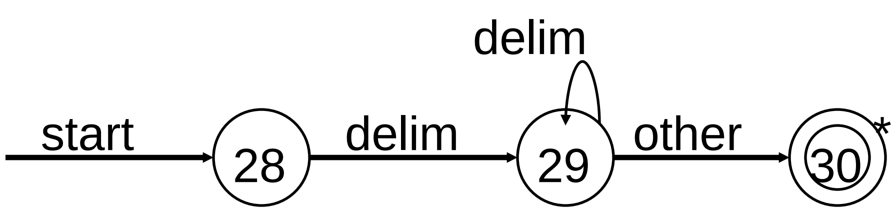
id :



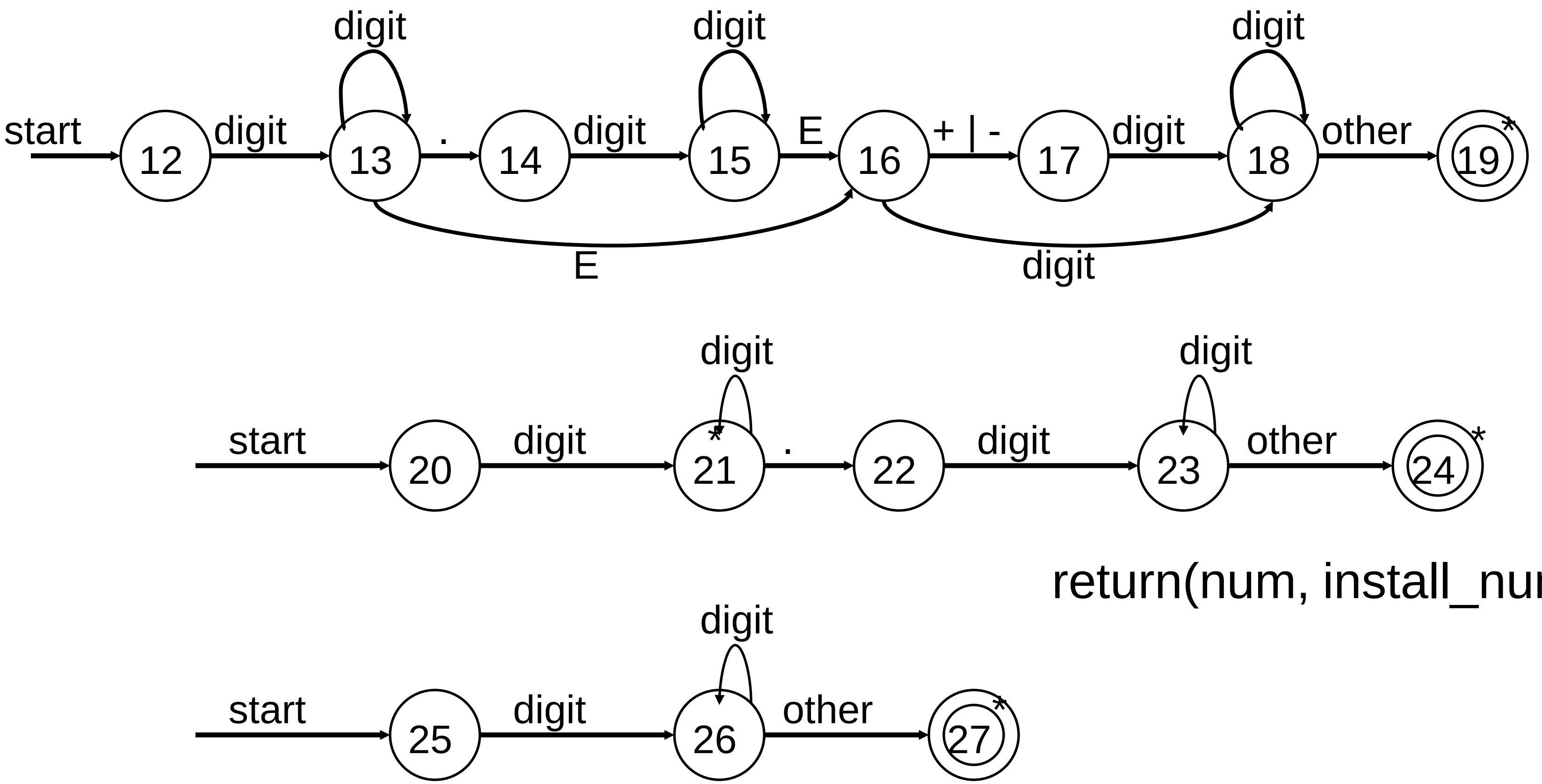

return( get_token(), install_id())

Either returns ptr or "0" if reserved

delim :



# Example TDs : Unsigned #s




return(num, install_num())

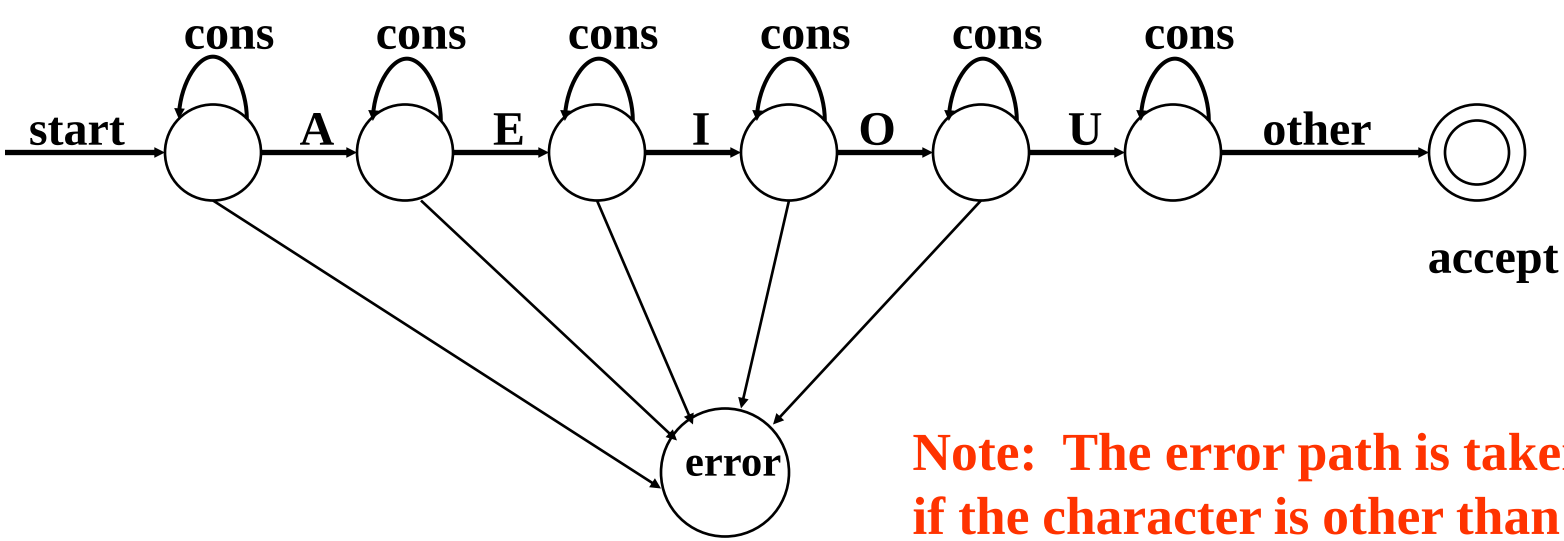
# QUESTION :

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

# Answer

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

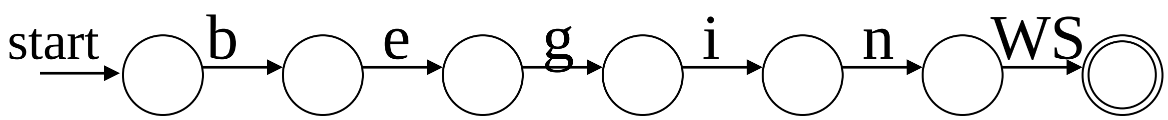
**string → cons* A cons* E cons* I cons* O cons* U cons***



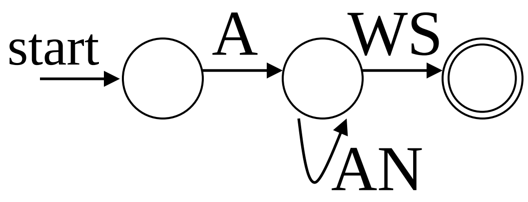

**Note: The error path is taken if the character is other than a cons or the vowel in the lex order.**

# Capturing Multiple Tokens

**Capturing keyword "begin"**



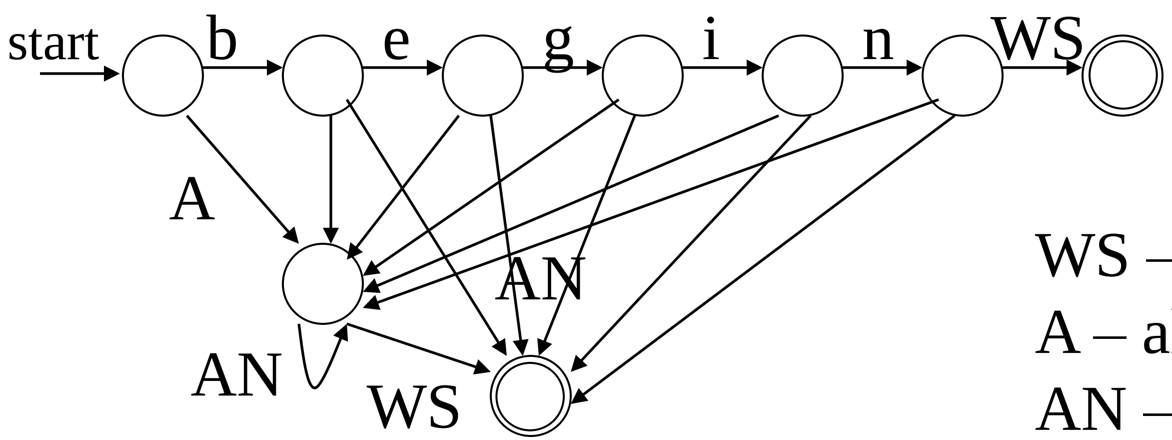

**Capturing variable names**




WS – white space
A – alphabetic
AN – alphanumeric

What if both need to happen at the same time?

# Capturing Multiple Tokens




WS – white space
A – alphabetic
AN – alphanumeric

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)

# 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

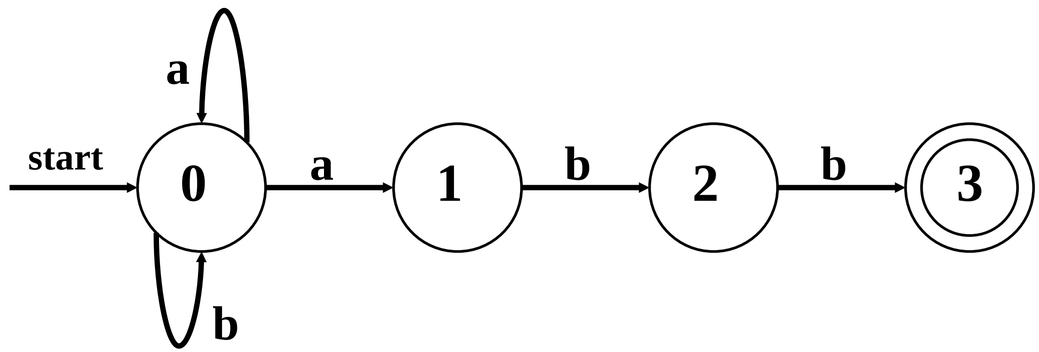
# Example – NFA : (a|b)*abb

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

$s_0 = 0$

$F = \{ 3 \}$

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

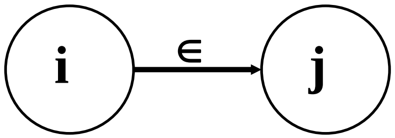



input

| state | a | b |
|---|---|---|
| 0 | { 0, 1 } | { 0 } |
| 1 | -- | { 2 } |
| 2 | -- | { 3 } |

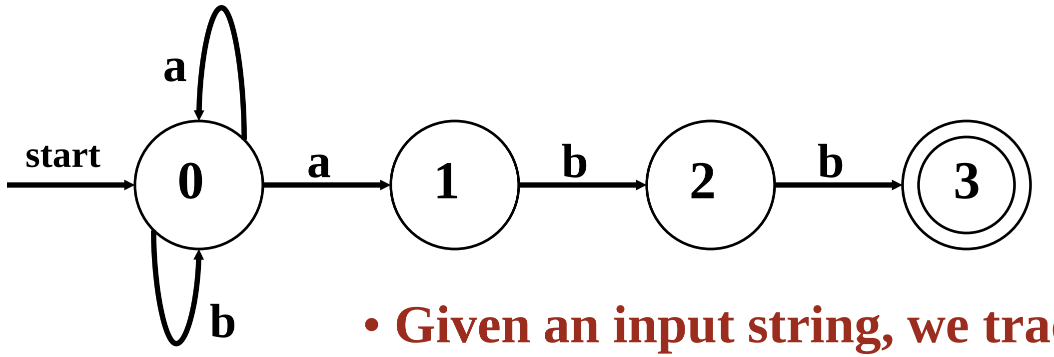
Transition Table

$\in$ (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 !**

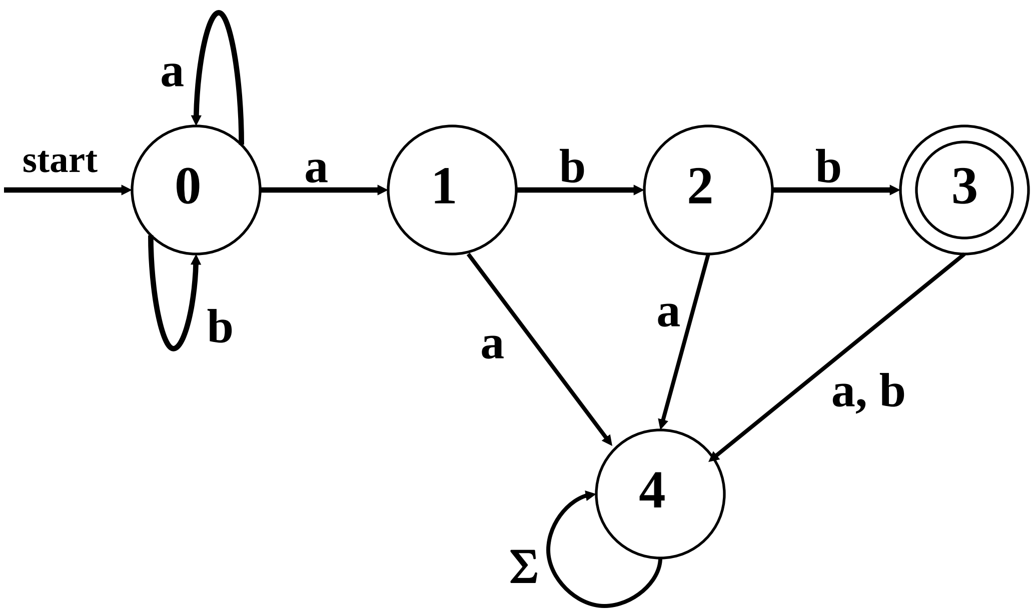
**-OR-**

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

**ACCEPT !**

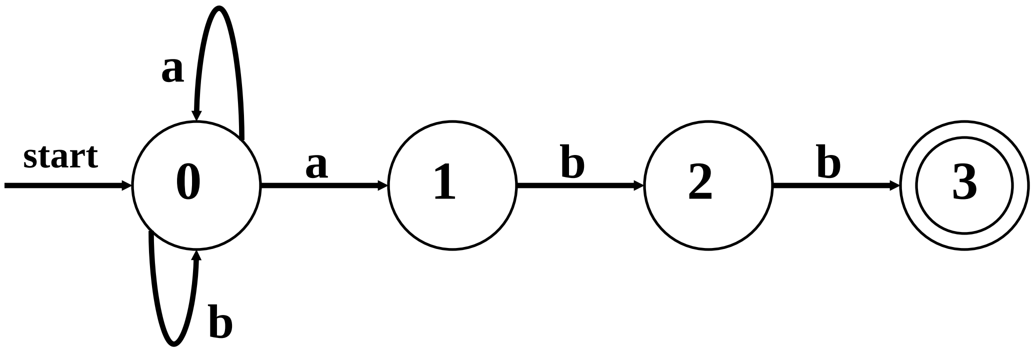
# Handling Undefined Transitions

**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 <u>not accepted</u> along the valid path:**

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

# Deterministic Finite Automata

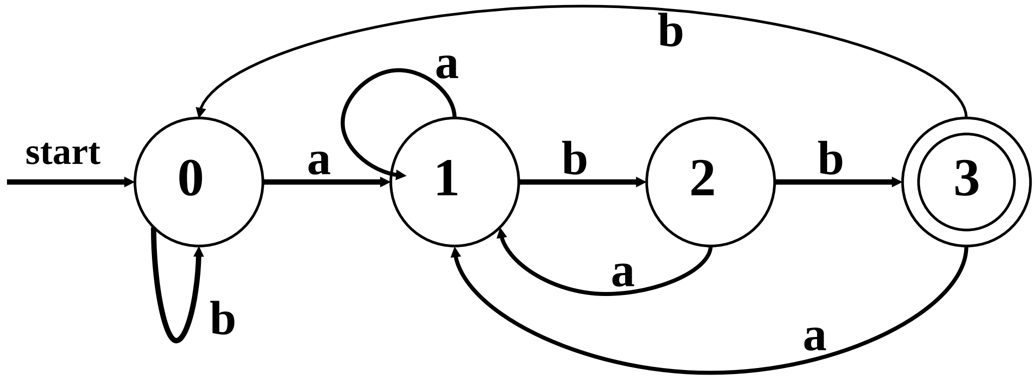
A DFA is an NFA with the following restrictions:

- $\in$ 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.

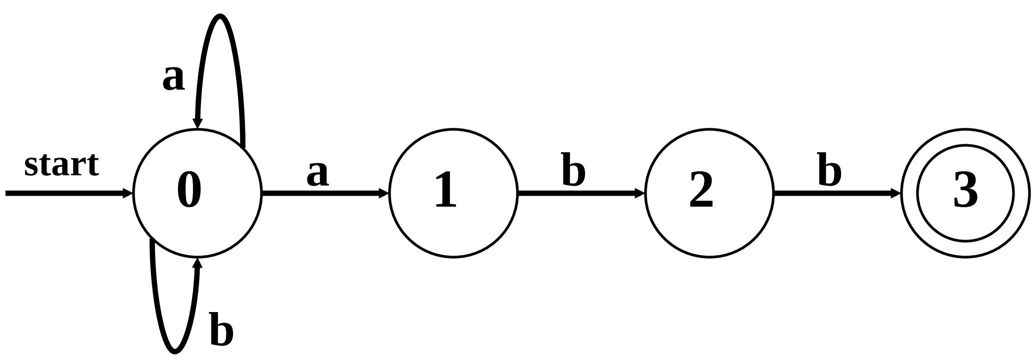
```
s ← s0
c ← nextchar;
while c ≠ eof do
   s ← move(s,c);
   c ← nextchar;
end;
if s is in F then return "yes"
                  else return "no"
```

# Example – DFA : (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)….(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\|) |

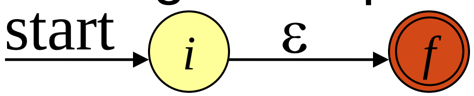
where |r| is the length of the regular expression.

# Converting Regular Expressions to NFAs

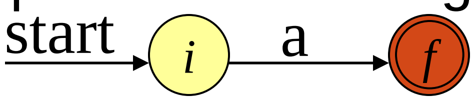
# Converting Regular Expressions to NFAs

## Thompson's Construction

- Empty string ε is a regular expression denoting { ε }



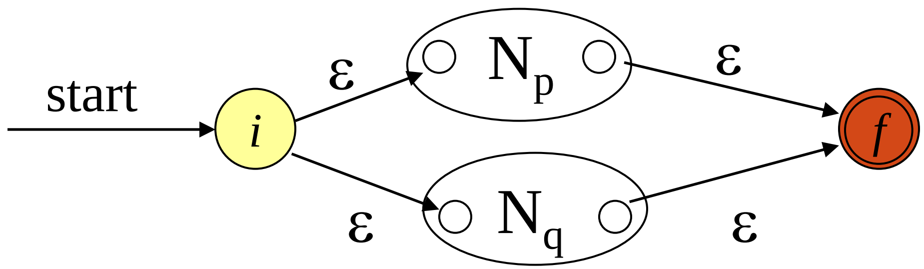

- *a* is a regular expression denoting {*a*} for any *a* in Σ



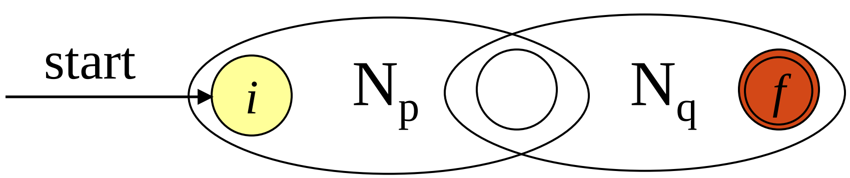
# Converting Regular Expressions to NFAs

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

P | Q (union)



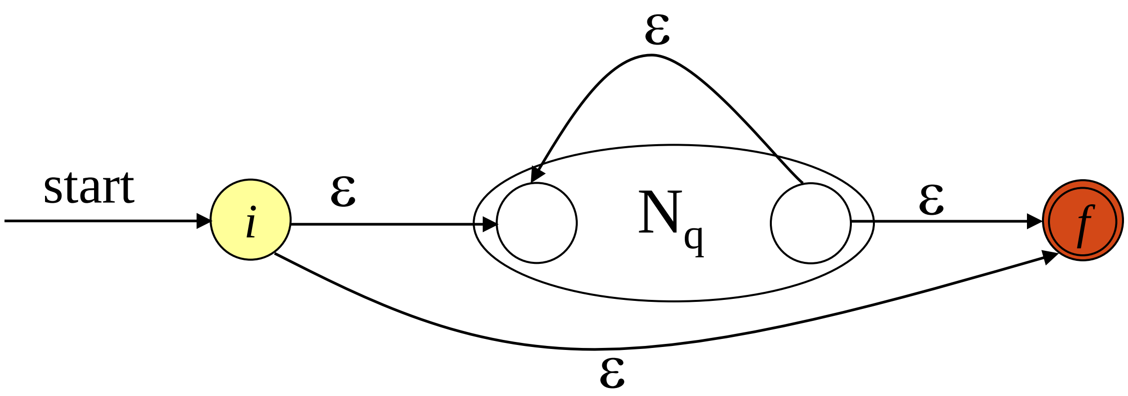

PQ (concatenation)



# NFAs

If Q is a regular expression with NFA $N_q$:

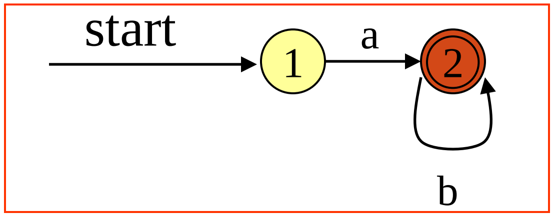
Q* (closure)



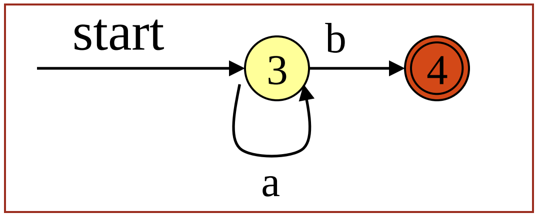
# Example (ab* | a*b)*

Starting with:

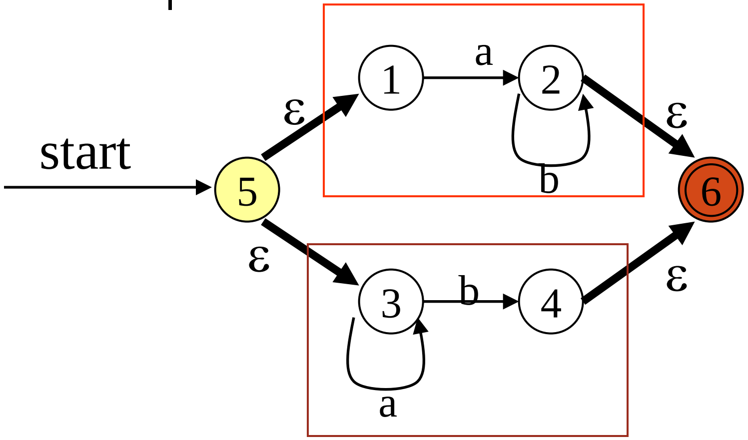
ab*



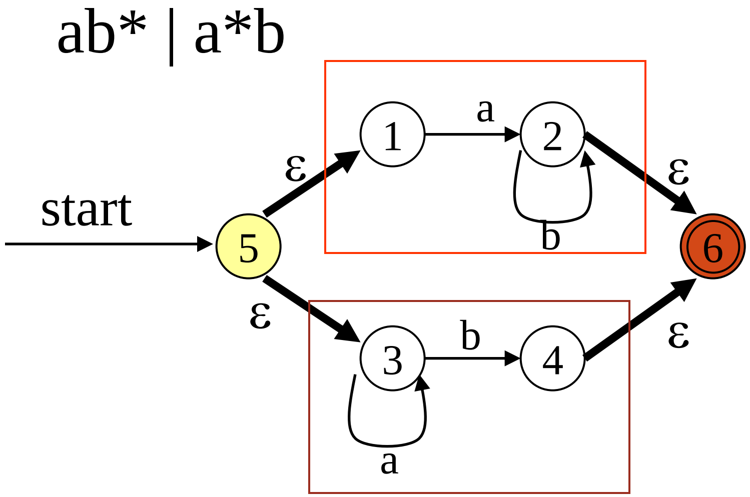

a*b

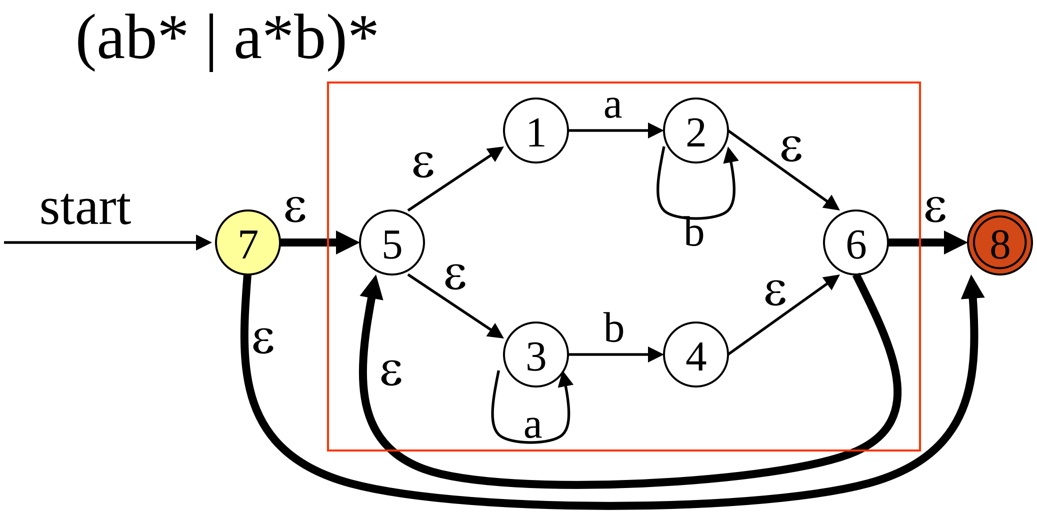



ab* | a*b



# Example (ab* | a*b)*



# Properties of Construction

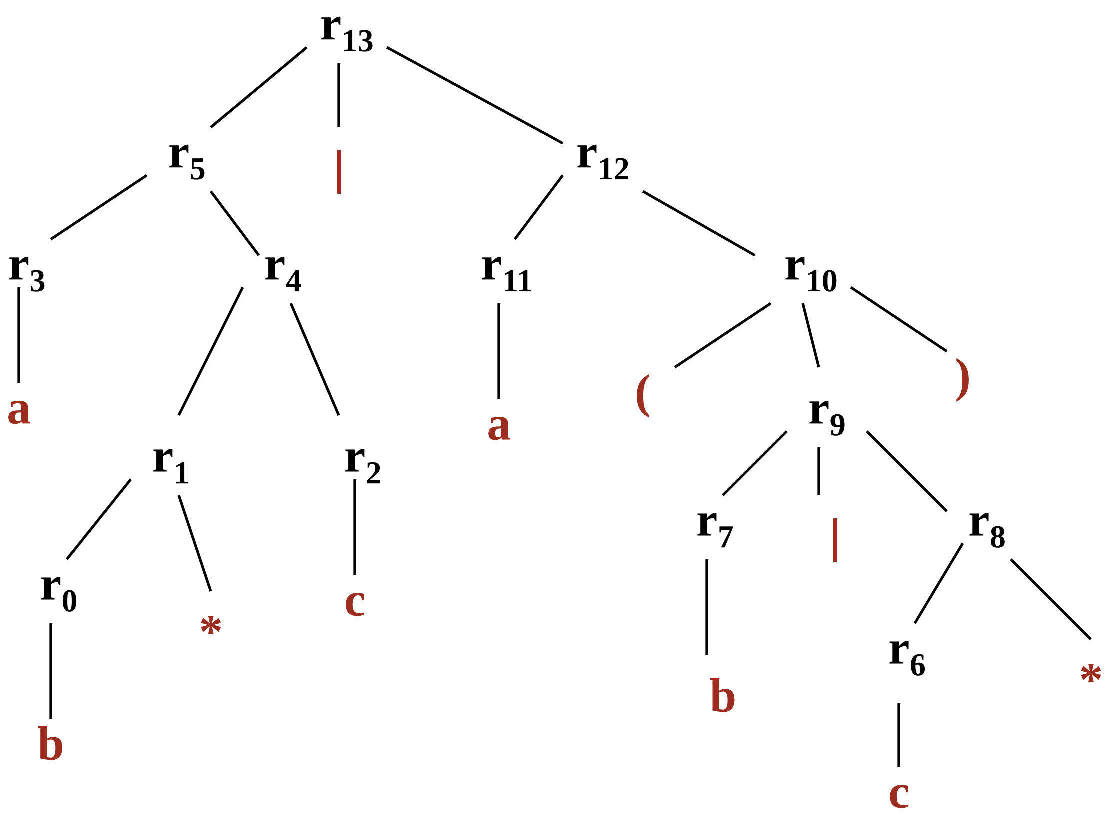
Let r be a regular expression, with NFA N(r), then

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 $\in$-transition

# Detailed Example

(ab*c) | (a(b|c*))

Parse Tree for this regular expression:

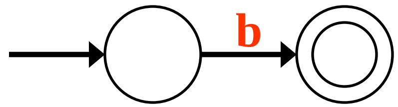



What is the NFA? Let’s construct it !

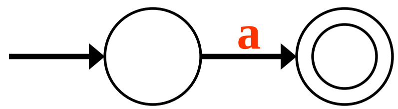
# Detailed Example – Construction(1)

**(ab*c) | (a(b|c*))**

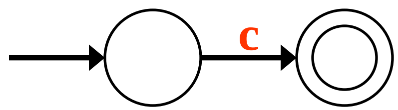
$r_0$:



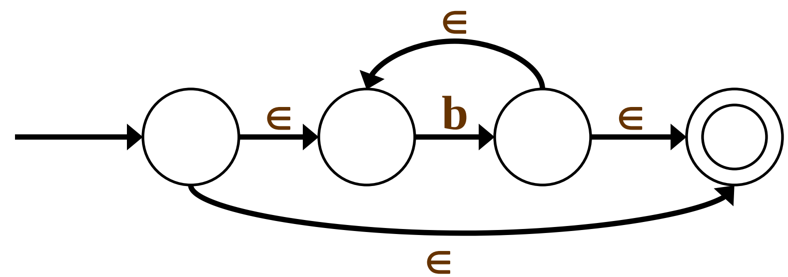

$r_3$:



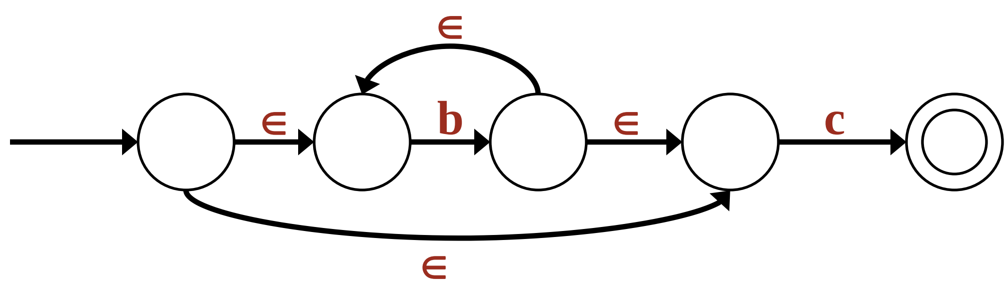

$r_2$:



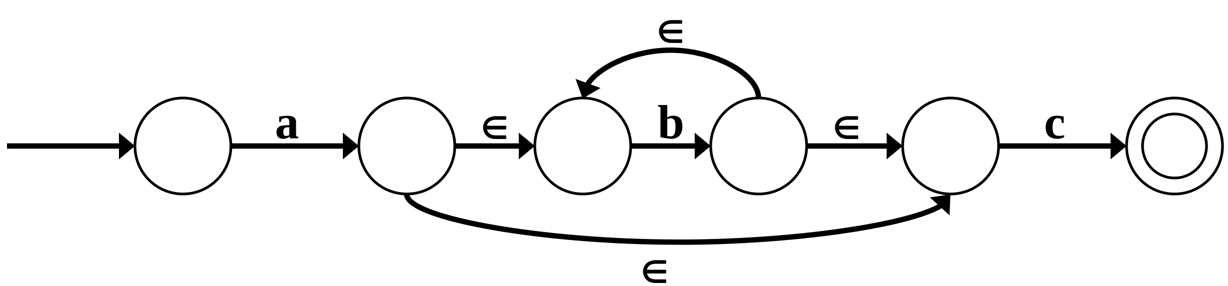

$r_1$:




$r_4 : r_1\, r_2$

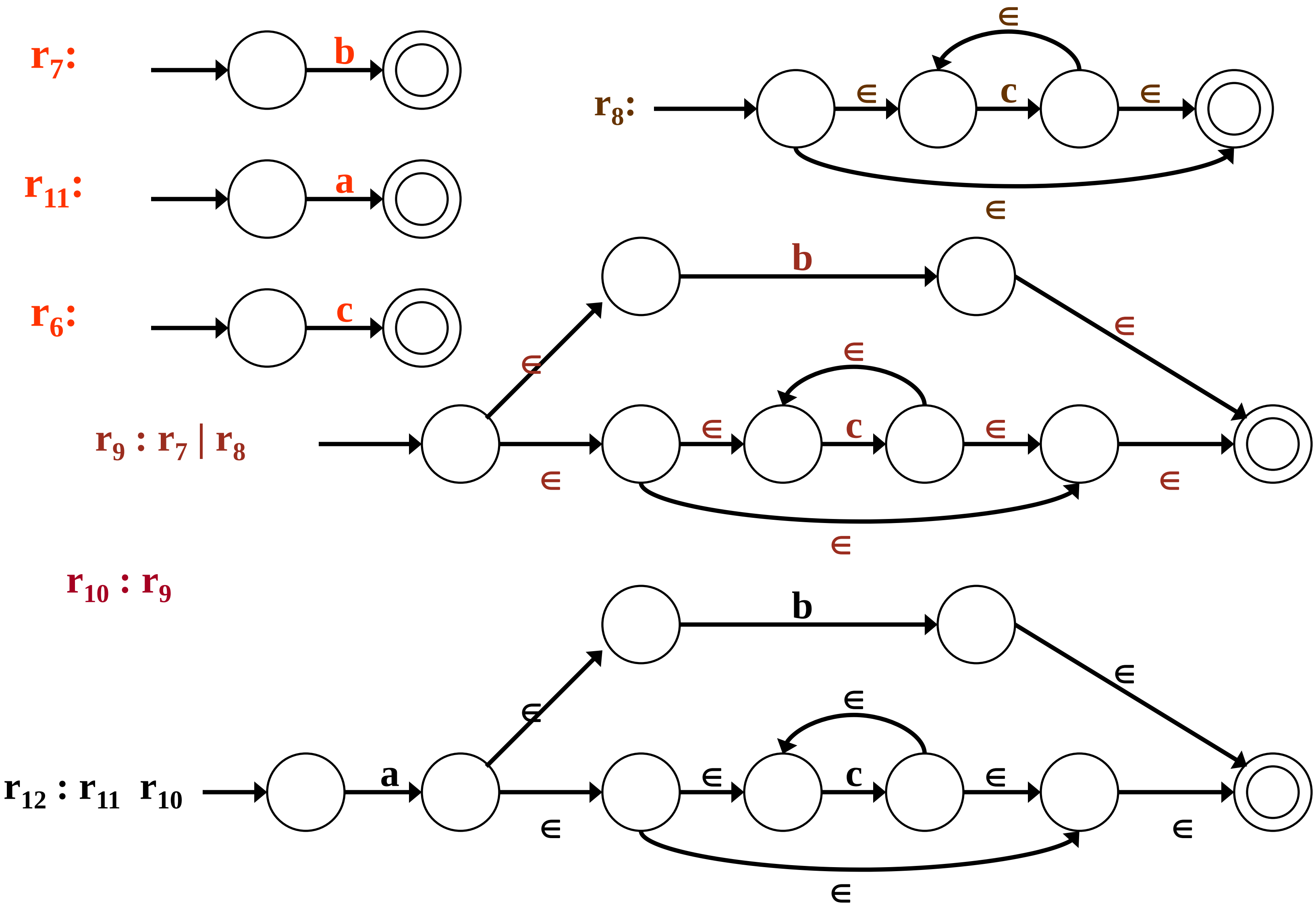



$r_5 : r_3\, r_4$



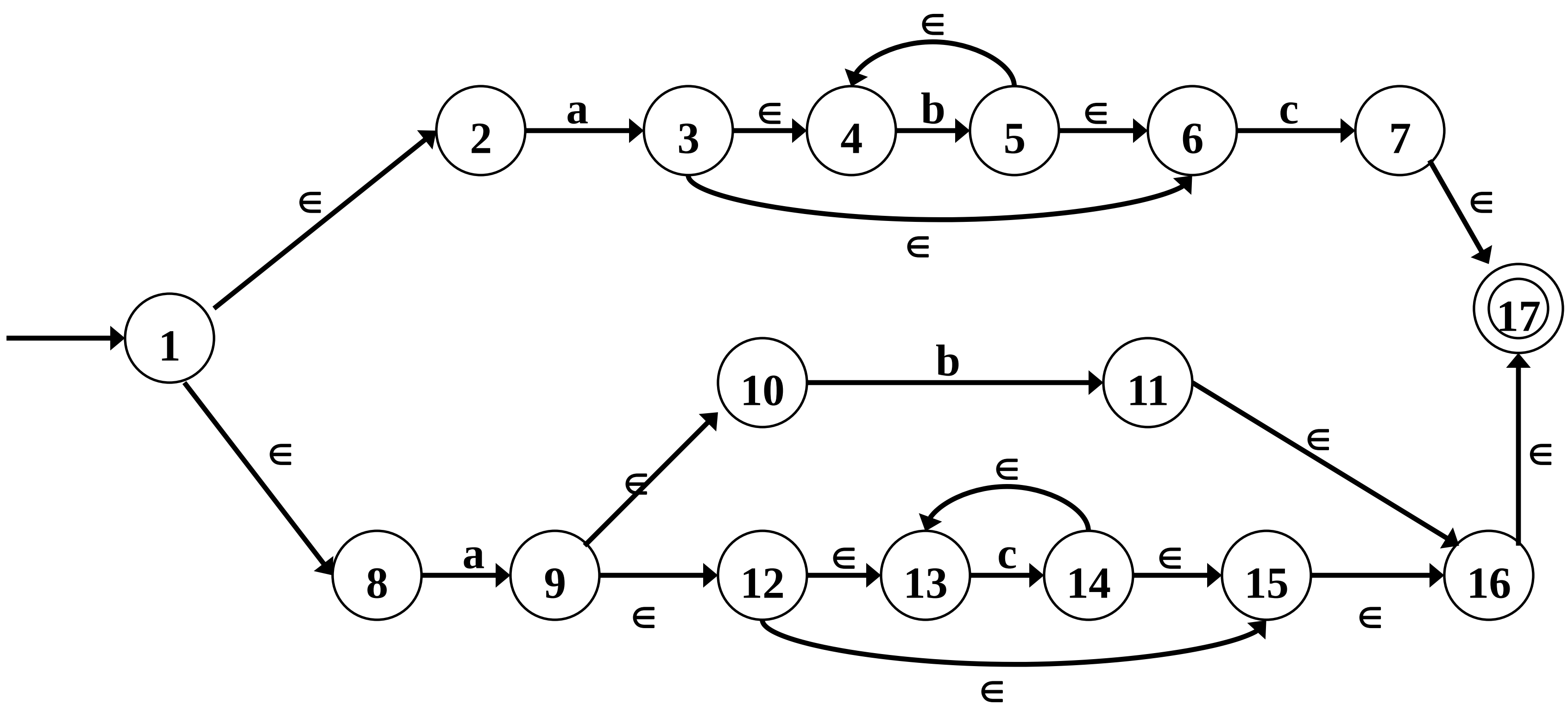
# Detailed Example – Construction(2)

**(ab*c) | (a(b|c*))**



# Detailed Example – Final Step

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



# Converting NFAs to DFAs

# Converting NFAs to DFAs (subset construction)

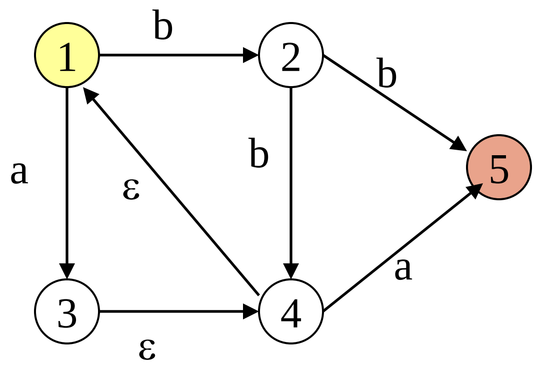
- **Idea**: Each state in the new DFA will correspond to some set of states from the NFA. The DFA will be in state $\{s_0, s_1, \ldots\}$ after input if the NFA could be in *any* of these states for the same input.

- **Input**: NFA N with state set $S_N$, alphabet $\Sigma$, start state $s_N$, final states $F_N$, transition function $T_N$: $S_N \times \{\Sigma \cup \varepsilon\} \rightarrow S_N$

- **Output**: DFA D with state set $S_D$, alphabet $\Sigma$, start state $s_D = \varepsilon\text{-closure}(s_N)$, final states $F_D$, transition function $T_D$: $S_D \times \Sigma \rightarrow S_D$

# Terminology: ε-closure

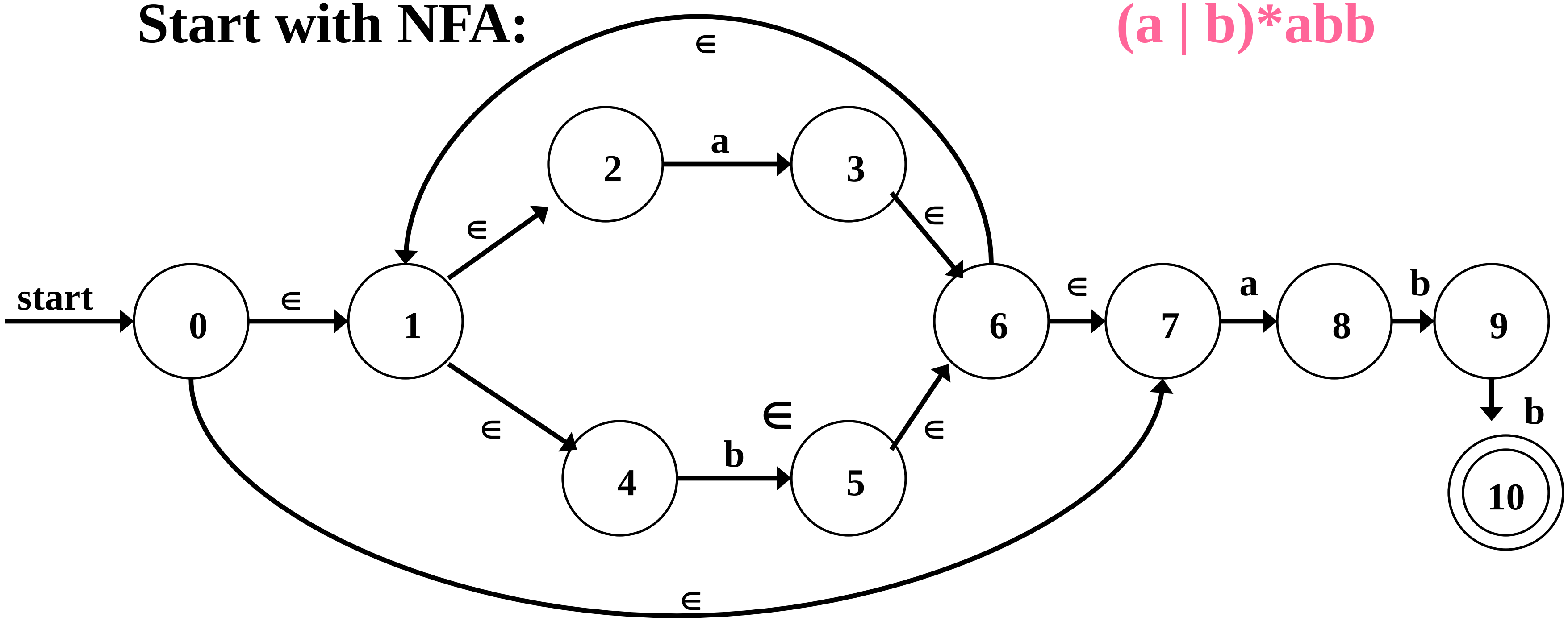
ε-closure(T) = T + all NFA states reachable from any state in T using only ε transitions.




ε-closure({1,2,5}) = {1,2,5}
ε-closure({4}) = {1,4}
ε-closure({3}) = {1,3,4}
ε-closure({3,5}) = {1,3,4,5}

# Illustrating Conversion – An Example

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




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

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

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

# Conversion Example – continued (1)

**$2^{nd}$, 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)

**$5^{th}$ , 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.**

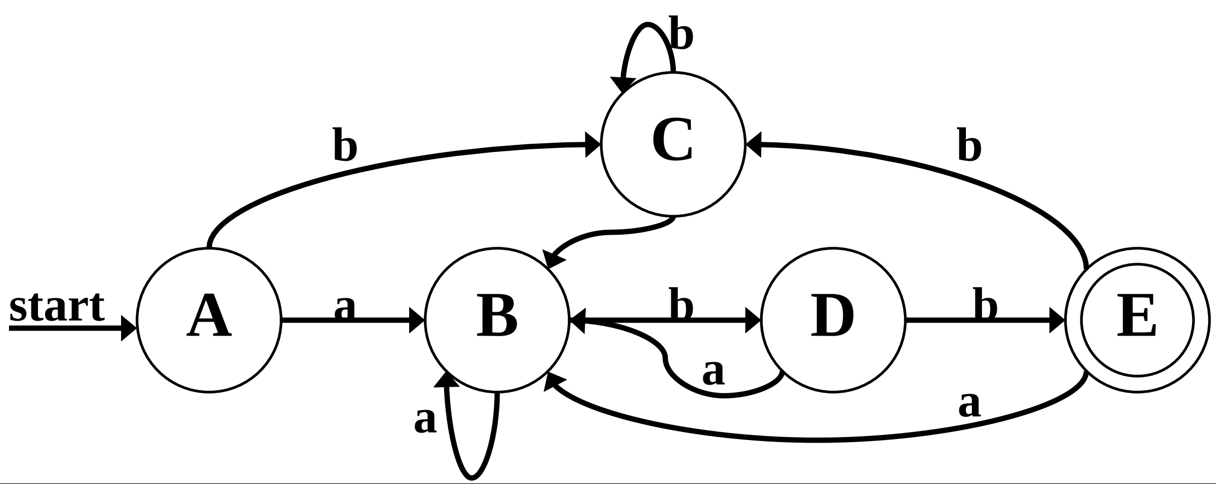
**b : ∈-closure(*move(E*,b)) = ∈-closure(*move(*{1,2,4,5,6,7,10},b))}**
**= {1,2,4,5,6,7} = C**

**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 $\in$-closure**

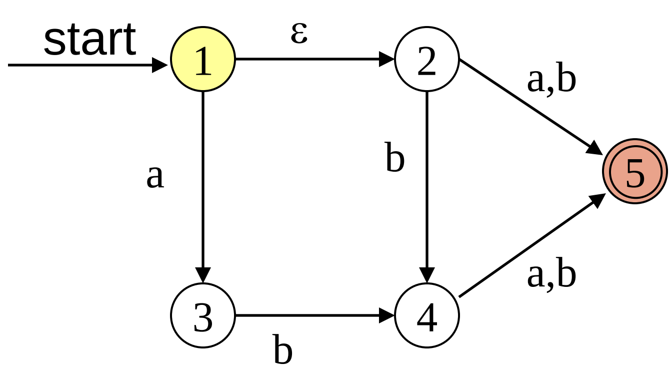
```
push all states in T onto stack;
initialize ∈-closure(T) to T;
while stack is not empty do begin
    pop t, the top element, off the stack;
    for each state u with edge from t to u labeled ∈ do
        if u is not in ∈-closure(T) do begin
            add u to ∈-closure(T) ;
            push u onto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

# Algorithm For Subset Construction – (2)

```
initially, ∈-closure(s0) 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 := ∈-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



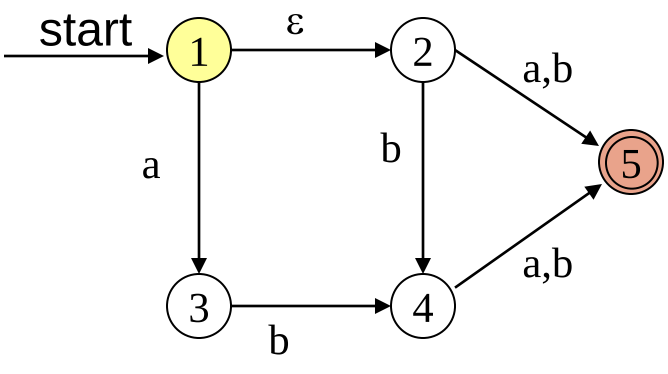

NFA N with

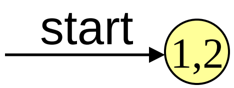
- State set $S_N = \{1,2,3,4,5\}$,
- Alphabet $\Sigma = \{a,b\}$
- Start state $s_N=1$,
- Final states $F_N=\{5\}$,
- Transition function $T_N$: $S_N \times \{\Sigma \cup \varepsilon\} \rightarrow S_N$

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

# Example 2: Subset Construction

NFA

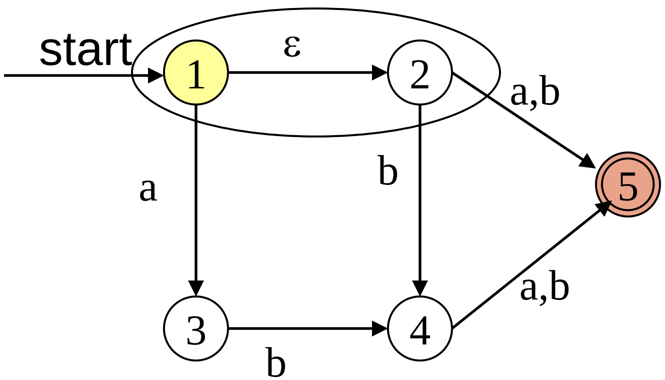


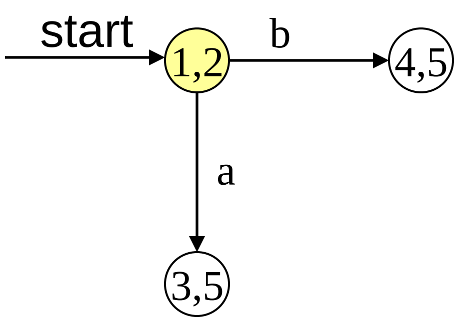




| T | ∈-closure(move(T, a)) | ∈-closure(move(T, b)) |
|---|---|---|
| {1,2} | | |
| | | |
| | | |
| | | |
| | | |

# Example 2: Subset Construction

NFA

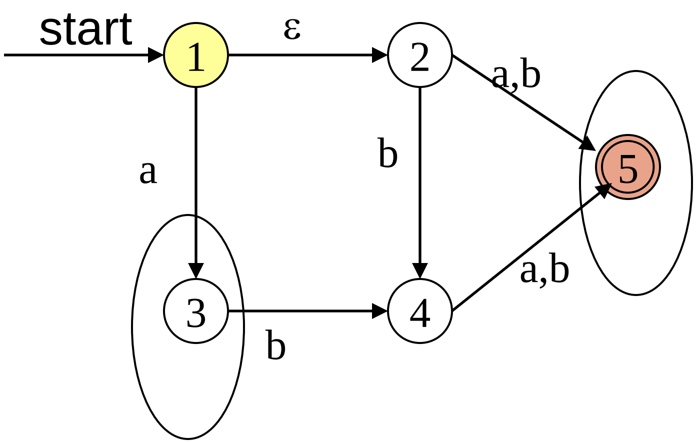


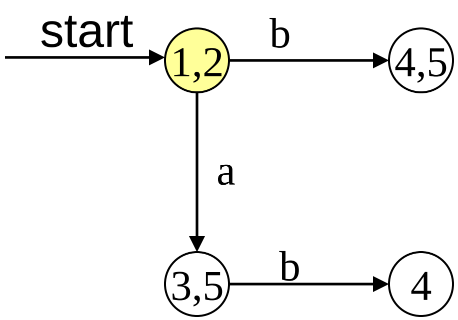




| T | ∈-closure(move(T, a)) | ∈-closure(move(T, b)) |
|---|---|---|
| {1,2} | {3,5} | {4,5} |
| {3,5} | | |
| {4,5} | | |
| | | |
| | | |

# Example 2: Subset Construction

NFA

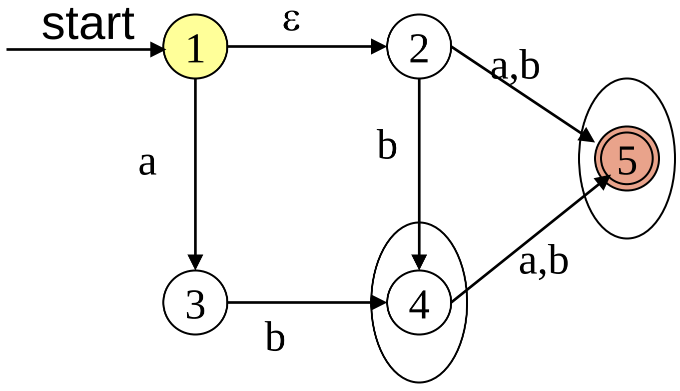


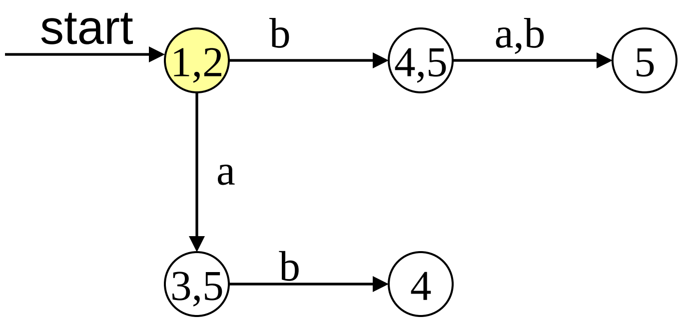




| T | ∈-closure(move(T, a)) | ∈-closure(move(T, b)) |
|---|---|---|
| {1,2} | {3,5} | {4,5} |
| {3,5} | - | {4} |
| {4,5} | | |
| {4} | | |
| | | |

# Example 2: Subset Construction

NFA

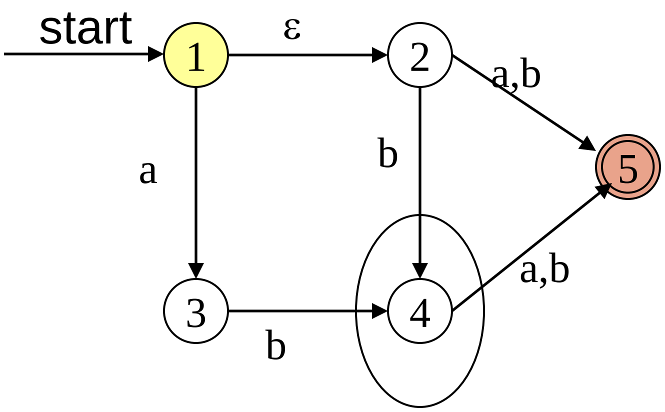


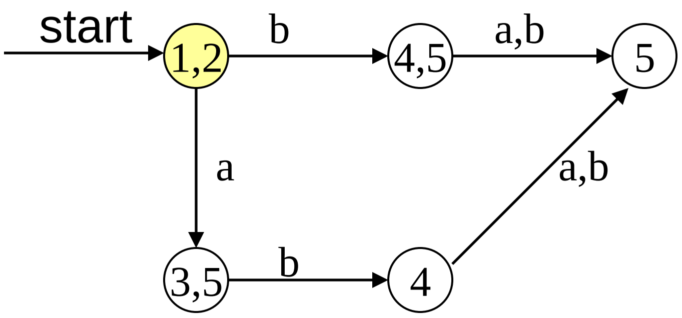




| T | ∈-closure(move(T, a)) | ∈-closure(move(T, b)) |
|---|---|---|
| {1,2} | {3,5} | {4,5} |
| {3,5} | - | {4} |
| {4,5} | {5} | {5} |
| {4} | | |
| {5} | | |

# Example 2: Subset Construction

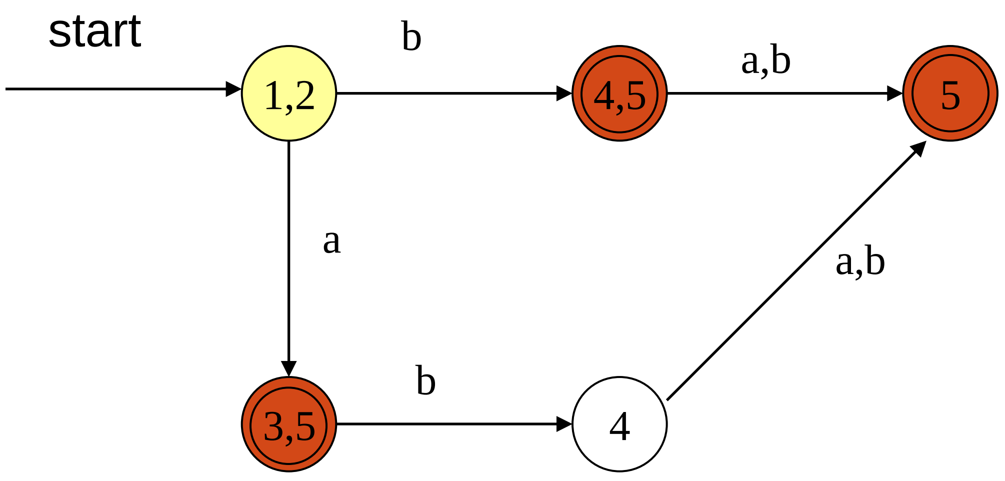
NFA



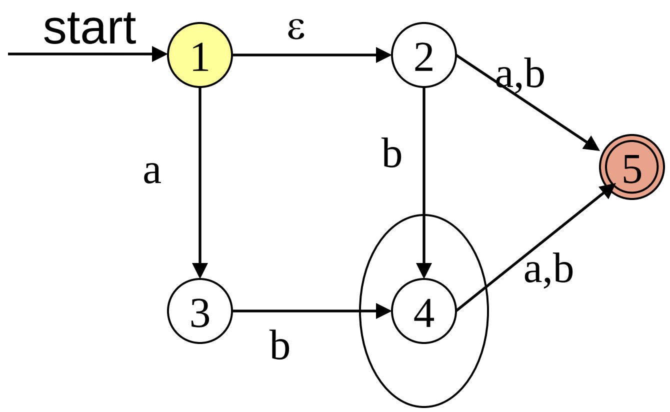




| T | ∈-closure(move(T, a)) | ∈-closure(move(T, b)) |
|---|---|---|
| {1,2} | {3,5} | {4,5} |
| {3,5} | - | {4} |
| {4,5} | {5} | {5} |
| {4} | {5} | {5} |
| {5} | - | - |

# Example 2: Subset Construction




NFA

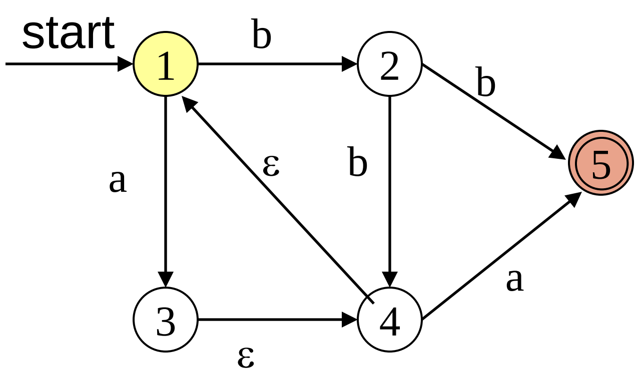



All final states since the
NFA final state is included

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

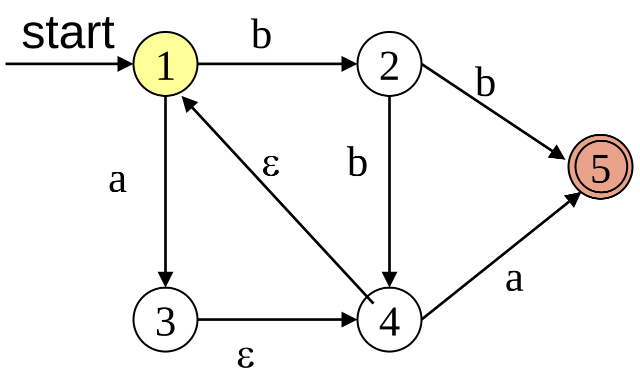
# Example 3: Subset Construction

NFA

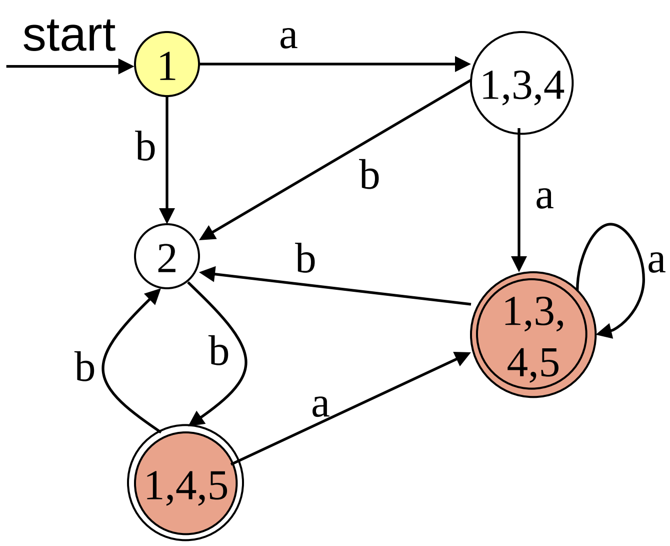


# Example 3: Subset Construction

NFA

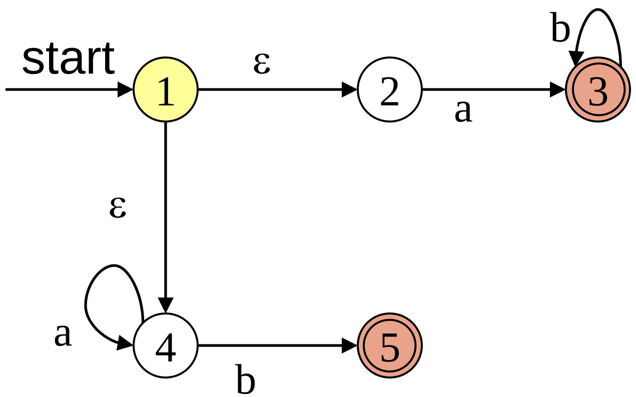



DFA

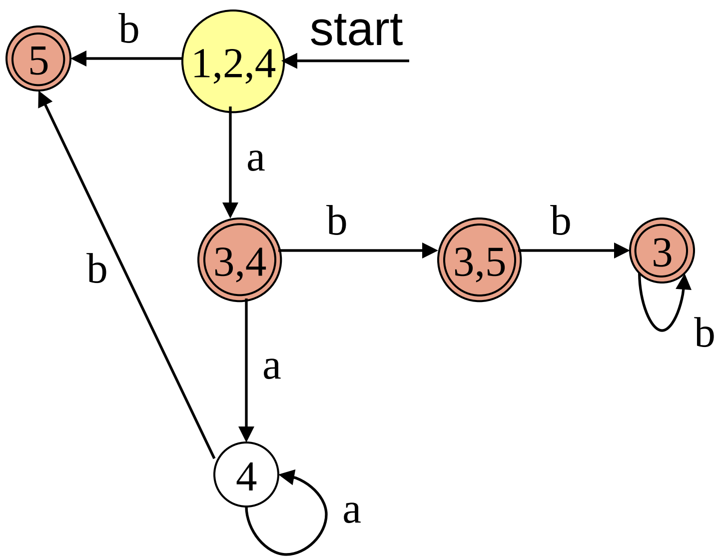


# Example 4: Subset Construction

NFA




DFA



**Thank You**

**Any Questions?**