

# Lexical Analysis and Lexical Analyzer Generators

## Chapter 3

AVAILABLE AT:

**Onebyzero Edu - Organized Learning, Smooth Career**

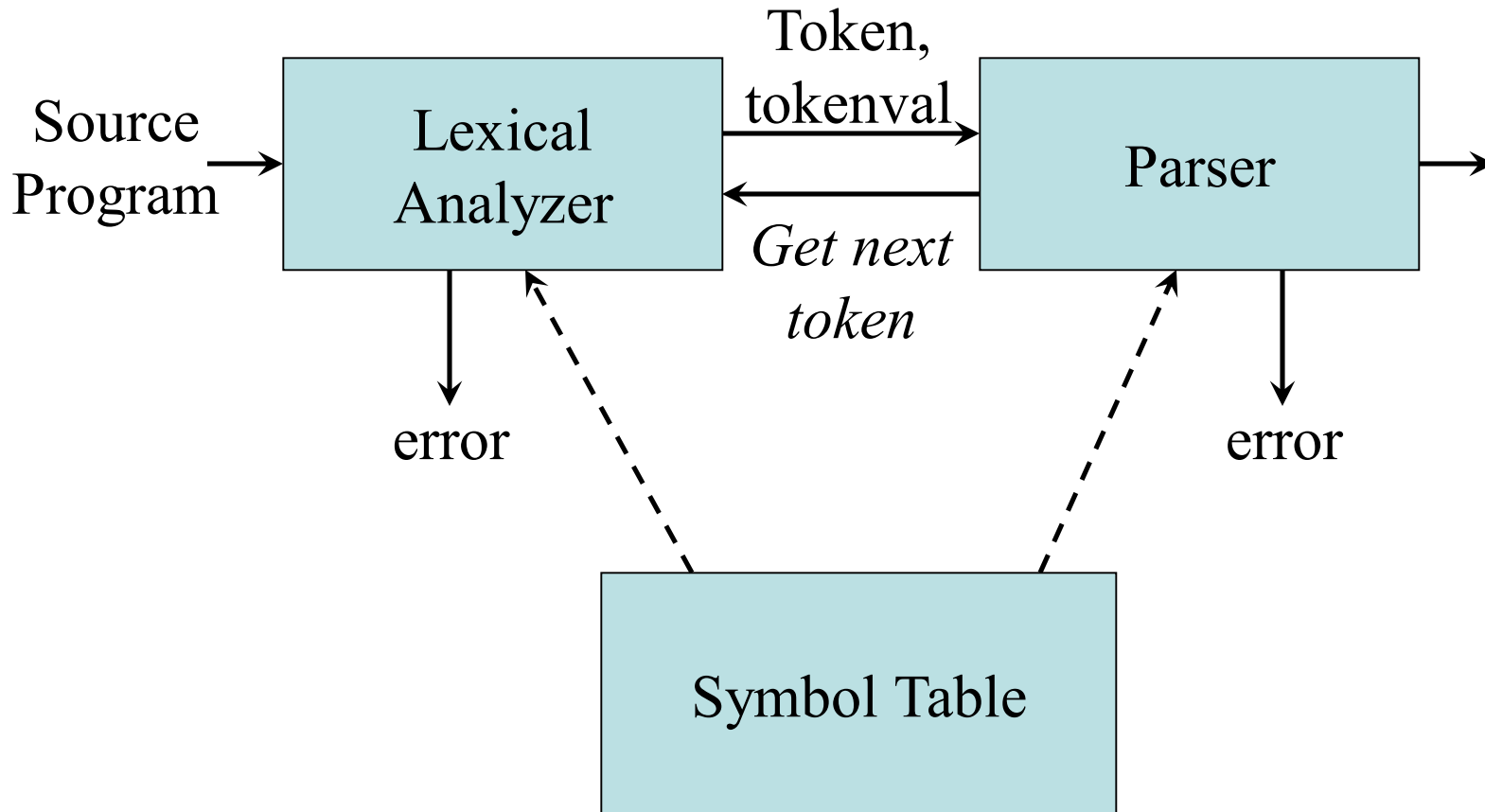
The Comprehensive Academic Study Platform for University Students in Bangladesh ([www.onebyzeroedu.com](http://www.onebyzeroedu.com))

Copyright Robert van Engelen, Florida State University

# The Reason Why Lexical Analysis is a Separate Phase

- Simplifies the design of the compiler
  - LL(1) or LR(1) parsing with 1 token lookahead would not be possible (multiple characters/tokens to match)
- Provides efficient implementation
  - Systematic techniques to implement lexical analyzers by hand or automatically from specifications
  - Stream buffering methods to scan input
- Improves portability
  - Non-standard symbols and alternate character encodings can be normalized (e.g. trigraphs)

# Interaction of the Lexical Analyzer with the Parser



# Attributes of Tokens

**y := 31 + 28\*x**

Lexical analyzer

**<id, “y”> <assign, > <num, 31> <+, > <num, 28> <\*, > <id, “x”>**

token

**tokenval**  
(token attribute)

Parser

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh ([www.onebyzeroedu.com](http://www.onebyzeroedu.com))

# Tokens, Patterns, and Lexemes

- A *token* is a classification of lexical units
  - For example: **id** and **num**
- *Lexemes* are the specific character strings that make up a token
  - For example: **abc** and **123**
- *Patterns* are rules describing the set of lexemes belonging to a token
  - For example: “*letter followed by letters and digits*” and “*non-empty sequence of digits*”

# Specification of Patterns for Tokens: *Definitions*

- An *alphabet*  $\Sigma$  is a finite set of symbols (characters)
- A *string*  $s$  is a finite sequence of symbols from  $\Sigma$ 
  - $|s|$  denotes the length of string  $s$
  - $\varepsilon$  denotes the empty string, thus  $|\varepsilon| = 0$
- A *language* is a specific set of strings over some fixed alphabet  $\Sigma$

# Specification of Patterns for Tokens: *String Operations*

- The *concatenation* of two strings  $x$  and  $y$  is denoted by  $xy$
- The *exponentiation* of a string  $s$  is defined by

$$s^0 = \varepsilon$$

$$s^i = s^{i-1}s \quad \text{for } i > 0$$

note that  $s\varepsilon = \varepsilon s = s$

AVAILABLE AT:

**Onebyzero Edu - Organized Learning, Smooth Career**

The Comprehensive Academic Study Platform for University Students in Bangladesh ([www.onebyzeroedu.com](http://www.onebyzeroedu.com))

# Specification of Patterns for Tokens: *Language Operations*

- *Union*

$$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$$

- *Concatenation*

$$LM = \{xy \mid x \in L \text{ and } y \in M\}$$

- *Exponentiation*

$$L^0 = \{\varepsilon\}; \quad L^i = L^{i-1}L$$

- *Kleene closure*

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

- *Positive closure*

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



# Specification of Patterns for Tokens: *Regular Expressions*

- Basis symbols:
  - $\varepsilon$  is a regular expression denoting language  $\{\varepsilon\}$
  - $a \in \Sigma$  is a regular expression denoting  $\{a\}$
- If  $r$  and  $s$  are regular expressions denoting languages  $L(r)$  and  $M(s)$  respectively, then
  - $r|s$  is a regular expression denoting  $L(r) \cup M(s)$
  - $rs$  is a regular expression denoting  $L(r)M(s)$
  - $r^*$  is a regular expression denoting  $L(r)^*$
  - $(r)$  is a regular expression denoting  $L(r)$
- A language defined by a regular expression is called a *regular set*

# Specification of Patterns for Tokens: *Regular Definitions*

- Regular definitions introduce a naming convention:

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

...

$$d_n \rightarrow r_n$$

where each  $r_i$  is a regular expression over

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

- Any  $d_j$  in  $r_i$  can be textually substituted in  $r_i$  to obtain an equivalent set of definitions

# Specification of Patterns for Tokens: *Regular Definitions*

- Example:

**letter**  $\rightarrow$  **A** | **B** | ... | **Z** | **a** | **b** | ... | **z**

**digit**  $\rightarrow$  **0** | **1** | ... | **9**

**id**  $\rightarrow$  **letter** ( **letter** | **digit** )<sup>\*</sup>

- Regular definitions are not recursive:

**digits**  $\rightarrow$  **digit digits** | **digit**      *wrong!*

# Specification of Patterns for Tokens: *Notational Shorthand*

- The following shorthands are often used:

$$\begin{aligned}
 r^+ &= rr^* \\
 r^? &= r \mid \varepsilon \\
 [\mathbf{a-z}] &= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \dots \mid \mathbf{z}
 \end{aligned}$$

- Examples:  
**digit**  $\rightarrow [0-9]$   
**num**  $\rightarrow \mathbf{digit}^+ (.\mathbf{digit}^+)? (\mathbf{E} (+ \mid -)? \mathbf{digit}^+ )?$

# Regular Definitions and Grammars

Grammar

$$\begin{aligned} \text{stmt} &\rightarrow \text{if } \text{expr} \text{ then } \text{stmt} \\ &\quad | \text{if } \text{expr} \text{ then } \text{stmt} \text{ else } \text{stmt} \\ &\quad | \epsilon \end{aligned}$$

$$\begin{aligned} \text{expr} &\rightarrow \text{term relop term} \\ &\quad | \text{term} \end{aligned}$$

$$\begin{aligned} \text{term} &\rightarrow \text{id} \\ &\quad | \text{num} \end{aligned}$$

Regular definitions

$$\text{if} \rightarrow \text{if}$$

$$\text{then} \rightarrow \text{then}$$

$$\text{else} \rightarrow \text{else}$$

$$\text{relop} \rightarrow < \mid <= \mid <> \mid > \mid >= \mid =$$

$$\text{id} \rightarrow \text{letter ( letter } \mid \text{ digit )}^*$$

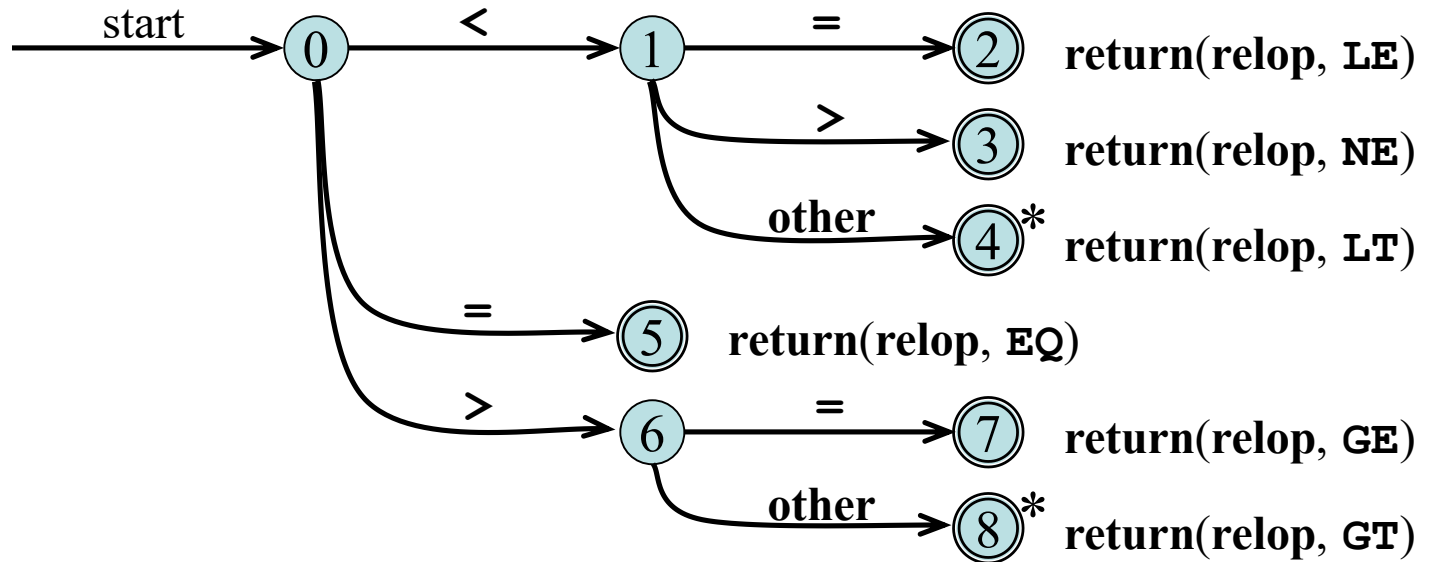
$$\text{num} \rightarrow \text{digit}^+ ( \text{digit}^+ )? ( \text{E ( + } \mid \text{ - )? digit}^+ )?$$

AVAILABLE AT:

Onebyzero Edu - Online Learning for Your Career  
The Comprehensive Academic Study Platform for University Students in Bangladesh ([www.onebyzeroedu.com](http://www.onebyzeroedu.com))

# Coding Regular Definitions in *Transition Diagrams*

**relop**  $\rightarrow < \mid <= \mid <> \mid > \mid >= \mid =$



**id**  $\rightarrow \text{letter} ( \text{letter} \mid \text{digit} )^*$



# Coding Regular Definitions in Transition Diagrams: Code

```

token nexttoken()
{ while (1) {
    switch (state) {
    case 0: c = nextchar();
        if (c==blank || c==tab || c==newline) {
            state = 0;
            lexeme_beginning++;
        }
        else if (c=='<') state = 1;
        else if (c=='=') state = 5;
        else if (c=='>') state = 6;
        else state = fail();
        break;
    case 1:
        ...
    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;
    }
}

```

Decides the  
next start state  
to check



```

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

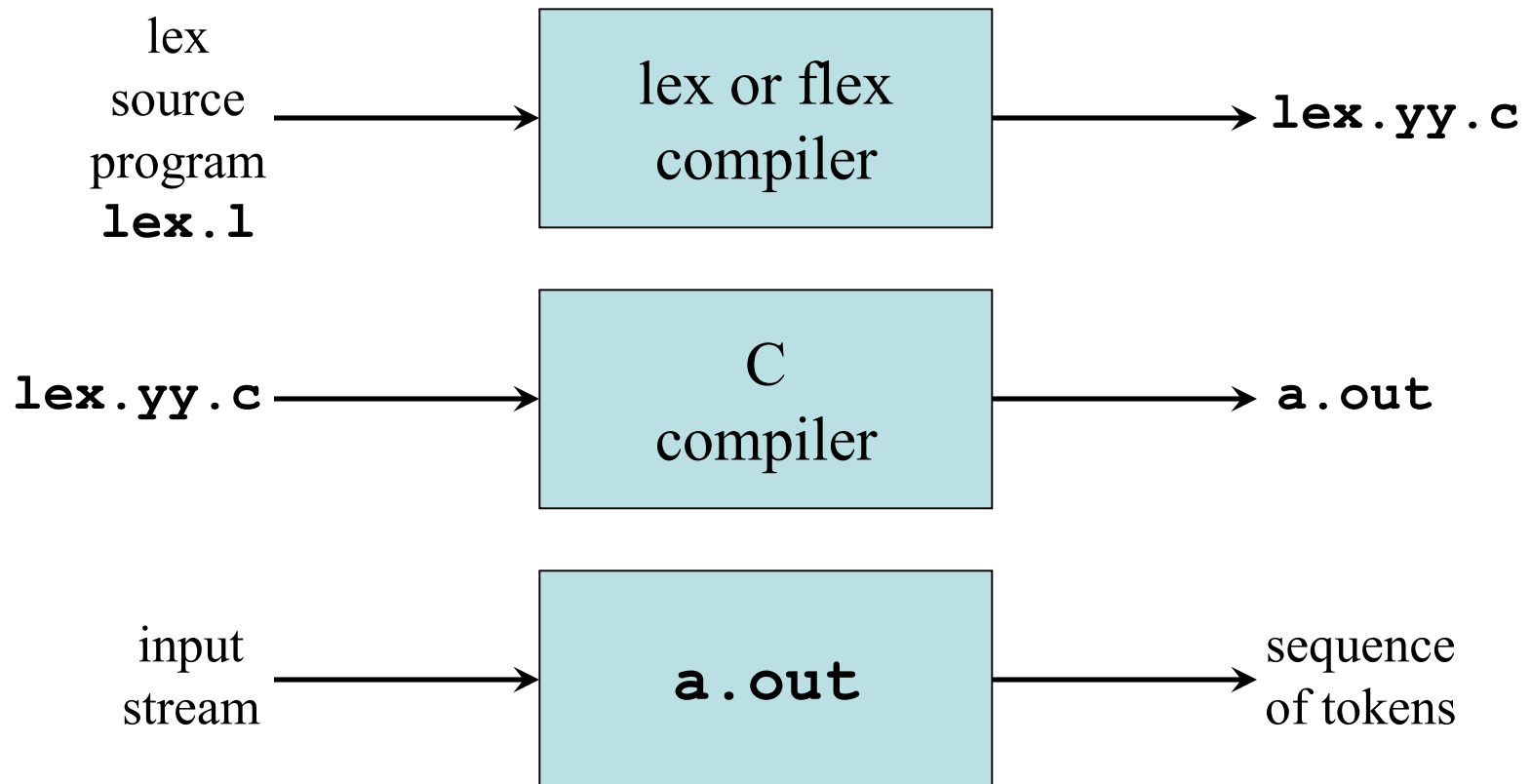
```

# The Lex and Flex Scanner Generators

- *Lex* and its newer cousin *flex* are scanner generators
- Systematically translate regular definitions into C source code for efficient scanning
- Generated code is easy to integrate in C applications



# Creating a Lexical Analyzer with Lex and Flex



# Lex Specification

- A *lex specification* consists of three parts:  
*regular definitions, C declarations in % { % }*  
*% %*  
*translation rules*  
*% %*  
*user-defined auxiliary procedures*
- The *translation rules* are of the form:  

$$\begin{array}{ll} p_1 & \{ action_1 \} \\ p_2 & \{ action_2 \} \\ \dots & \\ p_n & \{ action_n \} \end{array}$$

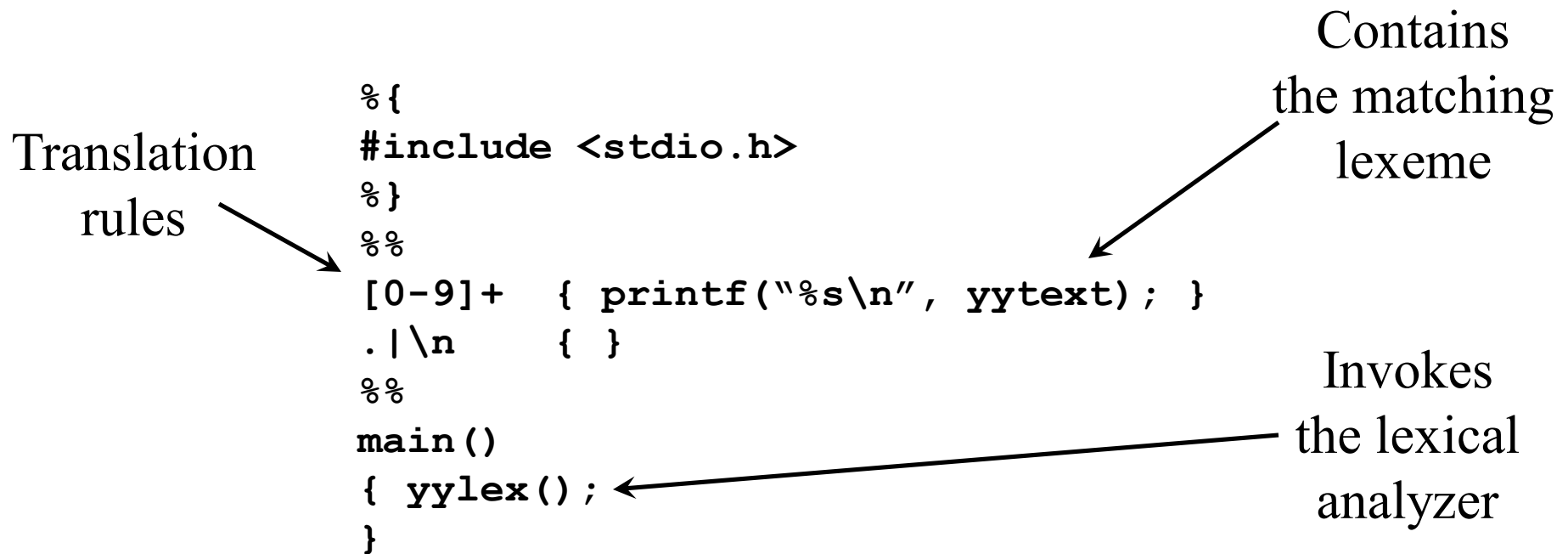
AVAILABLE AT:

# Regular Expressions in Lex

<b>x</b>	match the character <b>x</b>
<b>\.</b>	match the character <b>.</b>
<b>"string"</b>	match contents of string of characters
<b>.</b>	match any character except newline
<b>^</b>	match beginning of a line
<b>\$</b>	match the end of a line
<b>[xyz]</b>	match one character <b>x</b> , <b>y</b> , or <b>z</b> (use <b>\</b> to escape <b>-</b> )
<b>[^xyz]</b>	match any character except <b>x</b> , <b>y</b> , and <b>z</b>
<b>[a-z]</b>	match one of <b>a</b> to <b>z</b>
<b>r*</b>	closure (match zero or more occurrences)
<b>r+</b>	positive closure (match one or more occurrences)
<b>r?</b>	optional (match zero or one occurrence)
<b>r<sub>1</sub>r<sub>2</sub></b>	match <b>r<sub>1</sub></b> then <b>r<sub>2</sub></b> (concatenation)
<b>r<sub>1</sub>   r<sub>2</sub></b>	match <b>r<sub>1</sub></b> or <b>r<sub>2</sub></b> (union)
<b>( r )</b>	grouping
<b>r<sub>1</sub> \ r<sub>2</sub></b>	match <b>r<sub>1</sub></b> when followed by <b>r<sub>2</sub></b>

AVAILABLE AT:

# Example Lex Specification 1



```

lex spec.1
gcc lex.yy.c -ll
./a.out < spec.1

```

# Example Lex Specification 2

Translation  
rules



```
%{
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
}%
delim      [ \t]+
%%
\n          { ch++; wd++; nl++; }
^{delim}    { ch+=yyleng; }
{delim}     { ch+=yyleng; wd++; }
.           { ch++; }
%%
main()
{ yylex();
  printf("%8d%8d%8d\n", nl, wd, ch);
}
```

Regular  
definition



# Example Lex Specification 3

Translation rules

Regular definitions

```
%{
#include <stdio.h>
}%
digit      [0-9]
letter     [A-Za-z]
id         {letter}({letter}|{digit})*
%%
{digit}+   { printf("number: %s\n", yytext); }
{id}       { printf("ident: %s\n", yytext); }
.          { printf("other: %s\n", yytext); }
%%
main()
{ yylex();
}
```

# Example Lex Specification 4

```
%{ /* definitions of manifest constants */
#define LT (256)
...
%}
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
{ws}       { }
if         {return IF;}
then       {return THEN;}
else       {return ELSE;}
{id}       {yylval = install_id(); return ID;}
{number}   {yylval = install_num(); return NUMBER;}
"<"        {yylval = LT; return RELOP;}
"<="       {yylval = LE; return RELOP;}
"="        {yylval = EQ; return RELOP;}
"<>"       {yylval = NE; return RELOP;}
">"        {yylval = GT; return RELOP;}
">="       {yylval = GE; return RELOP;}
%%
```

Return  
token to  
parser

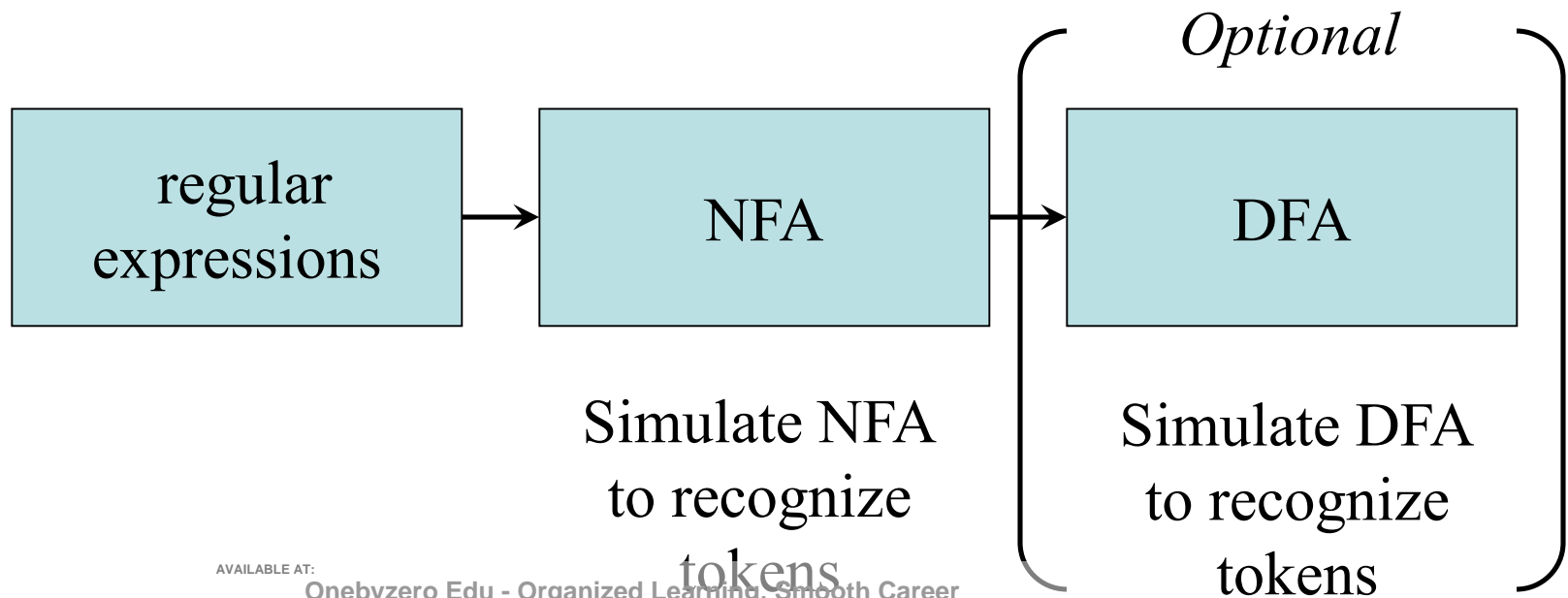
Token  
attribute

Install **yytext** as  
identifier in symbol table

```
int install_id()
```

# Design of a Lexical Analyzer Generator

- Translate regular expressions to NFA
- Translate NFA to an efficient DFA





# Nondeterministic Finite Automata

- An NFA is a 5-tuple  $(S, \Sigma, \delta, s_0, F)$  where

$S$  is a finite set of *states*

$\Sigma$  is a finite set of symbols, the *alphabet*

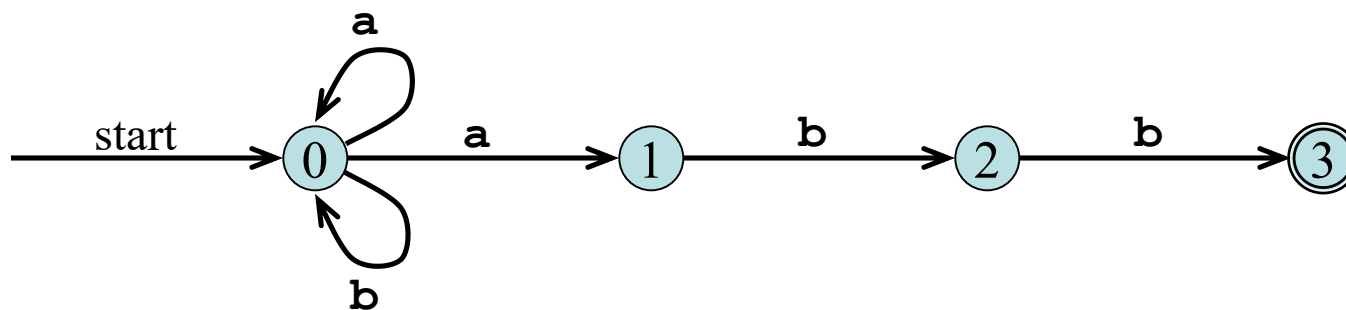
$\delta$  is a *mapping* from  $S \times \Sigma$  to a set of states

$s_0 \in S$  is the *start state*

$F \subseteq S$  is the set of *accepting* (or *final*) *states*

# Transition Graph

- An NFA can be diagrammatically represented by a labeled directed graph called a *transition graph*



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

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

$$s_0 = 0$$

$$F = \{3\}$$

# Transition Table

- The mapping  $\delta$  of an NFA can be represented in a *transition table*

$$\delta(0, \mathbf{a}) = \{0, 1\}$$

$$\delta(0, \mathbf{b}) = \{0\}$$

$$\delta(1, \mathbf{b}) = \{2\}$$

$$\delta(2, \mathbf{b}) = \{3\}$$



<i>State</i>	<i>Input</i> <b>a</b>	<i>Input</i> <b>b</b>
0	{0, 1}	{0}
1		{2}
2		{3}

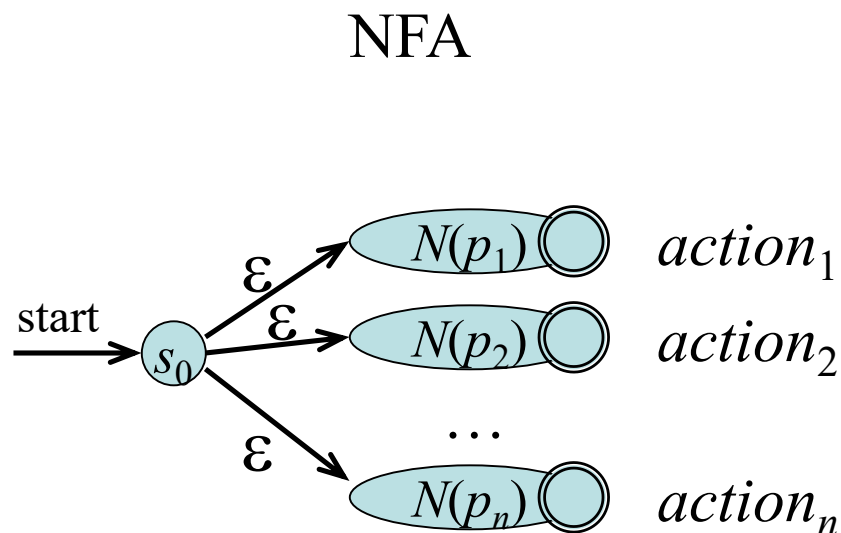
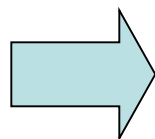
# The Language Defined by an NFA

- An NFA *accepts* an input string  $x$  if and only if there is some path with edges labeled with symbols from  $x$  in sequence from the start state to some accepting state in the transition graph
- A state transition from one state to another on the path is called a *move*
- The *language defined by* an NFA is the set of input strings it accepts, such as  $(\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}$  for the example NFA

# Design of a Lexical Analyzer Generator: RE to NFA to DFA

Lex specification with regular expressions

$p_1$	$\{ action_1 \}$
$p_2$	$\{ action_2 \}$
...	
$p_n$	$\{ action_n \}$

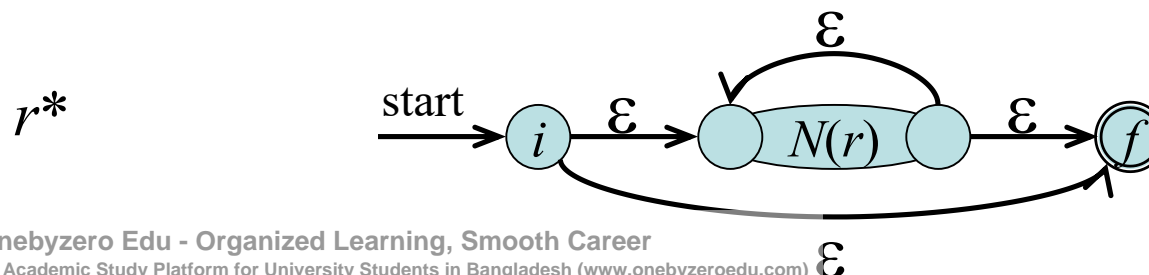
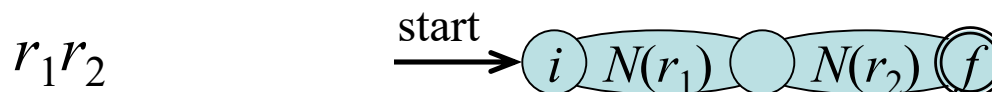
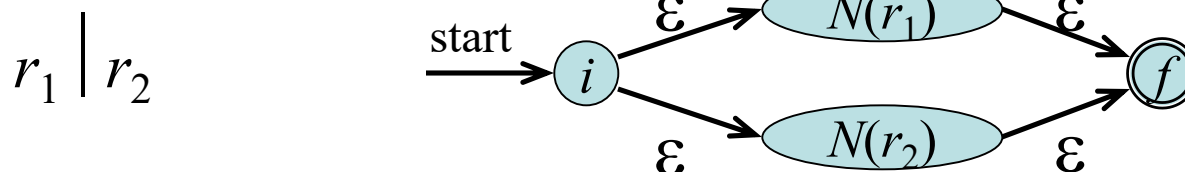
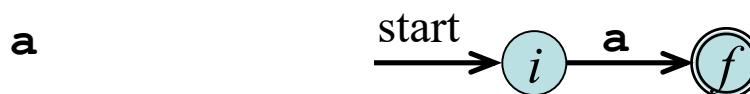
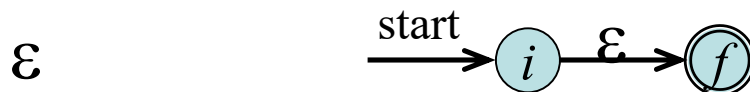


*Subset construction*

DFA

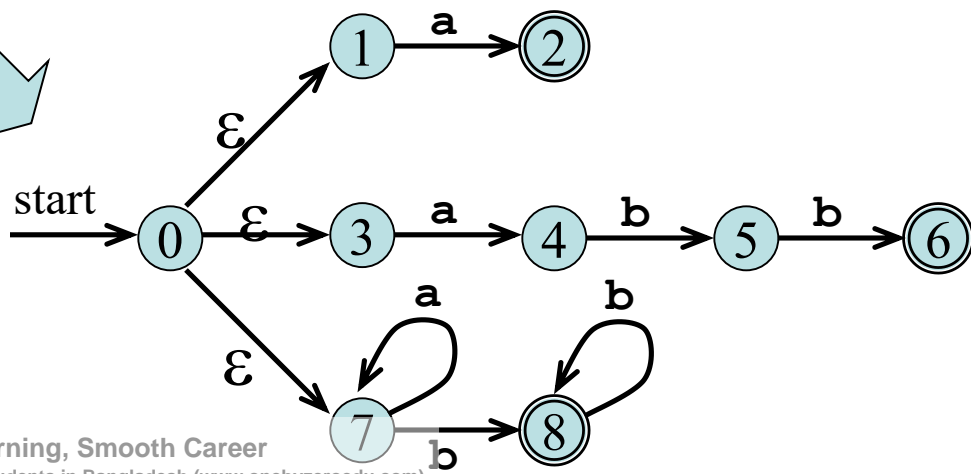
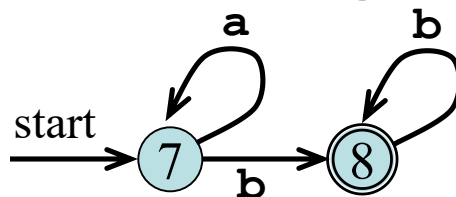
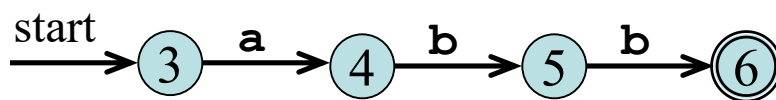
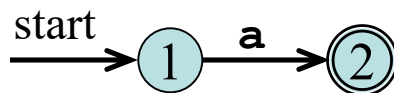
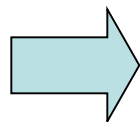
# From Regular Expression to NFA

## (Thompson's Construction)



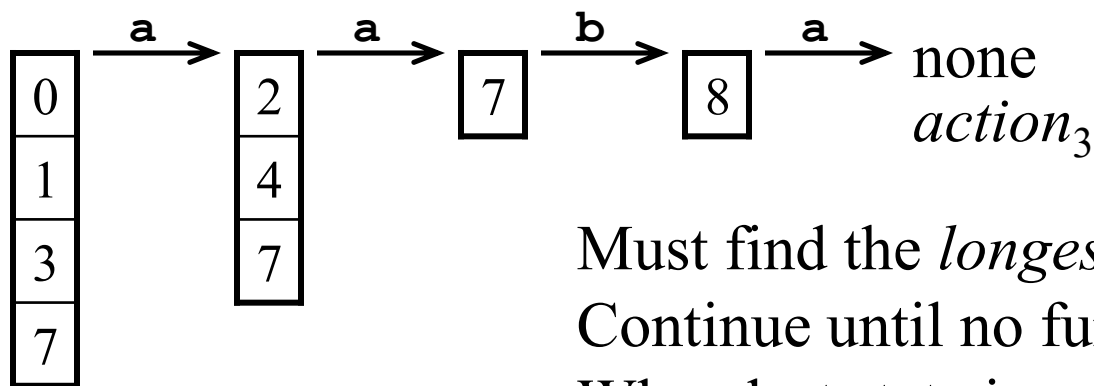
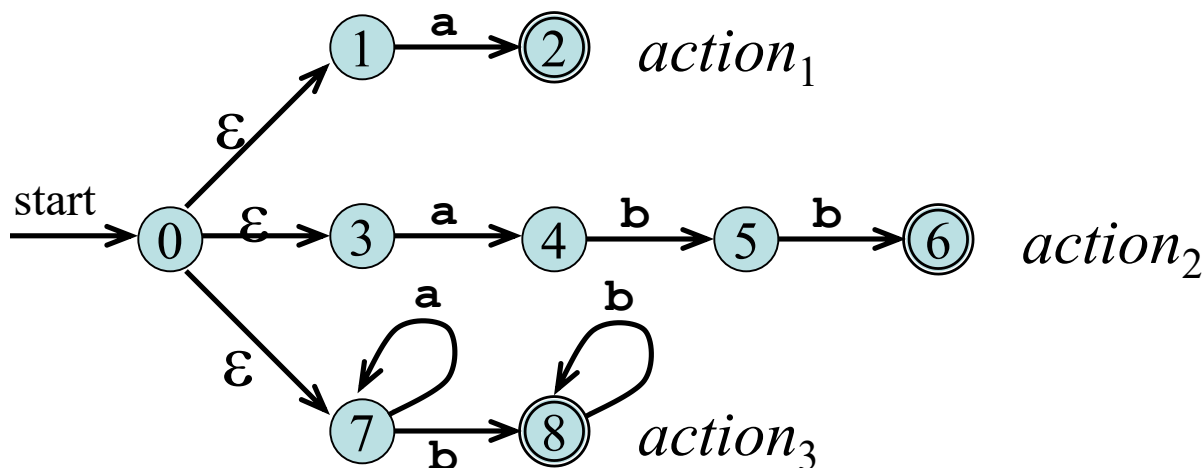
# Combining the NFAs of a Set of Regular Expressions

**a**       $\{ action_1 \}$   
**abb**     $\{ action_2 \}$   
**a\*b+**    $\{ action_3 \}$



# Simulating the Combined NFA

## Example 1



Must find the *longest match*:

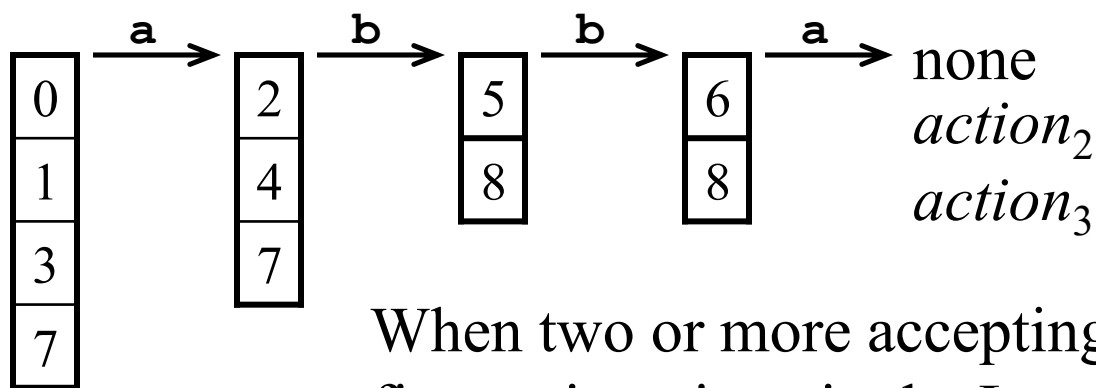
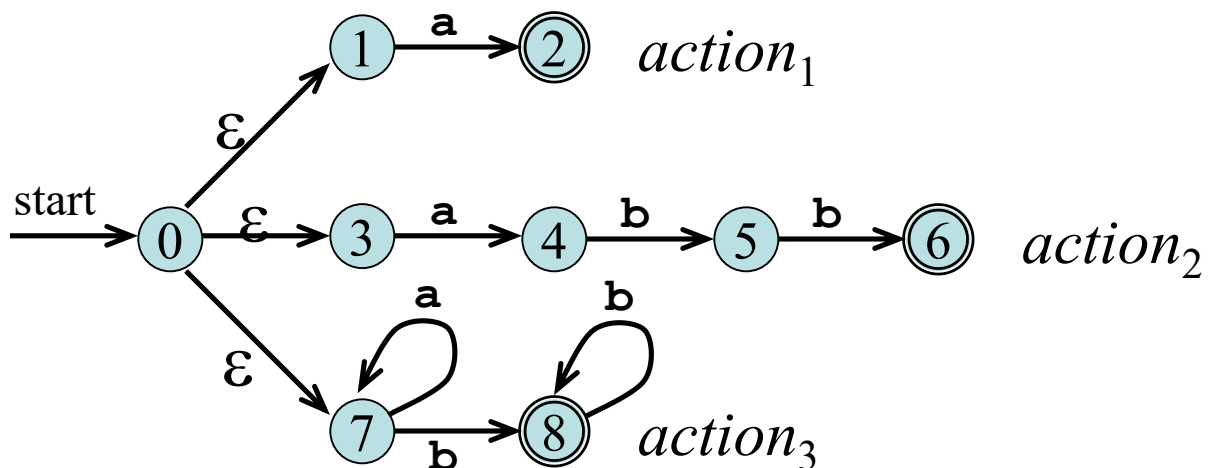
Continue until no further moves are possible

When last state is accepting: execute action



# Simulating the Combined NFA

## Example 2



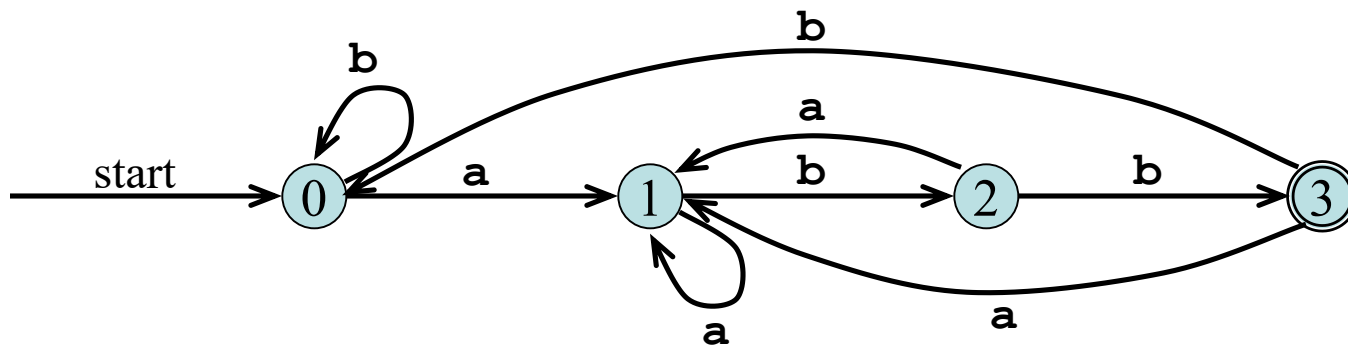
When two or more accepting states are reached, the first action given in the Lex specification is executed

# Deterministic Finite Automata

- A *deterministic finite automaton* is a special case of an NFA
  - No state has an  $\epsilon$ -transition
  - For each state  $s$  and input symbol  $a$  there is at most one edge labeled  $a$  leaving  $s$
- Each entry in the transition table is a single state
  - At most one path exists to accept a string
  - Simulation algorithm is simple

# Example DFA

A DFA that accepts  $(a \mid b)^*abb$



# Conversion of an NFA into a DFA

- The *subset construction algorithm* converts an NFA into a DFA using:

$$\varepsilon\text{-closure}(s) = \{s\} \cup \{t \mid s \rightarrow_{\varepsilon} \dots \rightarrow_{\varepsilon} t\}$$

$$\varepsilon\text{-closure}(T) = \bigcup_{s \in T} \varepsilon\text{-closure}(s)$$

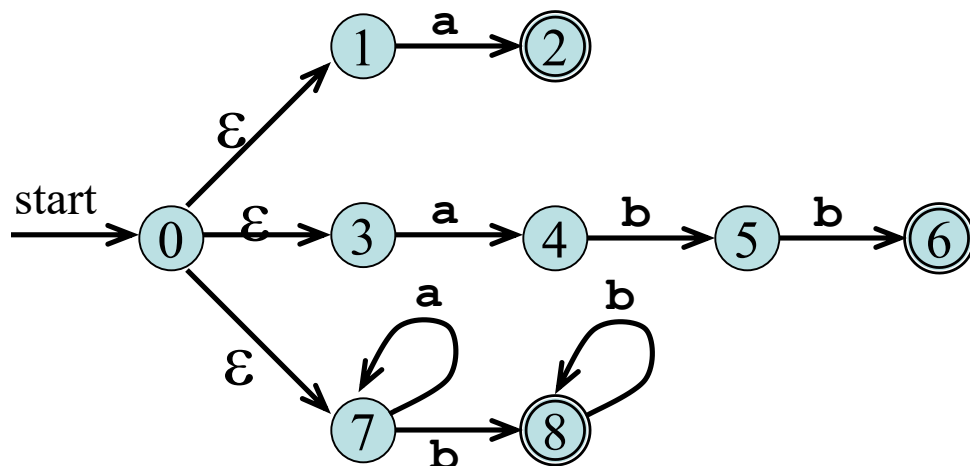
$$\text{move}(T, a) = \{t \mid s \rightarrow_a t \text{ and } s \in T\}$$

- The algorithm produces:

*Dstates* is the set of states of the new DFA  
consisting of sets of states of the NFA

*Dtran* is the transition table of the new DFA

# $\epsilon$ -closure and move Examples



$$\epsilon\text{-closure}(\{0\}) = \{0, 1, 3, 7\}$$

$$\text{move}(\{0, 1, 3, 7\}, \mathbf{a}) = \{2, 4, 7\}$$

$$\epsilon\text{-closure}(\{2, 4, 7\}) = \{2, 4, 7\}$$

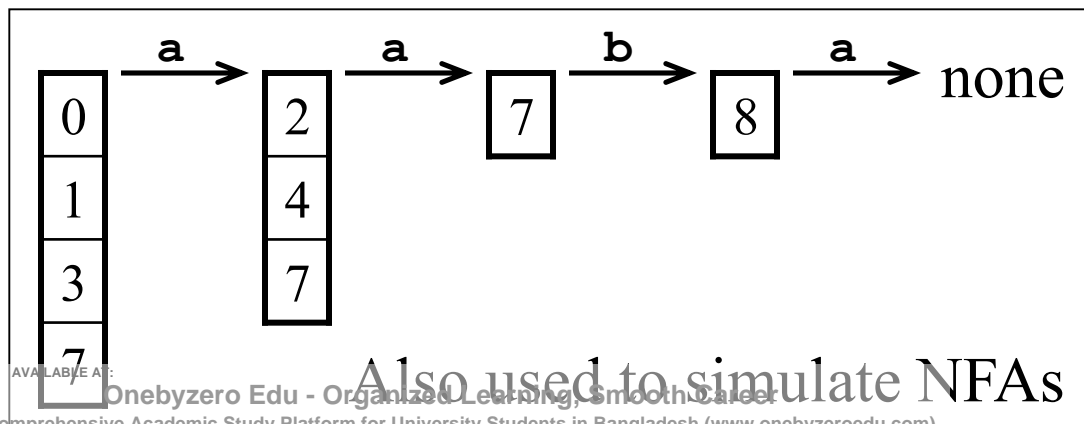
$$\text{move}(\{2, 4, 7\}, \mathbf{a}) = \{7\}$$

$$\epsilon\text{-closure}(\{7\}) = \{7\}$$

$$\text{move}(\{7\}, \mathbf{b}) = \{8\}$$

$$\epsilon\text{-closure}(\{8\}) = \{8\}$$

$$\text{move}(\{8\}, \mathbf{a}) = \emptyset$$



Also used to simulate NFAs

# Simulating an NFA using $\varepsilon$ -closure and *move*

```

 $S := \varepsilon\text{-closure}(\{s_0\})$ 
 $S_{prev} := \emptyset$ 
 $a := \text{nextchar}()$ 
while  $S \neq \emptyset$  do
     $S_{prev} := S$ 
     $S := \varepsilon\text{-closure}(\text{move}(S, a))$ 
     $a := \text{nextchar}()$ 
end do
if  $S_{prev} \cap F \neq \emptyset$  then
    execute action in  $S_{prev}$ 
    return “yes”
else    return “no”
  
```

# The Subset Construction Algorithm

Initially,  $\varepsilon\text{-closure}(s_0)$  is the only state in  $Dstates$  and it is unmarked  
**while** there is an unmarked state  $T$  in  $Dstates$  **do**

mark  $T$

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

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

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

add  $U$  as an unmarked state to  $Dstates$

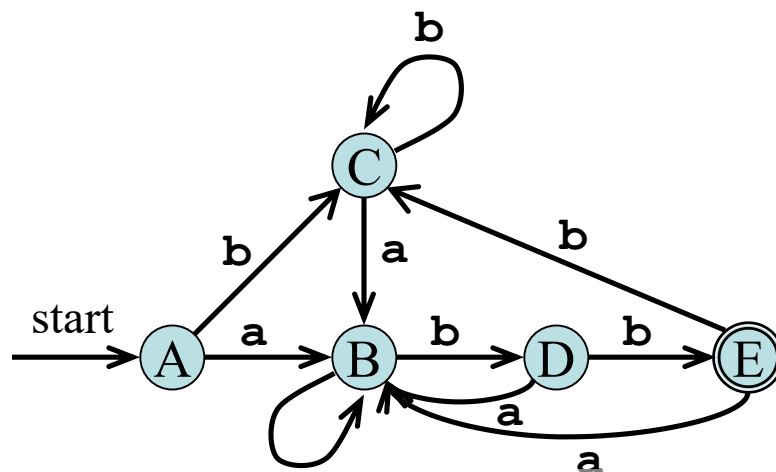
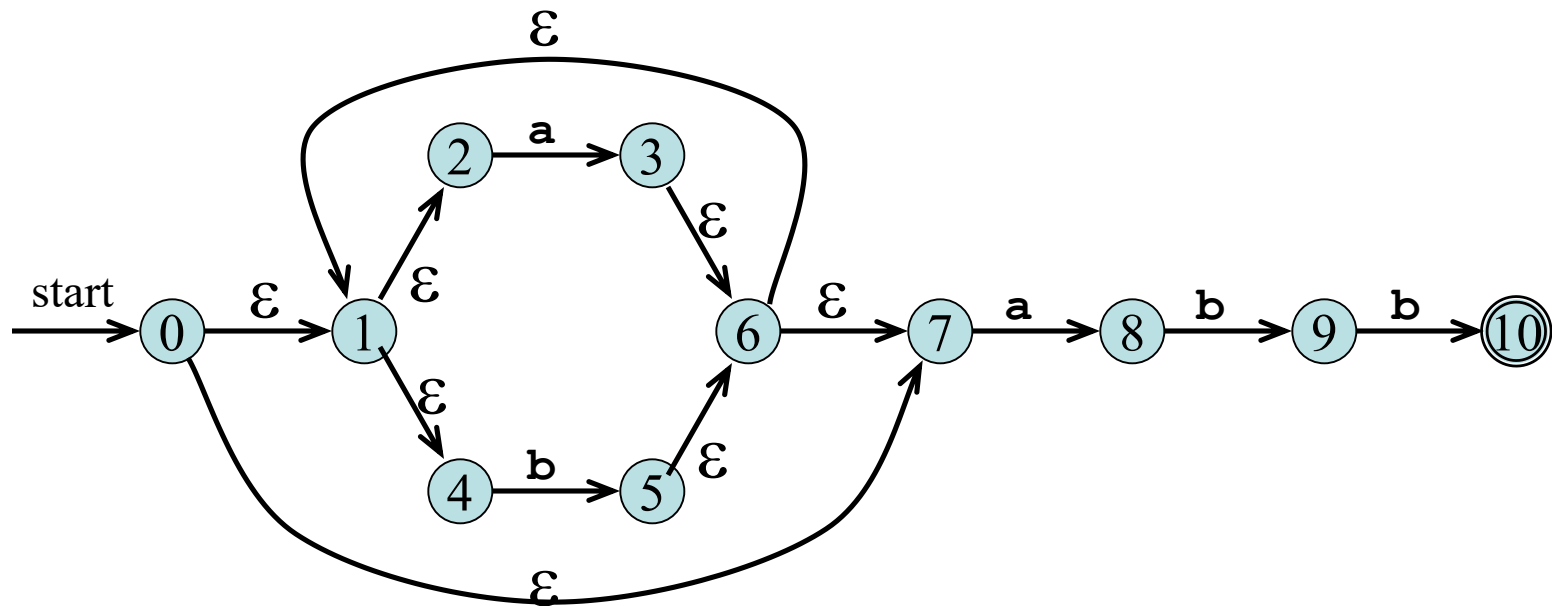
**end if**

$Dtran[T, a] := U$

**end do**

**end do**

# Subset Construction Example 1



*Dstates*

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

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

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

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

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

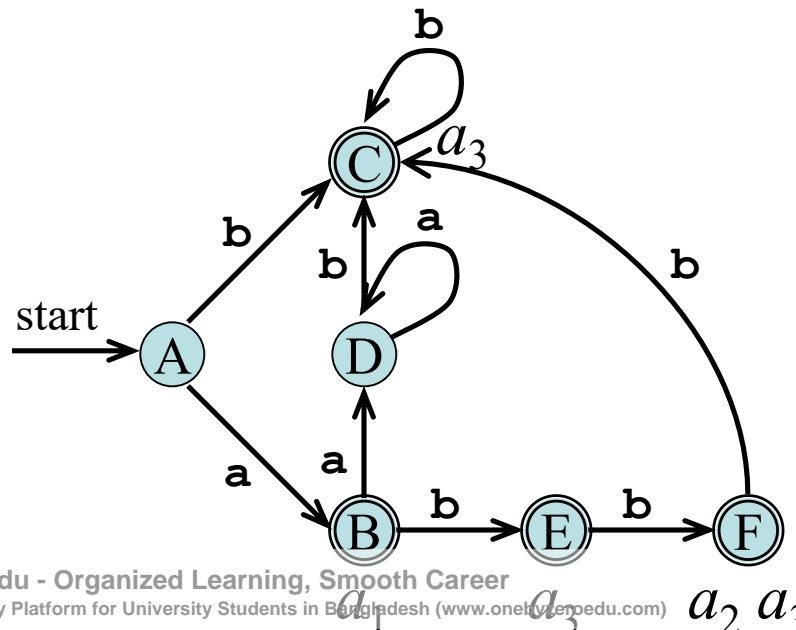
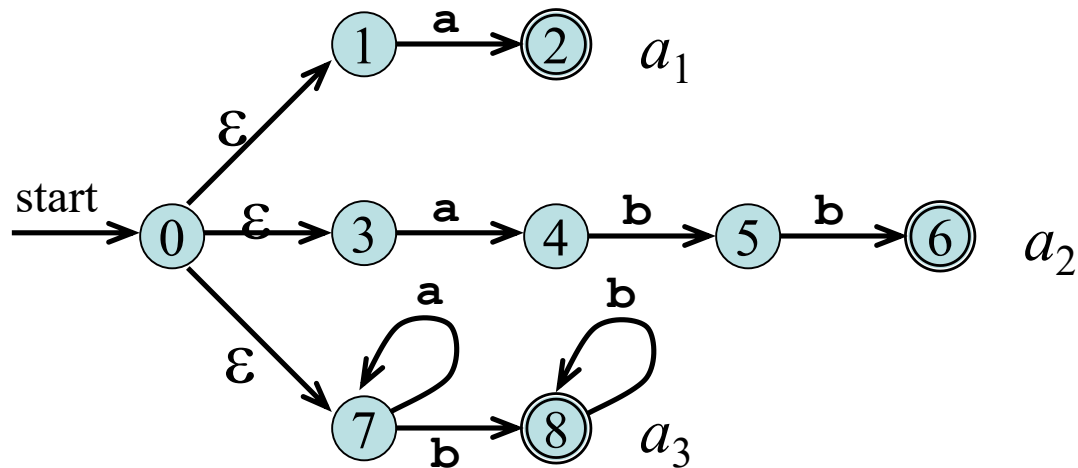
AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh ([www.onebyzeroedu.com](http://www.onebyzeroedu.com))



# Subset Construction Example 2



*Dstates*

A = {0,1,3,7}

B = {2,4,7}

C = {8}

D = {7}

E = {5,8}

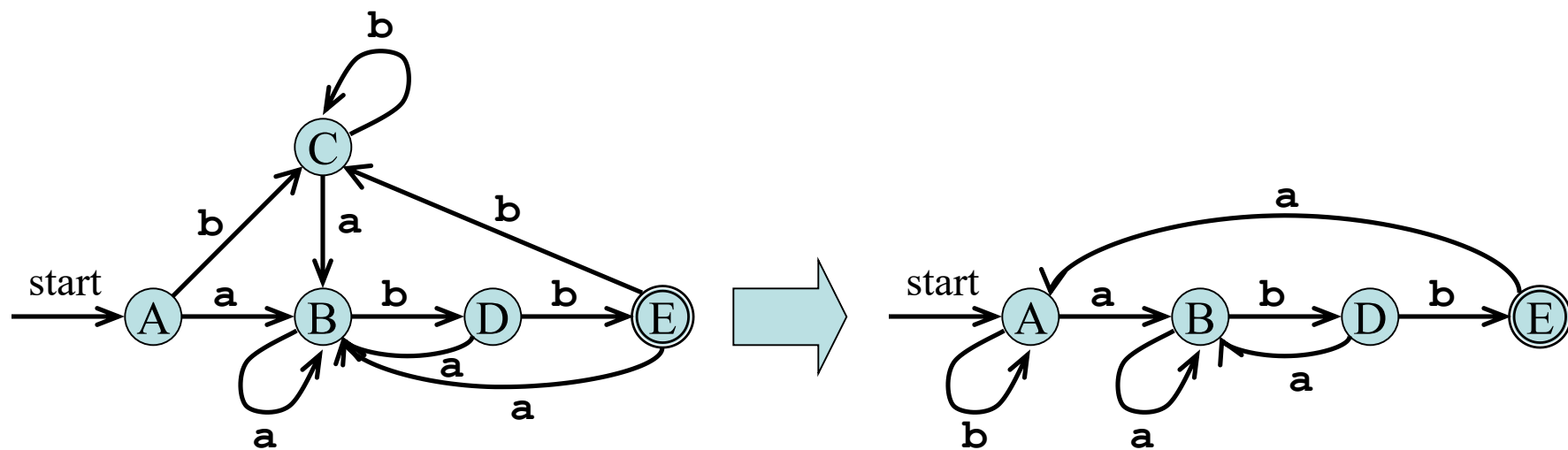
F = {6,8}

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

# Minimizing the Number of States of a DFA



# From Regular Expression to DFA Directly

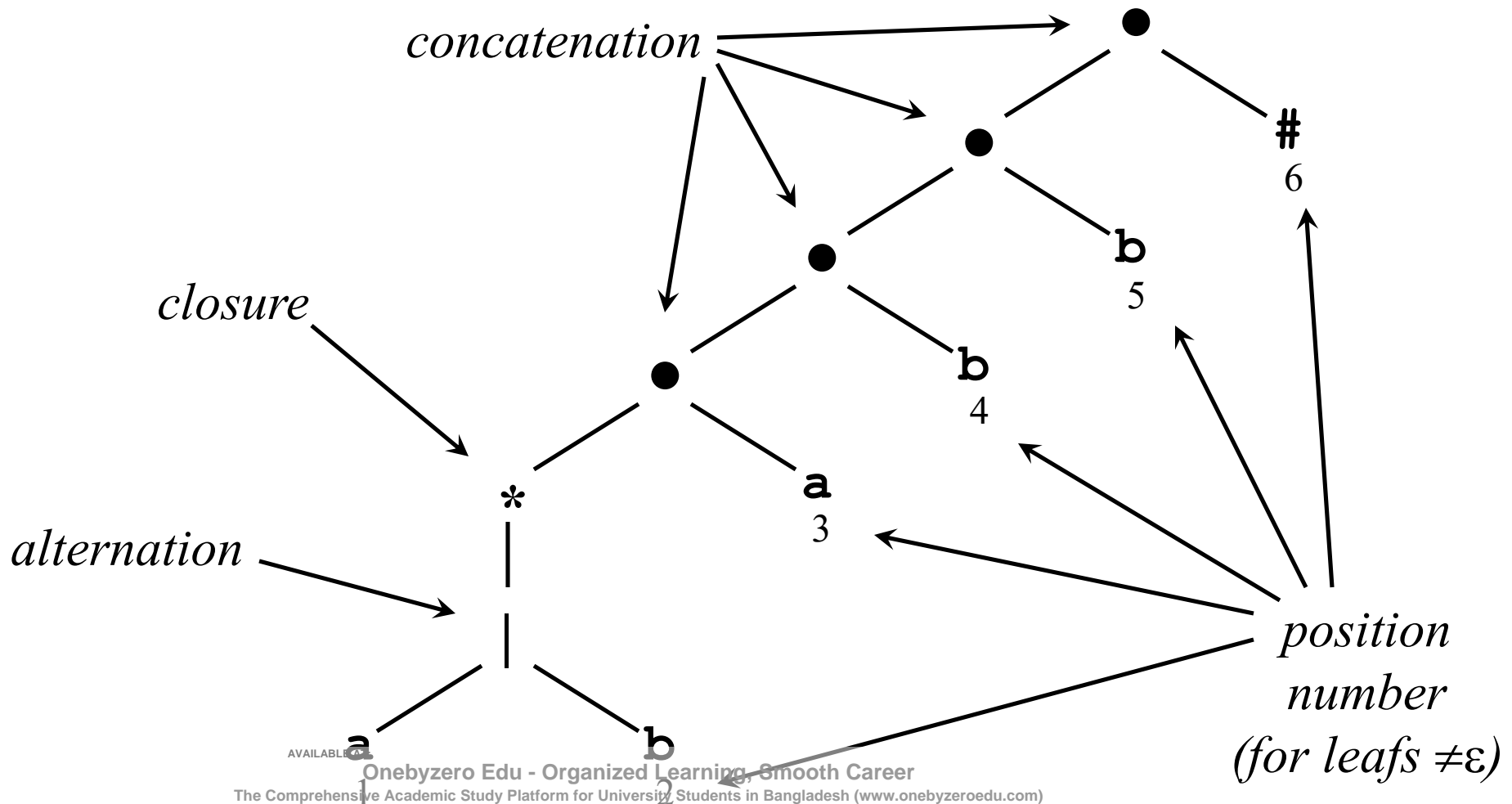
- The “*important states*” of an NFA are those without an  $\varepsilon$ -transition, that is if  $move(\{s\}, a) \neq \emptyset$  for some  $a$  then  $s$  is an important state
- The subset construction algorithm uses only the important states when it determines  $\varepsilon$ -closure( $move(T, a)$ )

# From Regular Expression to DFA Directly (Algorithm)

- Augment the regular expression  $r$  with a special end symbol  $\#$  to make accepting states important: the new expression is  $r\#$
- Construct a syntax tree for  $r\#$
- Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*

# From Regular Expression to DFA

## Directly: Syntax Tree of $(a|b)^*abb\#$



# From Regular Expression to DFA

## Directly: Annotating the Tree

- $nullable(n)$ : the subtree at node  $n$  generates languages including the empty string
- $firstpos(n)$ : set of positions that can match the first symbol of a string generated by the subtree at node  $n$
- $lastpos(n)$ : the set of positions that can match the last symbol of a string generated by the subtree at node  $n$
- $followpos(i)$ : the set of positions that can follow position  $i$  in the tree

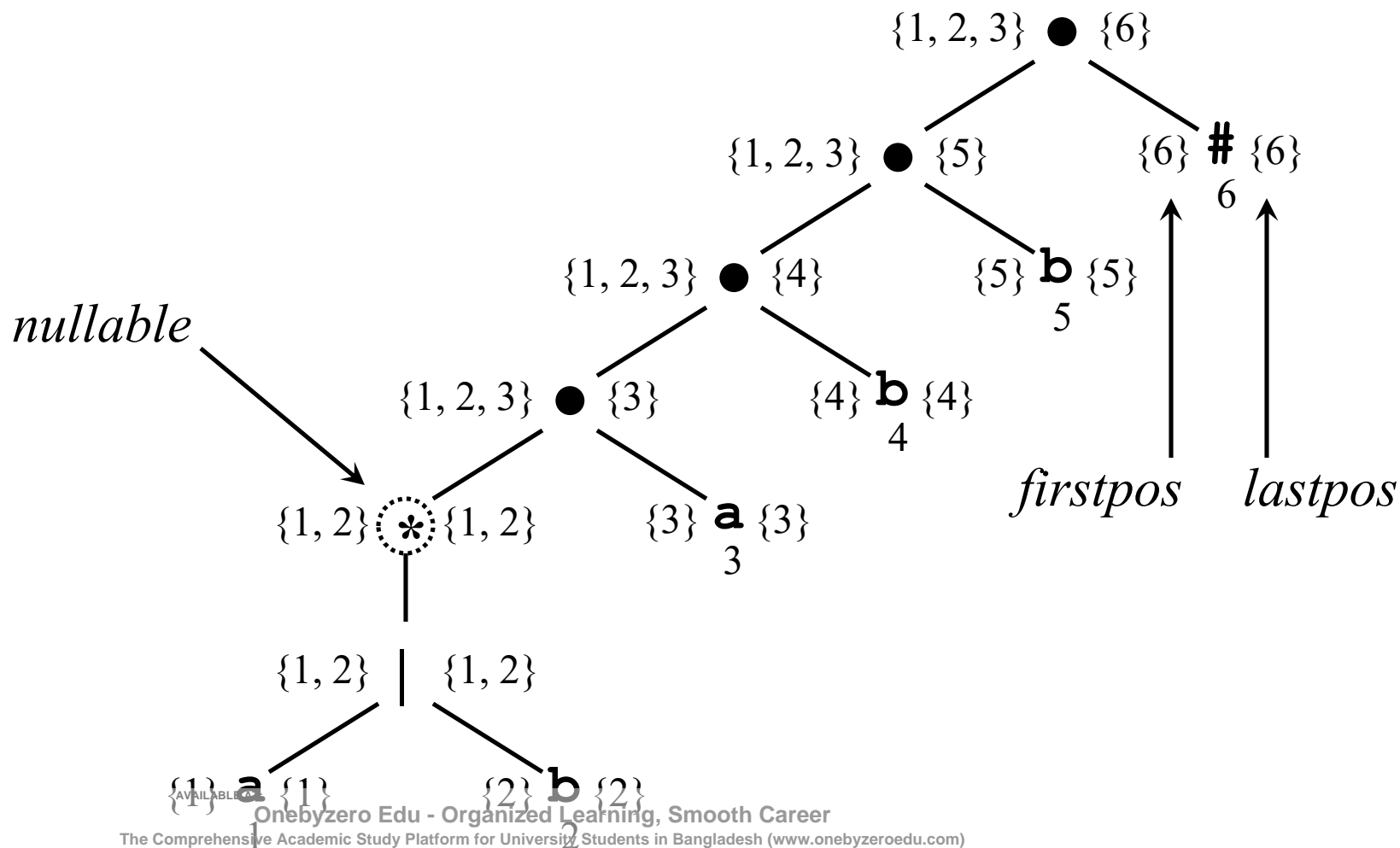
# From Regular Expression to DFA

## Directly: Annotating the Tree

Node $n$	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
Leaf $\varepsilon$	true	$\emptyset$	$\emptyset$
Leaf $i$	false	$\{i\}$	$\{i\}$
$\begin{array}{c}   \\ / \quad \backslash \\ c_1 \quad c_2 \end{array}$	$nullable(c_1)$ or $nullable(c_2)$	$firstpos(c_1)$ $\cup$ $firstpos(c_2)$	$lastpos(c_1)$ $\cup$ $lastpos(c_2)$
$\begin{array}{c} \bullet \\ / \quad \backslash \\ c_1 \quad c_2 \end{array}$	$nullable(c_1)$ and $nullable(c_2)$	<b>if</b> $nullable(c_1)$ <b>then</b> $firstpos(c_1) \cup firstpos(c_2)$ <b>else</b> $firstpos(c_1)$	<b>if</b> $nullable(c_2)$ <b>then</b> $lastpos(c_1) \cup lastpos(c_2)$ <b>else</b> $lastpos(c_2)$
$\begin{array}{c} * \\   \\ c_1 \end{array}$	true	$firstpos(c_1)$	$lastpos(c_1)$

# From Regular Expression to DFA

## Directly: Syntax Tree of $(a|b)^*abb\#$





# From Regular Expression to DFA

## Directly: *followpos*

```

for each node  $n$  in the tree do
    if  $n$  is a cat-node with left child  $c_1$  and right child  $c_2$  then
        for each  $i$  in  $lastpos(c_1)$  do
             $followpos(i) := followpos(i) \cup firstpos(c_2)$ 
        end do
    else if  $n$  is a star-node
        for each  $i$  in  $lastpos(n)$  do
             $followpos(i) := followpos(i) \cup firstpos(n)$ 
        end do
    end if
end do

```

# From Regular Expression to DFA

## Directly: Algorithm

$s_0 := \text{firstpos}(\text{root})$  where  $\text{root}$  is the root of the syntax tree

$Dstates := \{s_0\}$  and is unmarked

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

mark  $T$

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

let  $U$  be the set of positions that are in  $\text{followpos}(p)$

for some position  $p$  in  $T$ ,

such that the symbol at position  $p$  is  $a$

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

add  $U$  as an unmarked state to  $Dstates$

**end if**

$Dtran[T, a] := U$

**end do**

**end do**

AVAILABLE AT:

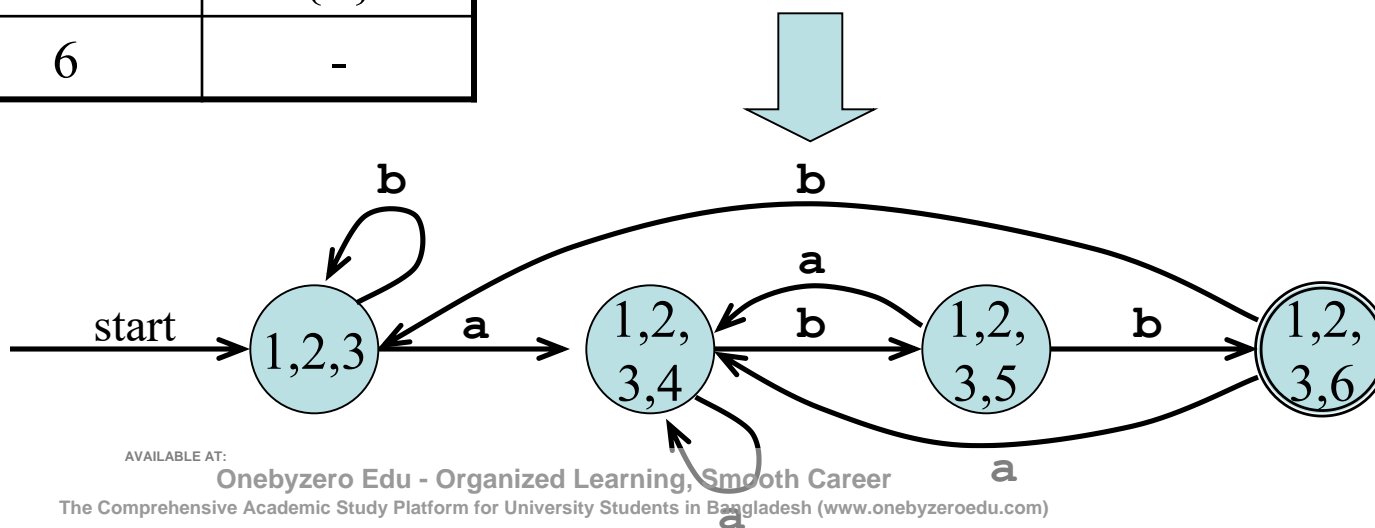
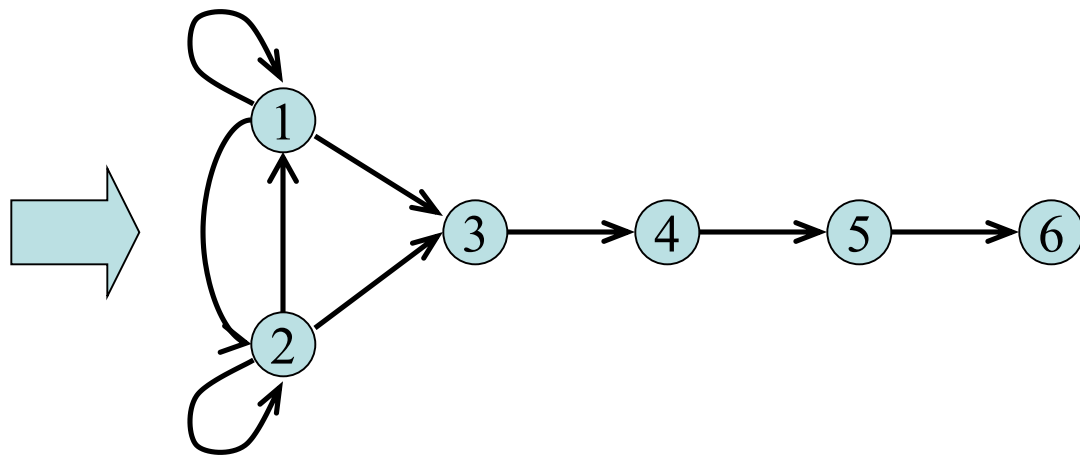
Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh ([www.onebyzeroedu.com](http://www.onebyzeroedu.com))

# From Regular Expression to DFA

## Directly: Example

Node	<i>followpos</i>
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-



AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh ([www.onebyzeroedu.com](http://www.onebyzeroedu.com))

# Time-Space Tradeoffs

<i>Automaton</i>	<i>Space (worst case)</i>	<i>Time (worst case)</i>
NFA	$O( r )$	$O( r  \times  x )$
DFA	$O(2^{ r })$	$O( x )$