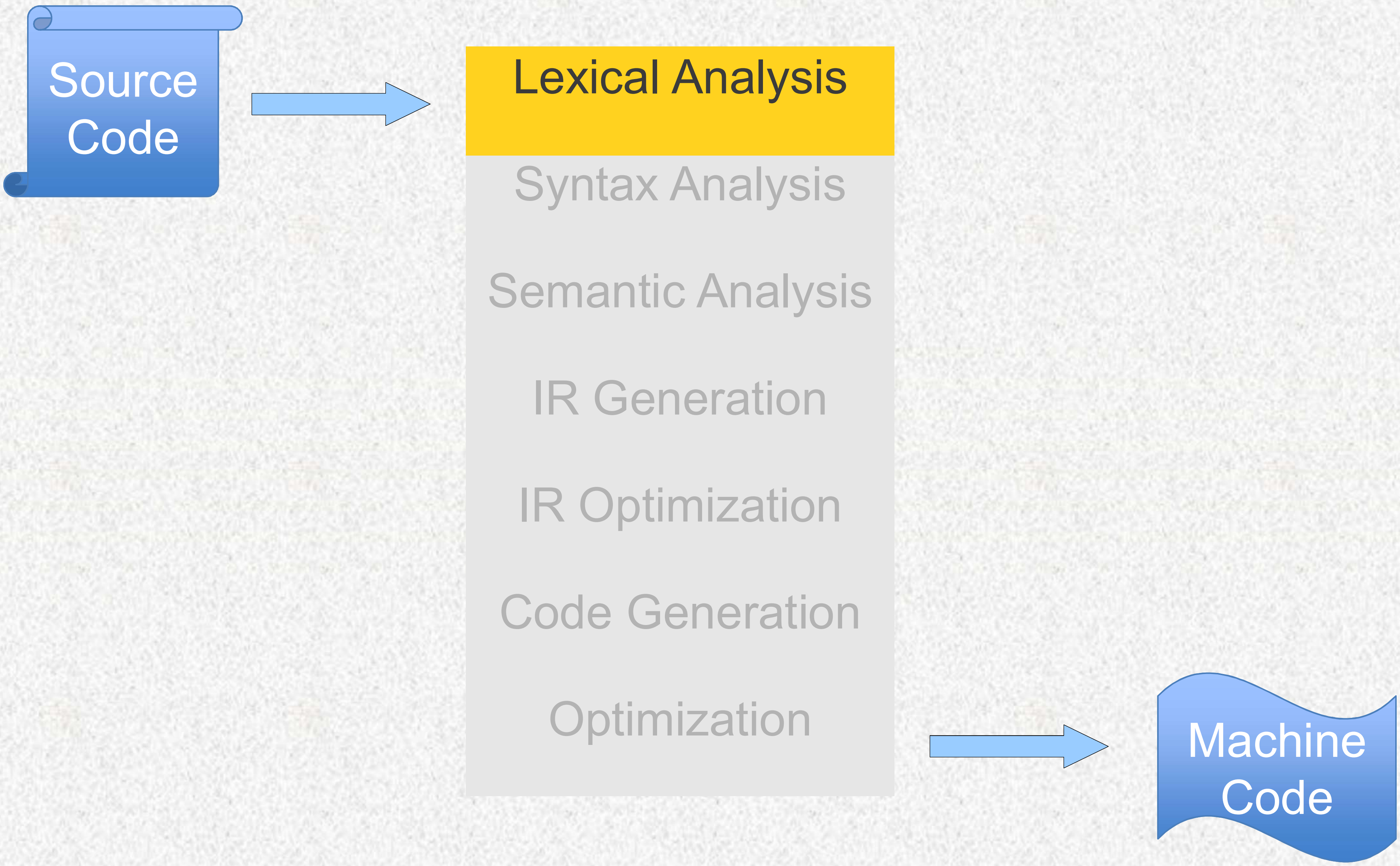
Lecture 3: Lexical Analysis Cont.

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Formalisms of tokens

Regular Expression

Finite Automaton

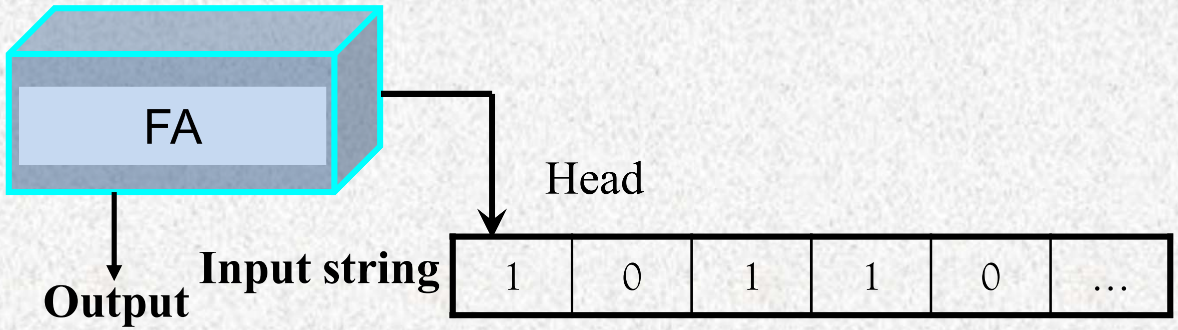
Implementing Regular Expressions

- Regular expressions can be implemented using **finite automata**.
  - Regular expressions = specification
  - Finite automata = implementation
- There are two main kinds of finite automata:
  - **NFA**s (**nondeterministic** finite automata)
  - **DFA**s (**deterministic** finite automata

Finite Automatons

- A finite automaton is a 5-tuple (S,Σ,δ,s_0,F)
  - A set of states S --- nodes
  - An input alphabet Σ
  - A transition function $\delta(S_i, a)=S_j$
  - A start state S_0
  - A set of accepting states $F \subseteq S$

Finite Automatons

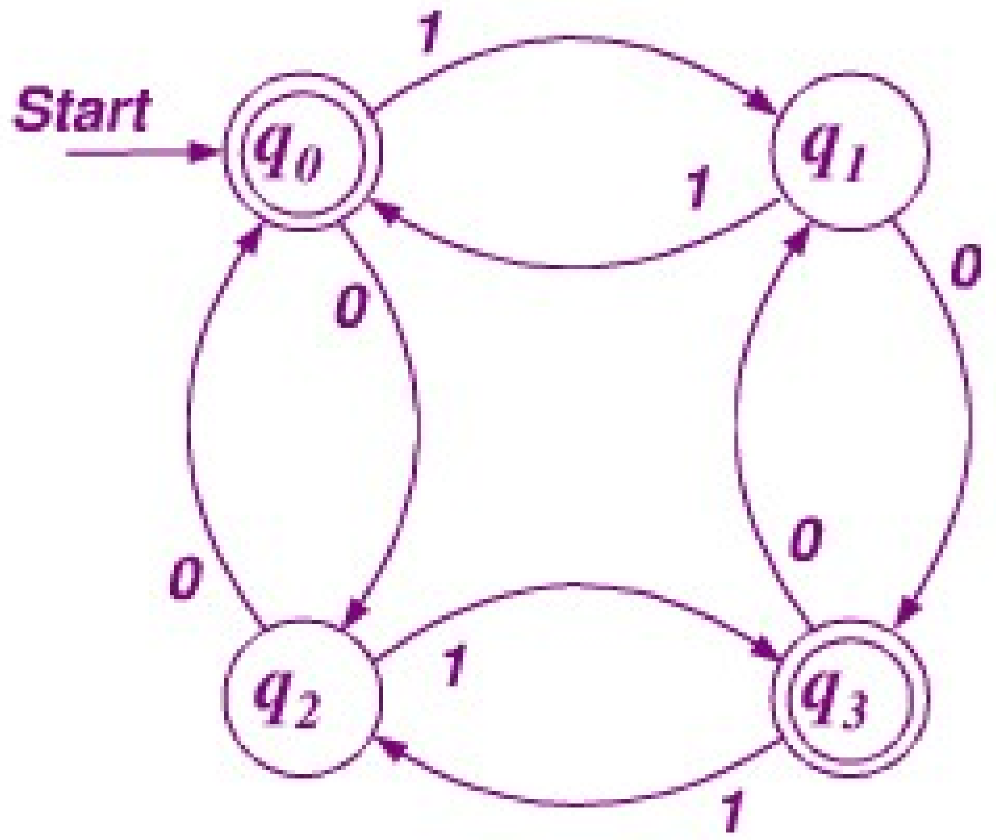


- Input: a string
- Output： accept if the scanning of input string reaches its EOF and the FA reaches an accepting state; reject otherwise

Strings accepted by an FA

- An FA *accepts* an input string *x* **iff** 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 More Complex Automaton

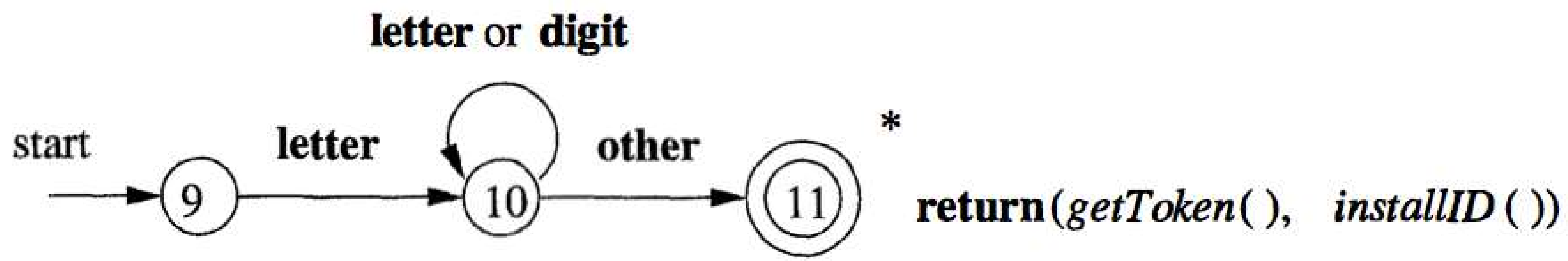


"1010": accept

"101": reject

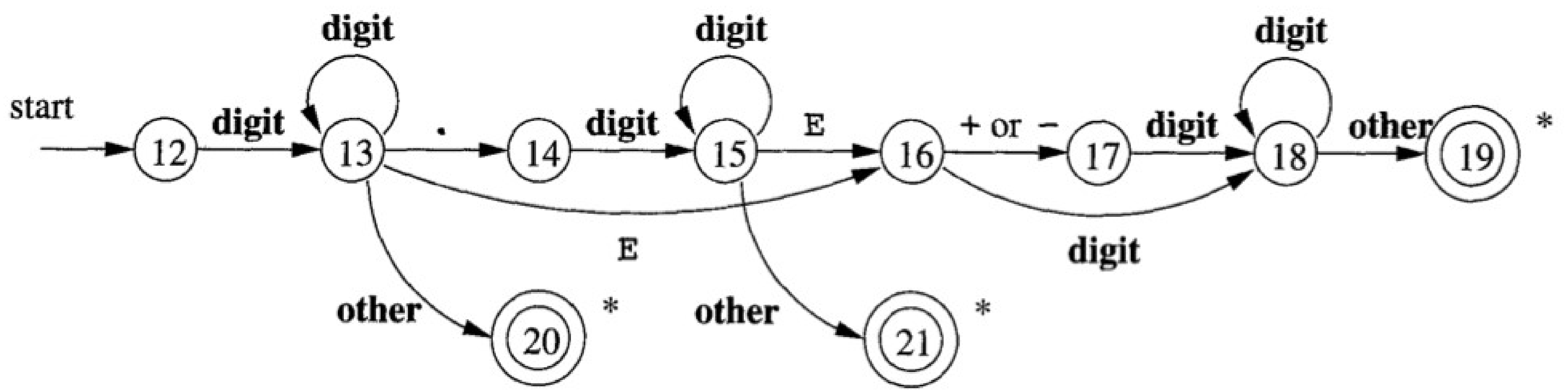
A state transition from one state to another on the path is called a *move*

A More Complex Automaton



h	i	1	2	3

A More Complex Automaton

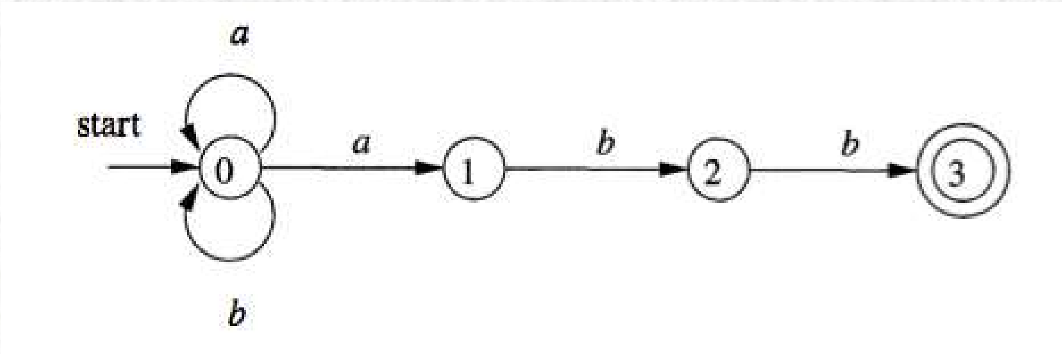


1	2	.	3	7	5

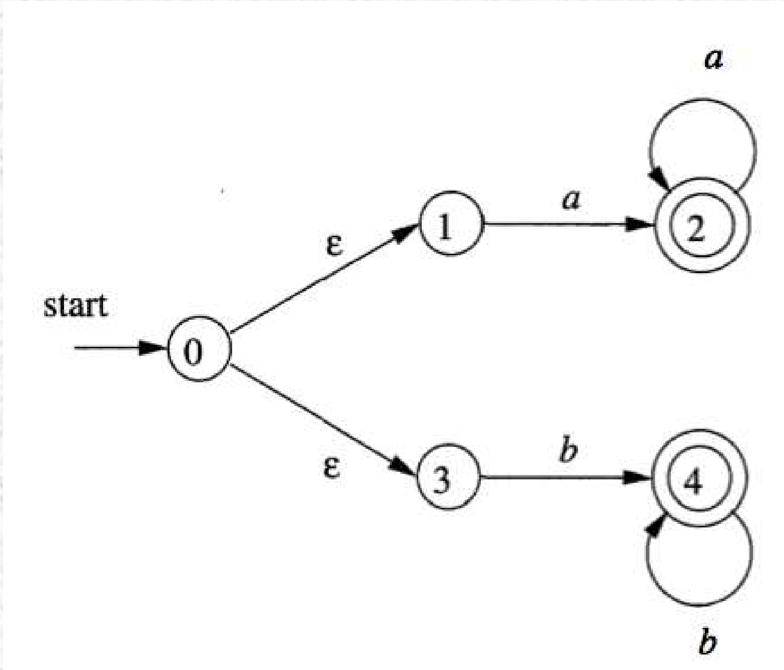
Language defined by an FA

- The *language defined by* an FA is the set of input strings it accepts, such as (**a**|**b**)***abb** for the example NFA

Languages defined by an FA



(a|b)*abb



aa*|bb*

Finite Automata

- Finite automata is a recognizer
- Given an input string, they simply say "yes" or "no" about each possible input string
  - **NFA**s (**nondeterministic** finite automata)
  - **DFA**s (**deterministic** finite automata
- To describe NFA or DFA, we have two methods
  - Transition diagram
  - Transition table

Nondeterministic Finite Automata (NFA)

- Definition: an NFA is a 5-tuple (S,Σ,δ,s_0,F) where
  - S is a finite set of *states*
  - Σ is a finite set of *input symbol alphabet*
  - δ is a *mapping* from $S \times \Sigma \cup \{\varepsilon\}$ to a set of states
  - $S_0 \subseteq S$ is the set of *start states*
  - $F \subseteq S$ is the set of *accepting* (or *final*) *states*

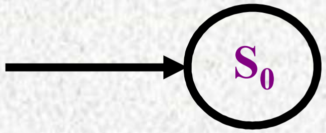
Nondeterministic Finite Automata (NFA)

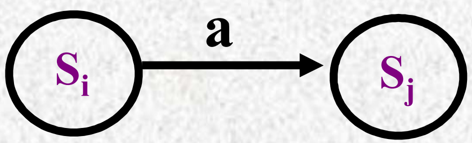
- Transition Graph

Node: **State**

- Non-terminal state: 

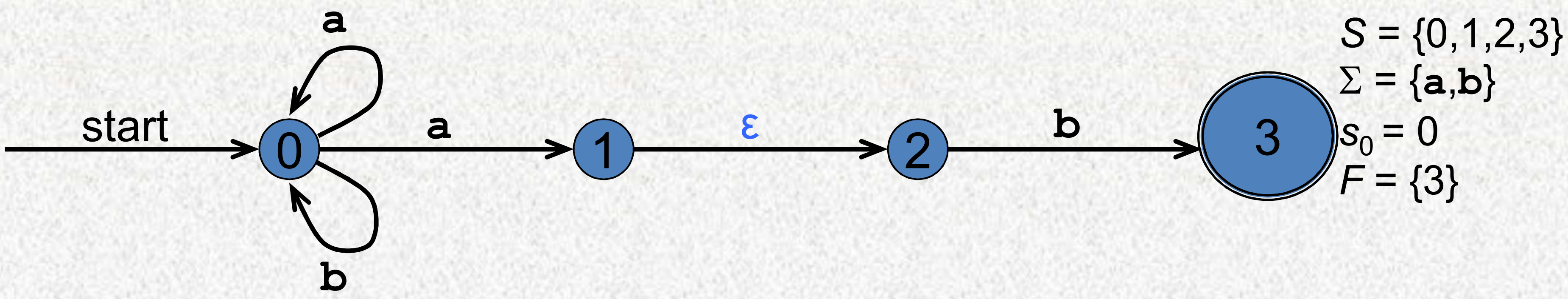
- Terminal state: 

- Starting state: 

Edge: **state transition** $f(S_i,a)=S_j$ 

Transition Graph

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



Nondeterministic Finite Automata (NFA)

- **Transit table**
  - Line：State
    - Starting state：in general, the first line，or label “+”;
    - Terminal state: “*” or “-” or “⊥”;
  - Column：All symbols in ∑
  - Cell：state transition mapping

Transition Table

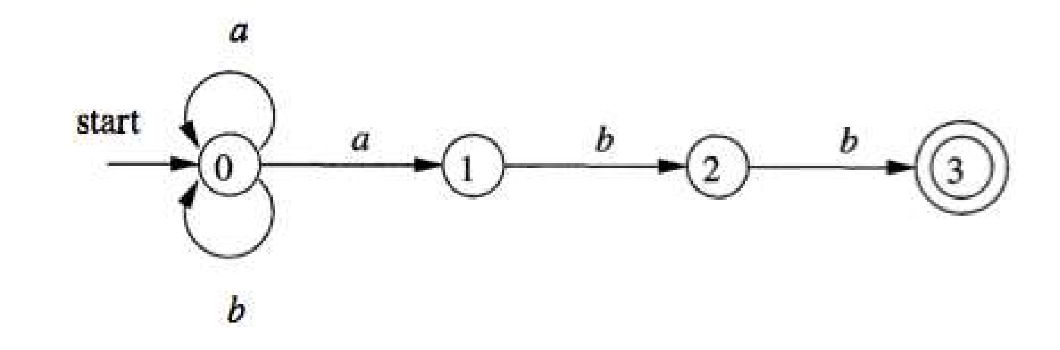
- The mapping δ of an NFA can be represented in a *transition table*

$\delta(0,\mathtt{a}) = \{0,1\}$
$\delta(0,\mathtt{b}) = \{0\}$
$\delta(1,\mathtt{b}) = \{2\}$
$\delta(2,\mathtt{b}) = \{3\}$

→

State	*Input* **a**	*Input* **b**
0	{0,1}	{0}
1		{2}
2		{3}

NFA Example 2



Transition Table

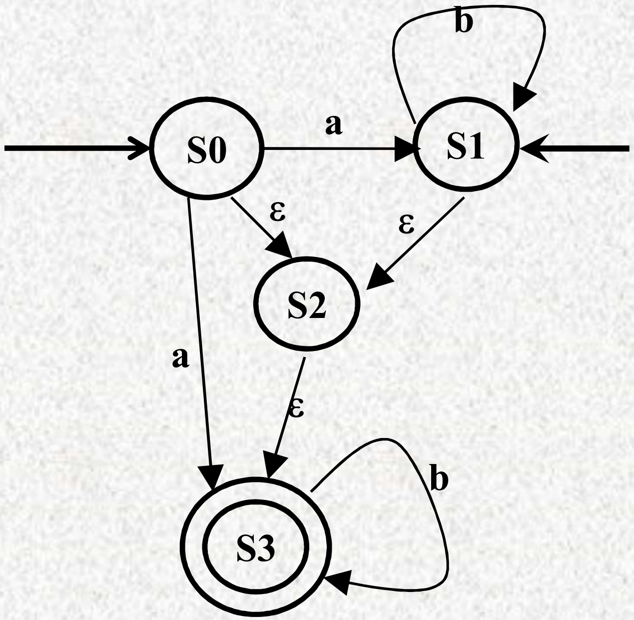
STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Acceptance of input strings

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

NFA Example 3



	a	b	ε
S0+	{S1,S3}		{S2}
S1+		{S1}	{S2}
S2			{S3}
S3-		{S3}	

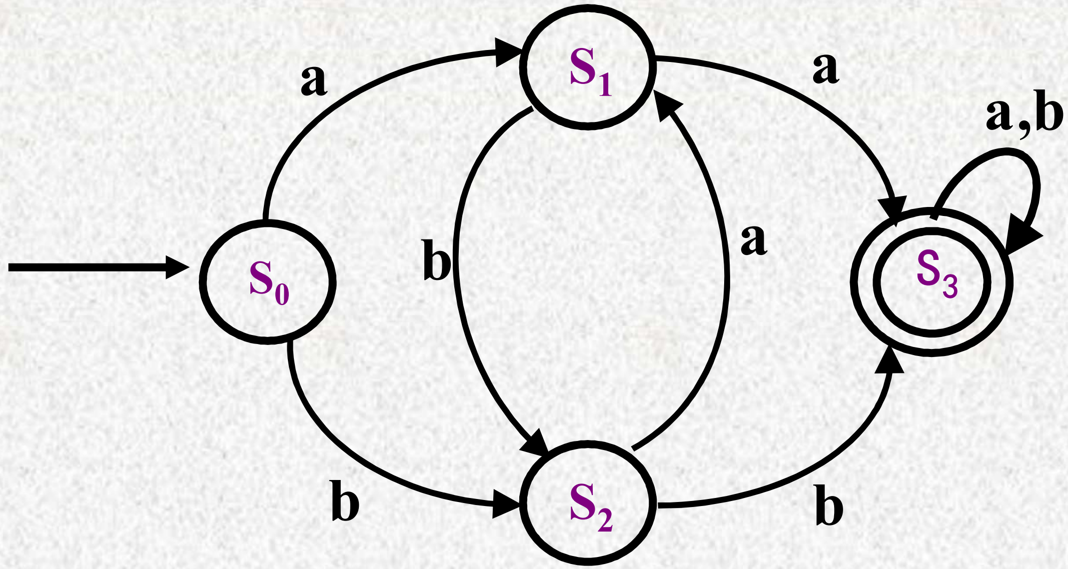
Deterministic Finite Automata (DFA)

- Definition: an DFA is a 5-tuple (S,Σ,δ,s_0,F), is a special case of NFA
  - There are no moves on input ε, and
  - For each state s and input symbol a, there is exactly one edge out of s labeled a.

Deterministic Finite Automata (DFA)

- DFA M=({S0, S1, S2, S3}, {a,b}, f, S0, {S3}), ：

f (S0, a)=S1	f (S2, a)=S1
f (S0, b)=S2	f (S2, b)= S3
f (S1, a)= S3	f (S3, a)= S3
f (S1, b)= S2	f (S3, b)= S3



Deterministic Finite Automata (DFA)

- For example, DFA $M=(\{0,1,2,3,4\},\{a,b\}, \delta,\{0\},\{3\})$
- $\delta(0, a) = 1$ $\delta(0, b) = 4$

$\delta(1, a) = 4$ $\delta(1, b) = 2$

$\delta(2, a) = 3$ $\delta(2, b) = 4$

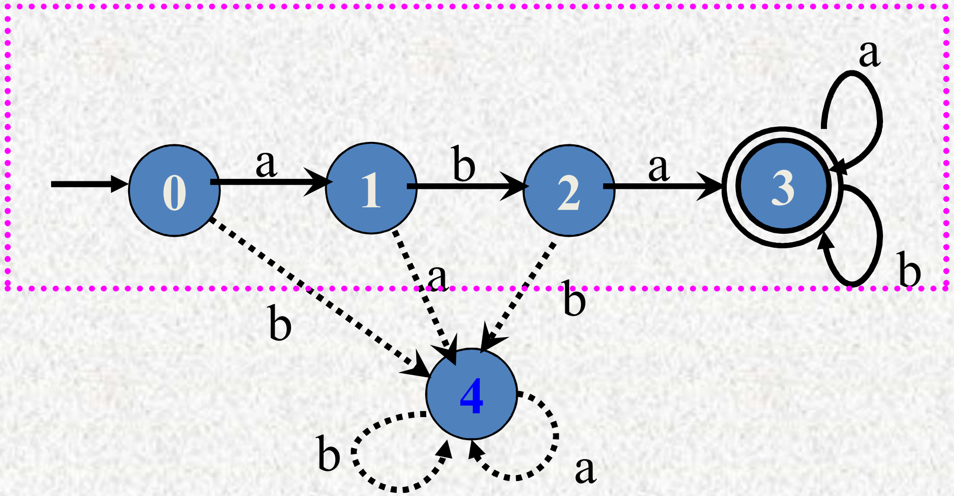
$\delta(3, a) = 3$ $\delta(3, b) = 3$

$\delta(4, a) = 4$ $\delta(4, b) = 4$

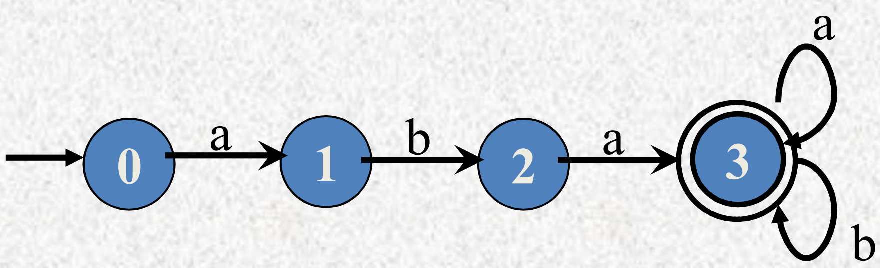
	a	b
0^+	1	4
1	4	2
2	3	4
3^-	3	3
4	4	4

Deterministic Finite Automata (DFA)

	a	b
0+	1	4
1	4	2
2	3	4
3−	3	3
4	4	4



Deterministic Finite Automata (DFA)



	a	b
0+	1	⊥
1	⊥	2
2	3	⊥
3⁻	3	3

↔

	a	b
0+	1	
1		2
2	3	
3⁻	3	3

Deterministic Finite Automata (DFA)

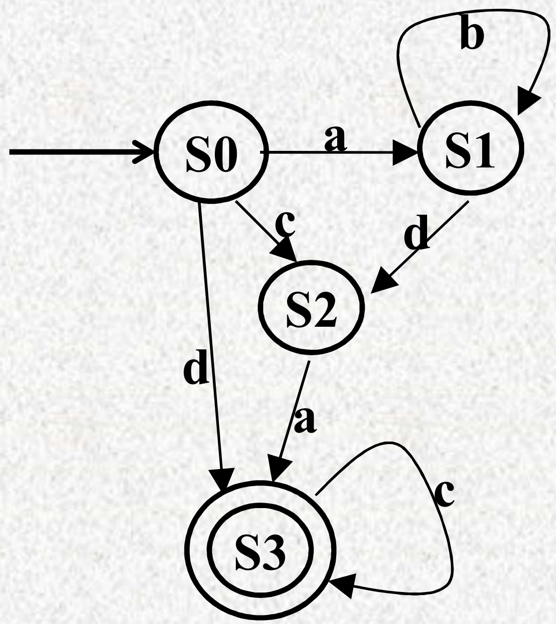
∑: {a, b, c, d}

S: {S0, S1, S2, S3}

Start: S0

Terminal: {S3}

f: {(S0,a)→ S1, (S0,c)→S2,
(S0,d)→S3, (S1,b)→S1,
(S1,d)→S2, (S2,a)→S3,
(S3, c)→S3}



NFA v.s. DFA

NFA v.s. DFA

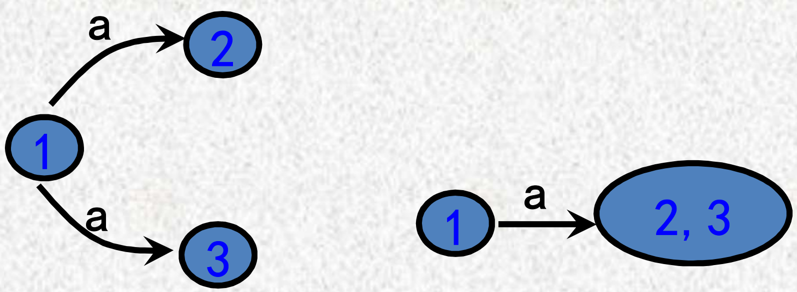
	DFA	NFA
Initial	Single starting state	A set of starting states
ε dege	Not allowed	Allowed
δ(S, a)	S' or ⊥	{S1, …, Sn} or ⊥
Implementation	Deterministic	Nondeterministic

- DFA accepts an input string with only one path
- NFA accepts an input string with possibly multiple paths

Construct DFA from NFA

- Construct DFA from NFA
  - For any NFA, there exists an equivalent DFA
  - Idea of construction: eliminate the uncertainty
  - Merge N states in NFA into one single state
    - Eliminate ε

    - Eliminate multiple mapping

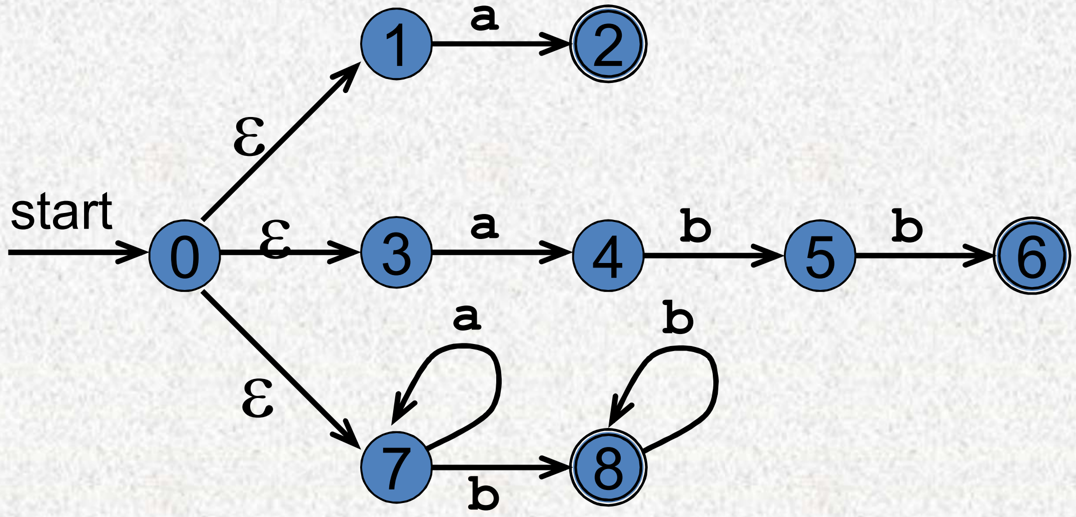


Construct DFA from NFA

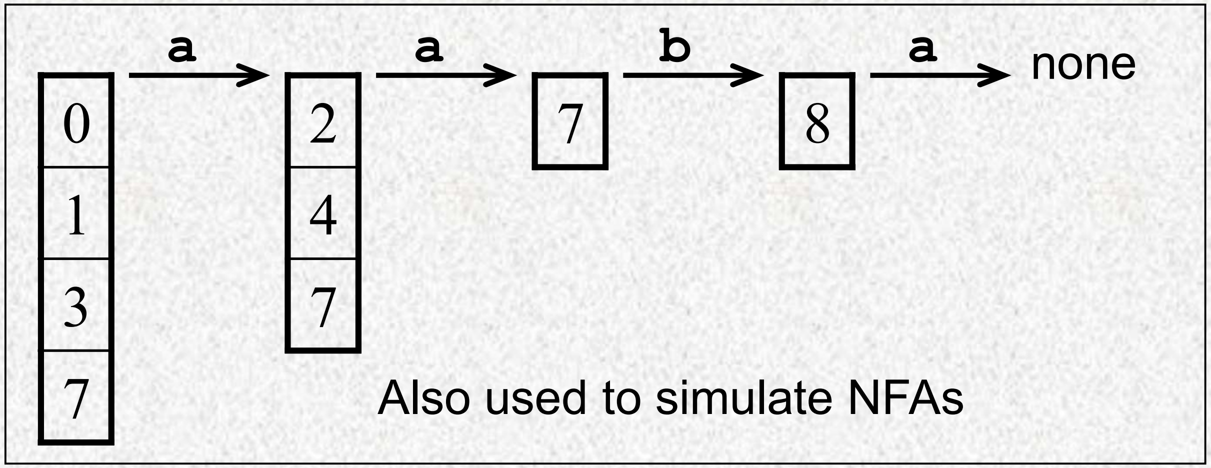
- **INPUT**: An NFA N.
- **OUTPUT**: A DFA D accepting the same language as N.
- **METHOD**: The algorithm constructs a transition table Dtran for D. Each state of D is a set of NFA states, and we construct Dtran so D will simulate "in parallel" all possible moves N can make on a given input string.

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

ε-*closure* and *move* Examples



ε-*closure*({0}) = {0,1,3,7}
move({0,1,3,7},**a**) = {2,4,7}
ε-*closure*({2,4,7}) = {2,4,7}
move({2,4,7},**a**) = {7}
ε-*closure*({7}) = {7}
move({7},**b**) = {8}
ε-*closure*({8}) = {8}
move({8},**a**) = $\emptyset$



Simulating the NFA

Algorithm 3.22: Simulating an NFA.

INPUT: An input string x terminated by an end-of-file character **eof**. An NFA N with start state s_0, accepting states F, and transition function *move*.

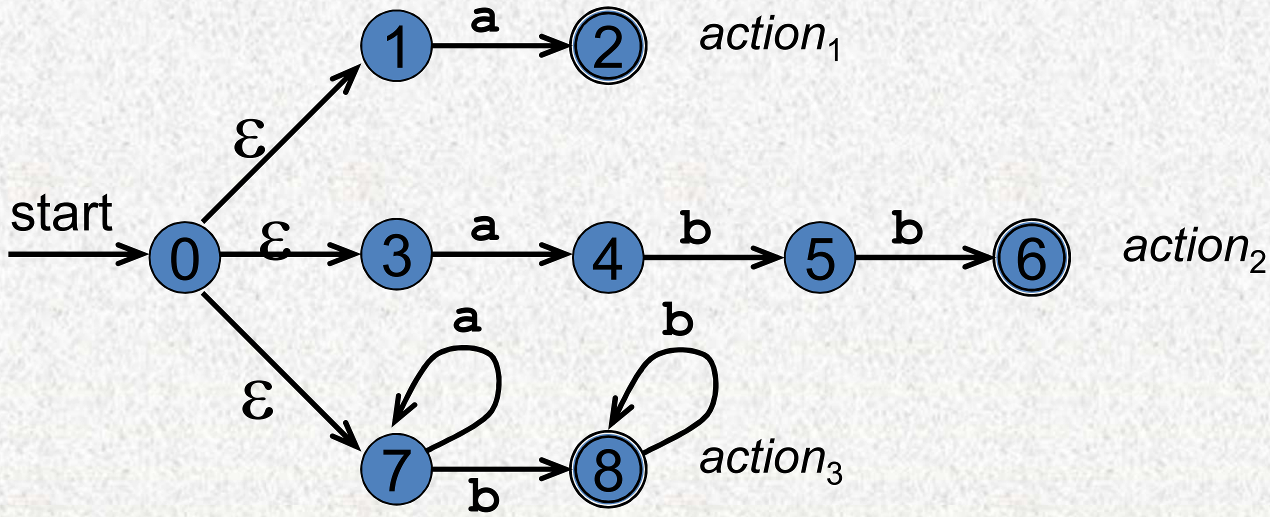
OUTPUT: Answer "yes" if M accepts x; "no" otherwise.

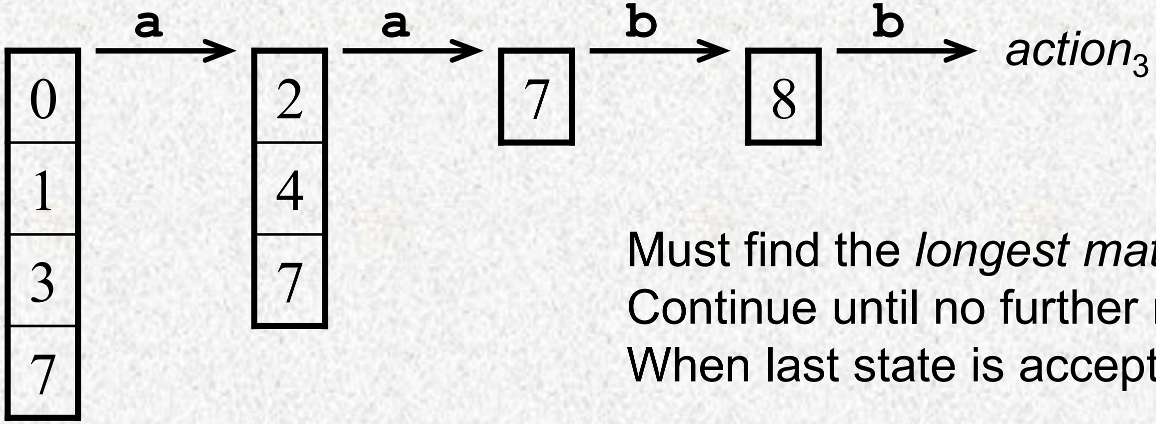
METHOD: The algorithm keeps a set of current states S, those that are reached from s_0 following a path labeled by the inputs read so far. If c is the next input character, read by the function *nextChar*(), then we first compute $move(S, c)$ and then close that set using ϵ-*closure*(). The algorithm is sketched in Fig. 3.37. □

```
S = ε-closure(s0);
c = nextChar();
while ( c != eof ) {
        S = ε-closure(move(S,c));
        c = nextChar();
}
if ( S ∩ F != ∅ ) return "yes";
else return "no";
```

Figure 3.37: Simulating an NFA

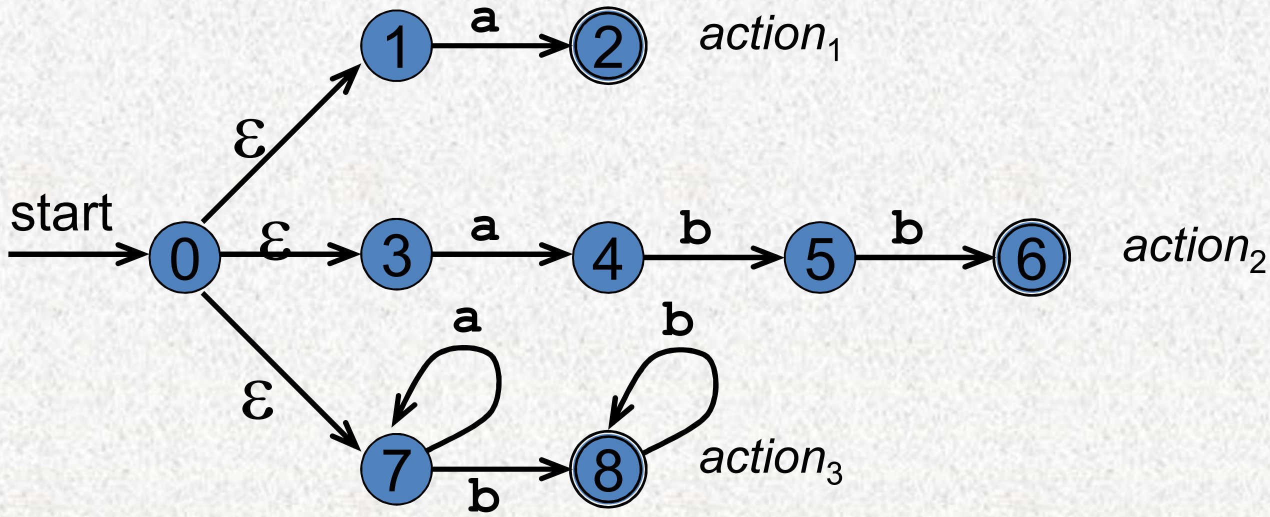
Simulating a NFA Example 1

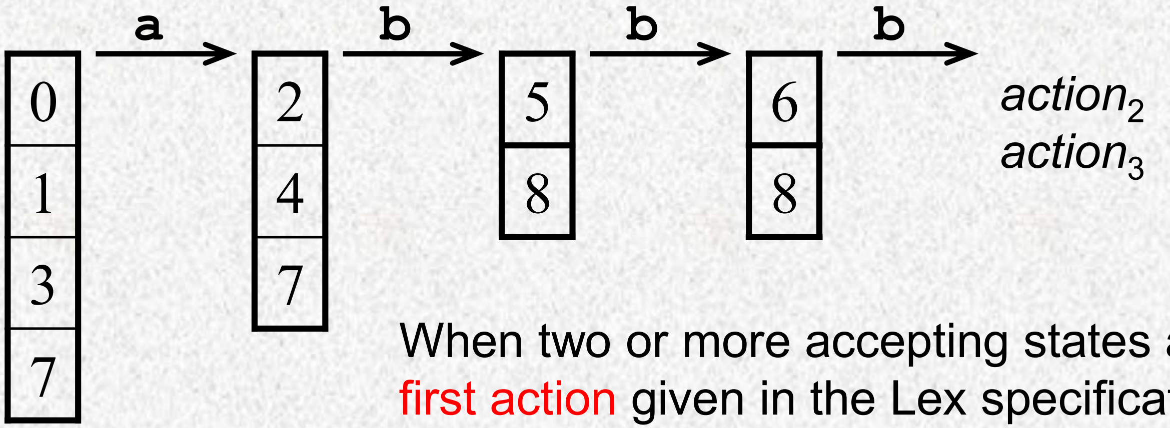




Must find the *longest match*:
Continue until no further moves are possible
When last state is accepting: execute action

Simulating a NFA Example 2





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

The Subset Construction Algorithm

- NFAs can be in many states at once, while DFAs can only be in a single state at a time.
- Key idea: **Make the DFA simulate the NFA**.
- Have the states of the DFA correspond to the *sets of states* of the NFA.
- Transitions between states of DFA correspond to transitions between *sets of states* in the NFA.

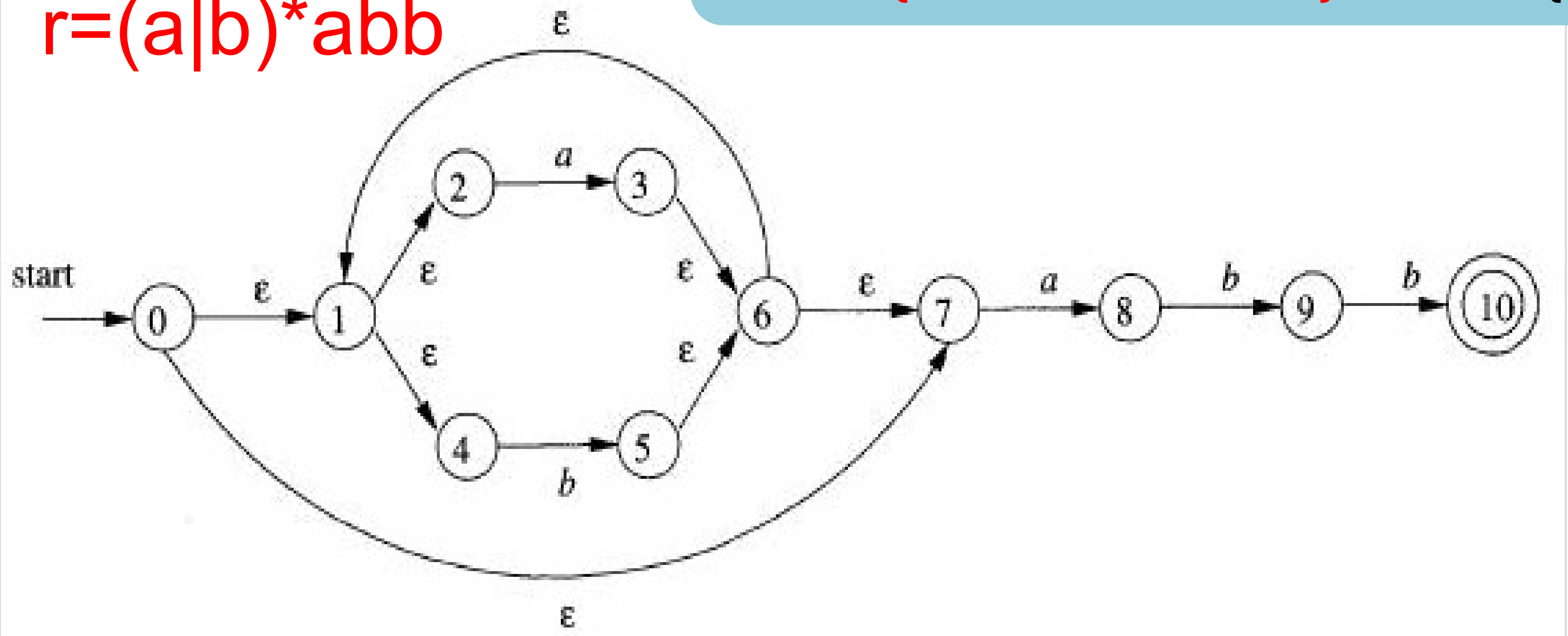
The Subset Construction Algorithm

```
initially, ε-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = ε-closure(move(T, a));
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```

Subset Construction Example 1

First，Initial state of NFA is ε-closure(0)，
i.e. A={0，1，2，4，7}， Σ = {a,b}

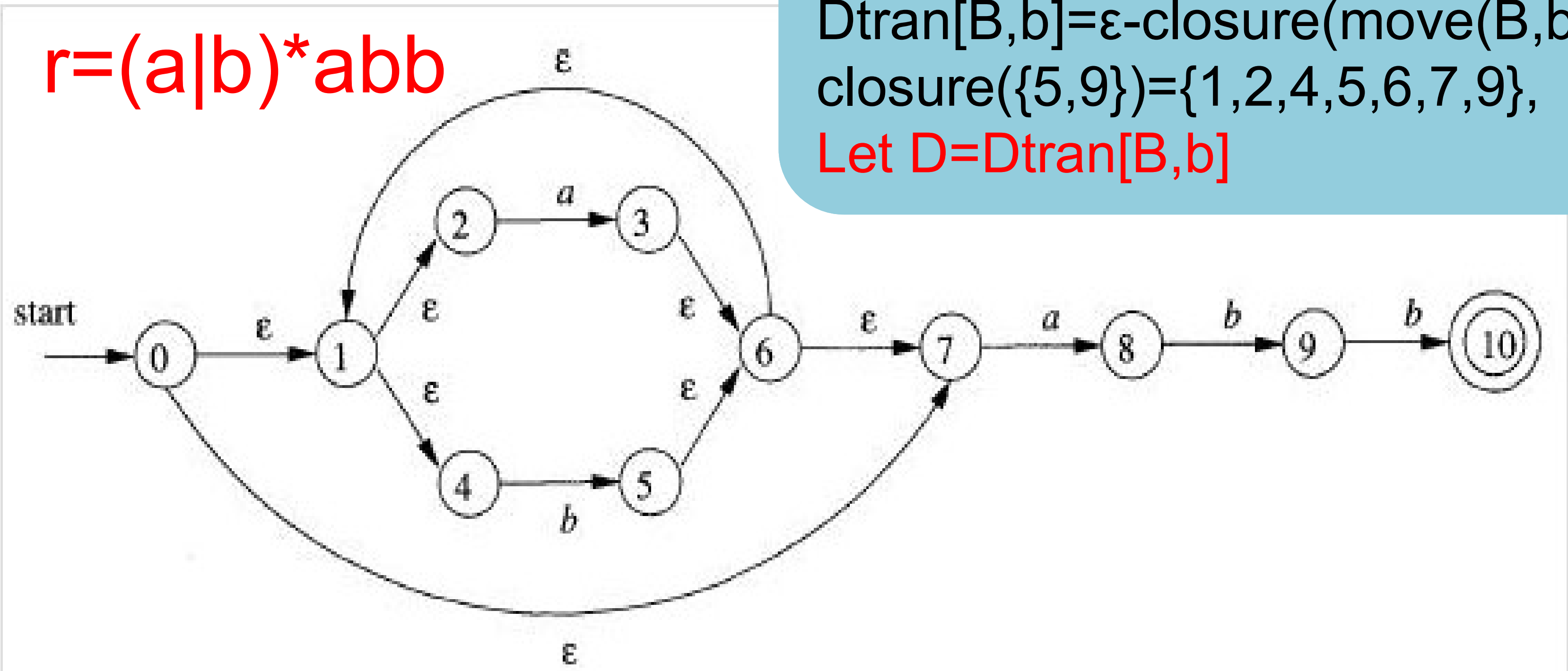
r=(a|b)*abb



Dtran[A,a]=ε-closure(move(A,a))=ε-closure({3,8})={1,2,3,4,6,7,8},
Let B=Dtran[A,a]
Dtran[A,b]=ε-closure(move(A,b))=ε-closure({5})={1,2,4,6,7},
Let C=Dtran[A,b]

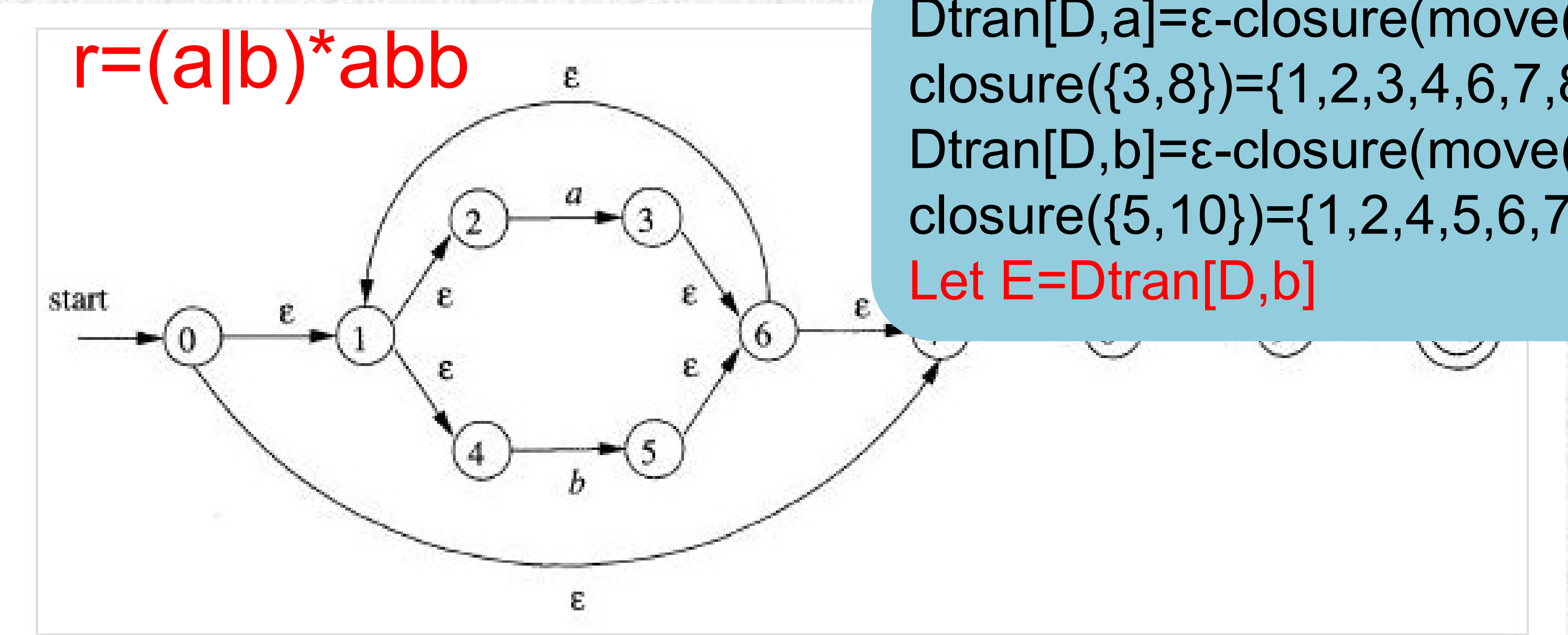
Subset Construction Example 1

Dtran[B,a]=ε-closure(move(B,a))=ε-closure({3,8})={1,2,3,4,6,7,8}=B
Dtran[B,b]=ε-closure(move(B,b))=ε-closure({5,9})={1,2,4,5,6,7,9},
Let D=Dtran[B,b]



Dtran[C,a]=ε-closure(move(C,a))=ε-closure({3,8})={1,2,3,4,6,7,8}=B
Dtran[C,b]=ε-closure(move(C,b))=ε-closure({5})={1,2,4,6,7}=C

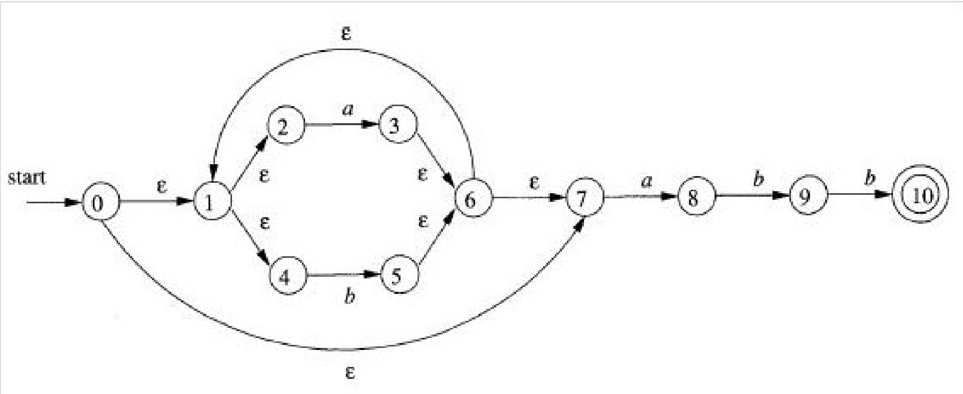
Subset Construction Example 1



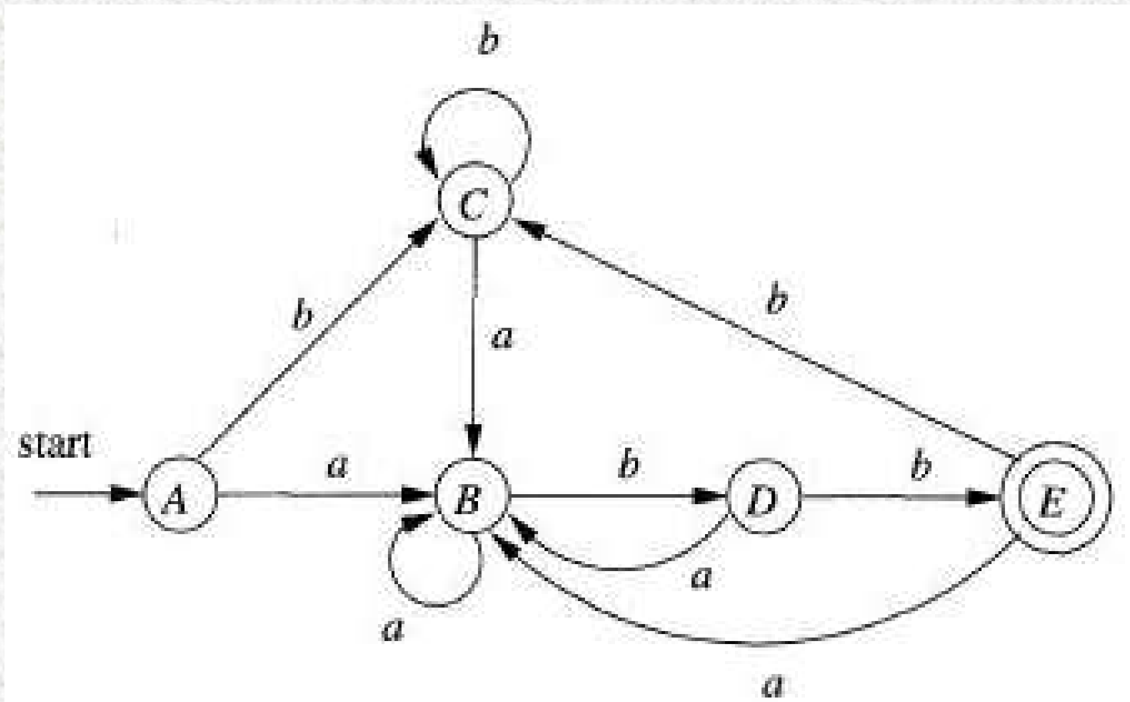
Dtran[D,a]=ε-closure(move(D,a))=ε-closure({3,8})={1,2,3,4,6,7,8}=B
Dtran[D,b]=ε-closure(move(D,b))=ε-closure({5,10})={1,2,4,5,6,7,10},
Let E=Dtran[D,b]

Dtran[E,a]=ε-closure(move(E,a))=ε-closure({3,8})={1,2,3,4,6,7,8}=B
Dtran[E,b]=ε-closure(move(E,b))=ε-closure({5})={1,2,4,6,7}=C

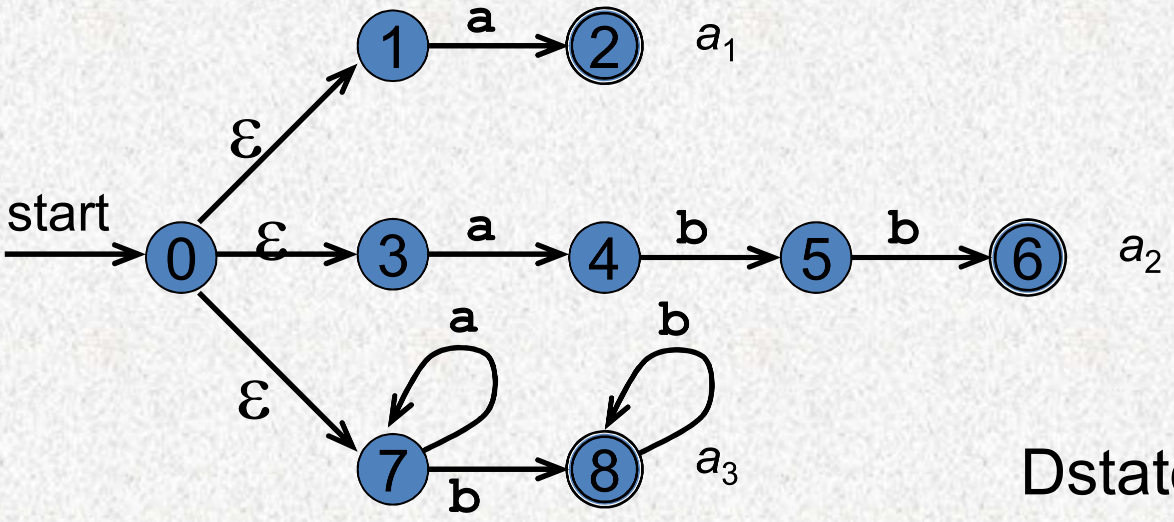
Subset Construction Example 1

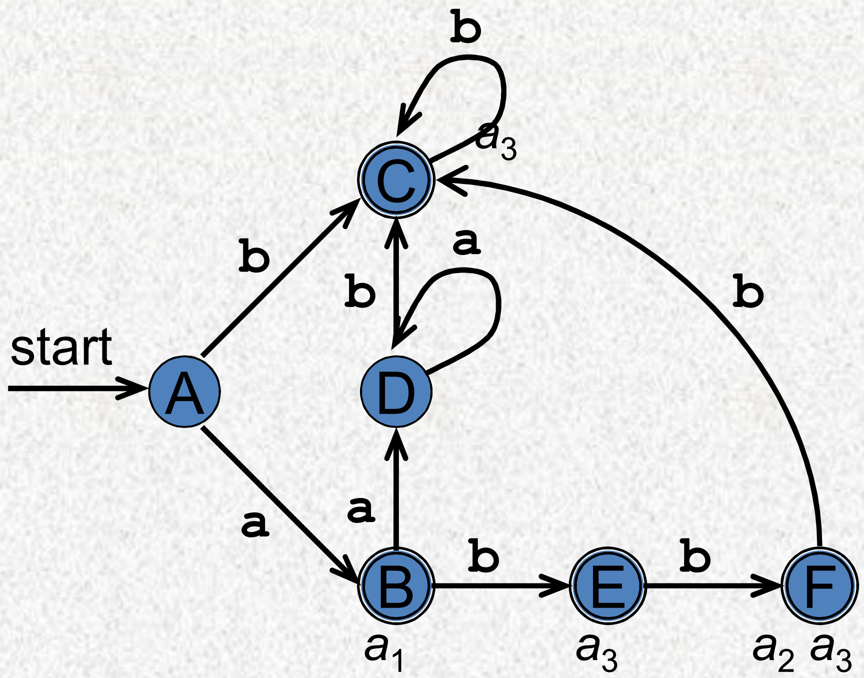


NFA STATE	DFA STATE	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,3,5,6,7,10\}$	E	B	C



Subset Construction Example 2





Dstates

A = {0,1,3,7}

B = {2,4,7}

C = {8}

D = {7}

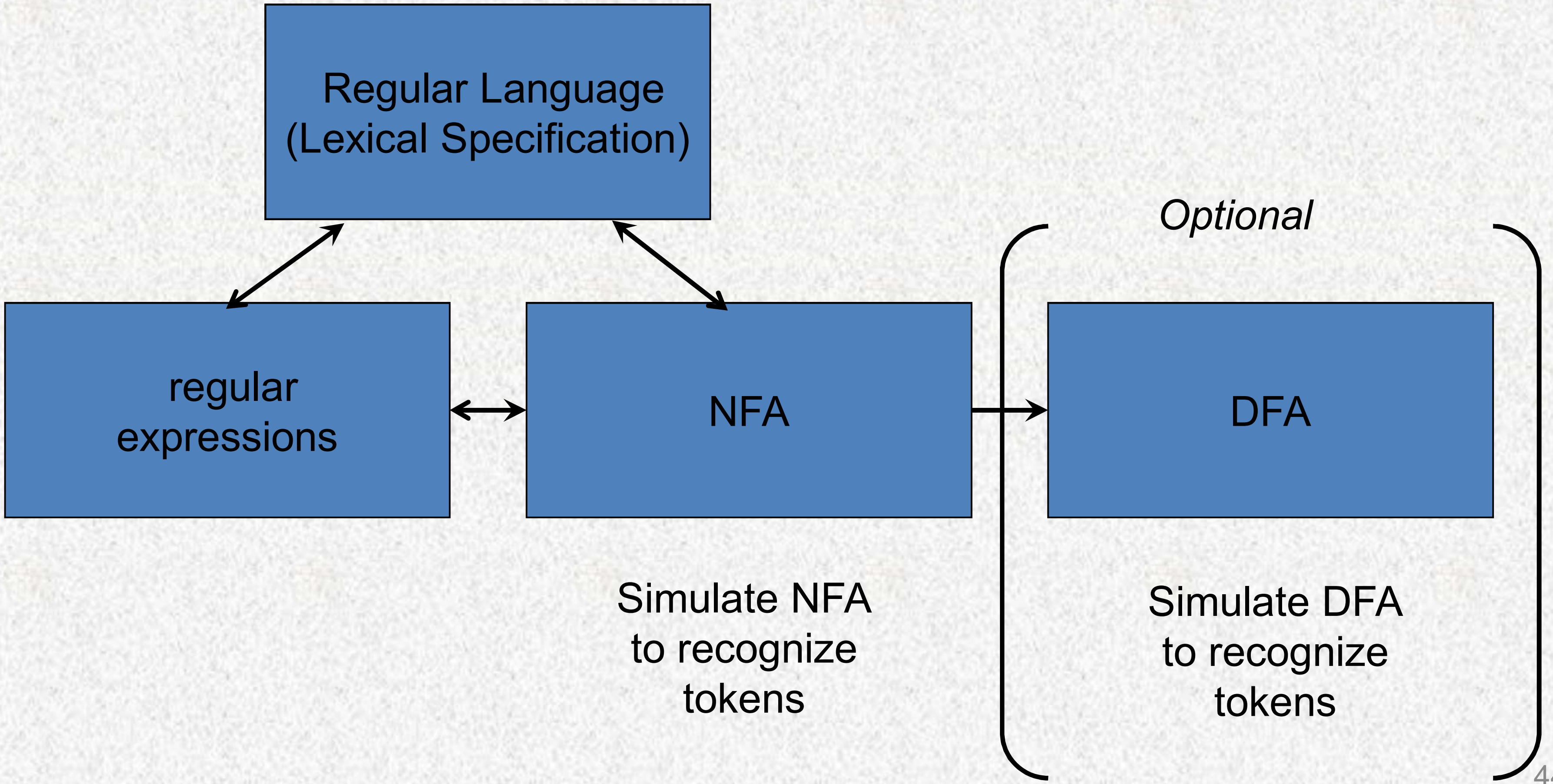
E = {5,8}

F = {6,8}

RE to NFA/DFA

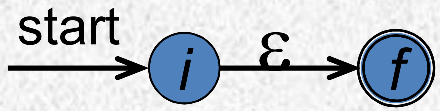
Design of a Lexical Analyzer Generator

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

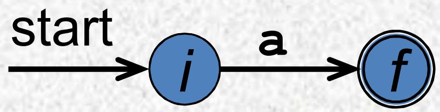


From Regular Expression to NFA (Thompson's Construction)

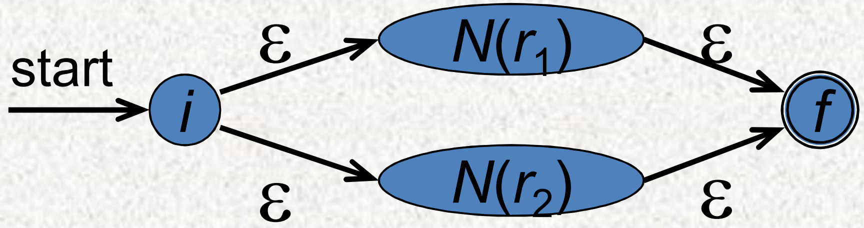
ε



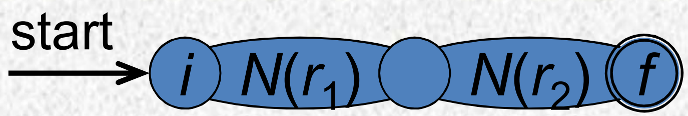
a $\in \Sigma$



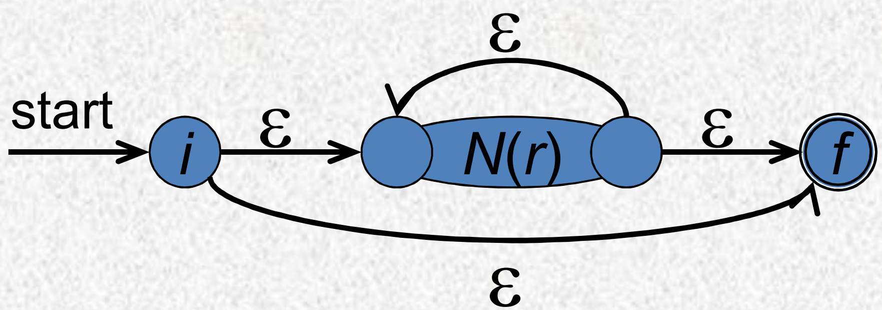
$r_1 \mid r_2$



$r_1 r_2$



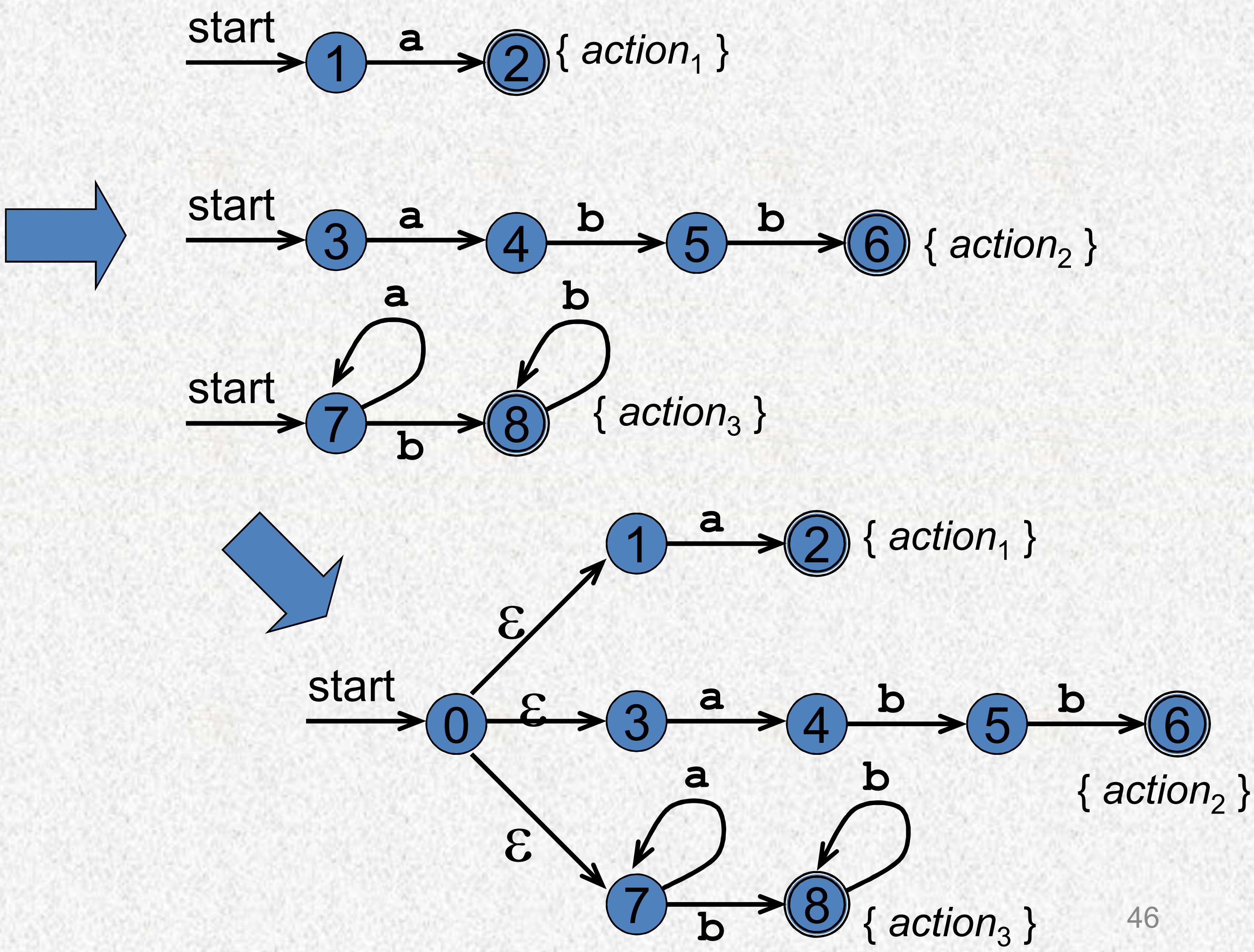
r^*



Combining the NFAs of a Set of Regular Expressions

a | abb | a*b+

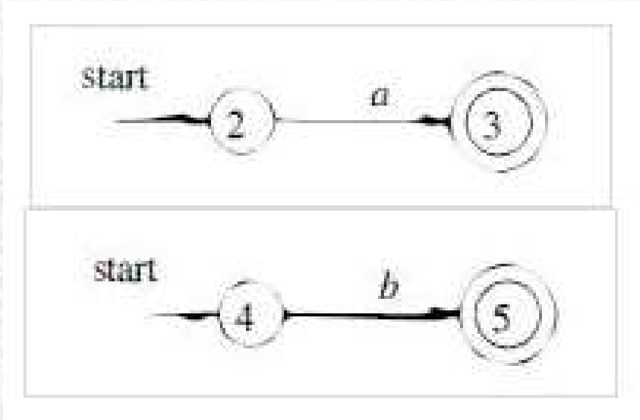
a	{ *action*$_1$ }
abb	{ *action*$_2$ }
a*b+	{ *action*$_3$ }



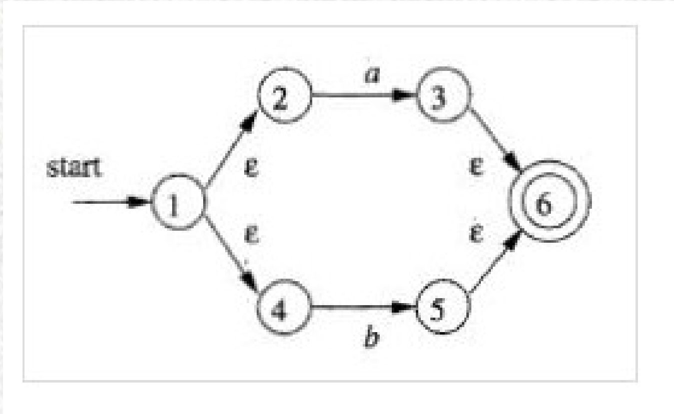
Combining the NFAs of a Set of Regular Expressions

r=(a|b)*abb

r1 = a，r2=b, we have NFA:



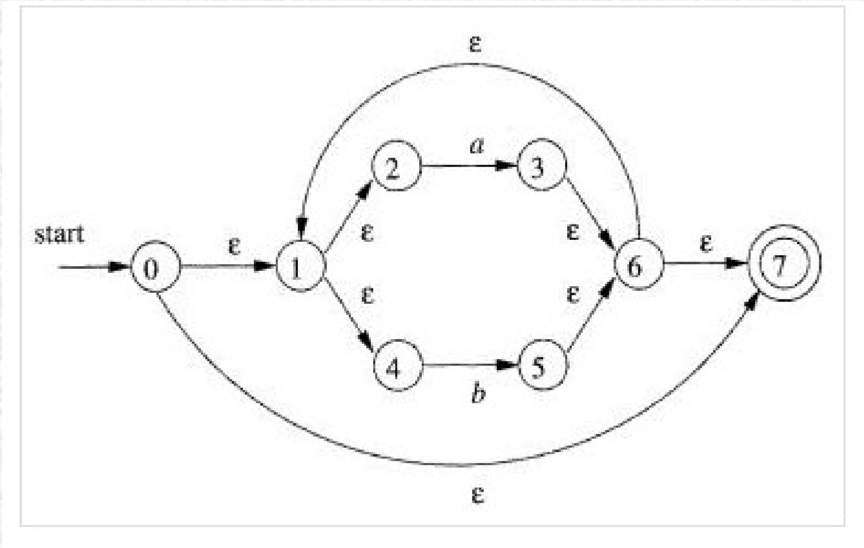
r3 = r1|r2, we have NFA:



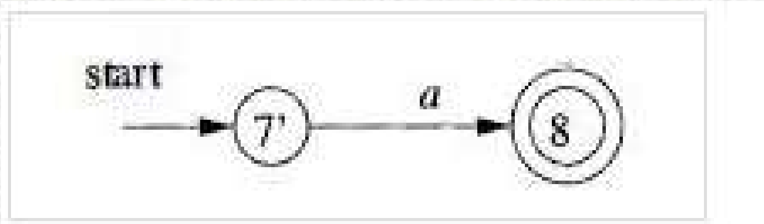
Combining the NFAs of a Set of Regular Expressions

r=(a|b)*abb

r5 = r3*, we have NFA:



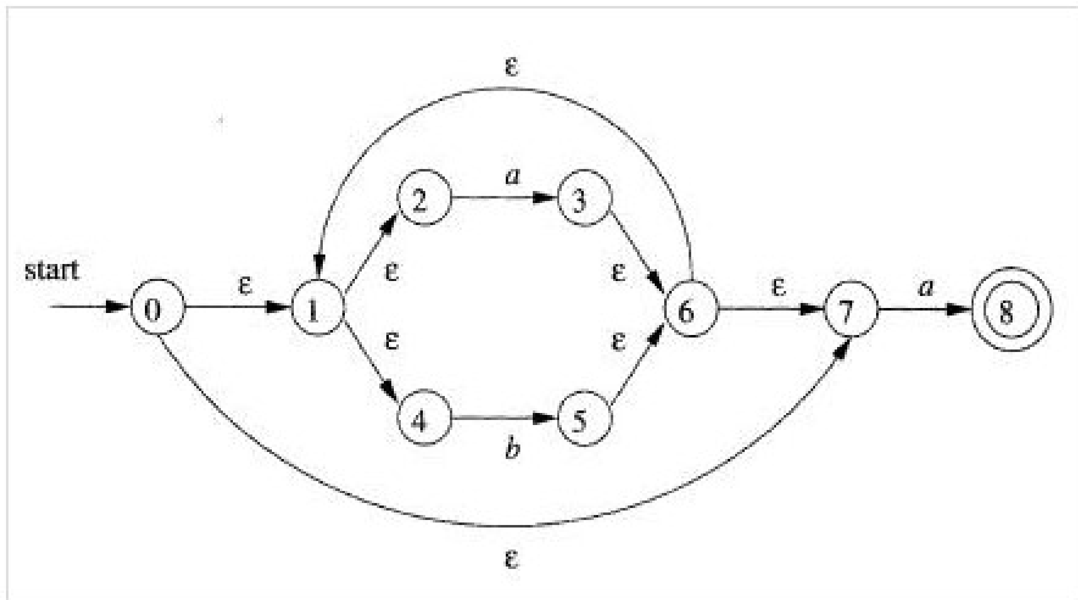
r6 = a, we have NFA:



Combining the NFAs of a Set of Regular Expressions

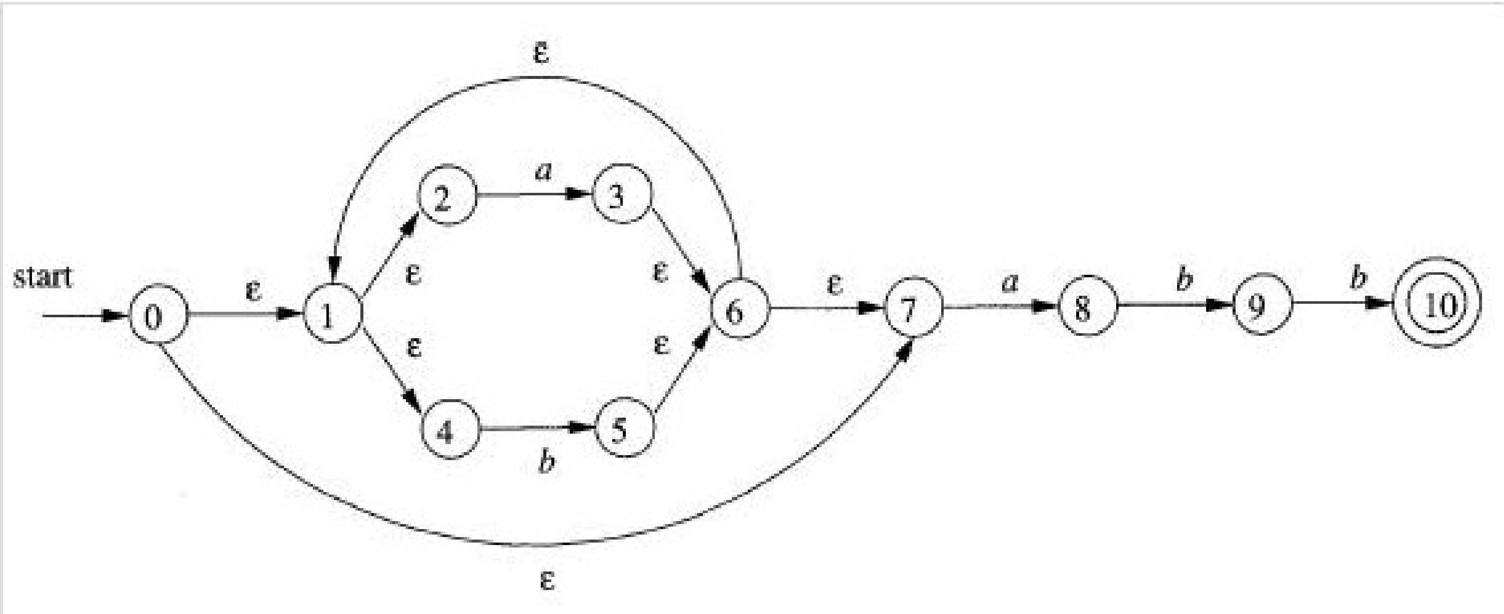
r=(a|b)*abb

r7 = r5r6, we have NFA:



Combining the NFAs of a Set of Regular Expressions

r=(a|b)*abb

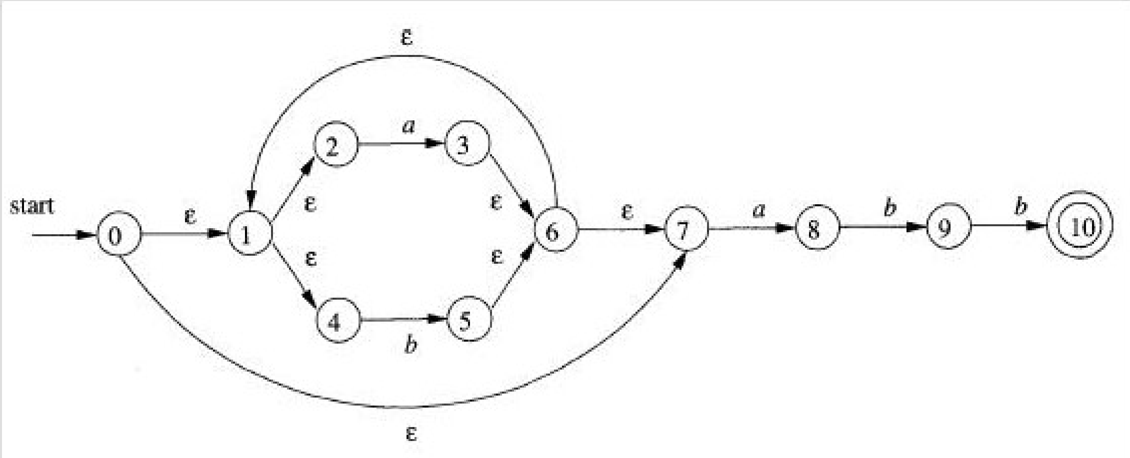


From NFA to DFA

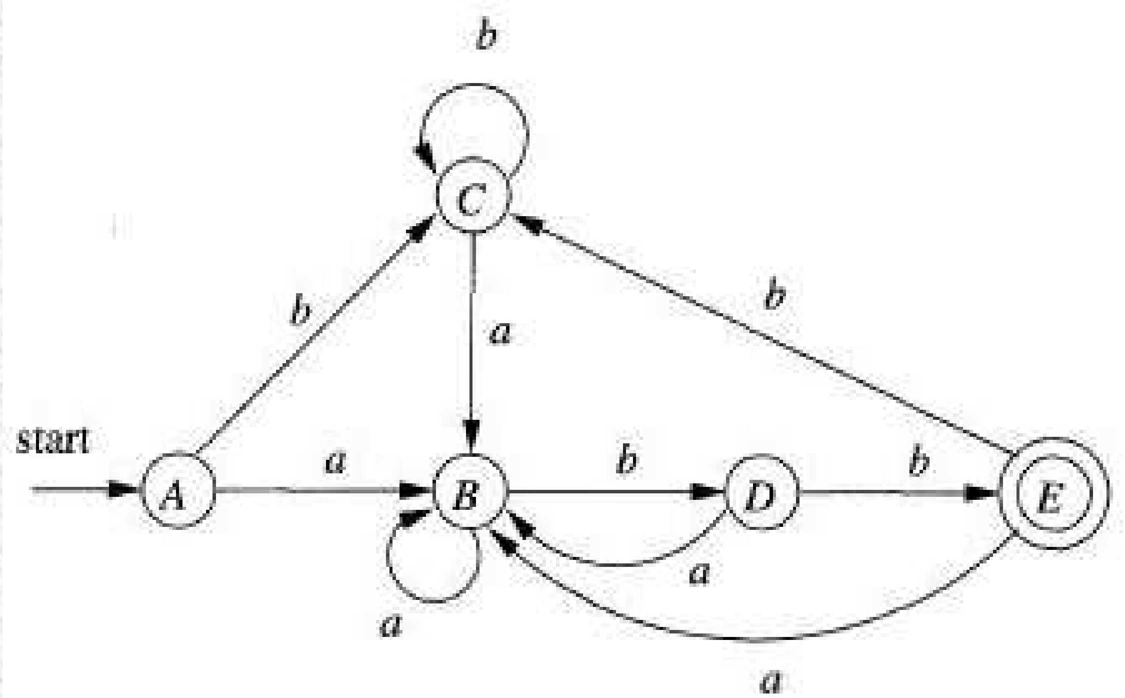
The Subset Construction Algorithm

```
initially, ε-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = ε-closure(move(T, a));
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```

From NFA to DFA

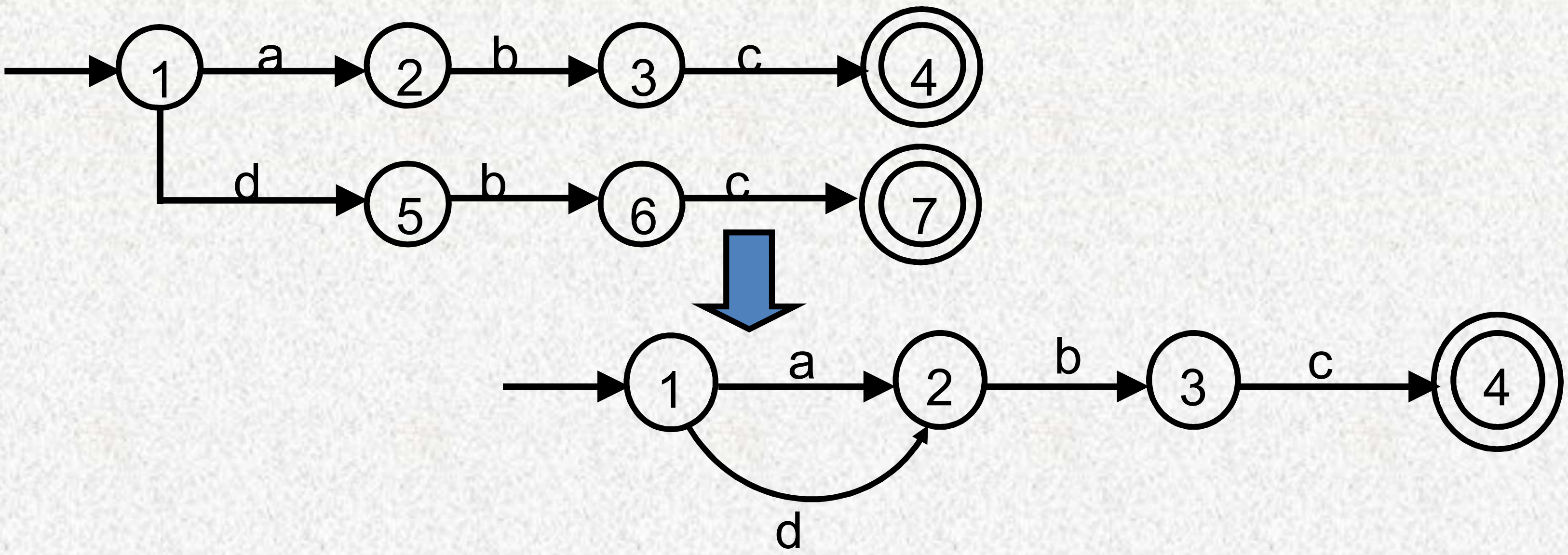


NFA STATE	DFA STATE	a	b
{0, 1, 2, 4, 7}	A	B	C
{1, 2, 3, 4, 6, 7, 8}	B	B	D
{1, 2, 4, 5, 6, 7}	C	B	C
{1, 2, 4, 5, 6, 7, 9}	D	B	E
{1, 2, 3, 5, 6, 7, 10}	E	B	C



Minimizing DFA

After conversion from NFA, the DFA may contain some equivalent states, which lead to low efficiency in the analysis



Minimizing DFA

- ***Lots*** of methods
- All involve finding **equivalent states**:
  - States that go to equivalent states under all inputs (sounds recursive)
- We will use the ***Partitioning Method***

Minimizing DFA

- Step 1
  - Start with an initial partition Π with two group: F and S-F (aceepting and nonaccepting)
- Step 2
  - Split Procedure
- Step 3
  - If ($\Pi_{new} = \Pi$)

    $\Pi_{final} = \Pi$ and continue step 4

    else

    $\Pi = \Pi_{new}$ and go to step 2
- Step 4
  - Construct the minimum-state DFA by Π_{final} group.
  - Delete the dead state

Split Procedure

```
initially, let Π_new = Π;
for ( each group G of Π ) {
        partition G into subgroups such that two states s and t
                are in the same subgroup if and only if for all
                input symbols a, states s and t have transitions on a
                to states in the same group of Π;
        /* at worst, a state will be in a subgroup by itself */
        replace G in Π_new by the set of all subgroups formed;
}
```

Minimizing the DFA

- DFA D=({0,1,2,3,4,5}, {a,b}, δ, 0, {0,1}),其中δ见表

states	a	b
0	1	2
1	1	4
2	1	3
3	3	2
4	0	5
5	5	4

Step 1:
A={0,1}, B={2,3,4,5}。

States	partition	a	b
0	A	1(A)	2(B)
1	A	1(A)	4(B)
2	B	1(A)	3(B)
3	B	3(B)	2(B)
4	B	0(A)	5(B)
5	B	5(B)	4(B)

Major operation: partition states into equivalent classes according to: final / non-final states; transition functions

Minimizing the DFA

- DFA D=({0,1,2,3,4,5}, {a,b}, δ, 0, {0,1}),

states	partition	a	b
0	A	1(A)	2(B)
1	A	1(A)	4(B)
2	B	1(A)	3(B)
3	B	3(B)	2(B)
4	B	0(A)	5(B)
5	B	5(B)	4(B)

states	a	b
0	1	2
1	1	4
2	1	3
3	3	2
4	0	5
5	5	4

Cannot be divided any more

states	partition	a	b
0	A	1(A)	2(B)
1	A	1(A)	4(B)
2	B	1(A)	3(C)
3	C	3(C)	2(B)
4	B	0(A)	5(C)
5	C	5(C)	4(B)

Minimizing the DFA

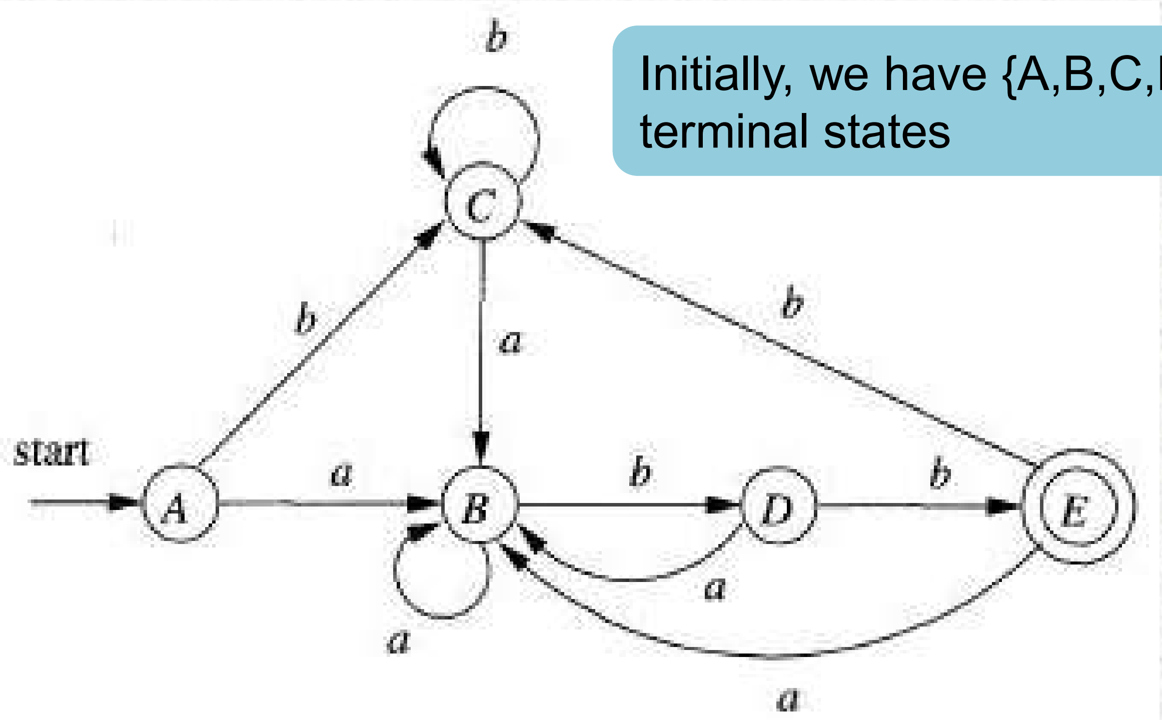
- **DFA D=({0,1,2,3,4,5}, {a,b}, δ, 0, {0,1}) is minimized to:**

DFA D′=({A,B,C}, {a,b}, δ， A， {A})， where δ is defined as follows

state	a	b
A	A	B
B	A	C
C	C	B

Minimizing the DFA-Example

- r=(a|b)*abb



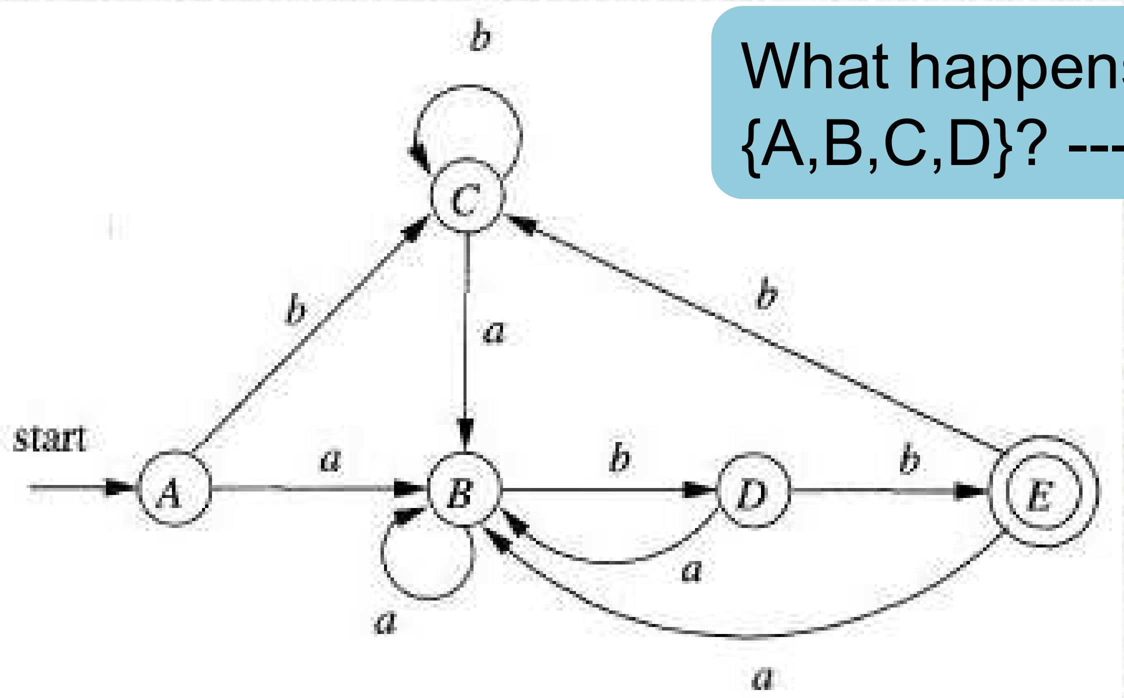
Initially, we have {A,B,C,D},{E}, which are for non-terminal and terminal states

{E} is not dividable, so we only consider {A, B, C, D}

Minimizing the DFA-Example

Is {A,B,C,D} dividable?

- **r=(a|b)*abb**

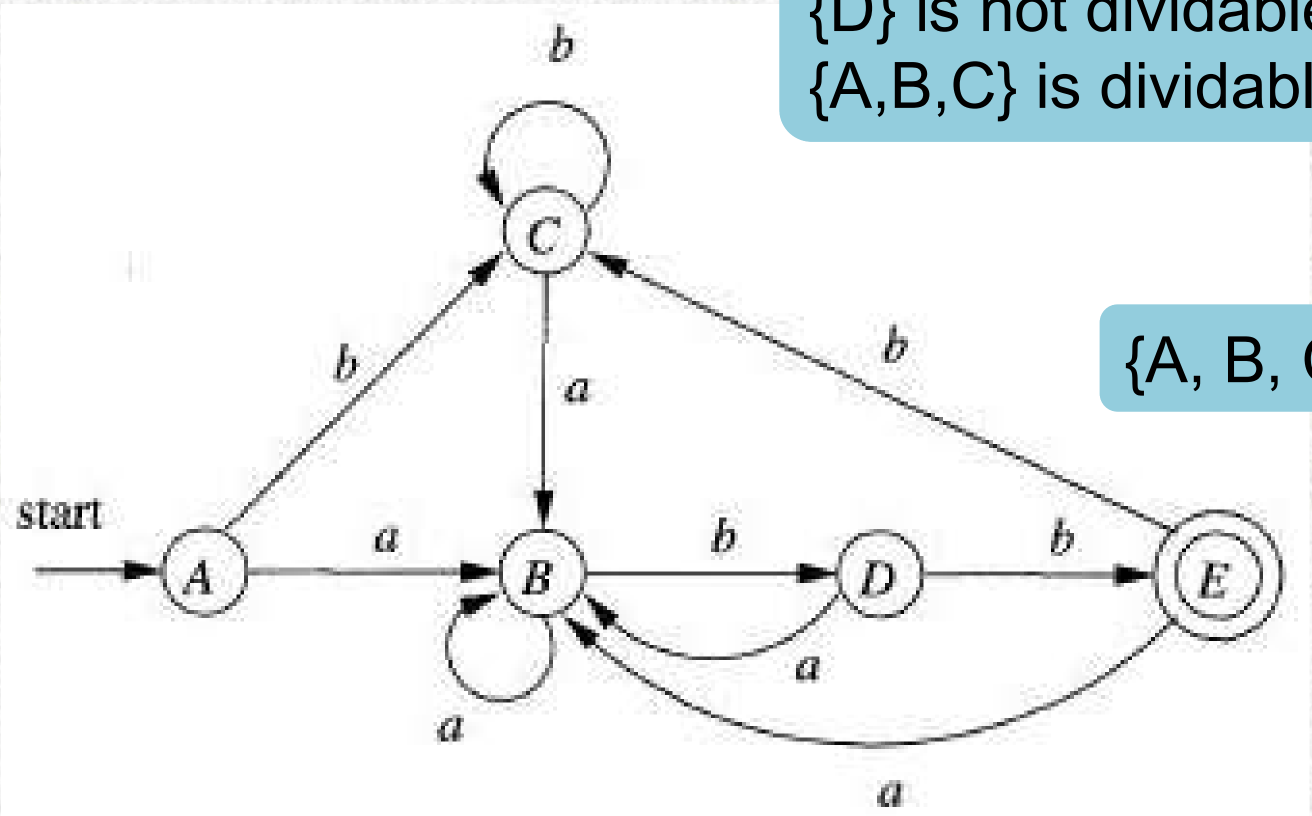


What happens when take in a under {A,B,C,D}? --- still with {A, B, C, D}

What happens when take in b under {A,B,C,D}? --- becomes {A,B,C}，{D}

Minimizing the DFA-Example

- r=(a|b)*abb



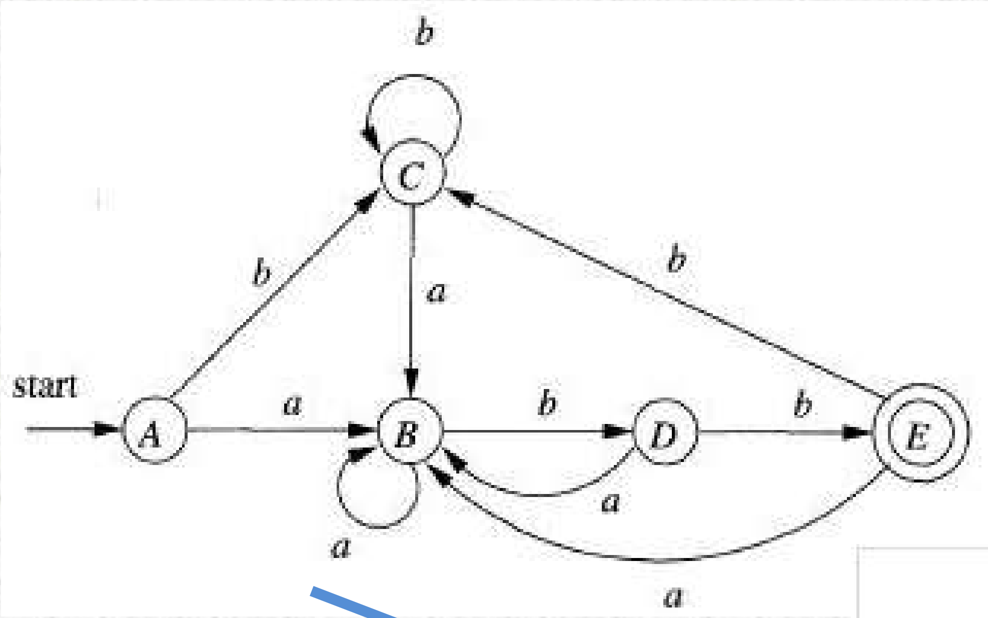
{D} is not dividable, so let us see whether {A,B,C} is dividable?

{A, B, C} becomes {A,C},{B}

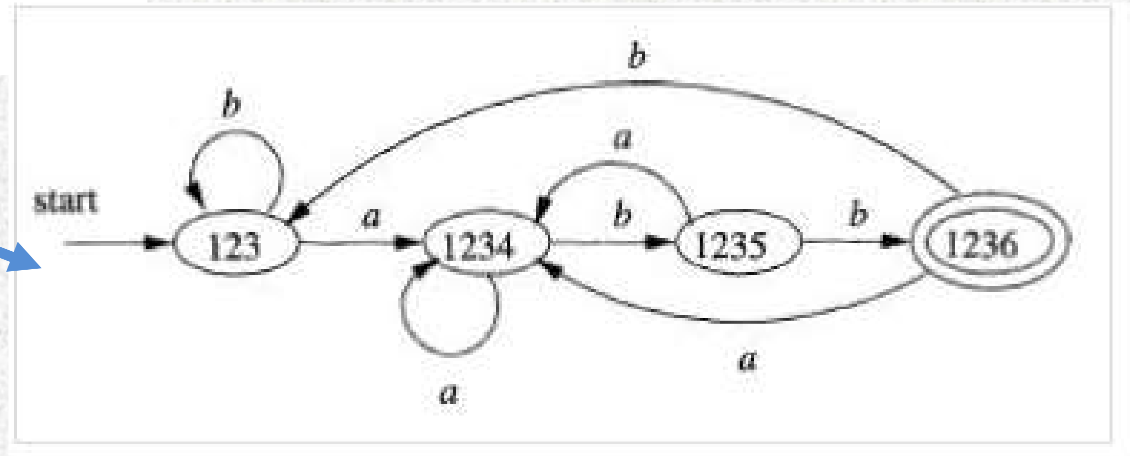
Minimizing the DFA-Example

- r=(a|b)*abb

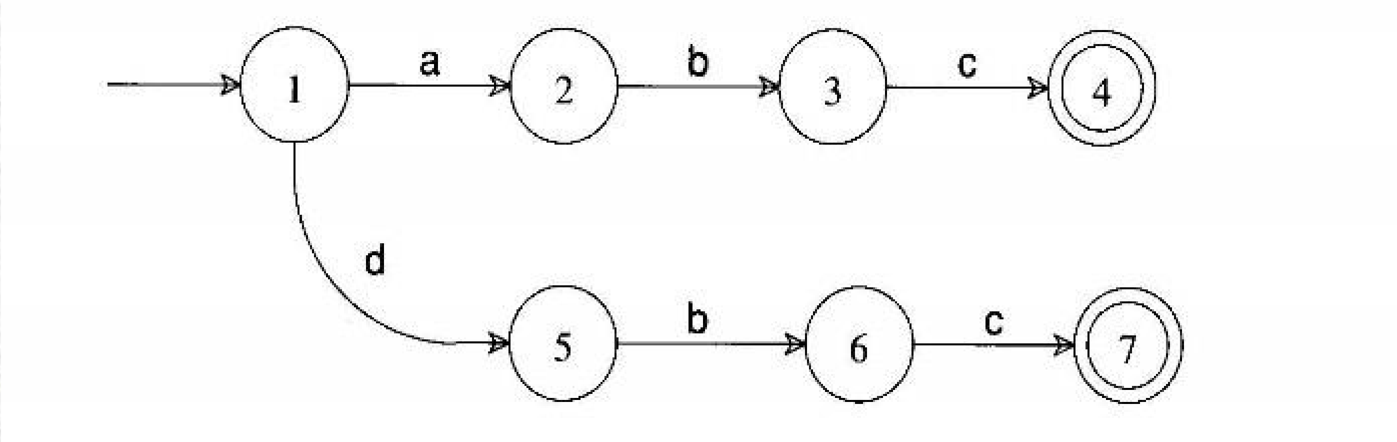
{A,C} is dividable?



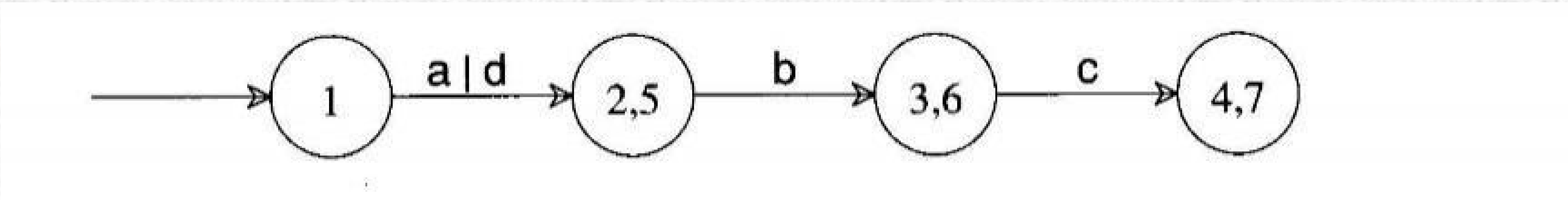
Finally, we have {A,C},{B},{D},{E}



Example

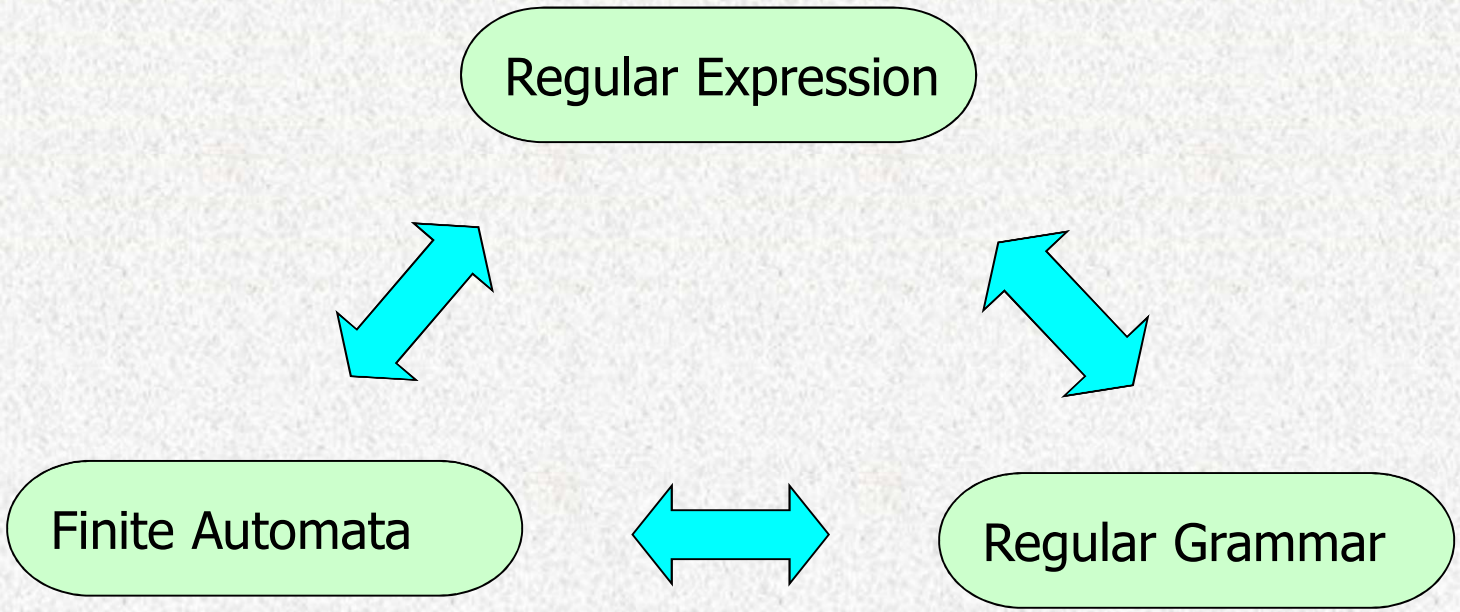


- initially, two sets {1, 2, 3, 5, 6}, {4, 7}.
- {1, 2, 3, 5, 6} splits {1, 2, 5}, {3, 6} on c.
- {1, 2, 5} splits {1}, {2, 5} on b.



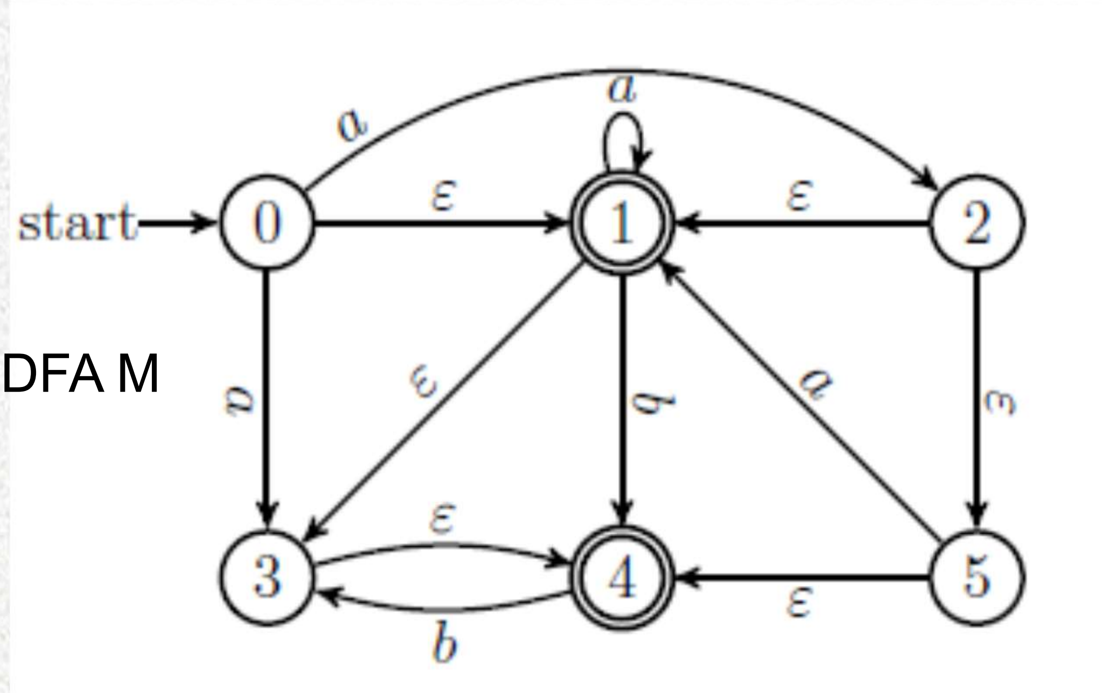
RE v.s. NFA/DFA

- RE， DFA(NFA)， L(RE) are equivalent to each other



Exercise

- Given an NFA N



(1) Simulate the NFA on input “aaabb”

(2) Convert the NFA N to its equivalent DFA M

(3) Minimize the DFA M

(4) Describe what can this DFA/NFA accept in natural language

(5) Write down the regular expression re, such that L(re) = L(N)

Homework-W3

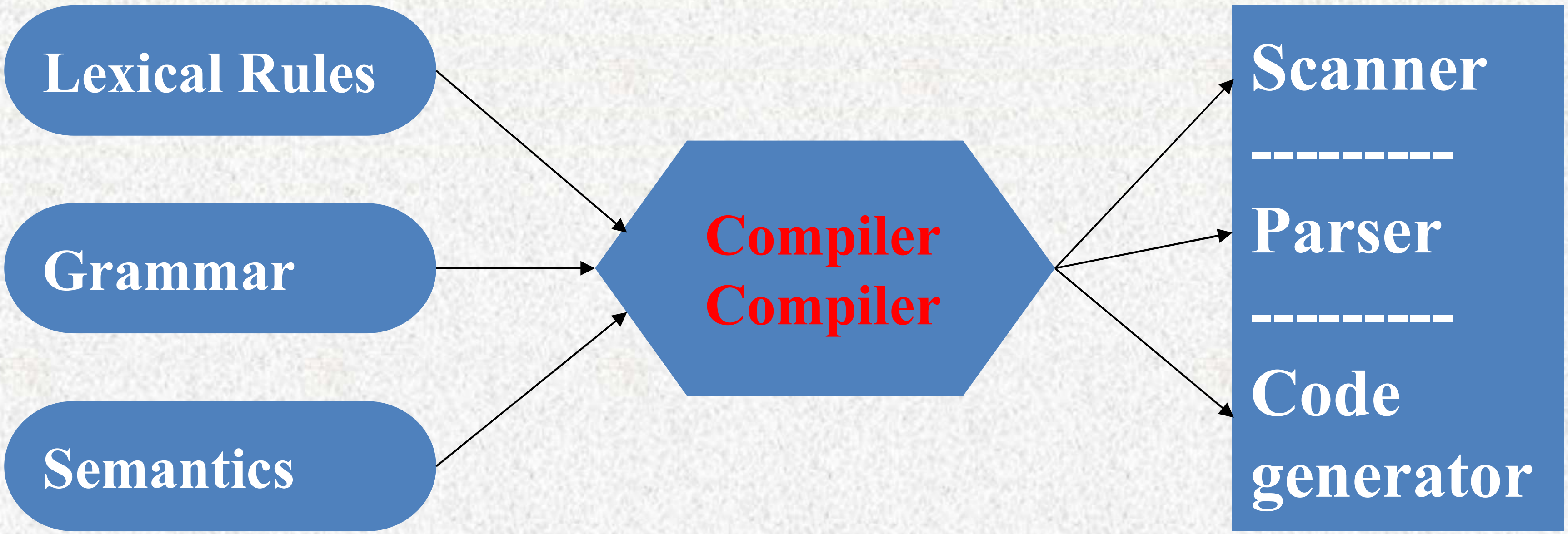
Homework – week 3

- pp. 125, Exercise 3.3.5 (c)(d)(f)(h)
- pp.152, Exercise 3.6.5
- pp. 166, Exercise 3.7.1 (b), Exercise 3.7.2 (b), Exercise 3.7.3 (d)
- pp. 172, Exercise 3.8.1
- pp.187, Exercise 3.9.4

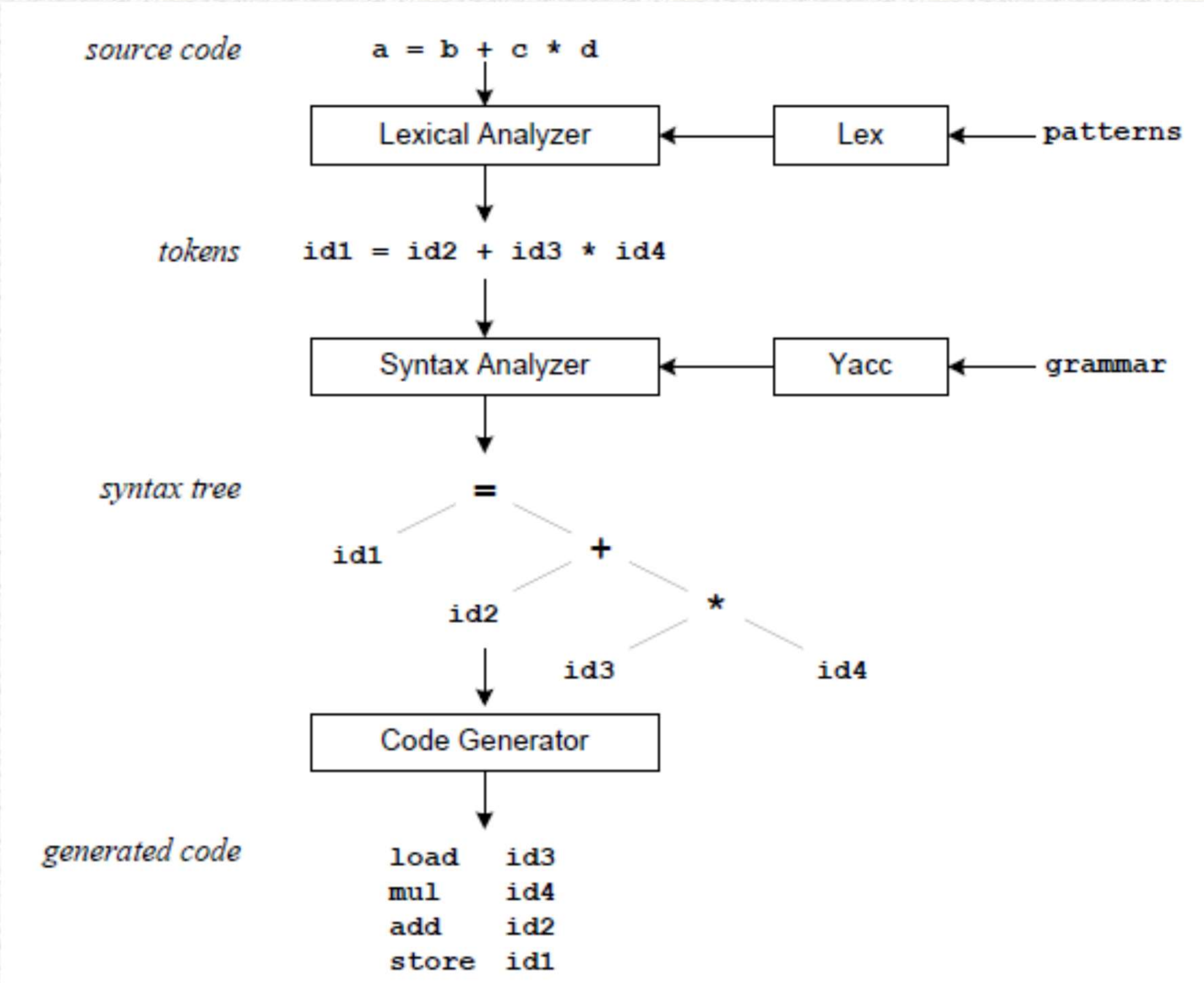
Lexical Analyzer Implementation

Overview

- Writing a compiler is difficult requiring lots of time and effort
- Construction of the scanner and parser is routine enough that the process may be automated



Overview


source code
a = b + c * d
Lexical Analyzer
Lex
patterns
tokens
id1 = id2 + id3 * id4
Syntax Analyzer
Yacc
grammar
syntax tree
=
id1
+
id2
*
id3
id4
Code Generator
generated code
load id3
mul id4
add id2
store id1

LEX

- Lex is a scanner generator
  - Input is description of patterns and actions
  - Output is a C program which contains a function yylex() which, when called, matches patterns and performs actions per input
  - Typically, the generated scanner performs lexical analysis and produces tokens for the (YACC-generated) parser

YACC

- What is **YACC** ?
  - **Tool which will produce a parser for a given grammar.**
  - YACC (Yet Another Compiler Compiler) is a program designed to compile a LALR(1) grammar and to produce the source code of the syntactic analyzer of the language produced by this grammar
  - Input is a grammar (rules) and actions to take upon recognizing a rule
  - Output is a C program and optionally a header file of tokens

LEX and YACC: a team

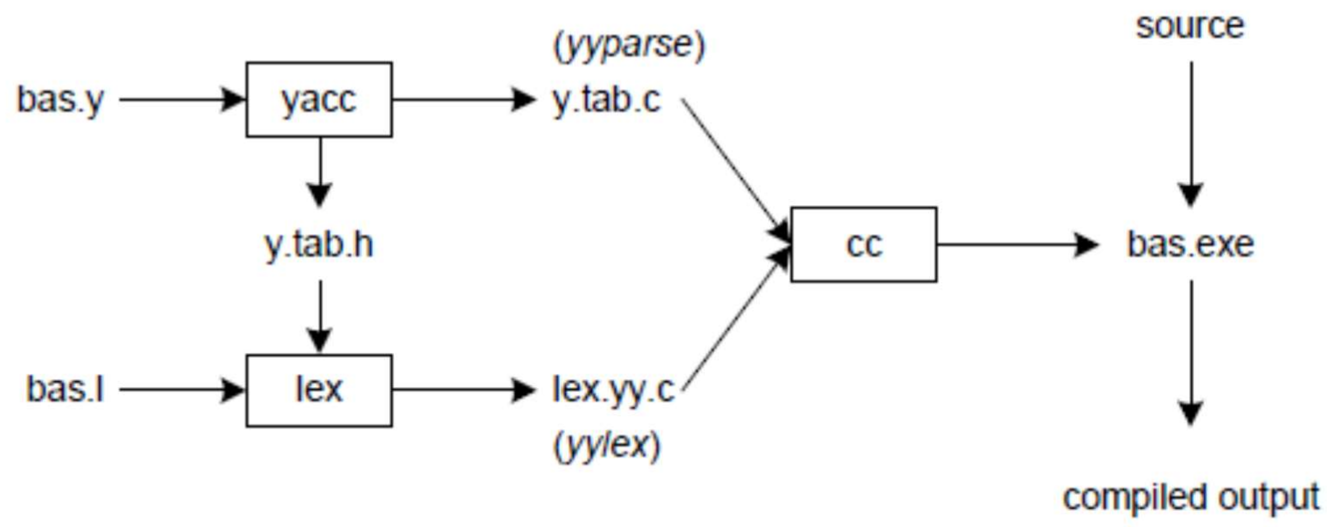


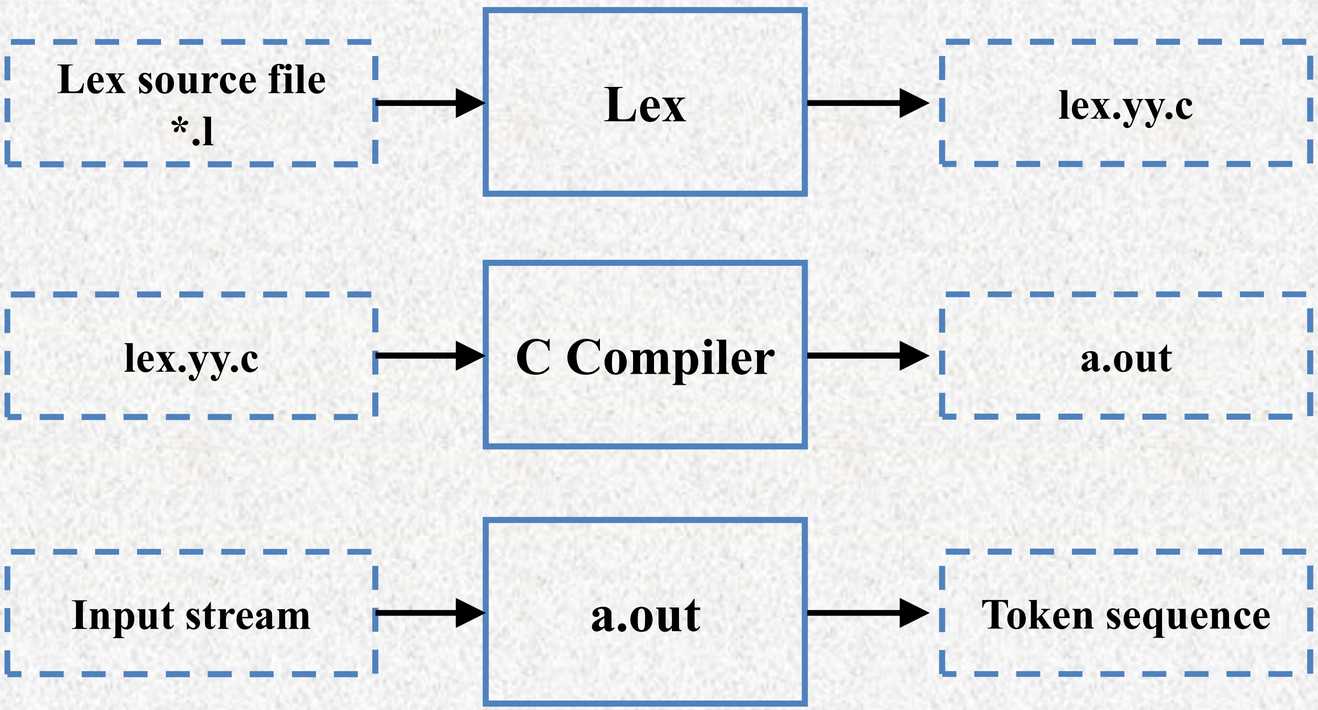
Figure 2: Building a Compiler with Lex/Yacc

```
yacc -d bas.y                   # create y.tab.h, y.tab.c
lex bas.l                       # create lex.yy.c
cc lex.yy.c y.tab.c -obas.exe   # compile/link
```

Availability

- lex, yacc on most UNIX systems
- bison: a yacc replacement from GNU
- flex: *f*ast *lex*ical analyzer
- BSD yacc
- Windows/MS-DOS versions exist

Lex



Create your lexical analyzer with Lex

Structure of Lex source file

```
Declaration
%%
Translation
   rules
%%
Auxiliary
functions
```

```
%{ constant
}%
```

Regular definition

Pattern {Action}:

- Pattern is a regular expression or regular definition
- Action is in C, describing the actions after matching the regular expression

```
%{ ID,NUM,IF,ADD
}%
```

```
letter  [A-Za-z]
digit  [0-9]
id     {letter}({letter}|{digit})*
num {digit}+
```

```
if   {return (IF);}
+   {return(ADD);}
{id} {yylval = strcpy(yytext,
       yylength); return(ID);  }
{num} {yylval = Change();
          return(NUM);}
```

Functions used in the action

```
int  Change()
{  /*Convert string into integer*/
}
```

yylval: value of the token
yytext: lexeme of the token
yyleng: length of the lexeme

Example: LEX

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
id          [_a-zA-Z][_a-zA-Z0-9]*
wspc        [ \t\n]+
semi        [;]
comma       [,]
%%
int         { return INT; }
char        { return CHAR; }
float       { return FLOAT; }
{comma}     { return COMMA; }          /* Necessary? */
{semi}      { return SEMI; }
{id}        { return ID;}
{wspc}      {;}
```

Example: Definitions

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%start  line
%token  CHAR, COMMA, FLOAT, ID, INT, SEMI
%%
```

Example: Rules

```
/* This production is not part of the "official"
 * grammar. It's primary purpose is to recover from
 * parser errors, so it's probably best if you leave
 * it here. */

line :  /* lambda */
     | line decl
     | line error {
        printf("Failure :-(\n");
        yyerrok;
        yyclearin;
            }
     ;
```

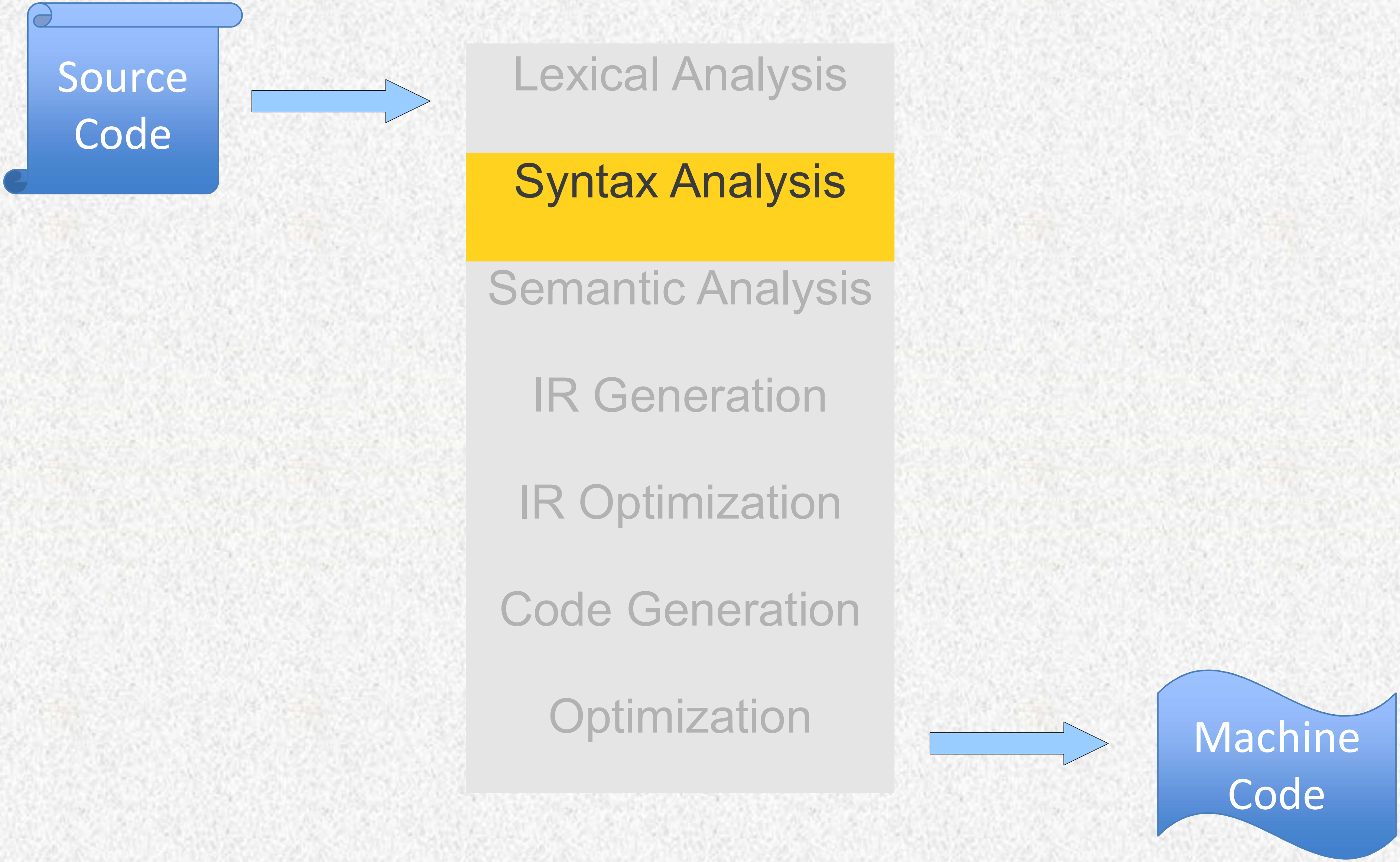
Example: Rules

```
decl :   type ID list { printf("Success!\n"); } ;

list :   COMMA ID list
     | SEMI
     ;
type :    INT | CHAR | FLOAT
     ;

%%
```

Example: Supplementary Code

```
extern FILE *yyin;
main()
{
    do {
        yyparse();
    } while(!feof(yyin));
}
yyerror(char *s)
{
   /* Don't have to do anything! */
}
```

Next Time


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code