

# Review for the Midterm

COMS W4115

Prof. Stephen A. Edwards

Fall 2006

Columbia University

Department of Computer Science

# The Midterm

70 minutes

4–5 problems

Closed book

One sheet of notes of your own devising

Comprehensive: Anything discussed in class is fair game

Little, if any, programming.

Details of ANTLR/C/Java/Prolog/ML syntax not required

Broad knowledge of languages discussed

# Topics

Structure of a Compiler

Scripting Languages

Scanning and Parsing

Regular Expressions

Context-Free Grammars

Top-down Parsing

Bottom-up Parsing

ASTs

Name, Scope, and Bindings

# Compiling a Simple Program

```
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

# What the Compiler Sees

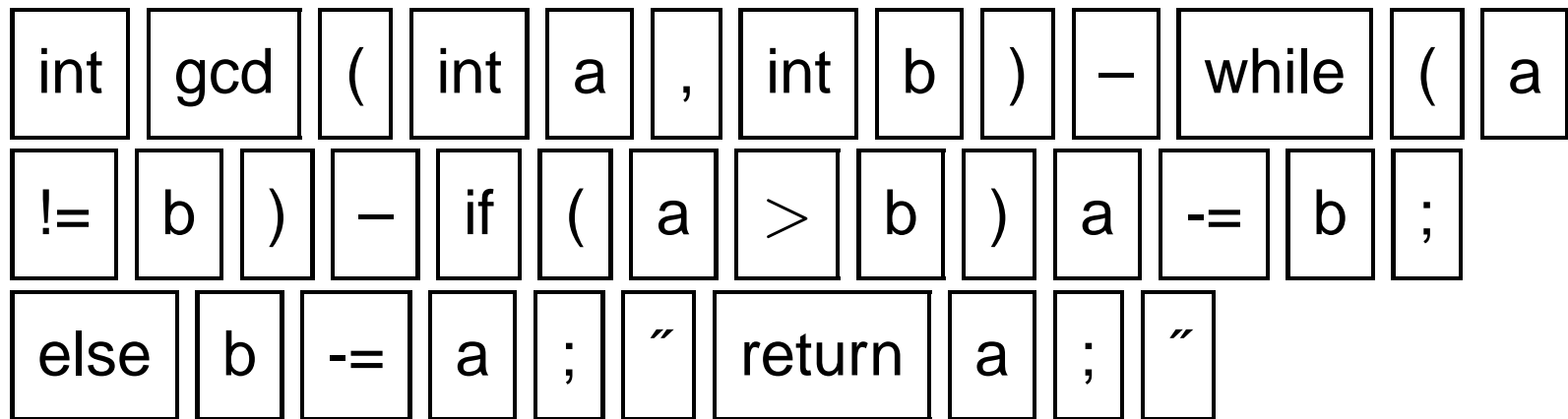
```
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

```
i n t   s p   g c d   (   i n t   s p   a   ,   s p   i
n t   s p   b   )   n l   {   n l   s p   s p   w h i l e   s p
(   a   s p   !   =   s p   b   )   s p   {   n l   s p   s p   s p   s p   i
f   s p   (   a   s p   >   s p   b   )   s p   a   s p   -   =   s p   b
;   n l   s p   s p   s p   s p   e l s e   s p   b   s p   -   =   s p
a   ;   n l   s p   s p   }   n l   s p   s p   r e t u r n   s p
a   ;   n l   }   n l
```

Text file is a sequence of characters

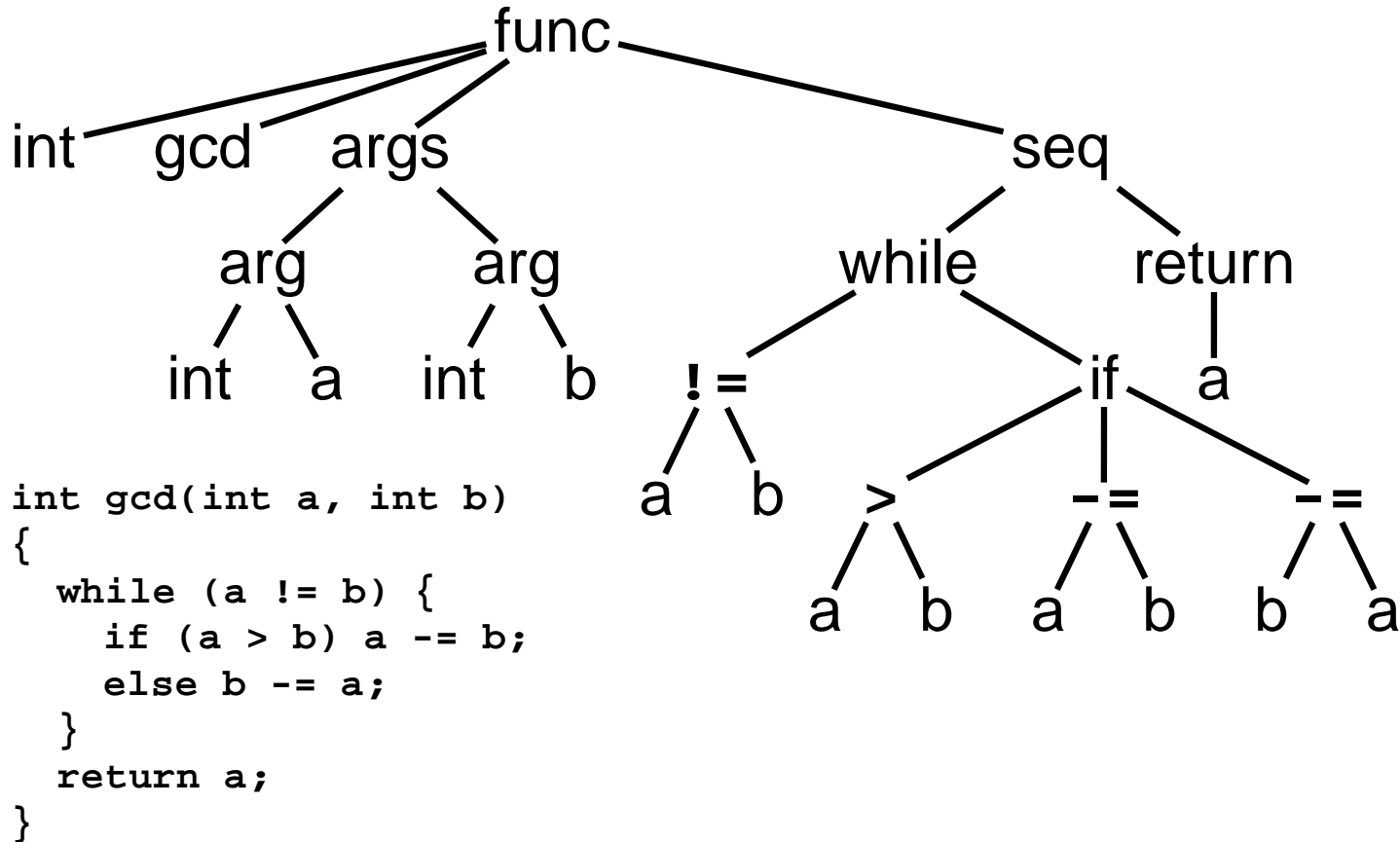
# Lexical Analysis Gives Tokens

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```



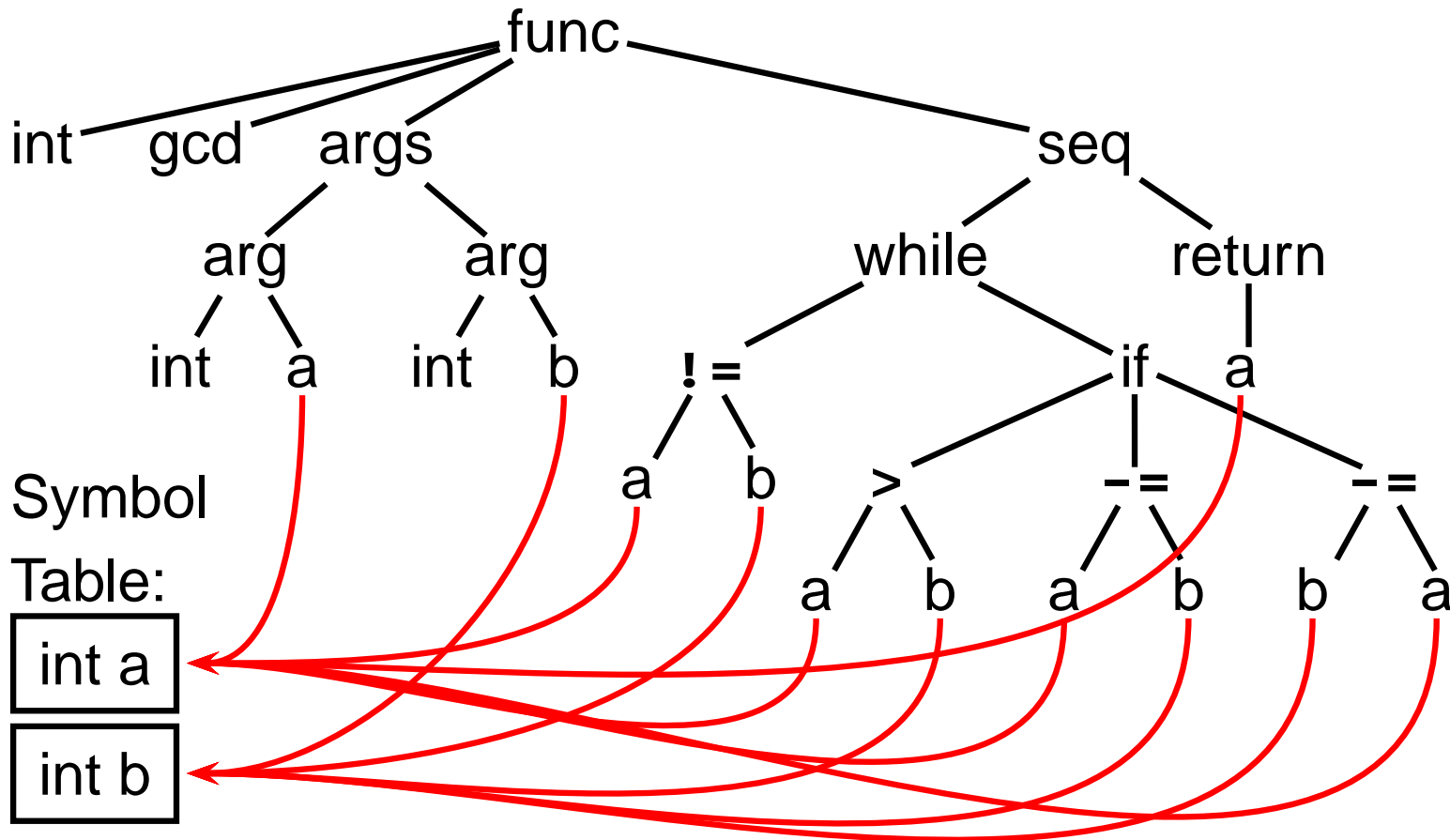
A stream of tokens. Whitespace, comments removed.

# Parsing Gives an AST



Abstract syntax tree built from parsing rules.

# Semantic Analysis Resolves Symbols



Types checked; references to symbols resolved



# Translation into 3-Address Code

```
L0: sne    $1,  a,  b
      seq    $0,  $1,  0
      btrue  $0,  L1    % while (a != b)
      sl     $3,  b,  a
      seq    $2,  $3,  0
      btrue  $2,  L4    % if (a < b)
      sub    a,   a,  b % a -= b
      jmp    L5
L4: sub    b,   b,  a % b -= a
L5: jmp    L0
L1: ret    a
```

```
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

Idealized assembly language w/ infinite registers

# Generation of 80386 Assembly

```
gcd:  pushl  %ebp                % Save frame pointer
      movl  %esp, %ebp
      movl  8(%ebp), %eax      % Load a from stack
      movl  12(%ebp), %edx    % Load b from stack
.L8:  cmpl  %edx, %eax
      je   .L3                % while (a != b)
      jle .L5                % if (a < b)
      subl %edx, %eax        % a -= b
      jmp .L8
.L5:  subl %eax, %edx        % b -= a
      jmp .L8
.L3:  leave                 % Restore SP, BP
      ret
```

# Scanning and Automata

# Describing Tokens

**Alphabet:** A finite set of symbols

Examples:  $\{ 0, 1 \}$ ,  $\{ A, B, C, \dots, Z \}$ , ASCII, Unicode

**String:** A finite sequence of symbols from an alphabet

Examples:  $\epsilon$  (the empty string), Stephen,  $\alpha\beta\gamma$

**Language:** A set of strings over an alphabet

Examples:  $\emptyset$  (the empty language),  $\{ 1, 11, 111, 1111 \}$ , all English words, strings that start with a letter followed by any sequence of letters and digits

# Operations on Languages

Let  $L = \{ \epsilon, \text{wo} \}$ ,  $M = \{ \text{man, men} \}$

**Concatenation:** Strings from one followed by the other

$LM = \{ \text{man, men, woman, women} \}$

**Union:** All strings from each language

$L \cup M = \{ \epsilon, \text{wo, man, men} \}$

**Kleene Closure:** Zero or more concatenations

$M^* = \{ \epsilon, M, MM, MMM, \dots \} =$   
 $\{ \epsilon, \text{man, men, manman, manmen, menman, menmen,}$   
 $\text{manmanman, manmanmen, manmenman, } \dots \}$

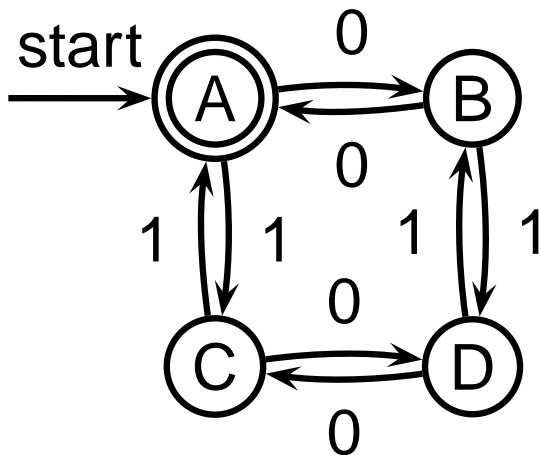
# Regular Expressions over an Alphabet $\Sigma$

A standard way to express languages for tokens.

1.  $\epsilon$  is a regular expression that denotes  $\{\epsilon\}$
2. If  $a \in \Sigma$ ,  $a$  is an RE that denotes  $\{a\}$
3. If  $r$  and  $s$  denote languages  $L(r)$  and  $L(s)$ ,
  - $(r)|(s)$  denotes  $L(r) \cup L(s)$
  - $(r)(s)$  denotes  $\{tu : t \in L(r), u \in L(s)\}$
  - $(r)^*$  denotes  $\cup_{i=0}^{\infty} L^i$  ( $L^0 = \emptyset$  and  $L^i = LL^{i-1}$ )

# Nondeterministic Finite Automata

“All strings containing an even number of 0’s and 1’s”



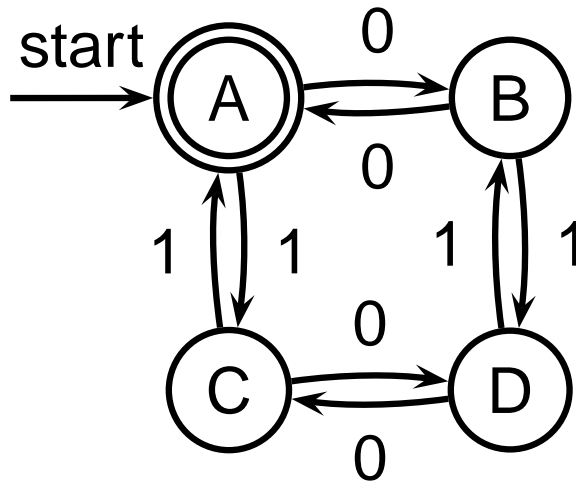
1. Set of states  $S: \{ \textcircled{\textcircled{A}}, \textcircled{B}, \textcircled{C}, \textcircled{D} \}$
2. Set of input symbols  $\Sigma: \{0, 1\}$
3. Transition function  $\sigma : S \times \Sigma_{\epsilon} \rightarrow 2^S$

state	$\epsilon$	0	1
A	–	{B}	{C}
B	–	{A}	{D}
C	–	{D}	{A}
D	–	{C}	{B}

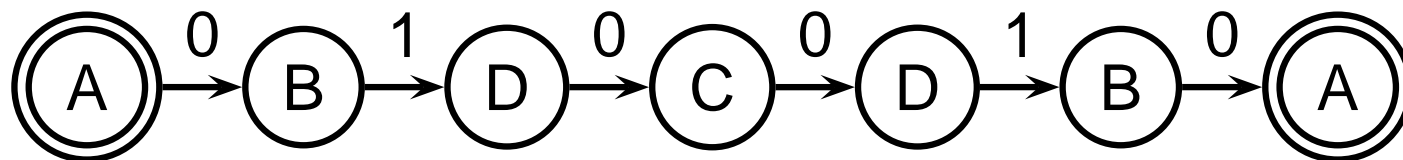
4. Start state  $s_0 : \textcircled{\textcircled{A}}$
5. Set of accepting states  $F: \{ \textcircled{\textcircled{A}} \}$

# The Language induced by an NFA

An NFA accepts an input string  $x$  iff there is a path from the start state to an accepting state that “spells out”  $x$ .

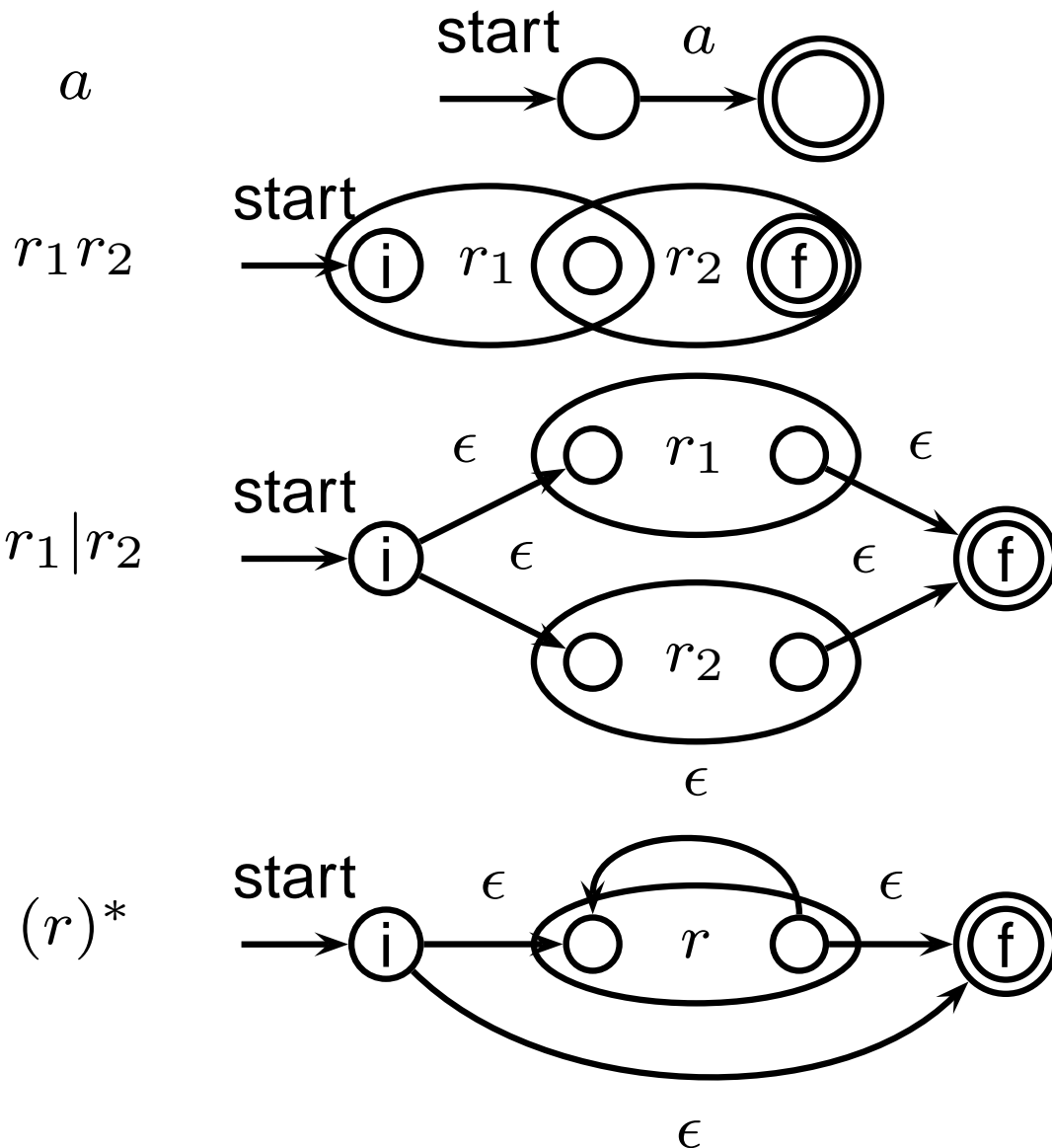


Show that the string “010010” is accepted.



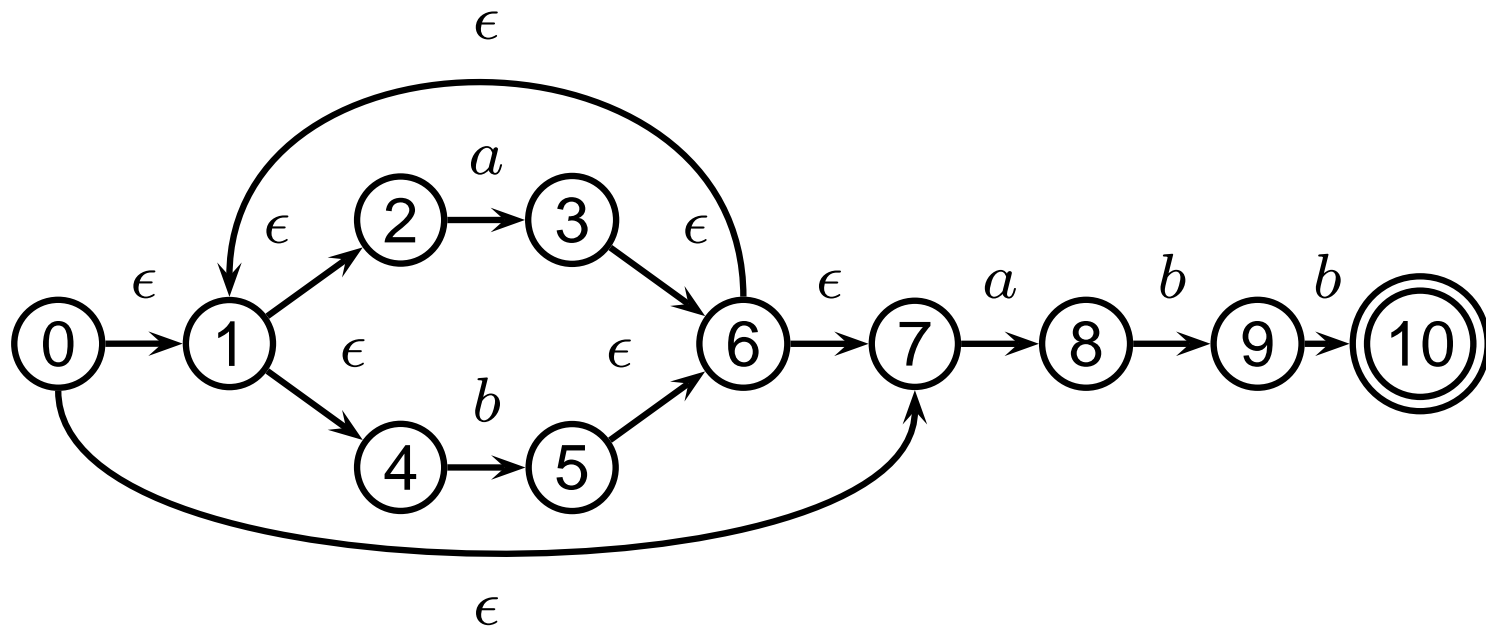


# Translating REs into NFAs

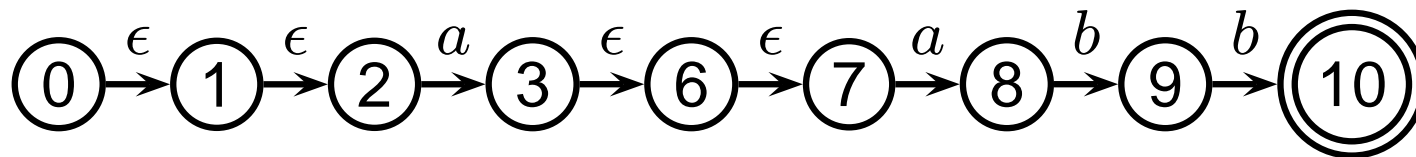


# Translating REs into NFAs

Example: translate  $(a|b)^*abb$  into an NFA



Show that the string “ $aabb$ ” is accepted.



# Simulating NFAs

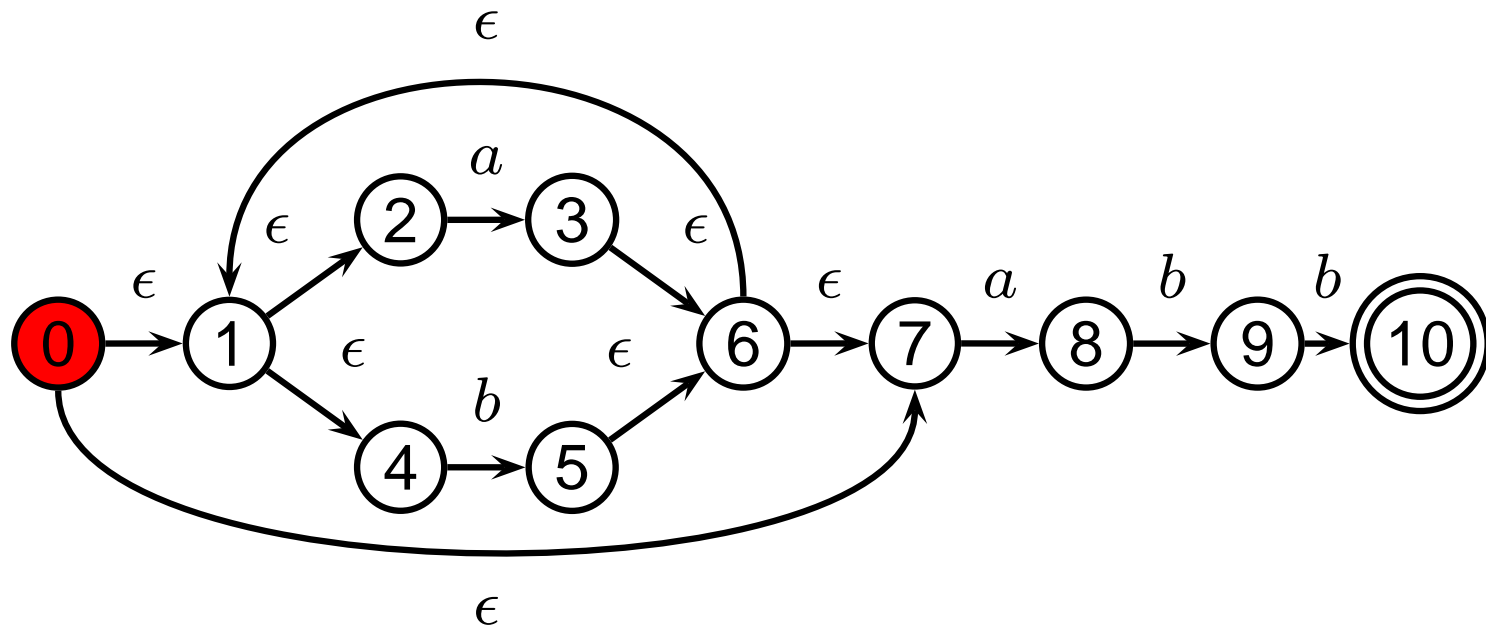
Problem: you must follow the “right” arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

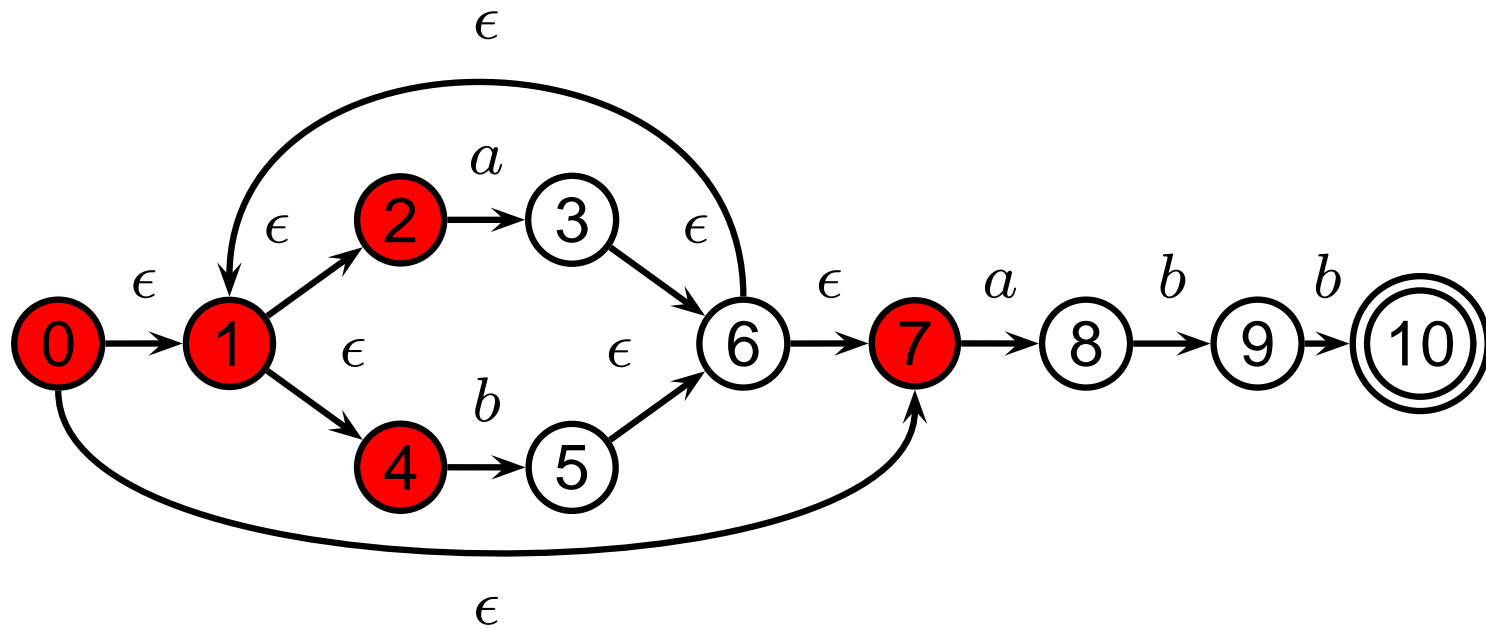
“Two-stack” NFA simulation algorithm:

1. Initial states: the  $\epsilon$ -closure of the start state
2. For each character  $c$ ,
  - New states: follow all transitions labeled  $c$
  - Form the  $\epsilon$ -closure of the current states
3. Accept if any final state is accepting

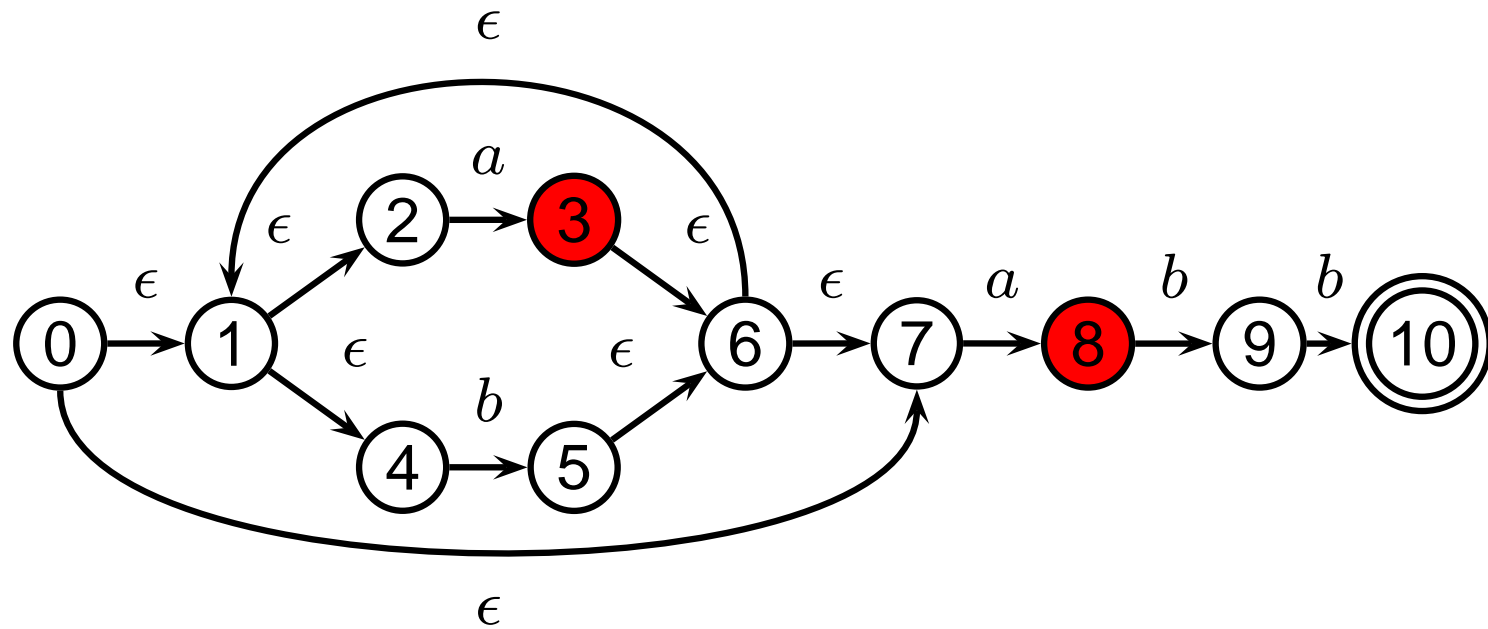
# Simulating an NFA: $\cdot aabb$ , Start



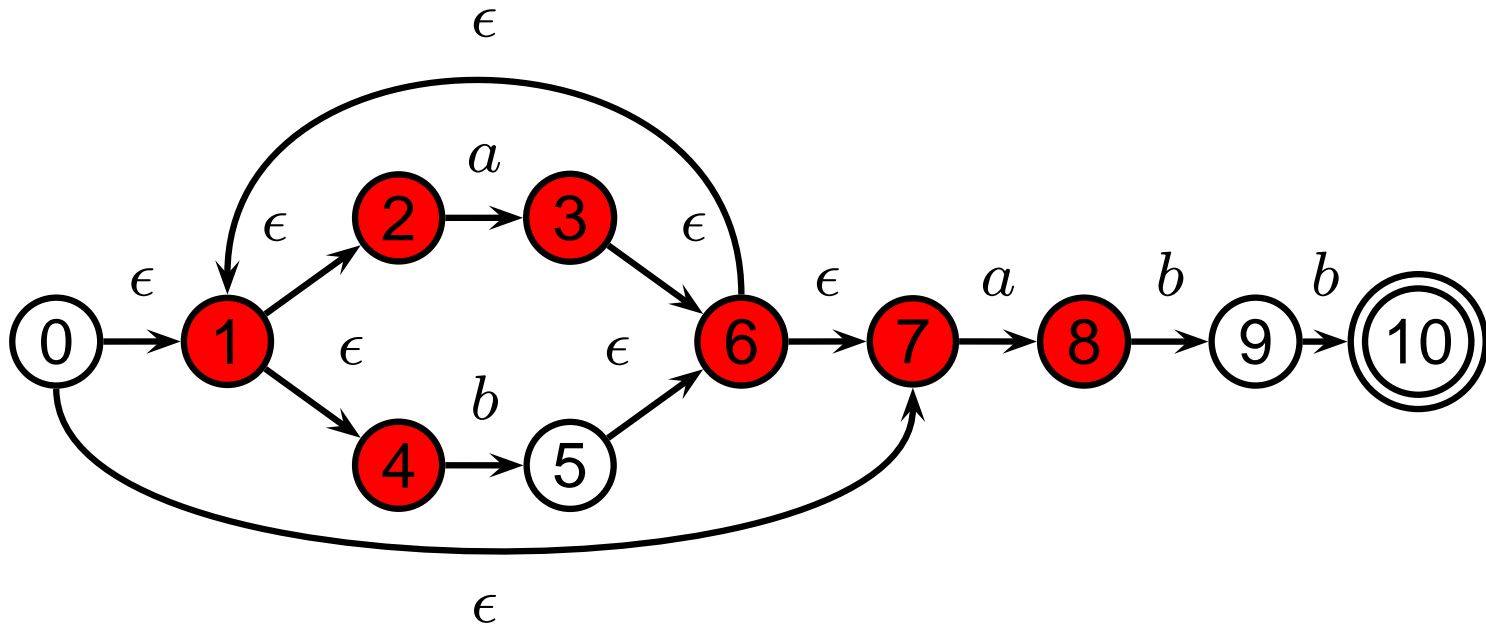
# Simulating an NFA: $\cdot aabb$ , $\epsilon$ -closure



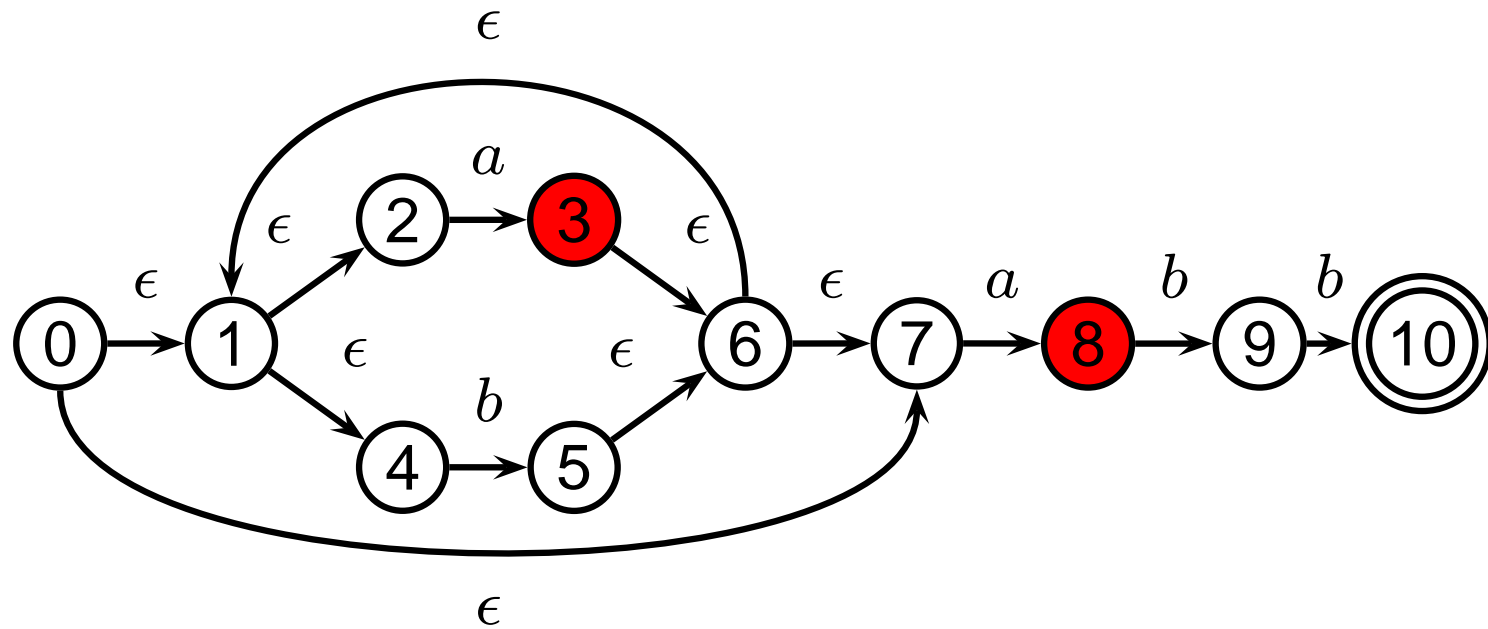
# Simulating an NFA: $a \cdot abb$



# Simulating an NFA: $a \cdot abb$ , $\epsilon$ -closure

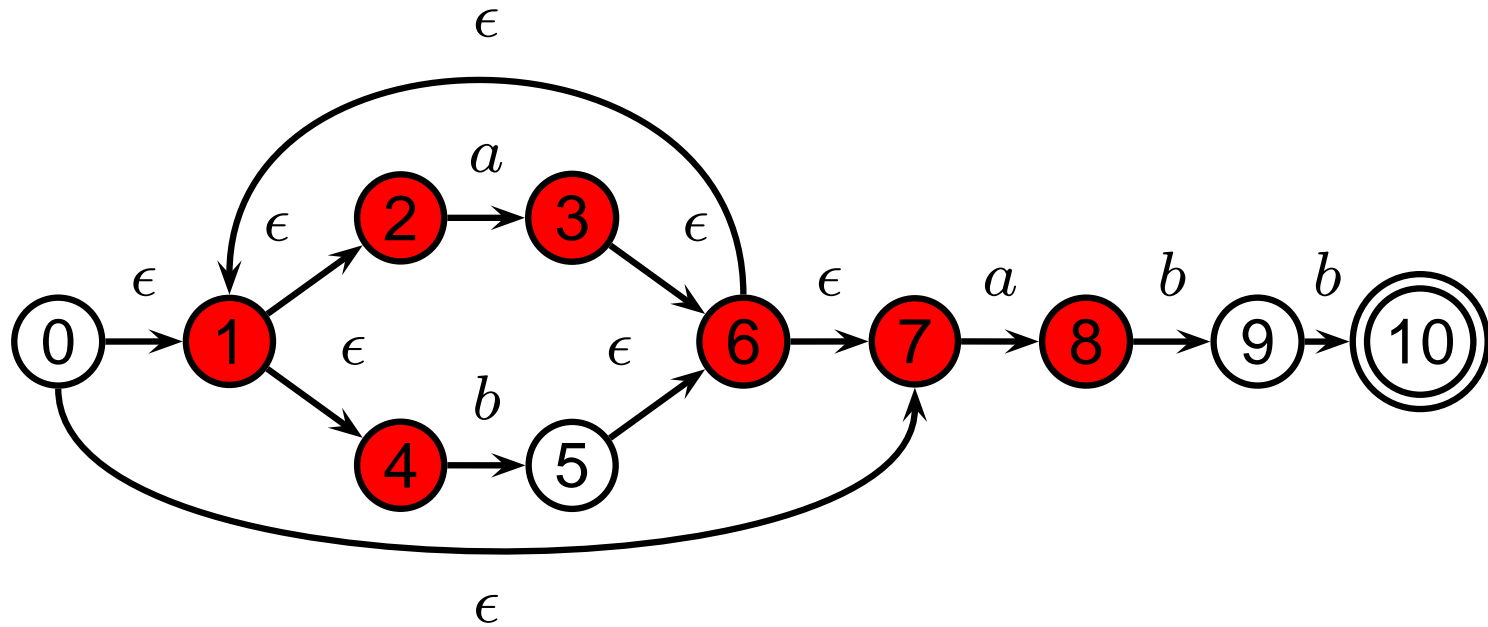


# Simulating an NFA: $aa \cdot bb$

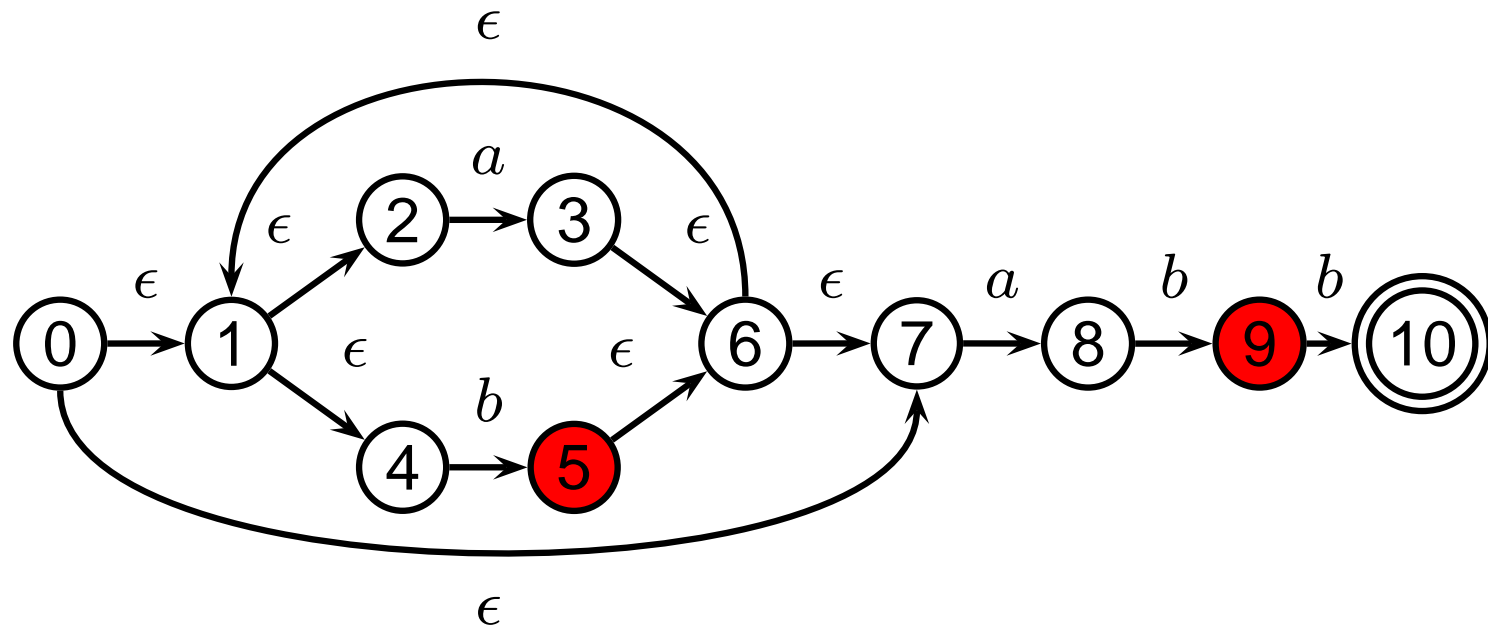




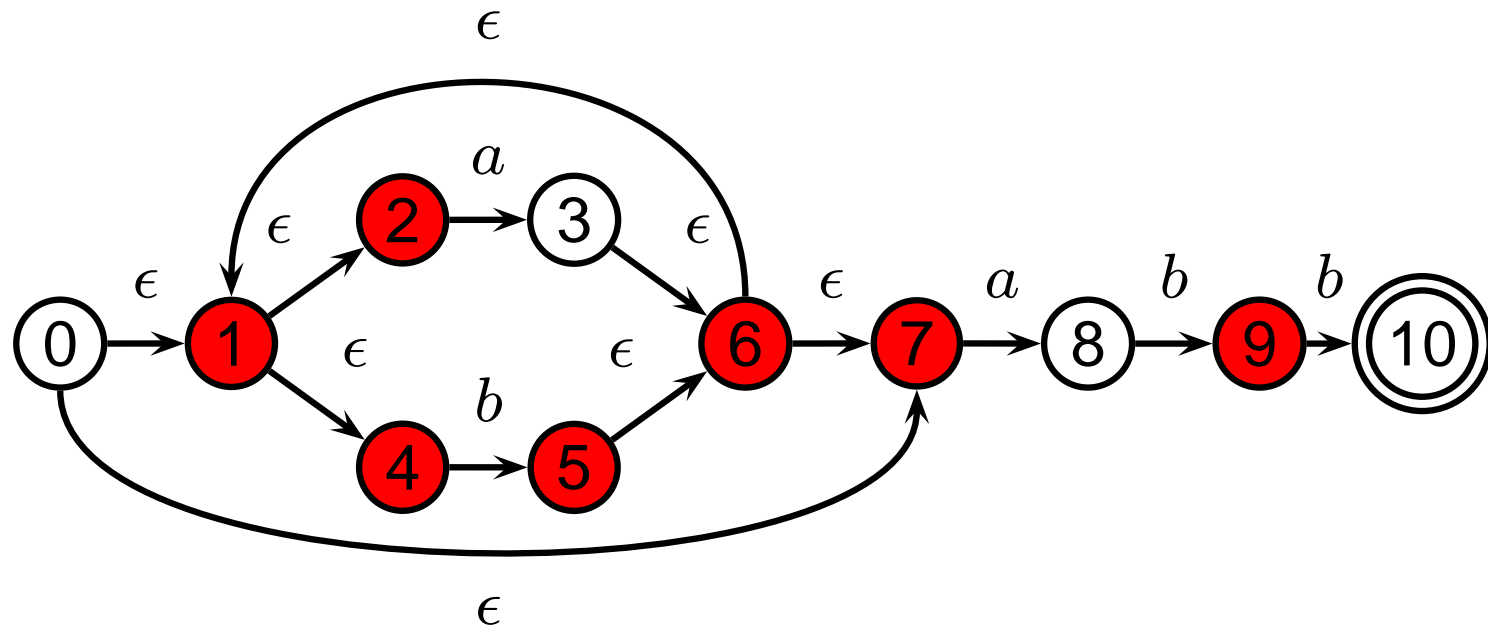
# Simulating an NFA: $aa \cdot bb$ , $\epsilon$ -closure



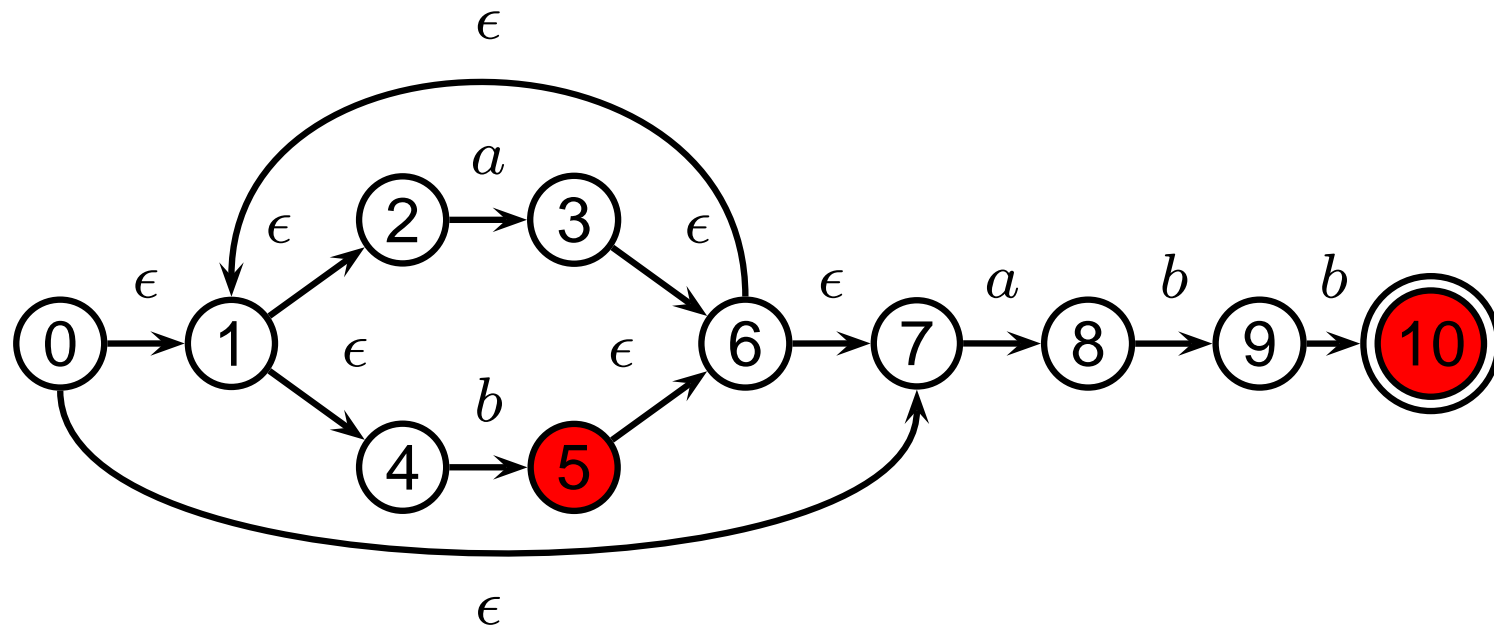
# Simulating an NFA: $aab \cdot b$



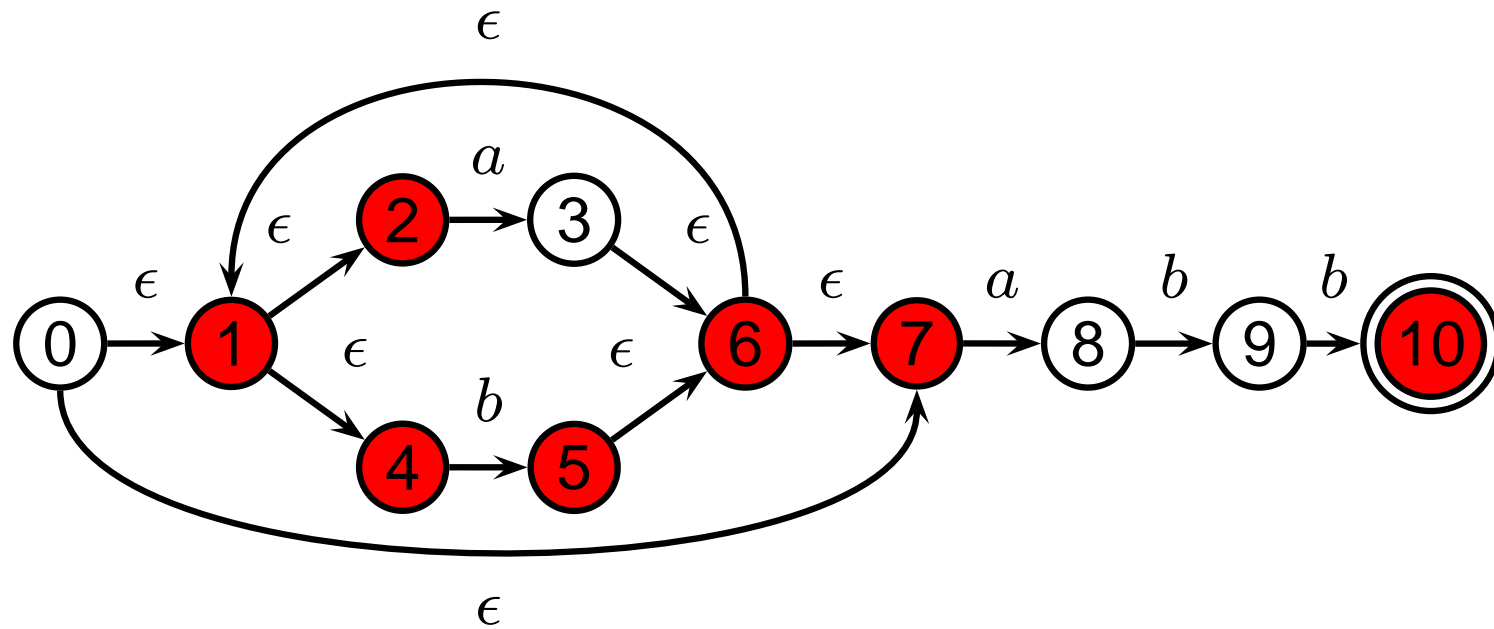
# Simulating an NFA: $aab \cdot b$ , $\epsilon$ -closure



# Simulating an NFA: $aabb$ .



# Simulating an NFA: *aabb.*, Done



# Deterministic Finite Automata

Restricted form of NFAs:

- No state has a transition on  $\epsilon$
- For each state  $s$  and symbol  $a$ , there is at most one edge labeled  $a$  leaving  $s$ .

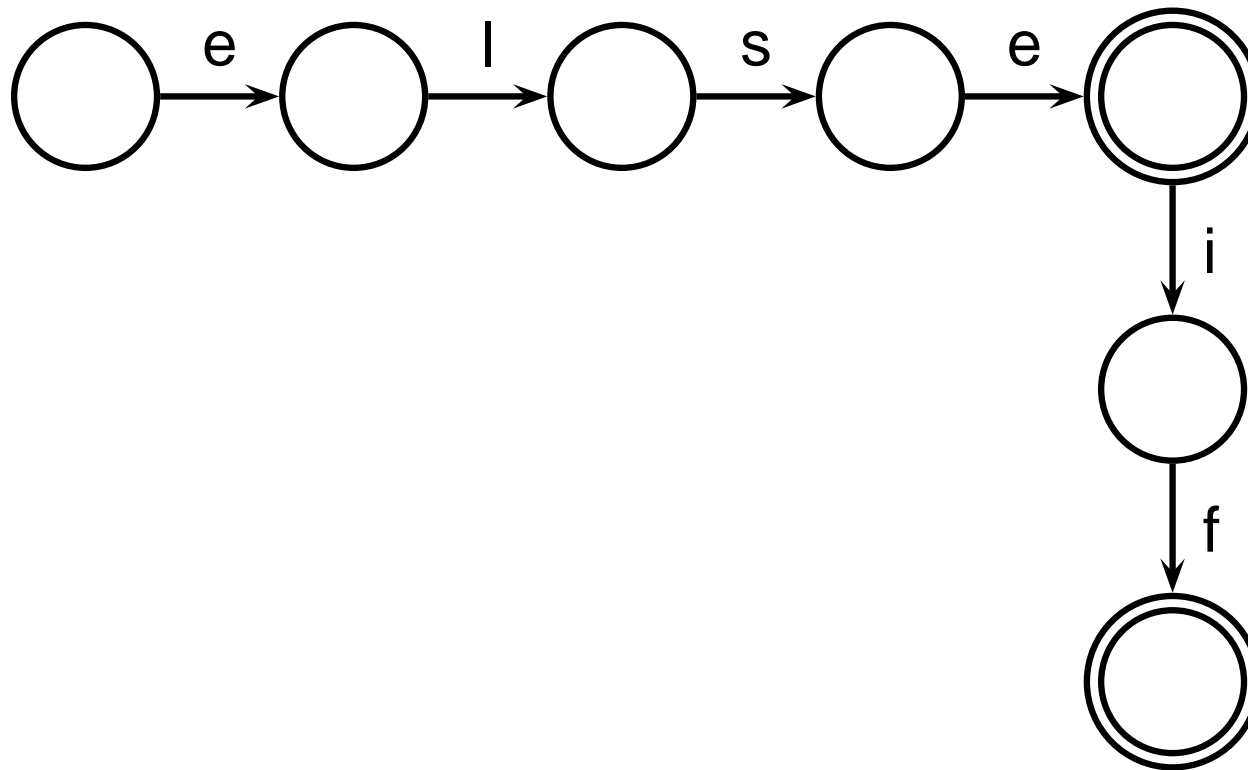
Differs subtly from the definition used in COMS W3261  
(Sipser, *Introduction to the Theory of Computation*)

Very easy to check acceptance: simulate by maintaining current state. Accept if you end up on an accepting state. Reject if you end on a non-accepting state or if there is no transition from the current state for the next symbol.

# Deterministic Finite Automata

`ELSE: "else" ;`

`ELSEIF: "elseif" ;`

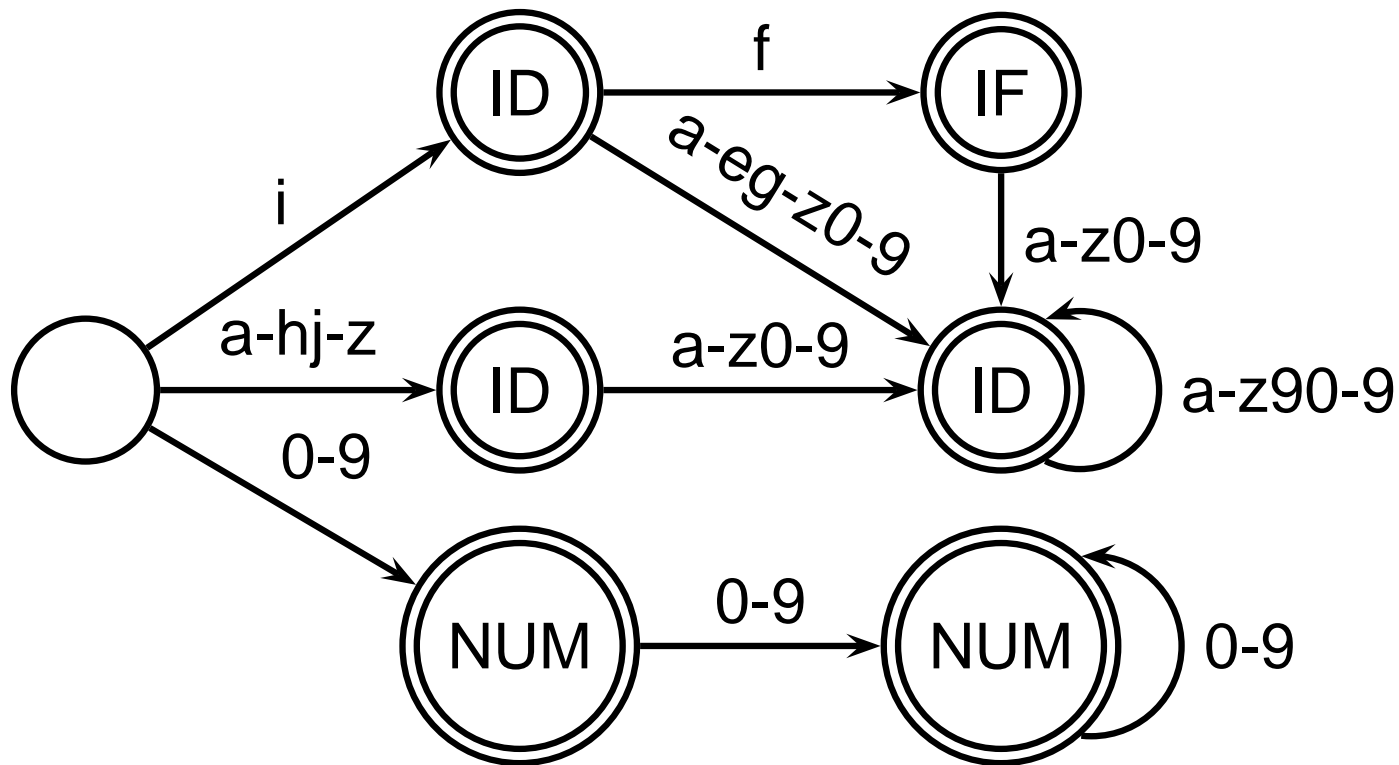


# Deterministic Finite Automata

IF: "if" ;

ID: 'a'..'z' ('a'..'z' | '0'..'9')

NUM: ('0'..'9')+





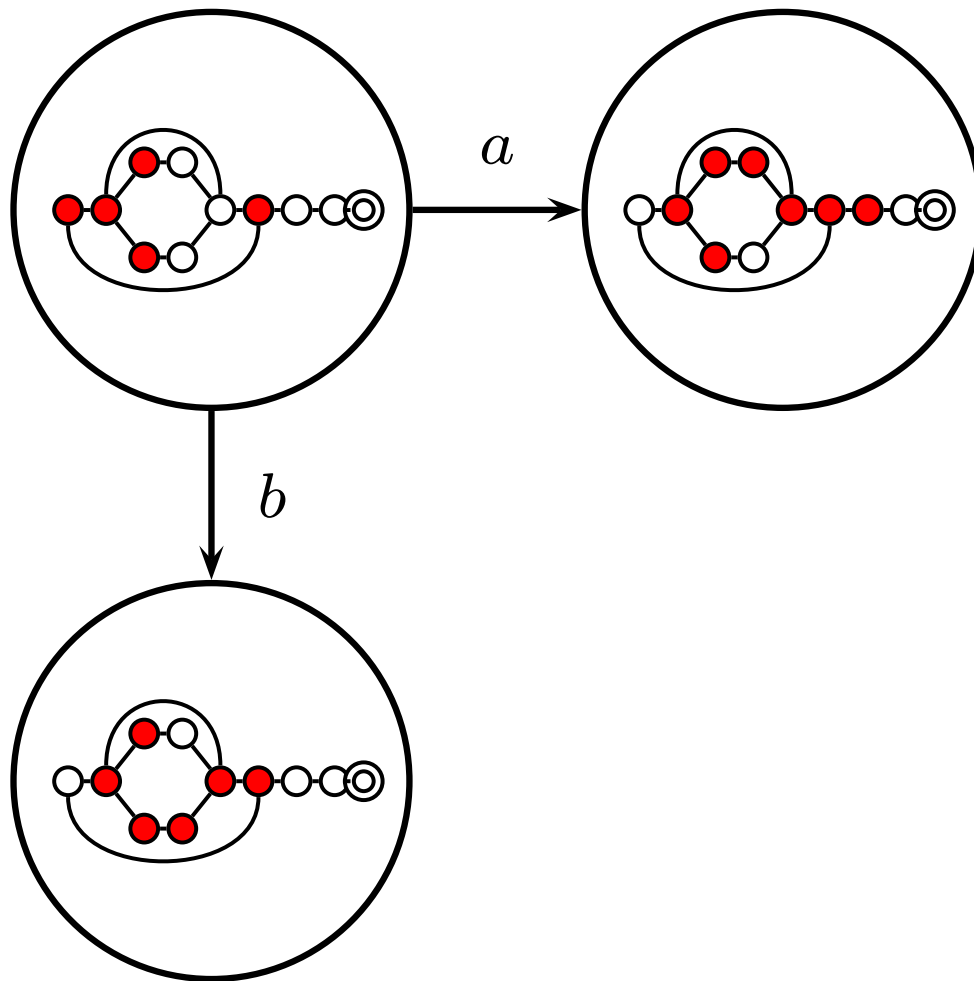
# Building a DFA from an NFA

Subset construction algorithm

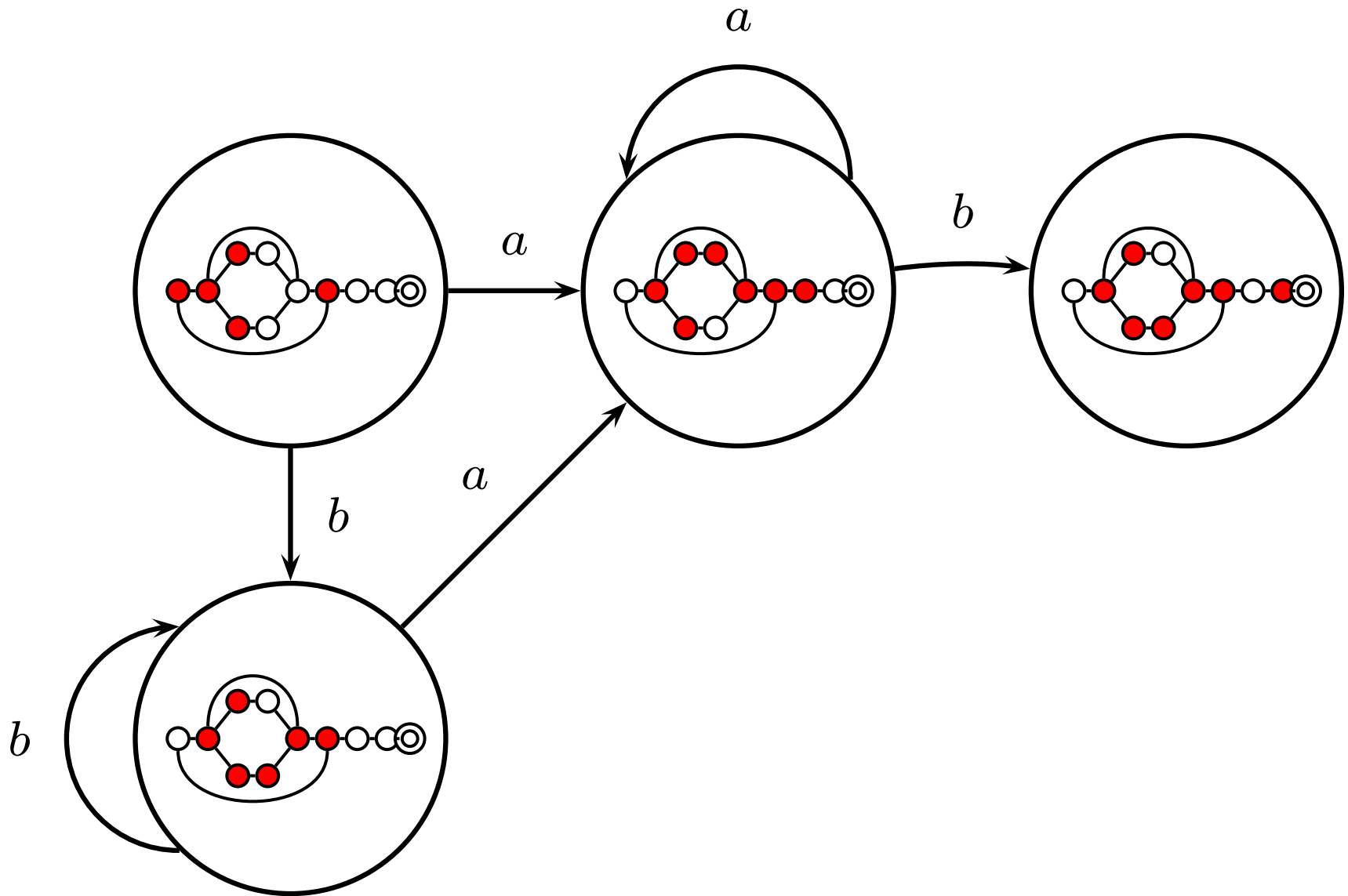
Simulate the NFA for all possible inputs and track the states that appear.

Each unique state during simulation becomes a state in the DFA.

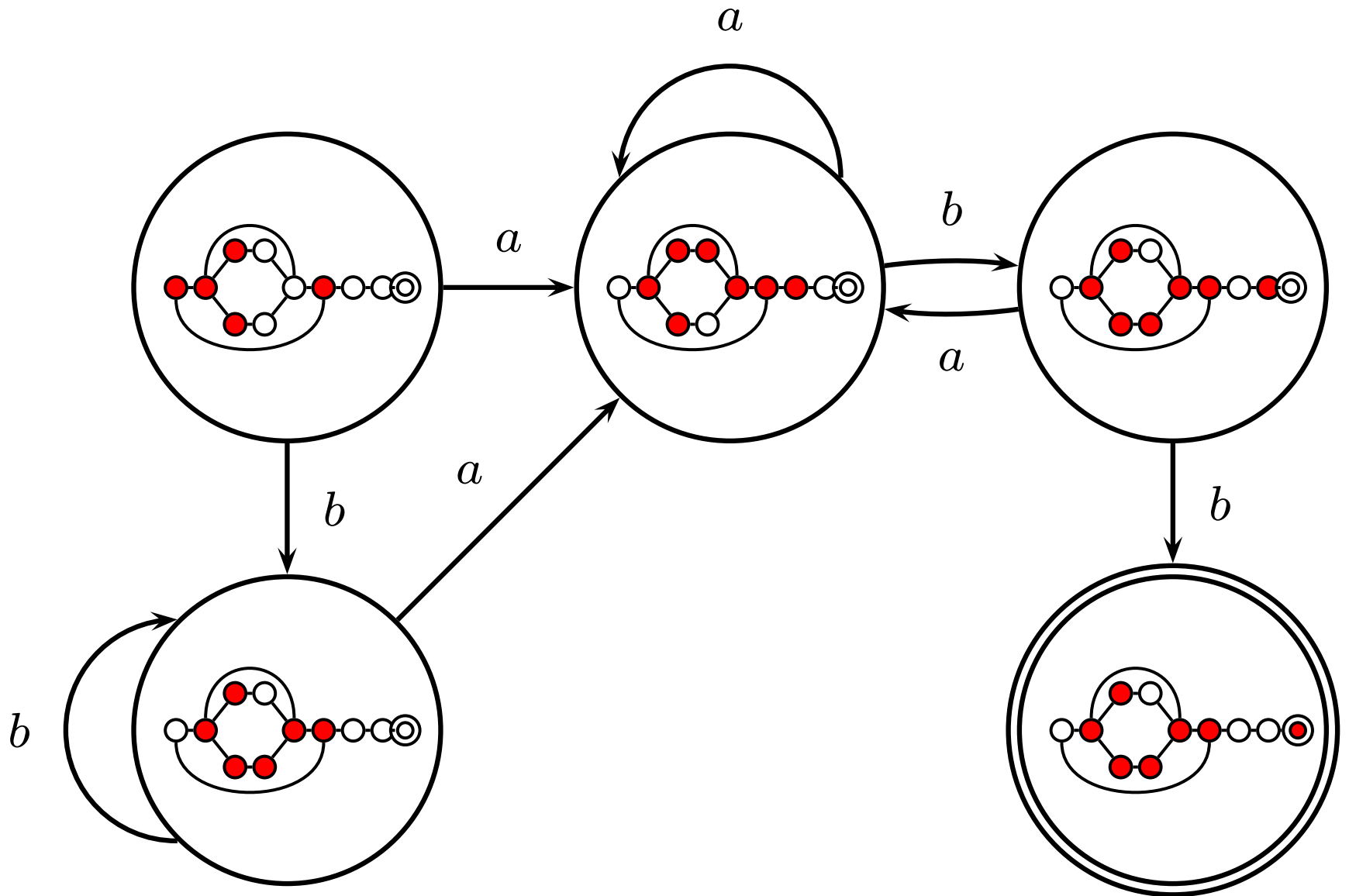
# Subset construction for $(a|b)^*abb$ (1)



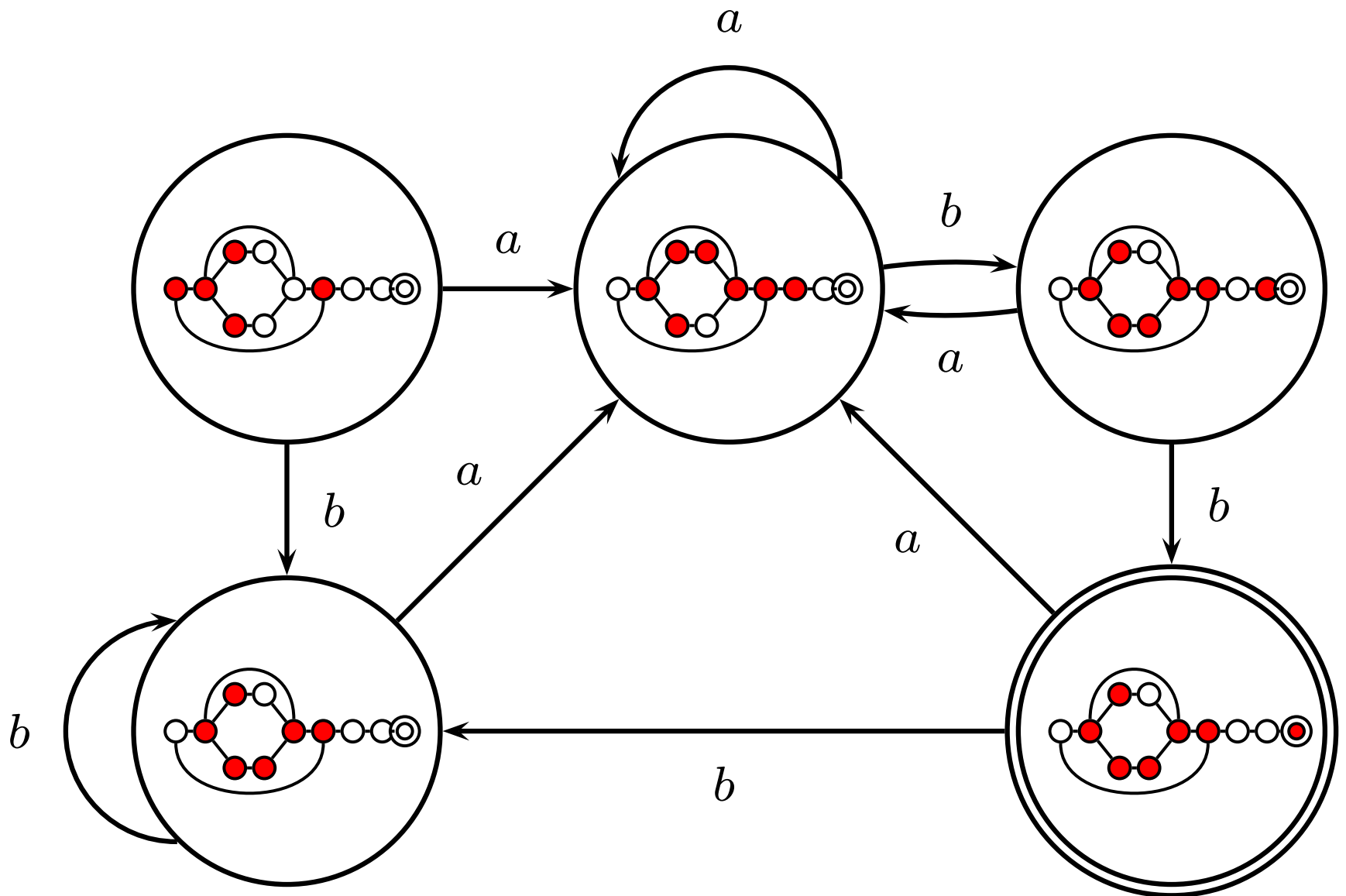
# Subset construction for $(a|b)^*abb$ (2)



# Subset construction for $(a|b)^*abb$ (3)



# Subset construction for $(a|b)^*abb$ (4)



# Grammars and Parsing

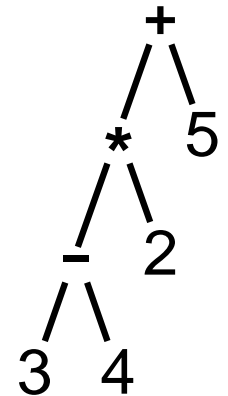
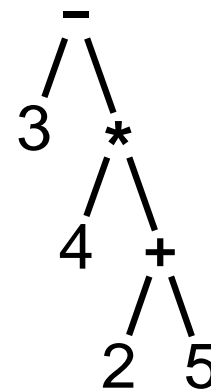
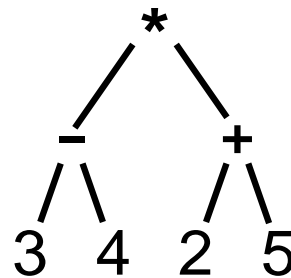
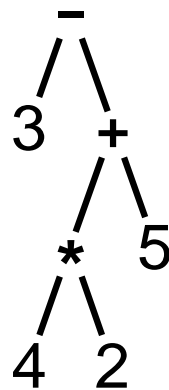
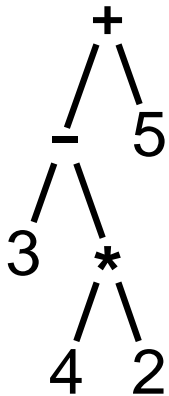
# Ambiguous Grammars

A grammar can easily be ambiguous. Consider parsing

$$3 - 4 * 2 + 5$$

with the grammar

$$e \rightarrow e + e \mid e - e \mid e * e \mid e / e$$



# Fixing Ambiguous Grammars

Original ANTLR grammar specification

```
expr
  : expr '+' expr
  | expr '-' expr
  | expr '*' expr
  | expr '/' expr
  | NUMBER
  ;
```

Ambiguous: no precedence or associativity.



# Assigning Precedence Levels

Split into multiple rules, one per level

```
expr : expr '+' expr  
      | expr '-' expr  
      | term ;
```

```
term : term '*' term  
      | term '/' term  
      | atom ;
```

```
atom : NUMBER ;
```

Still ambiguous: associativity not defined

# Assigning Associativity

Make one side or the other the next level of precedence

```
expr : expr '+' term
      | expr '-' term
      | term ;
```

```
term : term '*' atom
      | term '/' atom
      | atom ;
```

```
atom : NUMBER ;
```

# A Top-Down Parser

```
stmt : 'if' expr 'then' expr  
      | 'while' expr 'do' expr  
      | expr ':=' expr ;
```

```
expr : NUMBER | '(' expr ')' ;
```

```
AST stmt() {  
  switch (next-token) {  
  case "if" : match("if"); expr(); match("then"); expr();  
  case "while" : match("while"); expr(); match("do"); expr();  
  case NUMBER or "(" : expr(); match(":="); expr();  
  }  
}
```

# Writing LL(k) Grammars

Cannot have left-recursion

```
expr : expr '+' term | term ;
```

becomes

```
AST expr() {  
    switch (next-token) {  
        case NUMBER : expr(); /* Infinite Recursion */
```

# Writing LL(1) Grammars

Cannot have common prefixes

```
expr : ID '(' expr ')'  
      | ID '=' expr
```

becomes

```
AST expr() {  
    switch (next-token) {  
        case ID : match(ID); match('('); expr(); match(')');  
        case ID : match(ID); match('='); expr();
```

# Eliminating Common Prefixes

Consolidate common prefixes:

```
expr
  : expr '+' term
  | expr '-' term
  | term
  ;
```

becomes

```
expr
  : expr ( '+' term | '-' term )
  | term
  ;
```

# Eliminating Left Recursion

Understand the recursion and add tail rules

`expr`

```
: expr ( '+' term | '-' term )  
| term  
;
```

becomes

```
expr : term exprt ;  
exprt : '+' term exprt  
| '-' term exprt  
| /* nothing */  
;
```

# Bottom-up Parsing



# Rightmost Derivation

$$1 : e \rightarrow t + e$$

$$2 : e \rightarrow t$$

$$3 : t \rightarrow \mathbf{ld} * t$$

$$4 : t \rightarrow \mathbf{ld}$$

A rightmost derivation for  $\mathbf{ld} * \mathbf{ld} + \mathbf{ld}$ :

$$\begin{array}{l} e \\ t + e \\ t + t \\ t + \mathbf{ld} \\ \mathbf{ld} * t + \mathbf{ld} \\ \mathbf{ld} * \mathbf{ld} + \mathbf{ld} \end{array}$$

Basic idea of bottom-up parsing:  
construct this rightmost derivation  
**backward**.

# Handles

1 :  $e \rightarrow t + e$

$\mathbf{ld} * \mathbf{ld} + \mathbf{ld}$

2 :  $e \rightarrow t$

$\mathbf{ld} * t + \mathbf{ld}$

3 :  $t \rightarrow \mathbf{ld} * t$

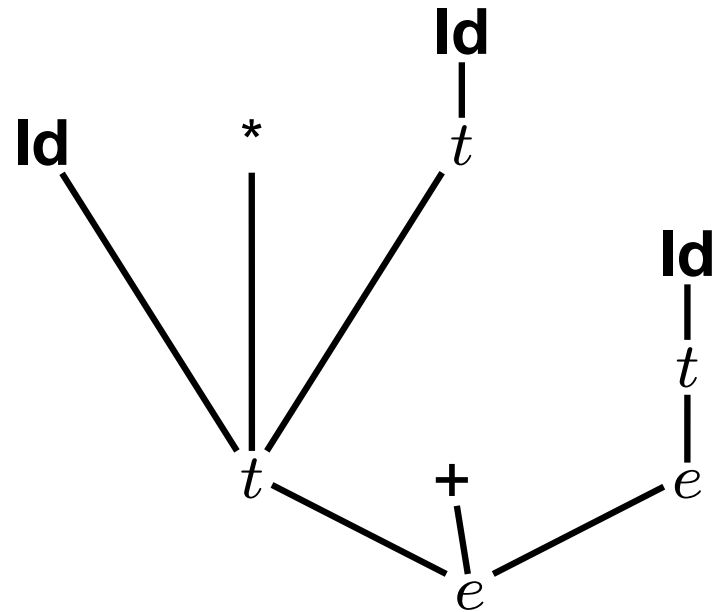
$t + \mathbf{ld}$

4 :  $t \rightarrow \mathbf{ld}$

$t + t$

$t + e$

$e$



This is a reverse rightmost derivation for  $\mathbf{ld} * \mathbf{ld} + \mathbf{ld}$ .

Each highlighted section is a **handle**.

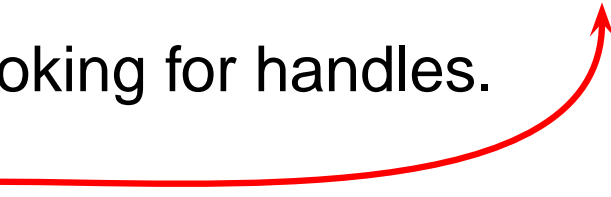
Taken in order, the handles build the tree from the leaves to the root.

# Shift-reduce Parsing

	stack	input	action
1 : $e \rightarrow t + e$			
2 : $e \rightarrow t$		<b>ld</b> * <b>ld</b> + <b>ld</b>	shift
3 : $t \rightarrow \mathbf{ld} * t$	<b>ld</b>	* <b>ld</b> + <b>ld</b>	shift
4 : $t \rightarrow \mathbf{ld}$	<b>ld</b> *	<b>ld</b> + <b>ld</b>	shift
	<b>ld</b> * <b>ld</b>	+ <b>ld</b>	reduce (4)
	<b>ld</b> * <i>t</i>	+ <b>ld</b>	reduce (3)
	<i>t</i>	+ <b>ld</b>	shift
	<i>t</i> +	<b>ld</b>	shift
	<i>t</i> + <b>ld</b>		reduce (4)
	<i>t</i> + <i>t</i>		reduce (2)
	<i>t</i> + <i>e</i>		reduce (1)
	<i>e</i>		accept

Scan input left-to-right, looking for handles.

An oracle tells what to do



# LR Parsing

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

3 :  $t \rightarrow \mathbf{ld} * t$

4 :  $t \rightarrow \mathbf{ld}$

	action				goto	
	<b>ld</b>	+	*	\$	<i>e</i>	<i>t</i>
0	s1				7	2
1	r4	r4	s3	r4		
2	r2	s4	r2	r2		
3	s1					5
4	s1				6	2
5	r3	r3	r3	r3		
6	r1	r1	r1	r1		
7				acc		

stack

input

action

0

**ld \* ld + ld \$**

shift, goto 1

1. Look at state on top of stack
2. and the next input token
3. to find the next action
4. In this case, shift the token onto the stack and go to state 1.

# LR Parsing

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

3 :  $t \rightarrow \mathbf{ld} * t$

4 :  $t \rightarrow \mathbf{ld}$

	action				goto	
	ld	+	*	\$	e	t
0	s1				7	2
1	r4	r4	s3	r4		
2	r2	s4	r2	r2		
3	s1					5
4	s1				6	2
5	r3	r3	r3	r3		
6	r1	r1	r1	r1		
7				acc		

stack

input

action

0

ld \* ld + ld \$

shift, goto 1

0 ld 1

\* ld + ld \$

shift, goto 3

0 ld 1 \* 3

ld + ld \$

shift, goto 1

0 ld 1 \* 3 ld 1

+ ld \$

reduce w/ 4

Action is reduce with rule 4

( $t \rightarrow \mathbf{ld}$ ). The right side is

removed from the stack to reveal

state 3. The goto table in state 3

tells us to go to state 5 when we

reduce a  $t$ :

stack

input

action

0 ld 1 \* 3 t 5

+ ld \$

# LR Parsing

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

3 :  $t \rightarrow \mathbf{ld} * t$

4 :  $t \rightarrow \mathbf{ld}$

	action				goto	
	<b>ld</b>	+	*	\$	<i>e</i>	<i>t</i>
0	s1				7	2
1	r4	r4	s3	r4		
2	r2	s4	r2	r2		
3	s1					5
4	s1				6	2
5	r3	r3	r3	r3		
6	r1	r1	r1	r1		
7				acc		

stack

0				
0	ld			
0	ld	*		
0	ld	*	ld	
0	ld	*	t	
0	t			
0	t	+		
0	t	+	ld	
0	t	+	t	
0	t	+	e	
0	e			

input

**ld \* ld + ld \$**

**\* ld + ld \$**

**ld + ld \$**

**+ ld \$**

**+ ld \$**

**+ ld \$**

**ld \$**

**\$**

**\$**

**\$**

**\$**

action

shift, goto 1

shift, goto 3

shift, goto 1

reduce w/ 4

reduce w/ 3

shift, goto 4

shift, goto 1

reduce w/ 4

reduce w/ 2

reduce w/ 1

accept

# Constructing the SLR Parse Table

The states are places we could be in a reverse-rightmost derivation. Let's represent such a place with a dot.

$$1 : e \rightarrow t + e$$

$$2 : e \rightarrow t$$

$$3 : t \rightarrow \mathbf{ld} * t$$

$$4 : t \rightarrow \mathbf{ld}$$

Say we were at the beginning ( $\cdot e$ ). This corresponds to

$$e' \rightarrow \cdot e$$

$$e \rightarrow \cdot t + e$$

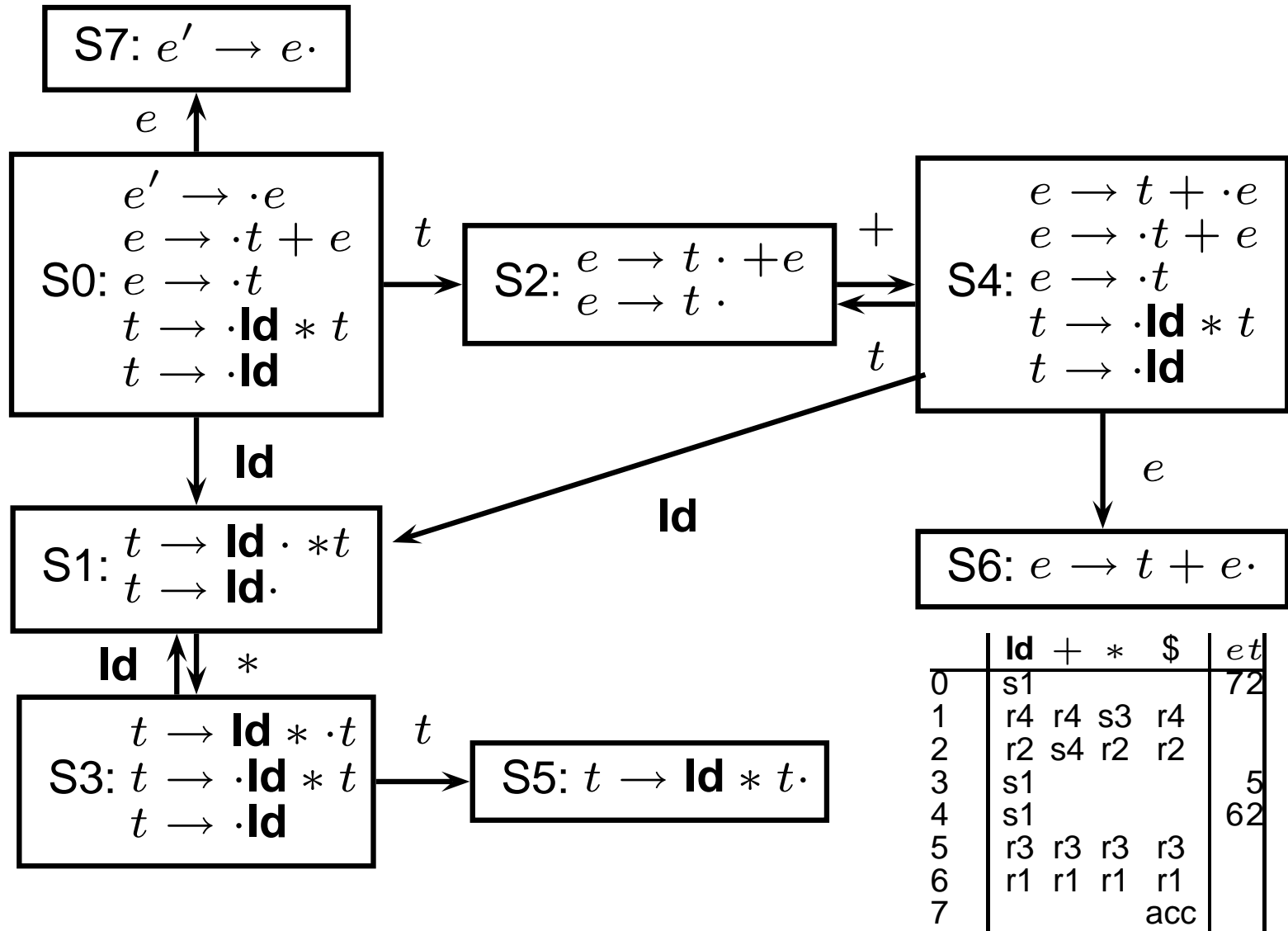
$$e \rightarrow \cdot t$$

$$t \rightarrow \cdot \mathbf{ld} * t$$

$$t \rightarrow \cdot \mathbf{ld}$$

The first is a placeholder. The second are the two possibilities when we're just before  $e$ . The last two are the two possibilities when we're just before  $t$ .

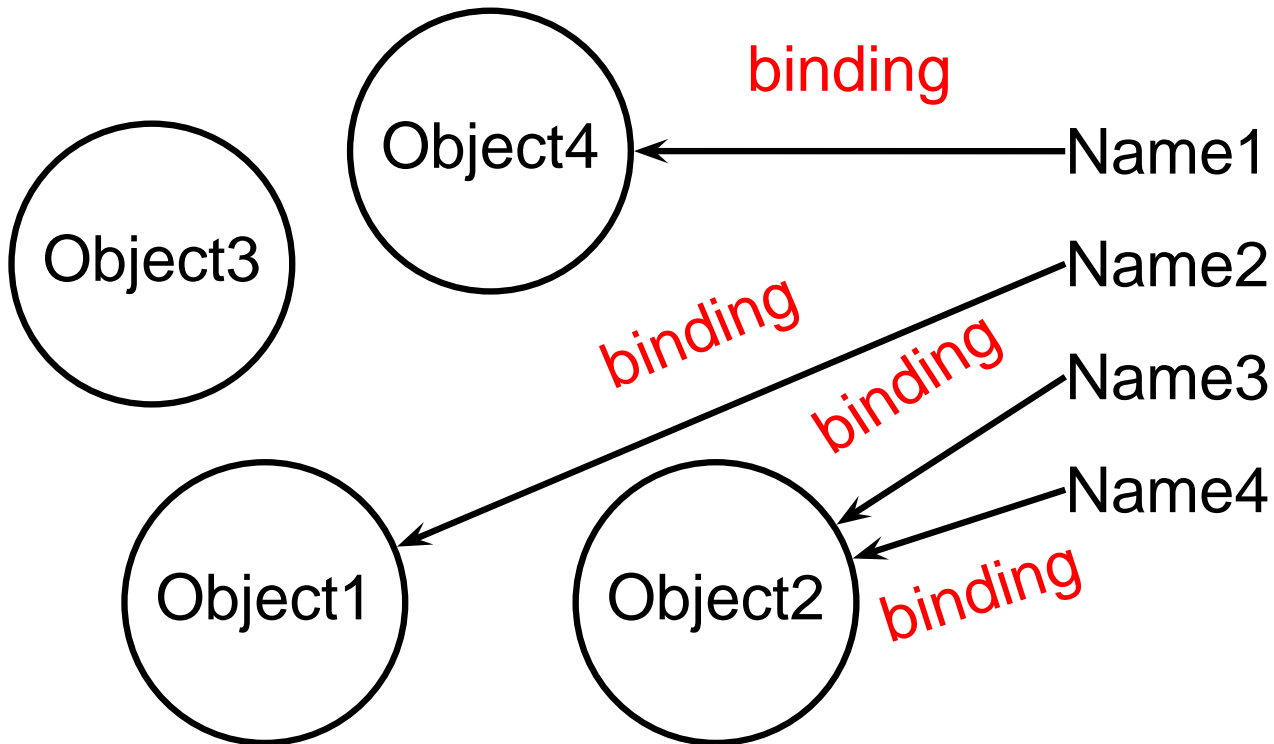
# Constructing the SLR Parsing Table



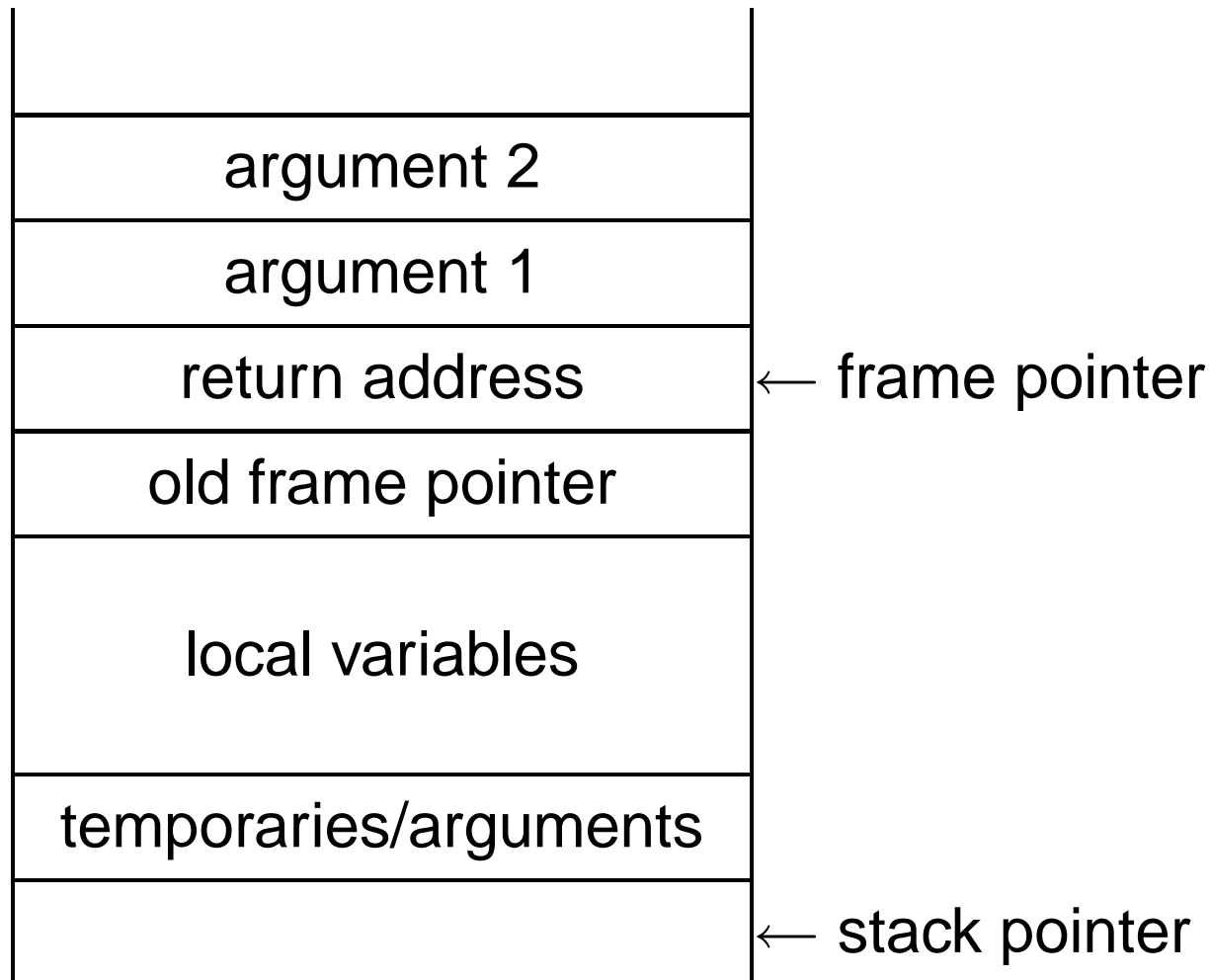


# **Names, Objects, and Bindings**

# Names, Objects, and Bindings

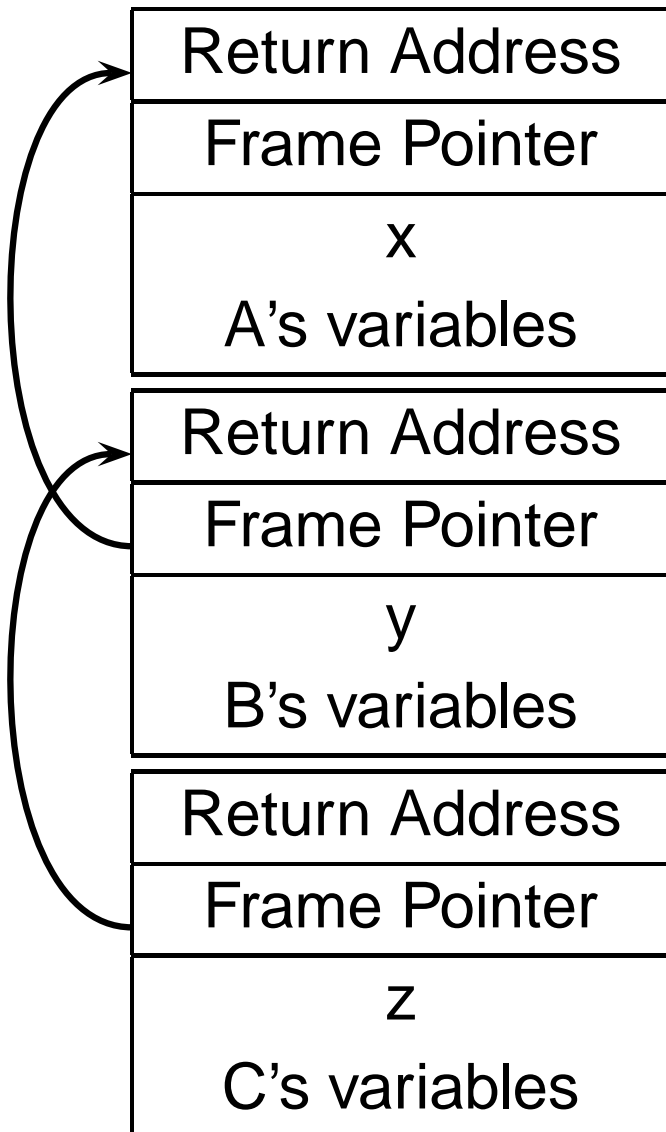


# Activation Records



↓ growth of stack

# Activation Records



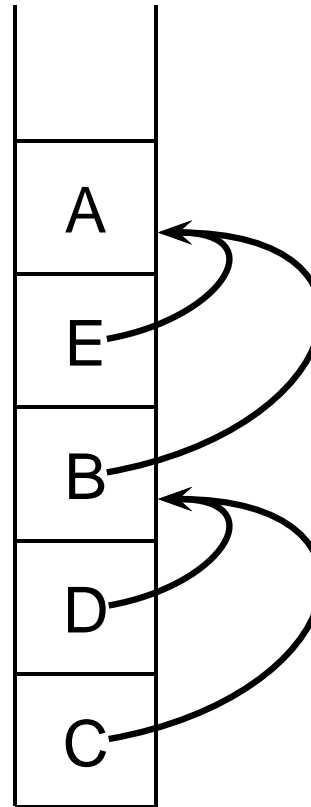
```
int A() {  
    int x;  
    B();  
}
```

```
int B() {  
    int y;  
    C();  
}
```

```
int C() {  
    int z;  
}
```

# Nested Subroutines in Pascal

```
procedure A;  
  procedure B;  
    procedure C;  
    begin .. end  
  
    procedure D;  
    begin C end  
  begin D end  
  
  procedure E;  
  begin B end  
begin E end
```



# Symbol Tables in Tiger

