

# Compiler Techniques

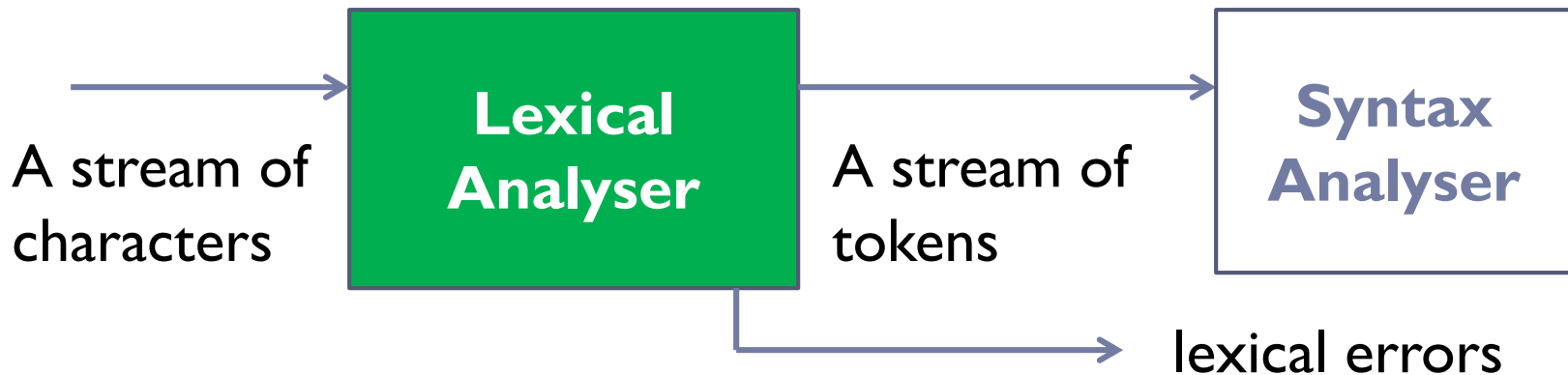
## 2. Lexical Analysis

Huang Shell Ying

# Lexical Analysis

[back](#)

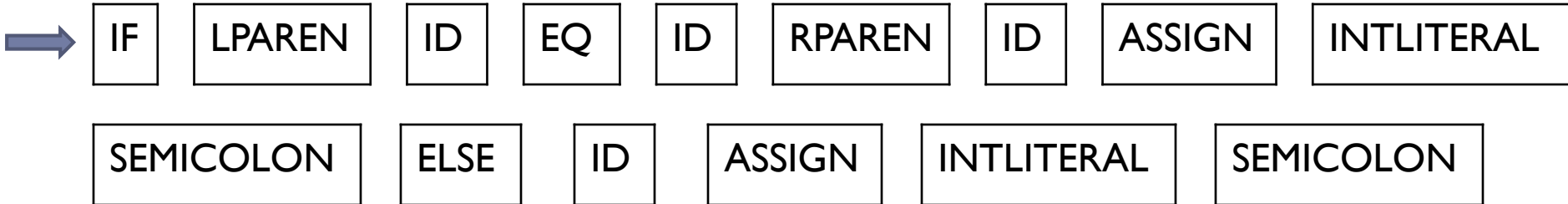
- ▶ The **lexical analyser** (a.k.a. “**lexer**” or “**scanner**”) transforms the input program from a sequence of characters into a sequence of tokens.



For example,      `if (i == j)    z = 0;`  
                      `else z = 1;`

# Lexical Analysis

\t	i	f		(	i	=	=	j	)	\t	z		=		0	;	\n
\t	e	l	s	e		z		=		1	;	\n					



- ▶ WHITESPACE and COMMENT are discarded by the lexer (not output by the lexer).
- ▶ Each programming language has a set of tokens.
- ▶ Different programming languages have different tokens.

# Tokens

[back](#)

- ▶ A token includes its **type** (such as IDENTIFIER) and its string value: **lexeme**.

Token type	lexeme	Token type	lexeme
WHILE	“While”	MINUS	“-”
NEQ	“!=”	LPAREN	“(”
WHITESPACE	“\t”, “ ”, “\n”	INTLITERAL	“10”, “123”, “0”
IDENTIFIER	“a2”, “i”, “f”		

- ▶ Some token types (like LPAREN) have a single lexeme, i.e., only one string belongs to this token type.
- ▶ Some token types (like IDENTIFIER) have many lexemes, i.e., a set of strings belong to this token type.

# Building a Lexical Analyser (Lexer)

---

**Step 1:** Define a finite set of tokens and describe which strings belong to each token

- ▶ Tokens describe all items of interest
- ▶ Choice of tokens depends on language, design of parser

**Step 2:** Implement the lexer—An implementation must do two things:

1. Recognise substrings corresponding to tokens
2. Return the type and the lexeme of the token

# A Lexer in Java

---

```
public class Lexer {
```

```
    public class Token {    // class for representing tokens
```

```
        public enum Type {
```

```
            /* keywords */  BOOLEAN, BREAK, ELSE, ...,
```

```
            /* punctuation symbols */  COMMA, LBRACKET, LCURLY, ...,
```

```
            /* operators */  DIV, EQEQ, EQL, ...,
```

```
            /* identifier */  ID,
```

```
            /* literals */  INT_LITERAL, STRING_LITERAL, ...,
```

```
            ...
```

```
        }
```

```
        Type type;
```

```
        String lexeme;
```

```
        public Token(Type type, String lexeme, ...) { ... }
```

```
    }
```

back

# A Lexer in Java

---

...

**private final** String source; // input string to be lexed

**private int** currentPos; // current position inside input

// called by the parser, return token starting at position curpos

**public** Token nextToken() { ... }

}

# Automatically Generating a Lexer

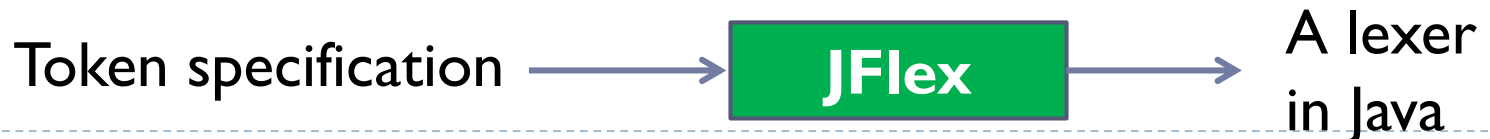
[back](#)

- ▶ The way to recognise tokens after reading some characters is the same for all languages.
- ▶ So a lexer generator can be used to automatically generate a lexer from a high-level specification:



- ▶ The token specification describes tokens by *regular expressions*.

In this course, we use the lexer generator JFlex





# Regular Expressions

---

- ▶ Regular expressions are a convenient way to specify various simple (possibly infinite) **sets of strings**.
- ▶ Using a regular expression, it takes only one line or a couple of lines (e.g., in Perl, PHP, Python, Ruby, Java, or .NET) of code to, say, check if the user's input is a valid email address.
- ▶ What is a string?
  - ▶ An **alphabet** is a finite set of characters.
  - ▶ A **string over an alphabet  $\Sigma$**  is a finite sequence of characters drawn from  $\Sigma$ .
- ▶ A regular expression **defines the structure** of a set of strings. Each set of strings forms a token class.

# Regular Expressions

---

- ▶ The definition of regular expressions starts with **a finite character set**, or **alphabet** (denoted by  $\Sigma$ ).  
E.g., The  $\Sigma$  of C programming language is the set of ASCII characters.

## Definition of regular expressions:

- ▶  $\lambda$  is a regular expression denoting an empty string, i.e.,  $\{\text{""}\}$ .
- ▶ The symbol  $s$ , where  $s \in \Sigma$ , is a regular expression denoting  $\{s\}$ .

For example, ' $<$ ' specifies  $\{<\}$ ;

# Regular Expressions

- ▶ If **A** and **B** are regular expressions, then **A|B** is a regular expression denoting the set of strings which are either in A or in B. **|** is the **alternation operator**<sup>1</sup>.

For example, **‘+’ | ‘-’** specifies {“+”, “-”};

- ▶ If **A** and **B** are regular expressions, then **A·B** is a regular expression denoting the set of strings which are the **concatenation** of **one** string from A and **one** string from B.

For example, **‘i’ · ‘f’** specifies {“if”}

A = {“a”, “b”}, B = {“0”, “1”}, then A·B = {“a0”, “a1”, “b0”, “b1”}

<sup>1</sup> In formal language theory and pattern matching, ‘alternation’ means the set union of two sets of strings.

# Regular Expressions

---

- ▶ If **A** is a regular expression then **A\*** is a regular expression representing all strings formed by the concatenation of **zero or more** selections from A.

The operator \* is called the **Kleene closure operator**.

For example, **a\*** specifies { "", "a", "aa", "aaa", ... }

Another example: if  $A = \{ "a", "b" \}$  then

$$A^* = \{ "", "a", "b", "aa", "ab", "bb", "ba", "aaa", \dots \}$$

- ▶ If **A** is a regular expression then **(A)** is a regular expression representing the same set of strings as A.
- ▶ **Operator precedence** in decreasing order:

$$(R), \quad R^*, \quad R_1R_2, \quad R_1|R_2$$

# Additional forms or operators

---

- ▶ To save ink, we usually omit the dot for concatenation.
- ▶ For single characters, we often omit the quotation marks except for special characters like '|', '\*', '\n'. E.g.,  $a^*$ ,  $a^*$
- ▶  $A^+$  is a regular expression representing all strings formed by the concatenation of **one or more** selections from  $A$ .  
 $A^* = A^+ | \lambda$  and  $A^+ = AA^*$ .

For example,  $(0|1|2|3|4|6|7)^+$  specifies octal integers.

- ▶ If  $k$  is a constant,  $A^k$  is a regular expression representing all strings formed by the **concatenation of  $k$**  selections from  $A$ .

For example,  $(0|1)^8$  specifies strings of 8 binary digits.

# Additional forms or operators

---

- ▶ A character class delimited by **[** and **]** represents **a single character** from the class. Ranges of characters may be separated by a **-**.

For example, decimal, octal digits: **[0-9]**, **[01234567]**

- ▶ **Not(s)** represents ( $\Sigma - s$ ), i.e., any character in  $\Sigma$  which is not s. **[^0-9]** means not a digit character.

For example, **' ' (Not('\n'))\* '\n'** or

**' ' [^\n']\* '\n'** specifies single line comments

- ▶ **?** Is the **optional** choice operator.

For example, **(+ | -)? [0-9]<sup>+</sup>** specifies signed integers

# Examples

Regular expression	Set of strings defined
$\lambda$	$\{""\}$
0	$\{"0"\}$
0   1	$\{"0", "1"\}$
0 1	$\{"01"\}$
(0   1) 0	$\{"00", "10"\}$
0   10	$\{"0", "10"\}$
0*	$\{ "", "0", "00", "000", \dots \}$
(ab) <sup>+</sup>	$\{ "ab", "abab", "ababab", \dots \}$
ab <sup>+</sup>	$\{ "ab", "abb", "abbb", \dots \}$
[abc] or [a-c] or a b c	$\{ "a", "b", "c" \}$
([a-z]   [0-9])* or [a-z0-9]*	$\{ "", "a", \dots, "z", "0", \dots, "9", "aa", \dots, "99", \dots \}$

# Test Yourself 2.1

---

1. If regular expression A is defined by `0|1|2|3|4|5|6|7|8|9`, what is, respectively, `AA`, `A*`, `A+`?
2. Write a regular expression that specifies a register in the assembly language of a machine. This machine has 16 registers specified by `r1`, `r2`, ..., `r10`, ..., `r16`.
3. To indicate a hexadecimal literal in python, use `0x` followed by a sequence of hexadecimal digits (0 to 9 and A to F, in either upper- or lowercase). Write a regular expression for hexadecimal literals.



# Automata

---

- ▶ We need an algorithm that can recognise (accept) all strings of specified regular expressions and reject those which do not belong to them.
- ▶ An **automaton** (plural: **automata** or **automatons**) is a **machine** that reads a string and decides whether it is a token specified by a regular expression.
- ▶ A finite automaton is essentially a graph, with nodes and transition edges.
- ▶ Nodes represent states and transition edges represent transitions between states.
- ▶ A **finite** automaton consists of the following:

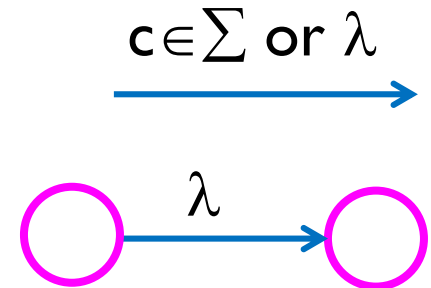
# Finite Automata

1. A **finite** set of **states**

a state: 

2. A **finite alphabet**, denoted by  $\Sigma$

3. A set of **transitions** from one state to another, labeled with characters in  $\Sigma$  or  $\lambda$



4. A special state with no predecessor state, called the **start** state

a start state: 

5. A subset of the states called the **accepting**, or **final** states

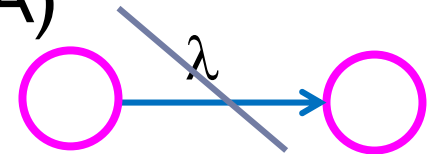
an accepting state: 

# Finite Automata

- ▶ Finite automata come in two flavours:
  - ▶ Deterministic finite automata (DFA)
  - ▶ Nondeterministic finite automata (NFA)

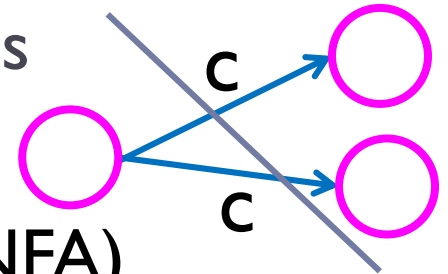
- ▶ **Deterministic finite automata (DFA)**

- ▶ Do not allow  $\lambda$  to label a transition
  - ▶ Do not allow the same character to label transitions from one state to several different states



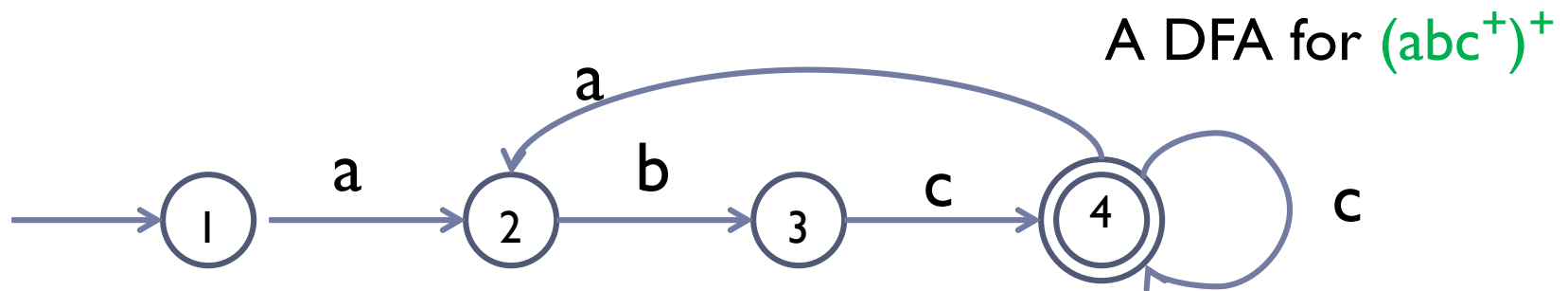
- ▶ **Nondeterministic finite automata (NFA)**

- ▶ Have no restrictions on the labels of transitions



# Finite Automata

- ▶ We can represent either a DFA or an NFA by a **transition graph**:



- ▶ or by a **transition table**:

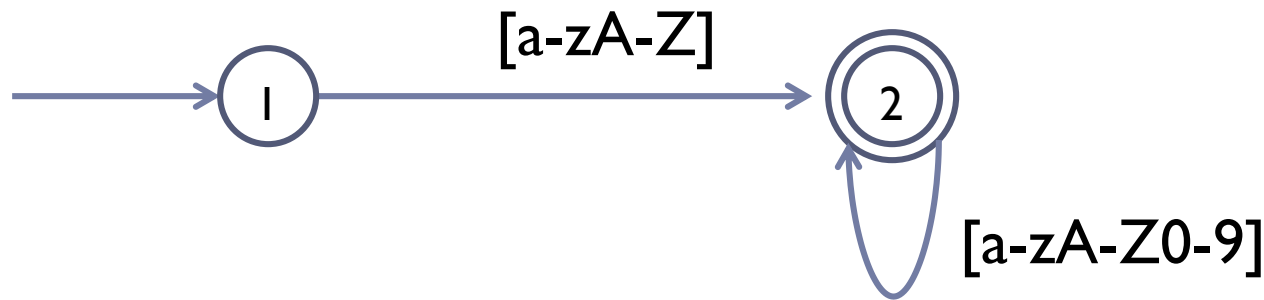
State 0: error state,  
i.e., a state-input not  
expected

	a	b	c	others
1, start	2	0	0	0
2	0	3	0	0
3	0	0	4	0
4, final	2	0	4	0

# Deterministic Finite Automata (DFA)

A DFA for **identifiers**:  $[a-zA-Z][a-zA-Z0-9]^*$

By a **transition diagram**



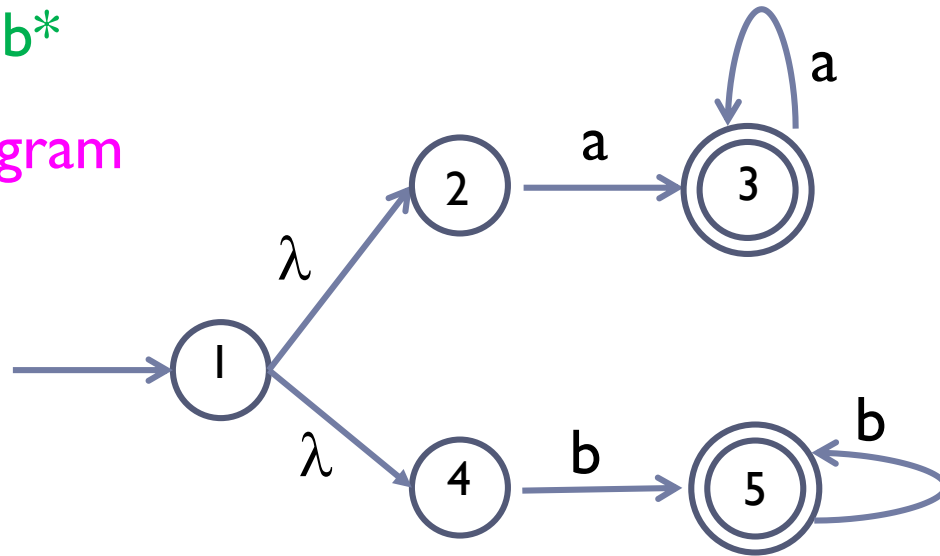
By a **transition table**

	a ...	z	A...	Z	0...	9	others
1, start	2...	2	2...	2	0...	0	0
2, final	2...	2	2...	2	2...	2	0

# Nondeterministic Finite Automata (NFA)

An NFA for  $aa^* \mid bb^*$

By a transition diagram



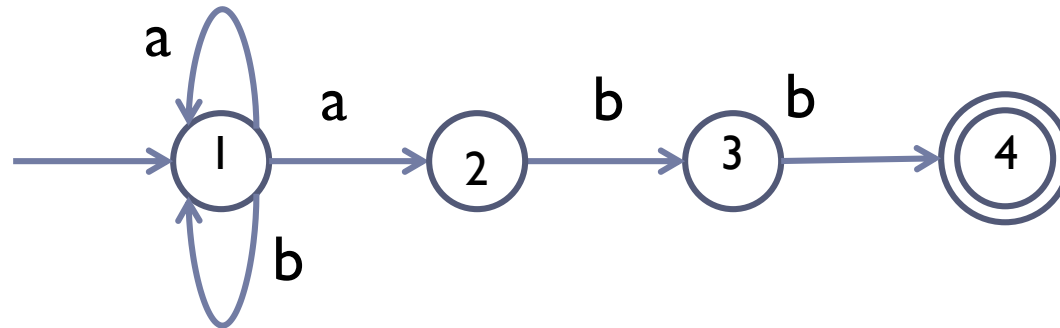
By a transition table

	a	b	$\lambda$	others
1, start	0	0	{2,4}	0
2	3	0	0	0
3, final	3	0	0	0
4	0	5	0	0
5, final	0	5	0	0

# Nondeterministic Finite Automata (NFA)

An NFA for  $(a|b)^*abb$

By a transition diagram



By a transition table

	a	b	others
1, start	{1,2}	1	0
2	0	3	0
3	0	4	0
4, final	0	0	0

[back](#)

# Coding the Deterministic Finite Automata

---

- ▶ A DFA can be coded in a **table-driven** form:

```
currentChar = read();
```

```
state = startState;      // the start state defined in slide 18
```

```
while true do
```

```
    nextState = T[state][currentChar];
```

```
    if (nextState == error) then break;    // state 0
```

```
    state = nextState;
```

```
    currentChar = read()
```

```
if (state in acceptingStates)
```

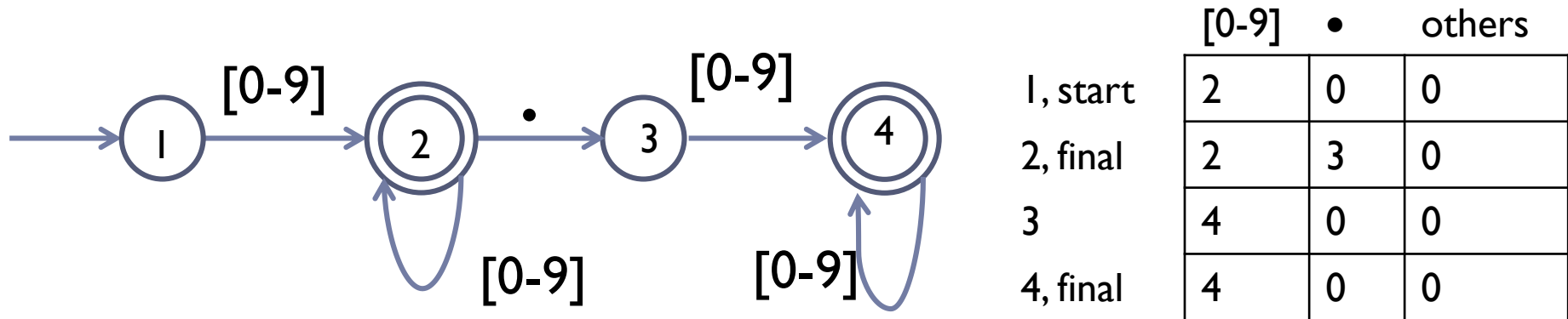
```
    then /* return or process the valid token */
```

```
    else /* signal a lexical error */
```



# Example

Regular expression:  $[0-9]^+ ( '.' [0-9]^+ | \lambda )$



Input: "0.5". What happens for the input "6", for "6."?

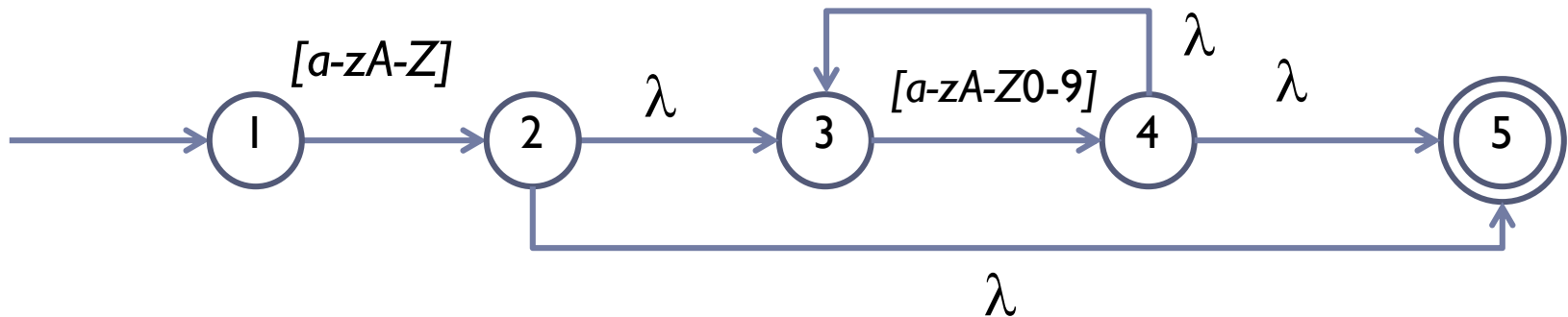
Problems with NFA—multiple destination states

Slide 23

back

# Conversion of an NFA to a DFA

- ▶ The transformation from an NFA **N** to a DFA **D** can be done by a **subset construction** algorithm.
- ▶ The  **$\lambda$ -successors** of a state **s** of **N** is the set of all states in **N** where **transition can be made from s without reading** any character.

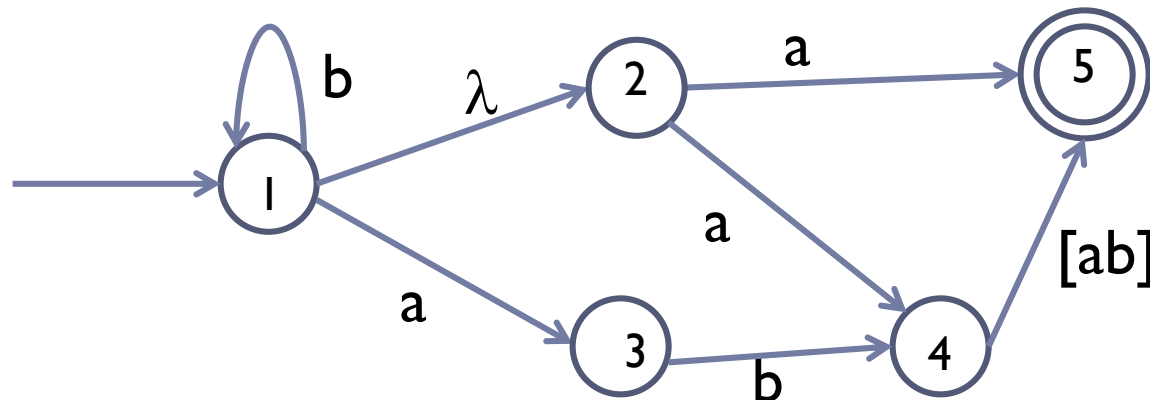


The  $\lambda$ -successors of state 2: {3, 5}

- ▶ The algorithm associates each state of **D** with a **set of states** of **N**.

# Conversion of an NFA to a DFA

- ▶ The start state of **D** is the set of all states consisting of the start state of **N** and its  $\lambda$ -successors.



An NFA **N**  
Start state: 1

A DFA **D**  
Start state: {1,2}

- ▶ An accepting (final) state of **D** is any set that contains an accepting (final) state of **N**.

# The Subset Construction Algorithm

**S** = the set of all states consisting of the start state of **N** and its  $\lambda$ -successors;

Put **S** into the work list and into **D**;

While work list is not empty

    Remove a state **S** =  $\{n1, n2, \dots\}$  from the work list;

    For (each character **c**  $\in \Sigma$ , if there is an outgoing edge from any state in **S** under **c** in **N**)

        put successor state(s) of  $n1, n2, \dots$  under **c** in **N** into **T**;

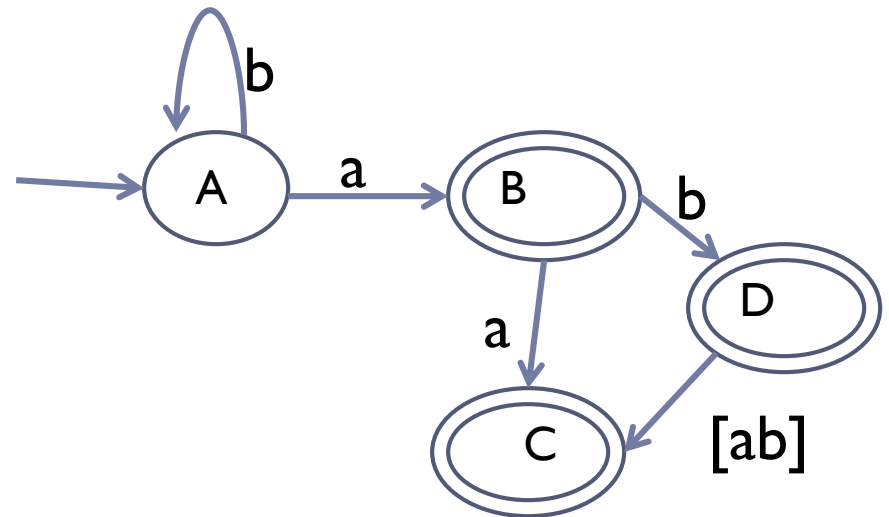
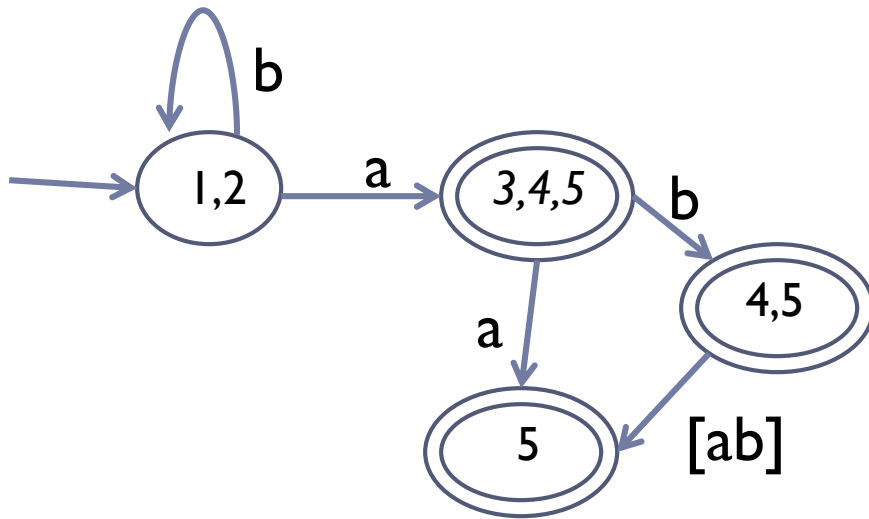
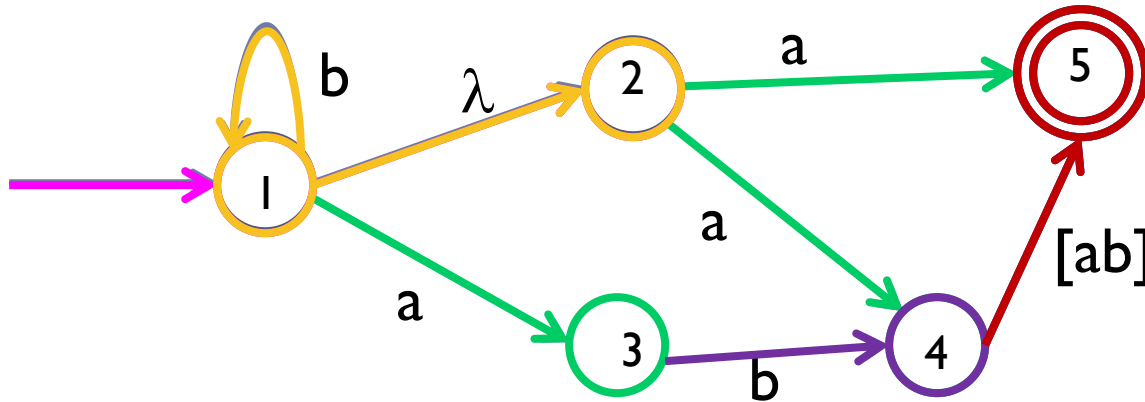
    // result: **T** =  $\{m1, m2, \dots\}$

    Put the  $\lambda$ -successors of  $m1, m2, \dots$  in **N** into **T**;

    Put **T** into the work list and into **D**;

    Add a transition from **S** to **T**, labelled with **c**, into **D**.

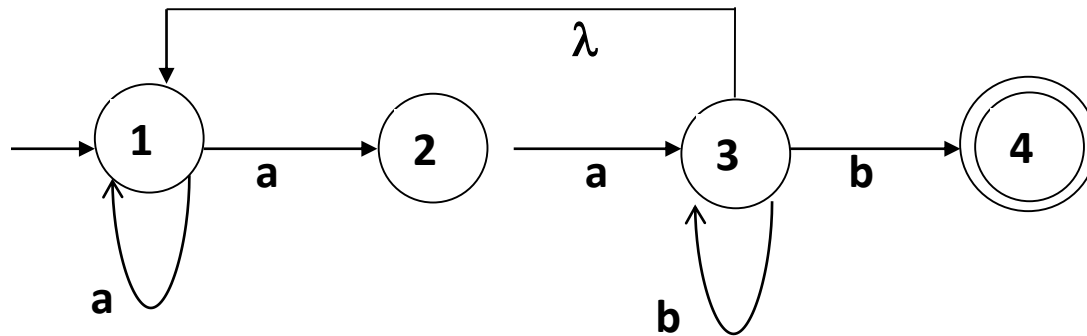
# Conversion of an NFA to a DFA



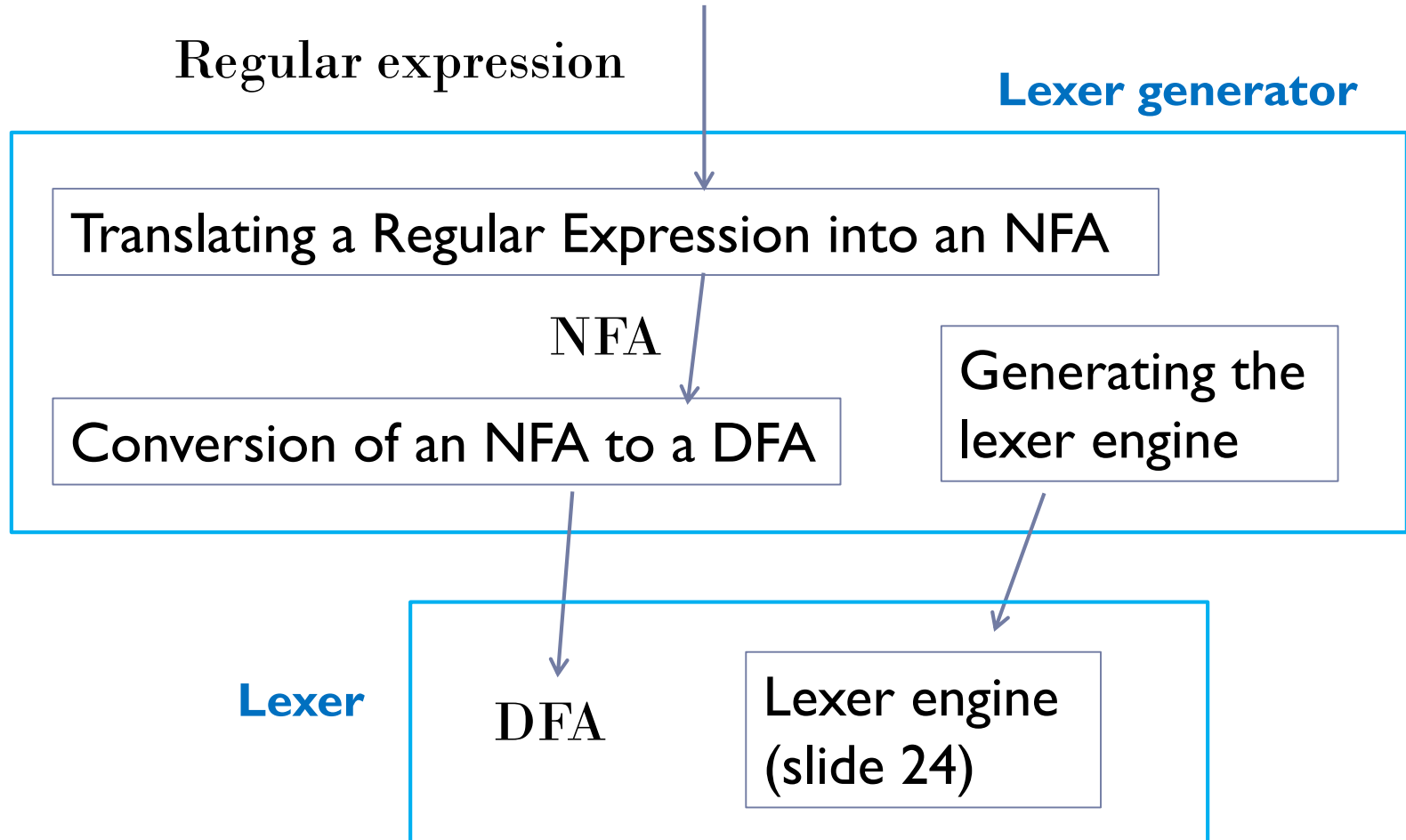
## Test Yourself 2.2

---

Convert this NFA to a DFA using the subset construction algorithm



# Lexer Generator and Lexer



# Translating Regular Expressions into NFAs

---

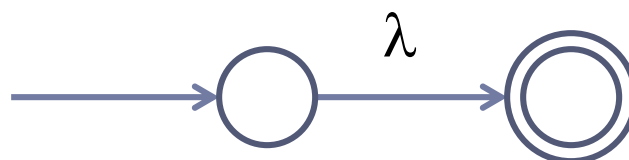
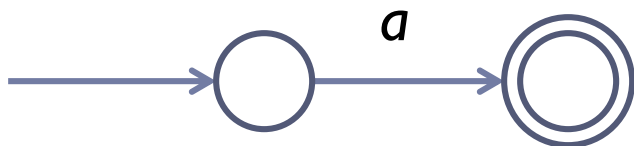
- ▶ The McNaughton-Yamada-Thompson method to construct NFAs from regular expressions produces NFAs that accept the same languages.
- ▶ An NFA constructed has the following properties:
  - 1) The NFA has at most twice as many states as there are operators and operands in the regular expression;
  - 2) The NFA has one start state and one accepting state. The accepting state has no outgoing transitions and the start state has no incoming transitions.
  - 3) Each state of the NFA other than the accepting state has either one outgoing edge on a symbol in  $\Sigma \cup \{\lambda\}$  or two outgoing edges, both on  $\lambda$ .

back

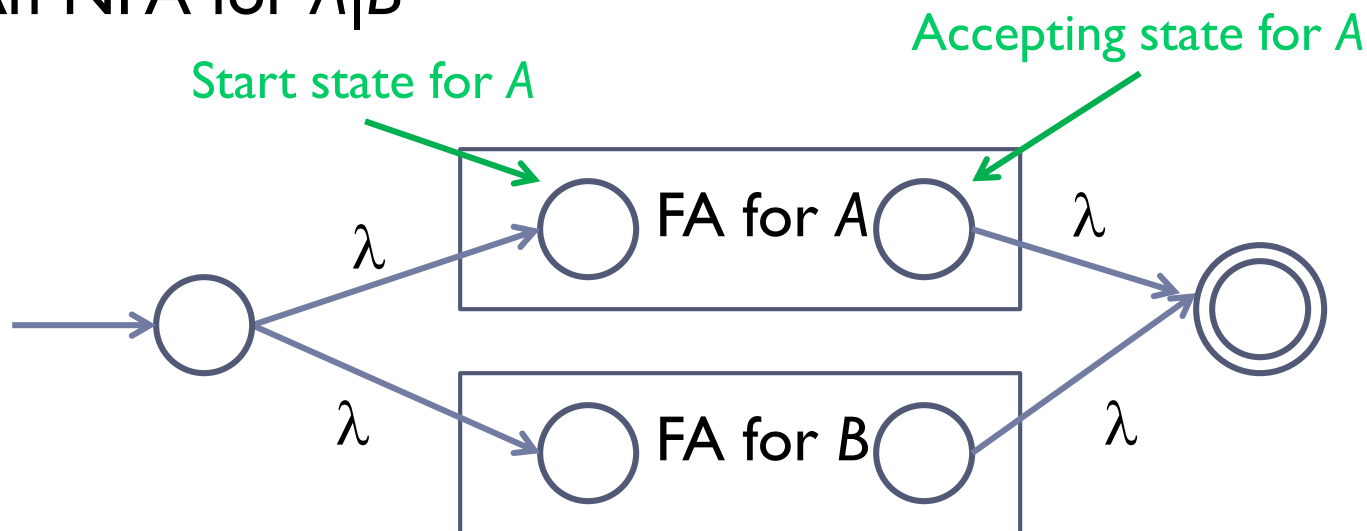


# Translating Regular Expressions into NFAs

## ► NFAs for $a$ and $\lambda$

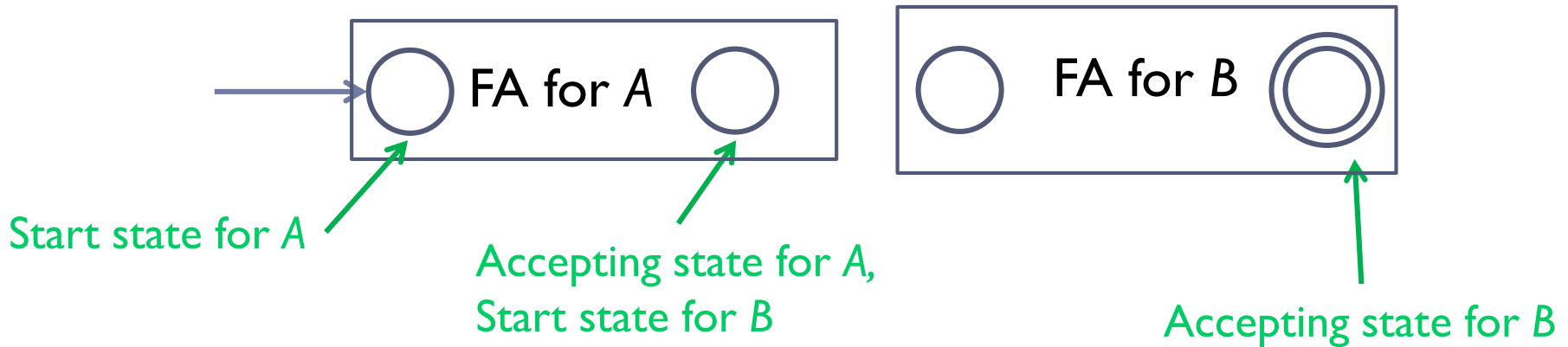


## ► An NFA for $A|B$

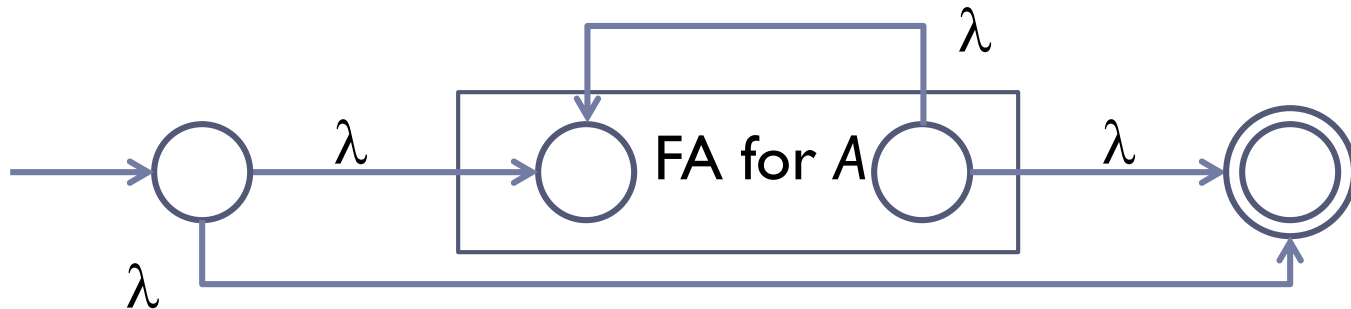


# Translating Regular Expressions into NFAs

## ► An NFA for $AB$

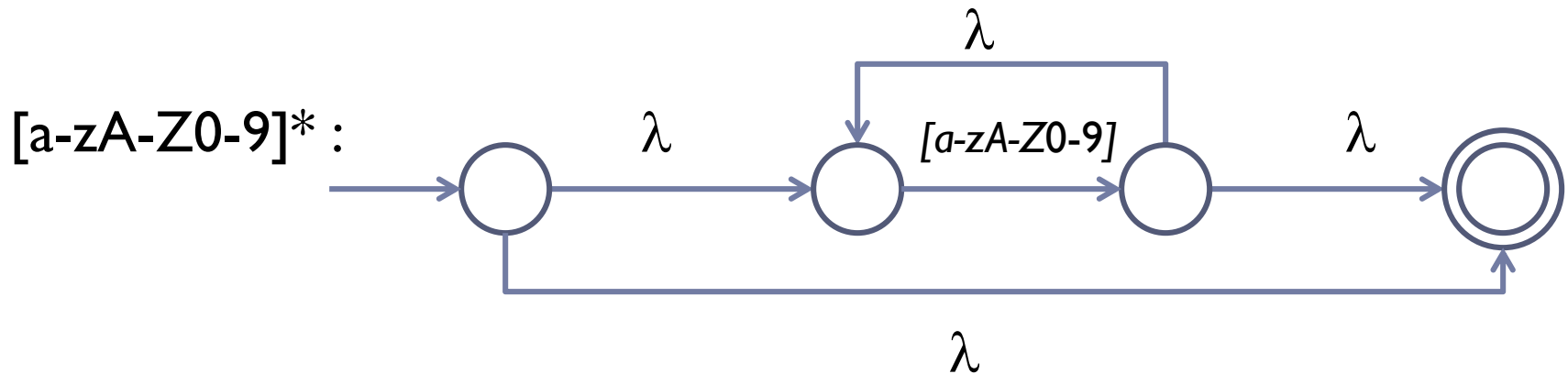


## ► An NFA for $A^*$



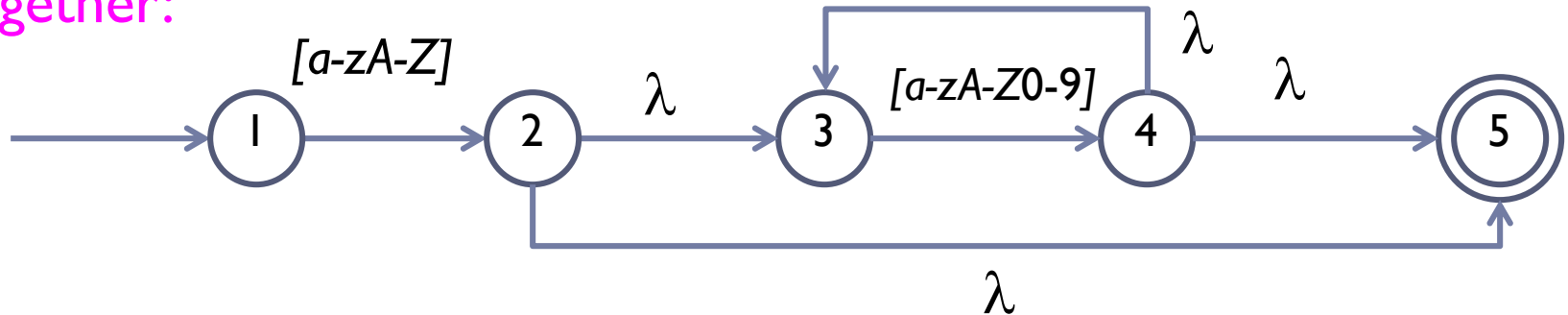
# Translating Regular Expressions into NFAs

Example:  $[a-zA-Z][a-zA-Z0-9]^*$

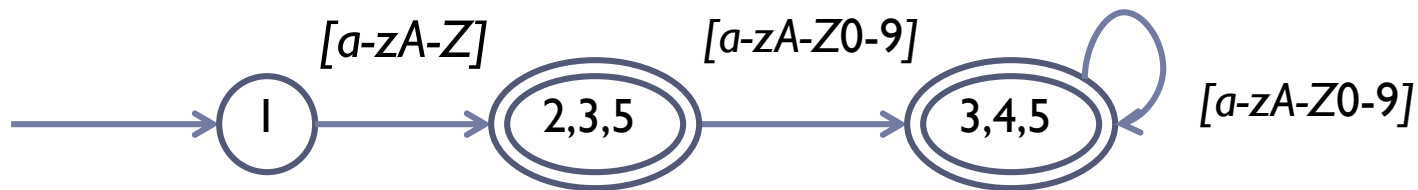


# Translating Regular Expressions into NFAs

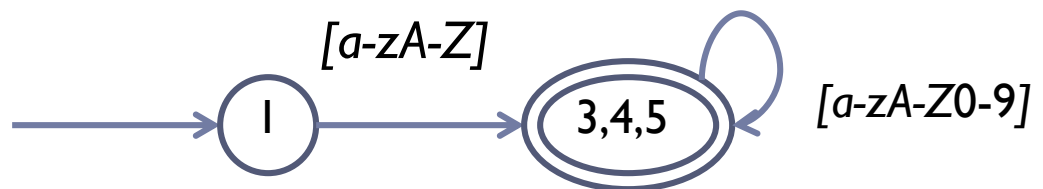
Together:



After subset construction:



After optimisation (not covered by the course):



## Test Yourself 2.3

---

Translate the regular expressions to NFAs using the McNaughton-Yamada-Thompson method.

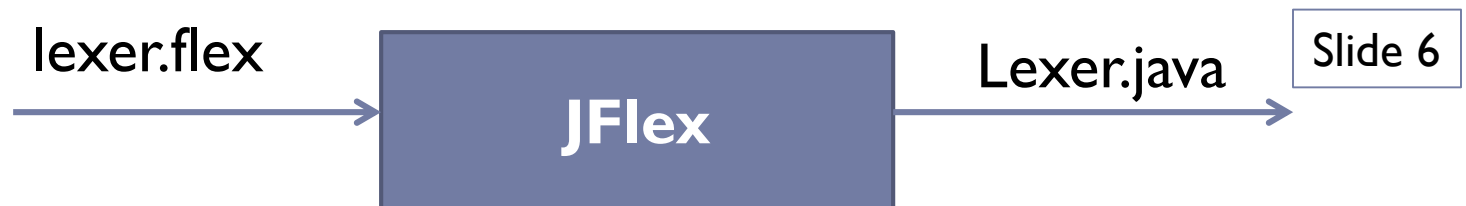
(1)  $a^*|bc$

(2)  $(a|b)c$

# Lexer Generators

---

- ▶ A very popular lexer generator, **Lex**, was developed by M.E. Lesk and E. Schmidt of AT&T Bell Laboratories. It is distributed as part of the Unix system.
- ▶ It was used primarily with programs written in C or C++ running under Unix.
- ▶ **Flex** is a widely used, freely distributed, reimplementations of Lex that produces faster and more reliable lexers.
- ▶ **JFlex** is a similar tool for use with Java.



Slide 6

# Lexer Generators

---

- ▶ A lexer specification that defines the tokens and how they are to be processed is presented to JFlex.
- ▶ JFlex generates a complete lexer coded in Java.
- ▶ This lexer is combined with other compiler components (syntax analyser, etc) to create a complete compiler.
- ▶ A lexer generator takes as its input a list of rules:

$R_1$	$\{A_1\}$
$R_2$	$\{A_2\}$
$R_3$	$\{A_3\}$
.....	

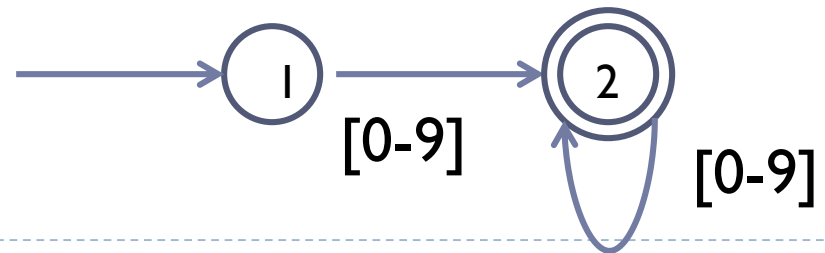
where  $R_i$  are regular expressions and  $A_i$  are snippets of Java code.

# Lexer Generators

- ▶ Intended meaning:
  - ▶ read input string one character at a time;
  - ▶ whenever the input read so far matches some  $R_i$ , execute the corresponding action  $A_i$ ;
  - ▶ then continue reading the input.
- ▶ The longest possible match between the input stream and  $R_i$  is chosen when matching  $R_i$ . For example, “123;” will be matched as one token with type INTEGER and lexeme “123” and another token SEMICOLON.

Slide 24


Regular expression:  $[0-9]^+$





# JFlex Regular Expression Syntax

---

- ▶ Concatenation is written without the dot •
- ▶ Alternation, repetition and non-empty repetition are written as  $a|b$ ,  $a^*$  and  $a^+$
- ▶ Negation  $\text{Not}(a)$  is written as  $!a$
- ▶ Single characters can be written without quotes, but only if they don't have special meaning (like  $*$  and  $+$ )
- ▶ Character classes:
  - $[0-9]$  is a character range (no quotes around 0 and 9!)
  - $[123]$  means  $'1' | '2' | '3'$  
  - Can be combined: E.g.,  $[a-z0-9]$ ,  $[0-9abc]$
  - If the character class starts with  $^$ , it is a negated character class:  $[\^0-9]$

# JFLex definition file

---

- ▶ This is the input file to the lexer generator.
- ▶ The general structure of JFLex definition files has three sections:

**User code**    -- copied to lexer.java before the  
lexer class declaration

%%

**Declarations** -- macro declarations: abbreviations to  
make lexical specifications easier to  
read and understand

%%

**Regular expression rules**

Slide 6

# Example of a simple arithmetic expression

%%

Digit = [0-9]

Alpha = [a-zA-Z\_]

%%

"+"

"-"

"\*"

"/"

"("

")"

("+"|"-"?) {Digit}+

{Alpha}({Alpha}|{Digit})\* { return new Token(**IDENTIFIER**, yytext()); }

[ \t\n]

In JFlex, when we want to use an abbreviation like Digit, we have to surround it by curly braces.

Slide 24

{ return new Token(**PLUS**, yytext()); }

{ return new Token(**MINUS**, yytext()); }

{ return new Token(**MULT**, yytext()); }

{ return new Token(**DIV**, yytext()); }

{ return new Token(**LPAREN**, yytext()); }

{ return new Token(**RPAREN**, yytext()); }

{ return new Token(**INTLIT**, yytext()); }

Slide 6

{ return new Token(**IDENTIFIER**, yytext()); }

{ /\* skip blank, tab and end of line chars \*/ }

## Example of a simple arithmetic expression

---

- ▶ Note that the method **yytext** (provided by JFlex) returns the text matched by the regular expression.
- ▶ From this specification, JFlex generates a class `Lexer` whose skeleton is:

```
public class Lexer {  
    public Lexer(InputStream in) {  
        ...  
    }  
    public Token nextToken() {  
        ...  
    }  
}
```

# Resolving ambiguities and error handling

---

- ▶ There often is more than one way to partition a given input string into tokens. For example “-12”.
- ▶ JFlex disambiguates this case using the longest match rule: If two different regular expressions  $R_i$  and  $R_j$  both match the start of the input, it chooses the one that matches the longer string. E.g., “breaker” is an identifier.
- ▶ If both  $R_i$  and  $R_j$  match the same number of characters, it prefers the one that occurs earlier in the specification. This is useful to introduce “catch all” rules for error reporting. E.g., “if” matches both a keyword and an identifier.

# Resolving ambiguities and error handling

---

- ▶ For instance, we could add the following rule at the end of our specification:
  - ```
{ System.err.println("Unexpected character " +  
                        yytext() + " " + "at line " +  
                        yyline + ", column " + yycolumn); }
```
- ▶ The dot character matches any input symbol.
- ▶ This rule will only fire if none of the other rules do.
- ▶ The action prints an explanatory error message and skips over the offending character.
- ▶ **yyline** and **yycolumn** are provided by JFlex, which contain information about the current source position.

# Processing Reserved Words

---

- ▶ Virtually all programming languages have **keywords** which are reserved. They are called **reserved words**.
- ▶ Keywords also match the lexical syntax of ordinary identifiers.
- ▶ One possible approach: Create distinct regular expressions for each reserved word before the rule for identifiers. For example,

%%

...

“if”

{return new Token(IF, “”); }

“break”

{return new Token(BREAK, “”): }

...

{Alpha}({Alpha}|{Digit})\* { return new Token(IDENTIFIER,  
yytext()); }

# Processing Reserved Words

---

- ▶ This approach increases the size of the transition table significantly.
- ▶ An alternative approach: after an apparent identifier is recognised, look up the lexeme in a keyword table to see if it is a keyword. For example,

Slide 6

```
{Alpha} ({Alpha}|{Digit})* { word = yytext();  
                             token = lookUp(keywordTable, word);  
                             return token;  
}
```

- ▶ The lookUp() method will return the token for a keyword if **word** matches one of the keywords, or return IDENTIFIER token if it does not match any keywords.



# Test Yourself 2.4

---

1. What does a lexer (lexical analyzer) do?
2. What are regular expressions used for in a compiler?
3. What are the differences between Deterministic Finite Automata and Nondeterministic Finite Automata?
4. Answer the questions on slide 25. Give the sequence of states when the DFA processes the two input strings respectively.
5. What is the Subset Construction algorithm used for?
6. What is the McNaughton-Yamada-Thompson method used for?

# Test Yourself 2.4

---

7. What is a lexer generator used for?
8. What are the input and output of a lexer?
9. What are the input and output of a lexer generator?
10. Write a regular expression that defines file names with the following pattern: it starts with a sequence of zero or more upper or lower case letters or a dash ('-') character, followed by a 2-digit number between 00 and 19, followed by a dash character, a 3-digit number between 000 and 999, and a sequence of zero or more upper or lower case letters or a dash character.

## Test Yourself 2.4

---

11. Write a regular expression for strings which represent fractions. A fraction has a numerator, a slash ('/') and a denominator. A numerator is an unsigned integer with an optional negative sign. A denominator is a positive integer. Leading zeros are not allowed in the numerator and the denominator. Some examples are: 3/10, -6/5, -1/100, and 500/2.
12. Write a regular expression for strings which are dates in the form of *dd/mm/yy*. The expressions *dd* should be in the range of 01 to 31, *mm* in the range of 01 to 12 and *yy* in the range of 00 to 99 respectively. There is no need to exclude dates like 31/04/00 or 29/02/11.

## Test Yourself 2.4

---

13. Use the McNaughton-Yamada-Thompson method to construct a nondeterministic finite automaton from each regular expression below.

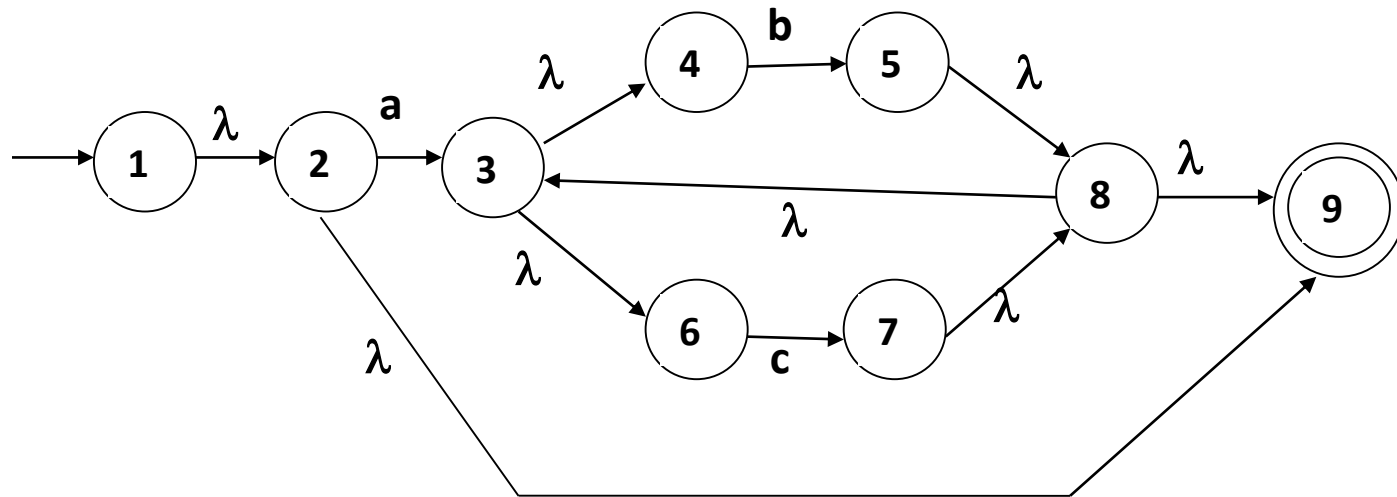
1)  $1(0/1)^*$

2)  $(a/b)^*c$

## Test Yourself 2.4

14. Transform each nondeterministic finite automaton below to the deterministic finite automaton with subset construction.

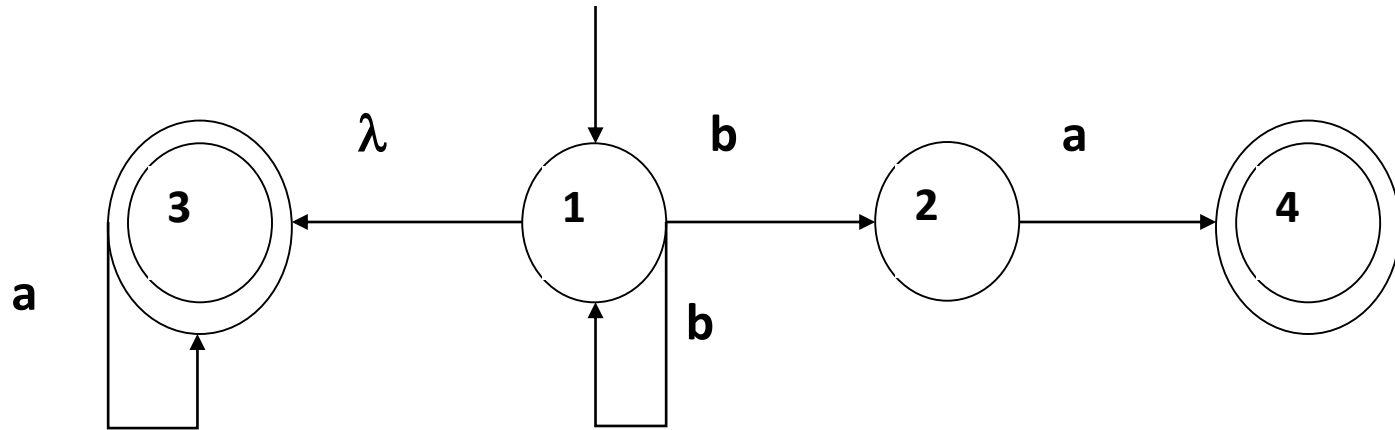
1)



# Test Yourself 2.4

---

2)



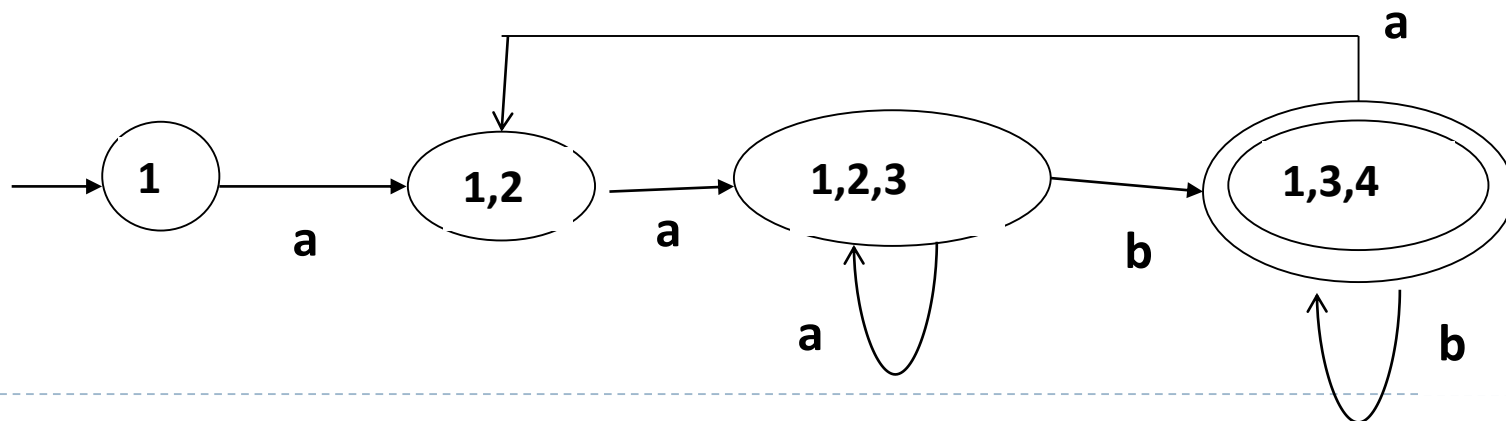
## Test Yourself 2.1 (answers)

---

1.  $AA$  is the concatenation of two decimal digits.  $AA = \{“00”, “01”, \dots “99”\}$ ,  $A^* =$  the set of strings of zero or more decimal digits,  $A^+ =$  the set of strings of one or more decimal digits
2. Regular expression:  $r([2-9] \mid 1(\lambda \mid [0-6]))$
3. Regular expression:  $0x[0-9a-fA-F]^+$

## Test Yourself 2.2 (answers)

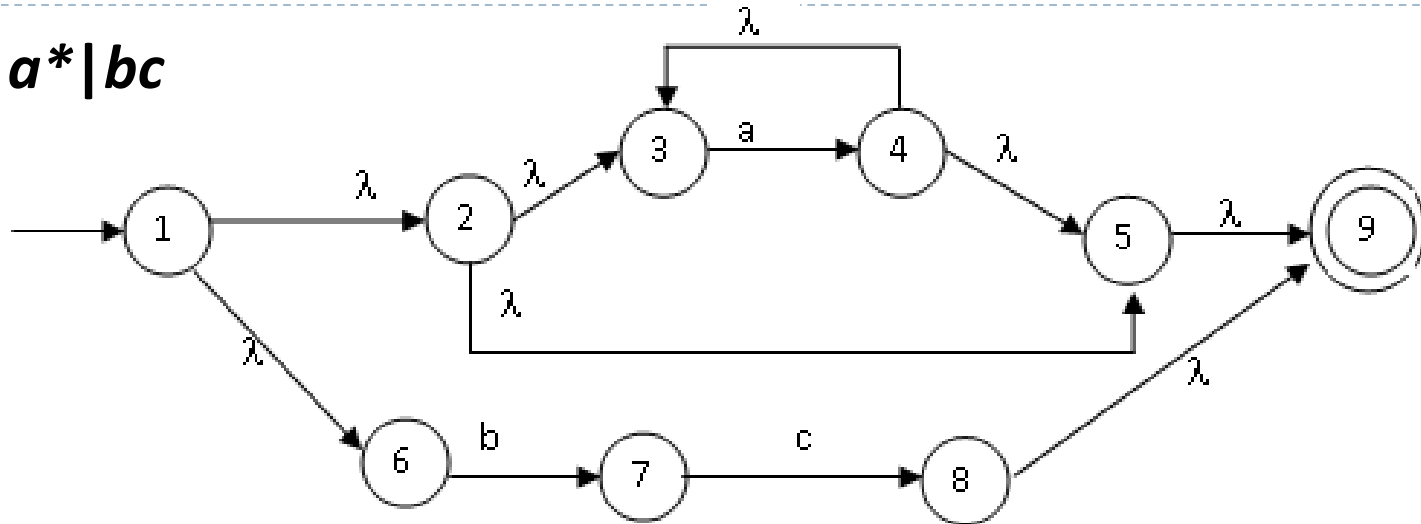
| NFA states/ DFA states<br>(current state) | Next input symbol |       |
|-------------------------------------------|-------------------|-------|
|                                           | a                 | b     |
| 1 (start state)                           | 1,2               |       |
| 1,2                                       | 1,2,3             |       |
| 1,2,3                                     | 1,2,3             | 1,3,4 |
| 1,3,4 (final state)                       | 1,2               | 1,3,4 |



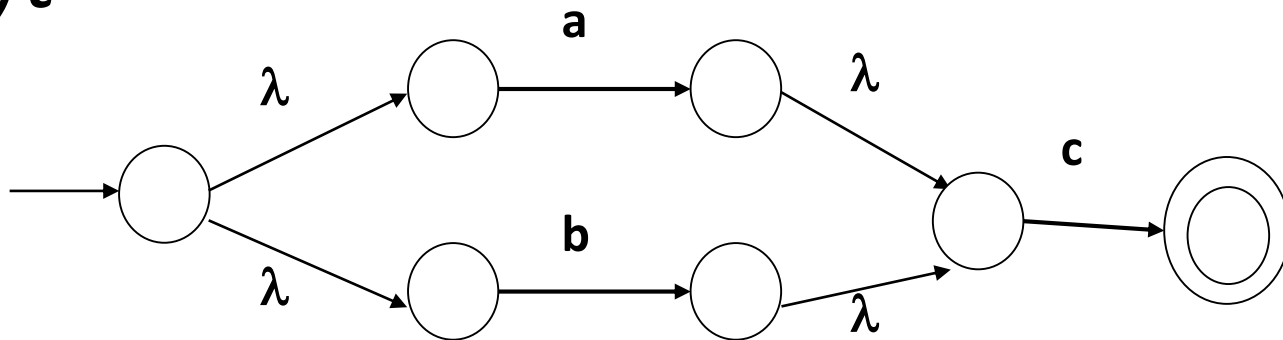


# Test Yourself 2.3 (answers)

1)  $a^*|bc$

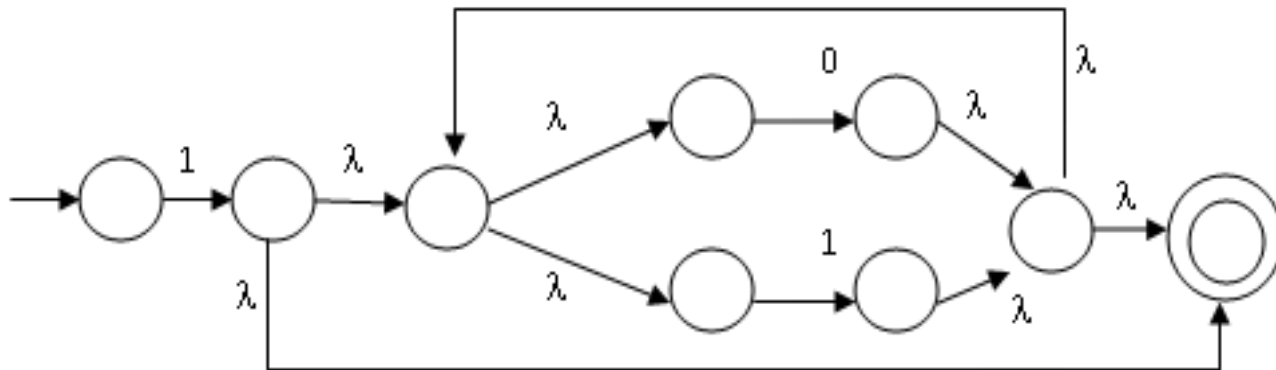


2)  $(a/b)c$



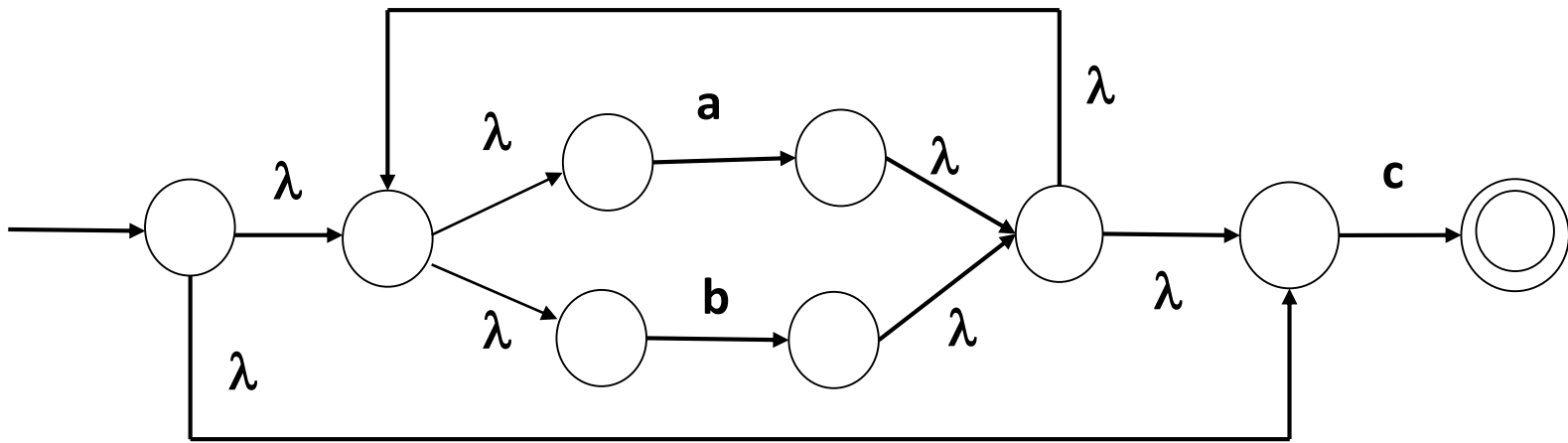
## Test Yourself 2.4 (selected answers)

10.  $[A-Za-z'\text{'}]^* [01][0-9] '\text{' } [0-9]^3 [A-Za-z'\text{'}]^*$
11.  $('\text{' }|\lambda)(0|[1-9][0-9]^*) '\text{' } [1-9][0-9]^*$
12.  $(0[1-9]|1[2][0-9]|3[01]) '\text{' } (0[1-9]|1[0-2]) '\text{' } [0-9]^2$
13.  $1) 1 (0|1)^*$



# Test Yourself 2.4 (selected answers)

13. 2)  $(a/b)^*c$



# Test Yourself 2.4 (selected answers)

14. 1)

| DFA states                           | NFA states       | a       | b                | c                |
|--------------------------------------|------------------|---------|------------------|------------------|
| $s_1$<br>(start state & final state) | 1, 2, 9          | 3, 4, 6 | none             | none             |
| $s_2$                                | 3, 4, 6          | none    | 5, 8, 9, 3, 4, 6 | 7, 8, 9, 3, 4, 6 |
| $s_3$<br>(final state)               | 5, 8, 9, 3, 4, 6 | none    | 5, 8, 9, 3, 4, 6 | 7, 8, 9, 3, 4, 6 |
| $s_4$<br>(final state)               | 7, 8, 9, 3, 4, 6 | none    | 5, 8, 9, 3, 4, 6 | 7, 8, 9, 3, 4, 6 |

2)

