中山大學
SUN YAT-SEN UNIVERSITY

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Compilation Principle
编 译 原 理

第2讲：词法分析(2)

张献伟

xianweiz.github.io

DCS290, 2/24/2022

Review Questions

- Q1: lexical analysis of “int b = a + 1”?
  <keyword, ‘int’>, <id, ’b’>, <op, ‘=‘>, <id, ‘a’>, <op, ‘+’>, <num, 1>
- Q2: RE of identifiers in C language?
  (_letter)(_letter|digit)*
- Q3: [-+]?[0-9]+\.?[0-9]+
  Floating point, or integers with two or more digits
- Q4: RE for numbers?
  [-+]?[0-9]*\.?[0-9]+
- Q5: L_1 and L_2 are two languages, what are L_1L_2, L_1^3, L_1^*?
  New languages of concatenation, L_1 3-time concat, closure
- Q6: Let Σ = {a, b}. Write RE to define language consisting of strings w such that, w of length even?
  r = ((a|b)(a|b))*

Summary: RE

- We have learnt how to specify tokens for lexical analysis[定义token]
  - Regular expressions
    - Concise notations for the string patterns

- Used in lexical analysis with some extensions[适度扩展]
  - To resolve ambiguities
  - To handle errors

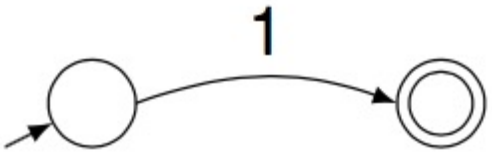
- REs is only a language specification[只是定义了语言]
  - An implementation is still needed
  - Next: to construct a token recognizer for languages given by regular expressions – by using **finite automata**[有穷自动机]

Impl. of Lexical Analyzer[实现]

- How do we go from specification to implementation?
  - RE → finite automata
- **Solution 1**: to implement using a tool — Lex (for C), Flex (for C++), Jlex (for java)
  - Programmer specifies tokens using REs
  - The tool generates the source code from the given REs
    - The Lex tool essentially does the following translation: REs (specification) ⇒ FAs (implementation)
- **Solution 2**: to write the code yourself
  - More freedom; even tokens not expressible through REs
  - But difficult to verify; not self-documenting; not portable; usually not efficient
  - Generally not encouraged

Transition Diagram[转换图]

- REs → transition diagrams
  - By hand
  - Automatic



- Node[节点]: state
  - Each state represents a condition that may occur in the process
  - Initial state (Start): only one, circle marked with 'start →'
  - Final state (Accepting): may have multiple, double circle

- Edge[边]: directed, labeled with symbol(s)
  - From one state to another on the input

Finite Automata[有穷自动机]

- **Regular Expression** = specification[正则表达是定义]
- **Finite Automata** = implementation[自动机是实现]

- Automaton (pl. automata): a machine or program
- **Finite automaton** (FA): a program with a finite number of states

- Finite Automata are similar to transition diagrams
  - They have states and labelled edges
  - There are one unique start state and one or more than one final states

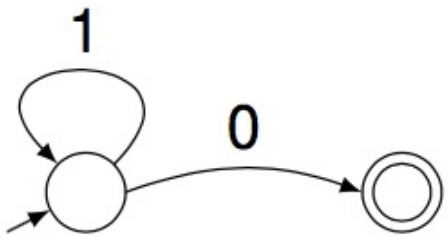
FA: Language

- An FA is a program for classifying strings (accept, reject)
  - In other words, a program for recognizing a language
  - The Lex tool essentially does the following translation: REs (specification) ⇒ FAs (implementation)
  - For a given string ‘x’, if there is transition sequence for ‘x’ to move from start state to certain accepting state, then we say ‘x’ is accepted by the FA
    - Otherwise, rejected

- Language of FA = set of strings accepted by that FA
  - L(FA) ≡ L(RE)

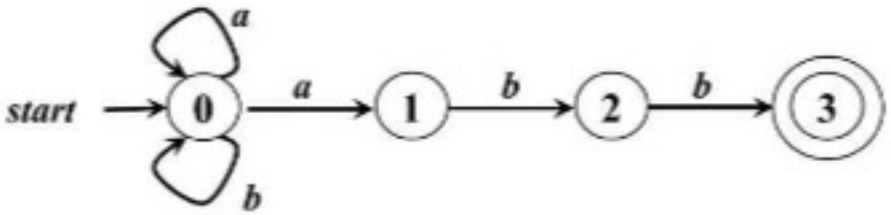
Example

- Are the following strings acceptable?
  - 0 ✓
  - 1 ✗
  - 11110 ✓
  - 11101 ✗
  - 11100 ✗
  - 1111110 ✓



- What language does the state graph recognize? ∑ = {0, 1}

  Any number of '1's followed by a single 0



L(FA): all strings of ∑ {a, b}, ending with 'abb'

L(RE) = (a|b)*abb

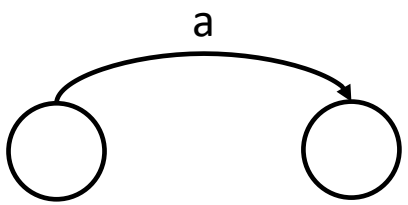
DFA and NFA

- Deterministic Finite Automata (**DFA**): the machine can exist in only one state at any given time[确定]
  - One transition per input per state
  - No ε-moves
  - Takes only one path through the state graph

- Nondeterministic Finite Automata (**NFA**): the machine can exist in multiple states at the same time[非确定]
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
  - Can choose which path to take
    - An NFA accepts if some of these paths lead to accepting state at the end of input

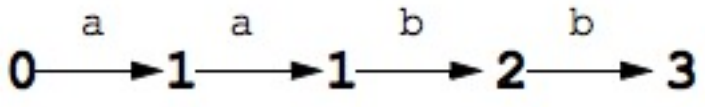
State Graph

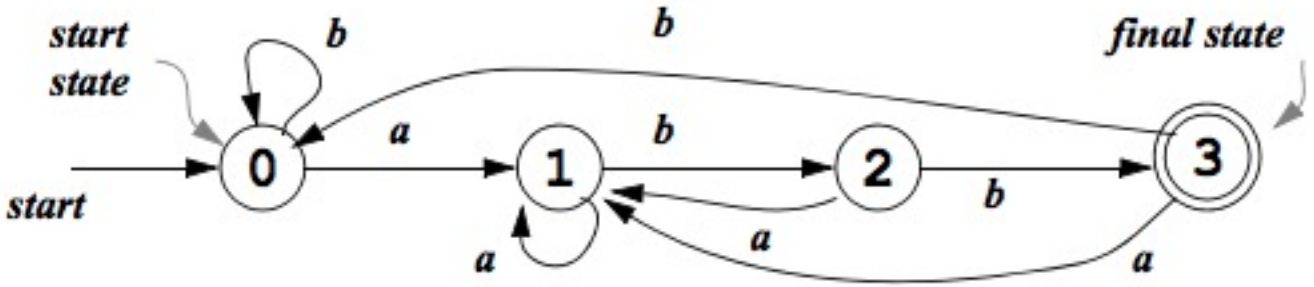
- 5 components （∑, S, n, F, δ）
  - An input alphabet Σ
  - A set of states *S*
  - A start state $n \in S$
  - A set of accepting states $F \subseteq S$
  - A set of transitions δ: $S_a \xrightarrow{\text{input}} S_b$



Example: DFA

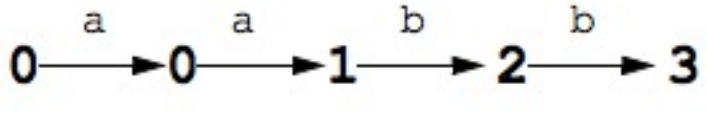
- There is **only one** possible sequence of moves --- either lead to a final state and accept or the input string is rejected
  - Input string: aabb
  - Successful sequence:

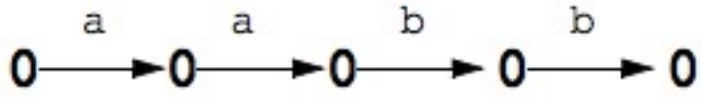


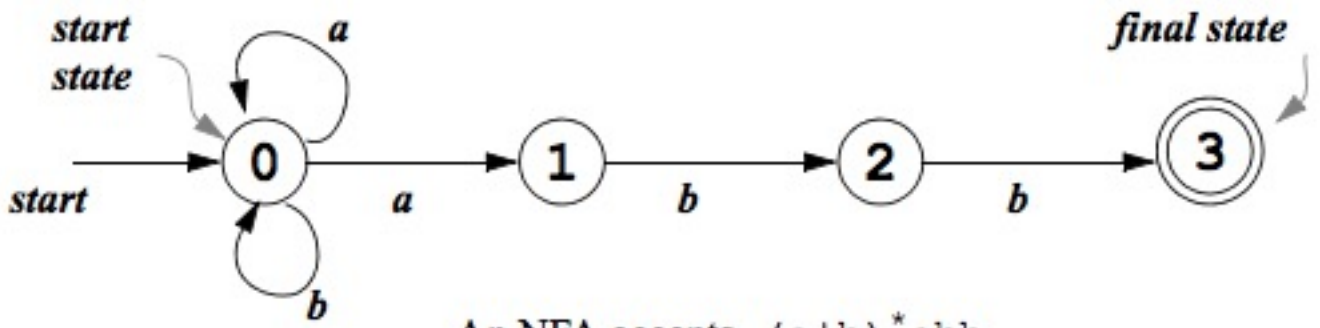


A DFA accepts $(a|b)^*abb$

Example: NFA

- There are **many possible** moves: to accept a string, we only need one sequence of moves that lead to a final state
  - Input string: aabb
  - Successful sequence: 

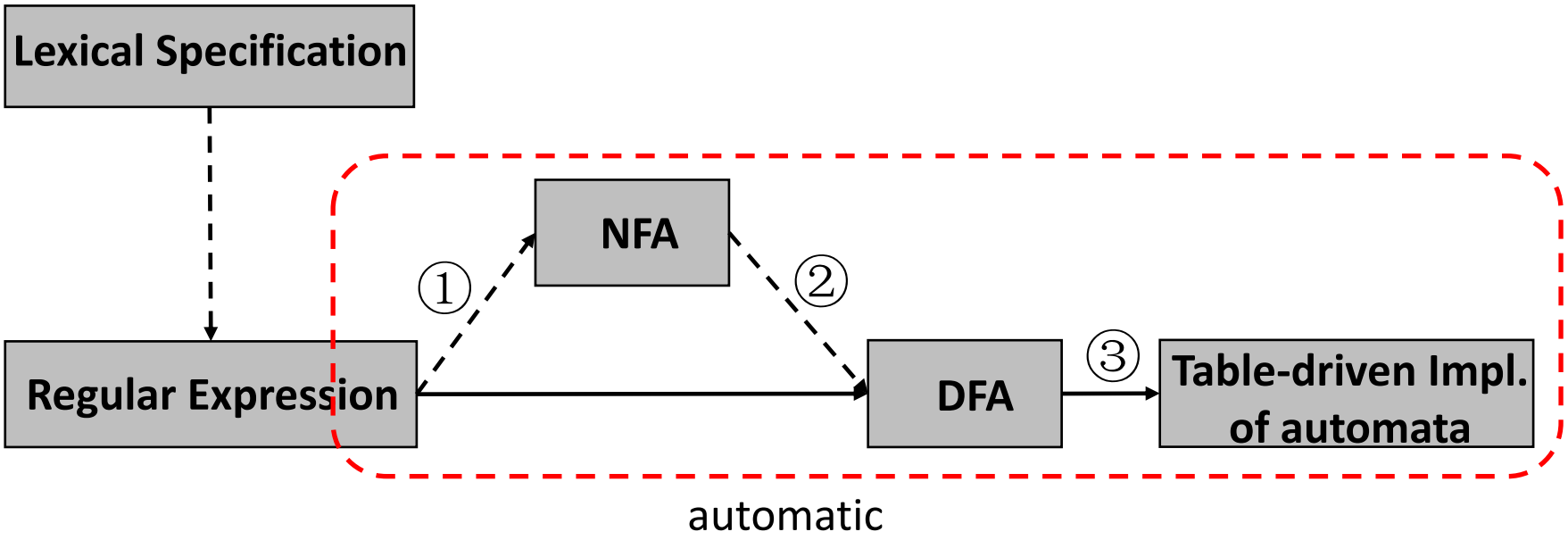
  - Unsuccessful sequence: 



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

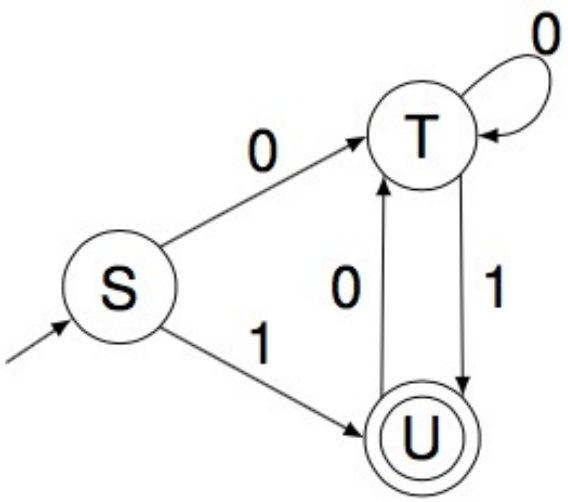
Conversion Flow[转换流程]

- Outline: RE → NFA → DFA → Table-driven Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs
  - ② Converting NFAs to DFAs



DFA → Table

- FA can also be represented using transition table



alphabet

state

	0	1
S	**T**	**U**
T	**T**	**U**
U	**T**	**x**

Table-driven Code:

```
DFA() {
    state = "S";
    while (!done) {
        ch = fetch_input();
        state = Table[state][ch];
        if (state == "x")
            print("reject");
    }
    if (state ∈ F)
        printf("accept");
    else
        printf("reject");
}
```

Q: which is/are accepted?

111

000

001

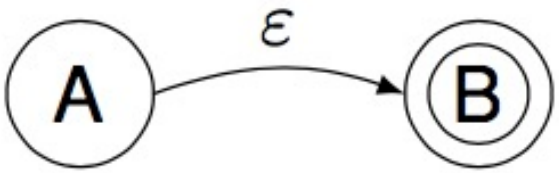
Discussion

- Implementation is efficient[表格是一种高效实现]
  - Table can be automatically generated
  - Need finite memory O(S x ∑)
    - Size of transition table
  - Need finite time O(input length)
    - Number of state transitions

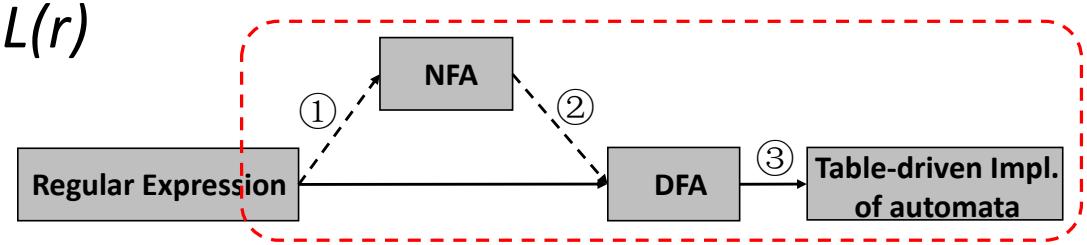
- Pros and cons of table[表格实现的优劣]
  - Pro: can easily find the transitions on a given state and input
  - Con: takes a lot of space, when the input alphabet is large, yet most states do not have any moves on most of the input symbols

RE → NFA

- NFA can have ε-moves
  - Edges labelled with ε
  - move from state A to state B without reading any input

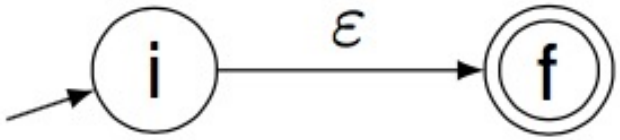


- **M-Y-T algorithm** to convert any RE to an NFA that defines the same language
  - Input: RE *r* over alphabet ∑
  - Output: NFA accepting *L(r)*

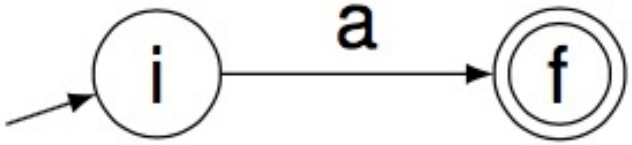


RE → NFA (cont.)

- Step 1: processing atomic REs
  - ε expression[空]
    - *i* is a new state, the start state of NFA
    - *f* is another new state, the accepting state of NFA

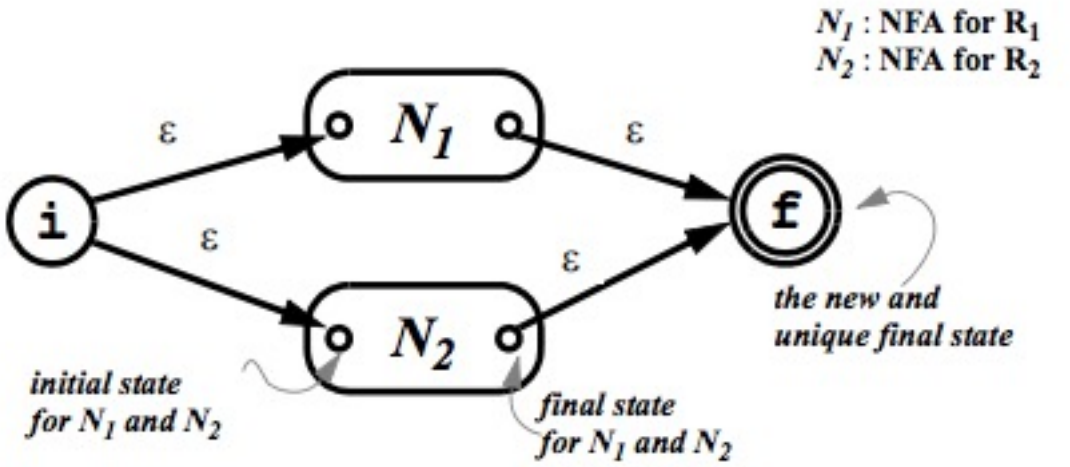


  - Single character RE *a*[单字符]

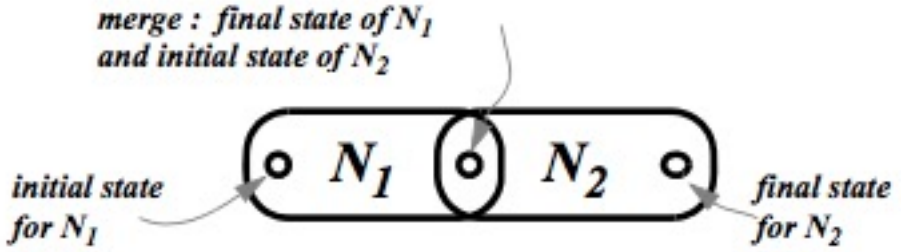


RE → NFA (cont.)

- Step 2: processing compound REs[组合]
  - $R = R_1 \mid R_2$

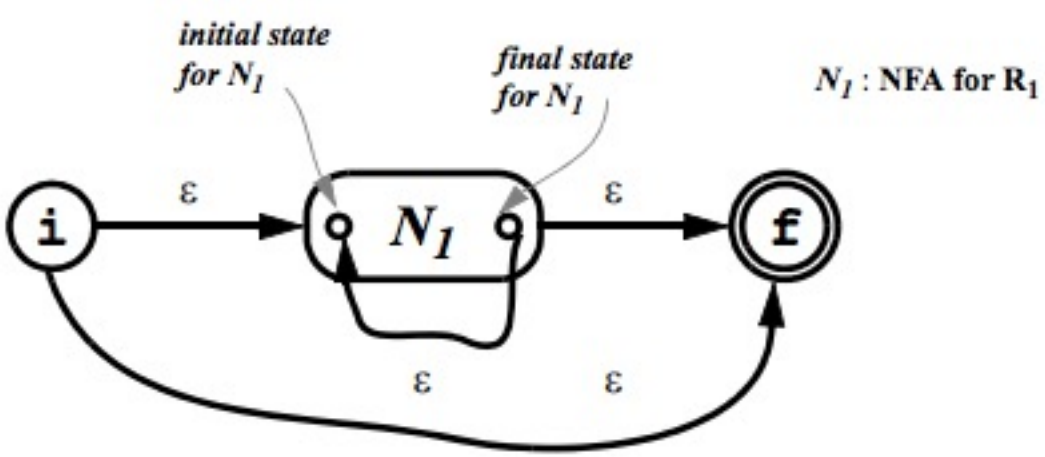


  - $R = R_1R_2$



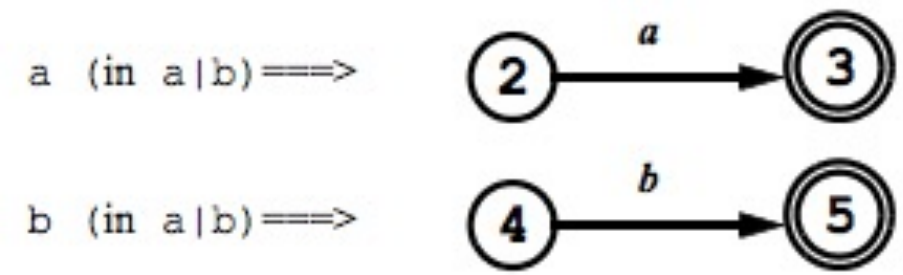
RE → NFA (cont.)

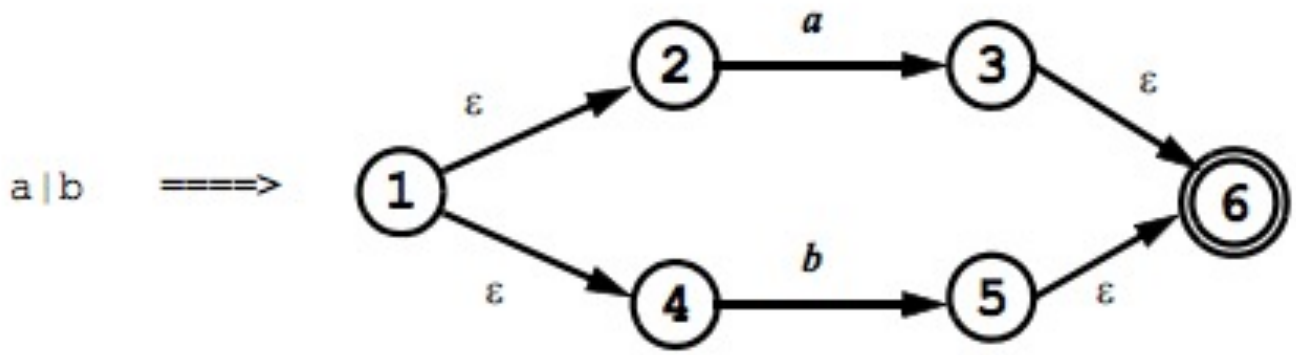
- Step 2: processing compound REs
  - $R = R_1^*$



Example

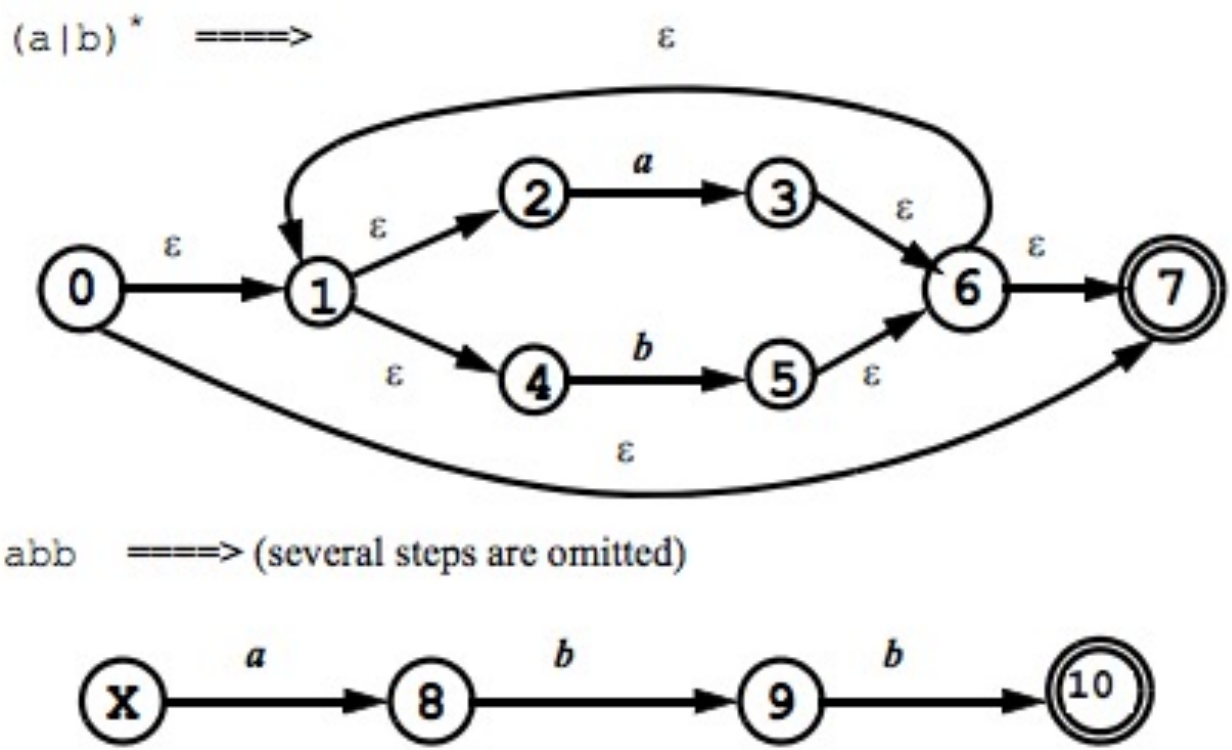
- Convert "(a|b)*abb" to NFA





Example (cont.)

- Convert "(a|b)*abb" to NFA



Example (cont.)

- Convert "(a|b)*abb" to NFA