Compilation Principle
编 译 原 理

第2讲：词法分析(2)

张献伟

xianweiz.github.io

DCS290, 3/4/2021

Review Questions

Q1: input and output of lexical analysis?

character stream → tokens

Q2: how to denote a token?

<class, lexeme>

Q3: atomic and compound REs?

atomic: ε, {a}

compound: R1|R2, R1R2, R1*

Q4: (+|-)?([0-9])*(0|2|4|6|8)

even numbers

Q5: RE of identifiers in C language?

(_letter)(_letter|digit)*

Alphabet Operations[字母表运算]

- Product[乘积]： $\Sigma_1 \Sigma_2 = \{ab \mid a \in \Sigma_1, b \in \Sigma_2\}$
  - E.g., {0, 1}{a, b} = {0a, 0b, 1a, 1b}

- Power[幂]: $\Sigma^n = \Sigma^{n-1} \Sigma$, $n \geq 1$; $\Sigma^0 = \{\varepsilon\}$
  - Set of strings of length n
  - $\{0, 1\}^3$ = {0, 1}{0, 1}{0, 1} = {000, 001, 010, 011, 100, 101, 110, 111}

- Positive Closure[正闭包]: $\Sigma^+ = \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$
  - {a, b, c}+ = {a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, ...}

- Kleene Closure[闭包]: $\Sigma = \Sigma^0 \cup \Sigma^+$

Regular Expressions

- Atomic[原子]
  - ε is a RE: $L(\varepsilon) = \{\varepsilon\}$
  - If $a \in \Sigma$, then a is a RE: $L(a) = \{a\}$

- Compound[组合]
  - If both r and s are REs, corr. to languages $L(r)$ and $L(s)$, then：
  - $r|s$ is a RE: $L(r|s) = L(r) \cup L(s)$
  - rs is a RE: $L(rs) = L(r)L(s)$
  - r^* is a RE: $L(r^*) = (L(r))^*$
  - (r) is a RE: $L((r)) = L(r)$

Different REs of the Same Language

- (a|b)* = ?
  - L((a|b)*) = (L(a|b))* = (L(a) ∪ L(b))* = ({a} ∪ {b})* = {a, b}*
  - = $\{a, b\}^0 + \{a, b\}^1 + \{a, b\}^2 + \ldots$
  - = {ε, a, b, aa, ab, ba, bb, aaa, …}

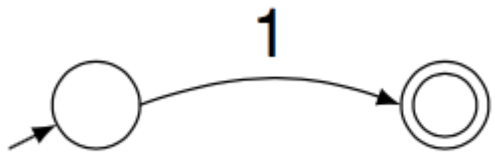
- (a*b*)* = ?
  - L((a*b*)*) = (L(a*b*))* = (L(a*)L(b*))*
  - = L({ε, a, aa, …}{ε, b, bb, …})*
  - = L({ε, a, b, aa, ab, bb, …})*
  - = ε + {ε, a, b, aa, ab, bb, …} + $\{\varepsilon, a, b, aa, ab, bb, \ldots\}^2$ + $\{\varepsilon, a, b, aa, ab, bb, \ldots\}^3$ + …

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

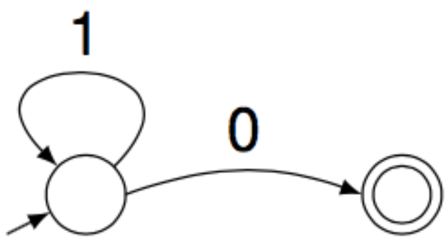
FA: Language

- 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

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

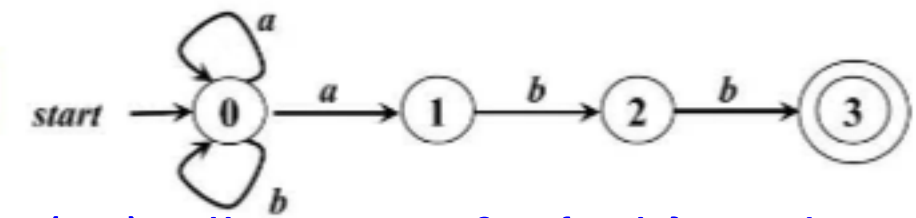
Example

- 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

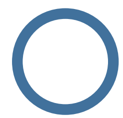
DFA and NFA

- 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

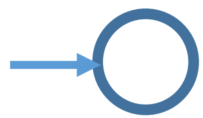
- Nondeterministic Finite Automata (**NFA**): the machine can exist in multiple states at the same time[非确定]
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
  - Can choose which path to take
    - An NFA accepts if some of these paths lead to accepting state at the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

State Graph

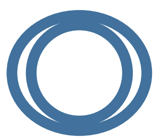
- 5 components （∑, S, n, F, δ）
  - An input alphabet Σ
  - A set of states S



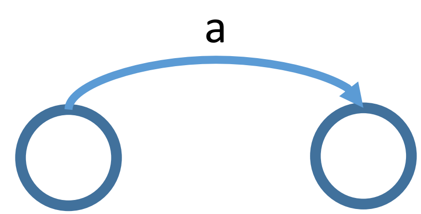
  - A start state $n \in S$



  - A set of accepting states $F \subseteq S$

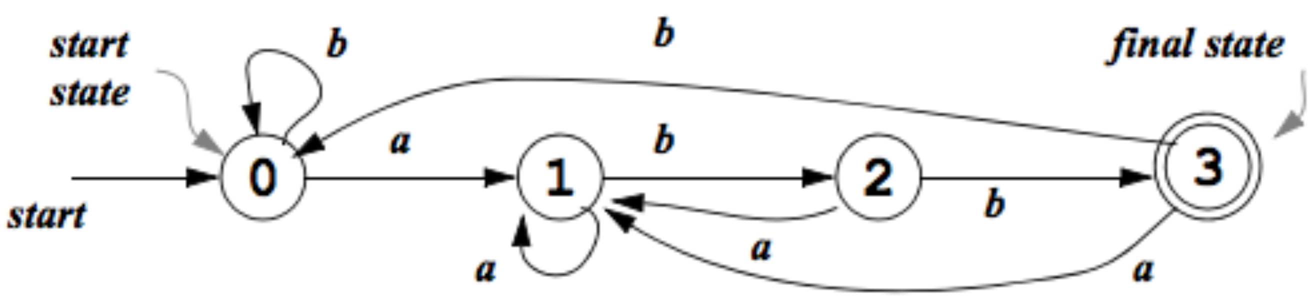


  - A set of transitions δ: $S_a \xrightarrow{\text{input}} S_b$



Example: DFA

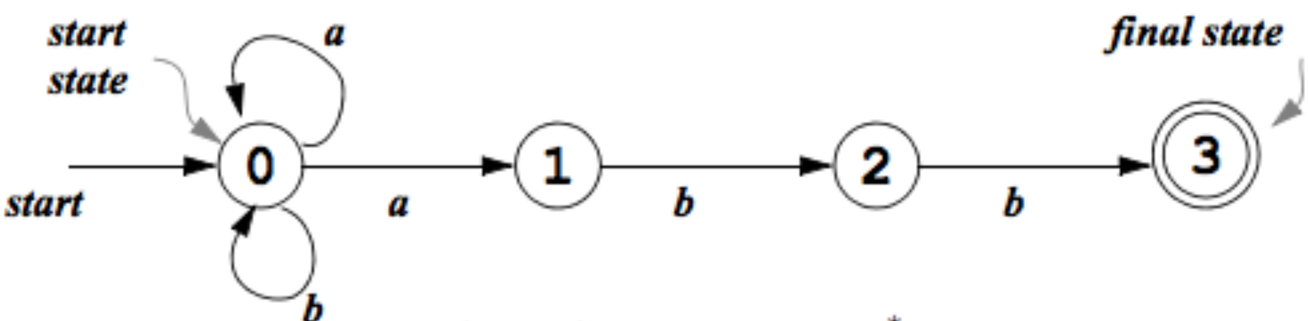
- There is **only one** possible sequence of moves --- either lead to a final state and accept or the input string is rejected
  - Input string: aabb
  - Successful sequence: 0 $\xrightarrow{a}$ 1 $\xrightarrow{a}$ 1 $\xrightarrow{b}$ 2 $\xrightarrow{b}$ 3



A DFA accepts (a|b)*abb

Example: NFA

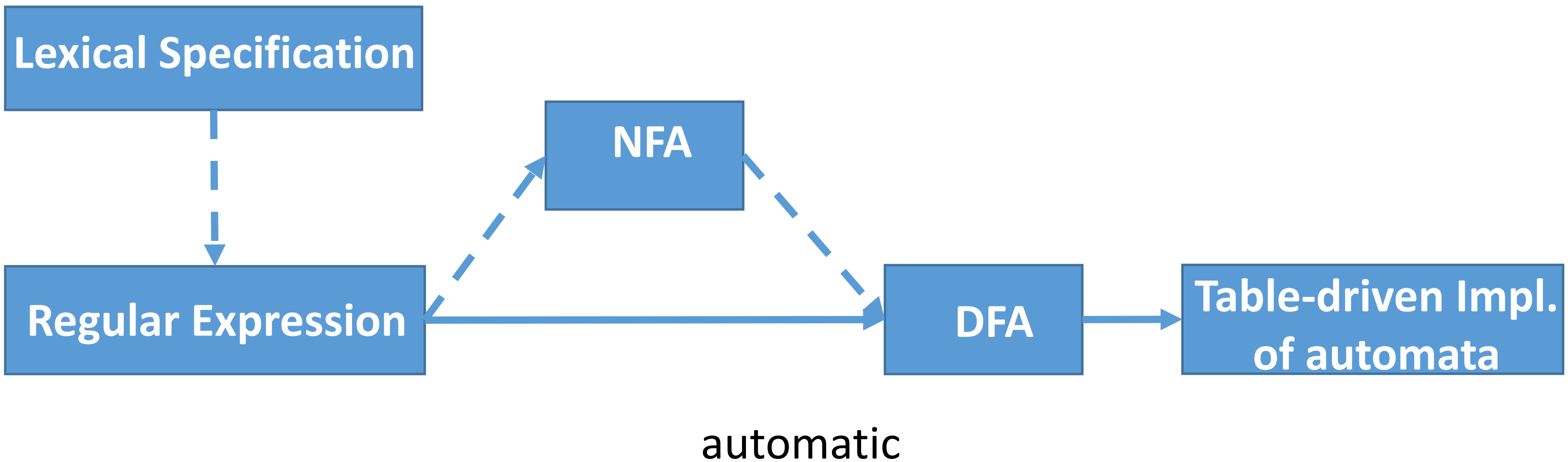
- There are **many possible** moves: to accept a string, we only need one sequence of moves that lead to a final state
  - Input string: aabb
  - Successful sequence: $0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$
  - Unsuccessful sequence: $0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$



An NFA accepts (a|b)*abb

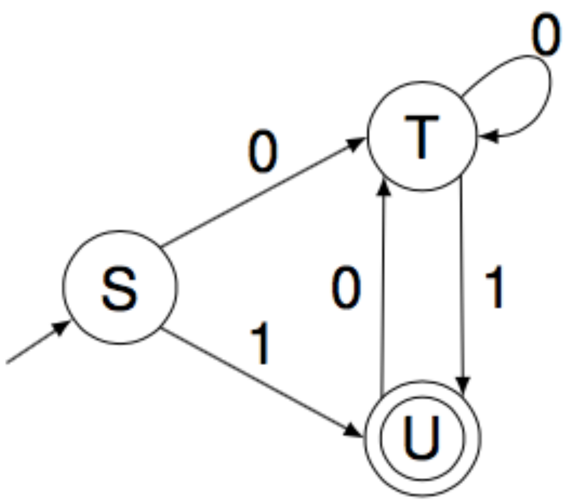
Conversion Flow[转换流程]

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

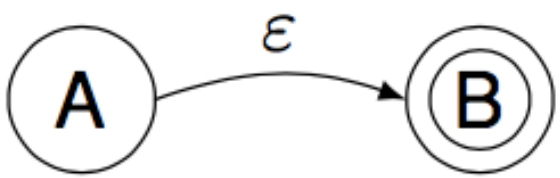
16

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

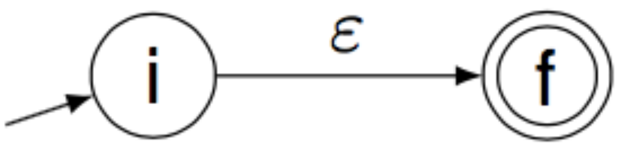
- NFA can have ε-moves
  - Edges labelled with ε
  - move from state A to state B without reading any input



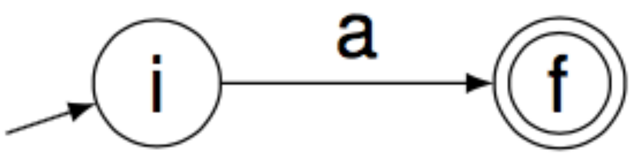
- **M-Y-T algorithm** to convert any RE to an NFA that defines the same language
  - Input: RE *r* over alphabet ∑
  - Output: NFA accepting *L(r)*

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

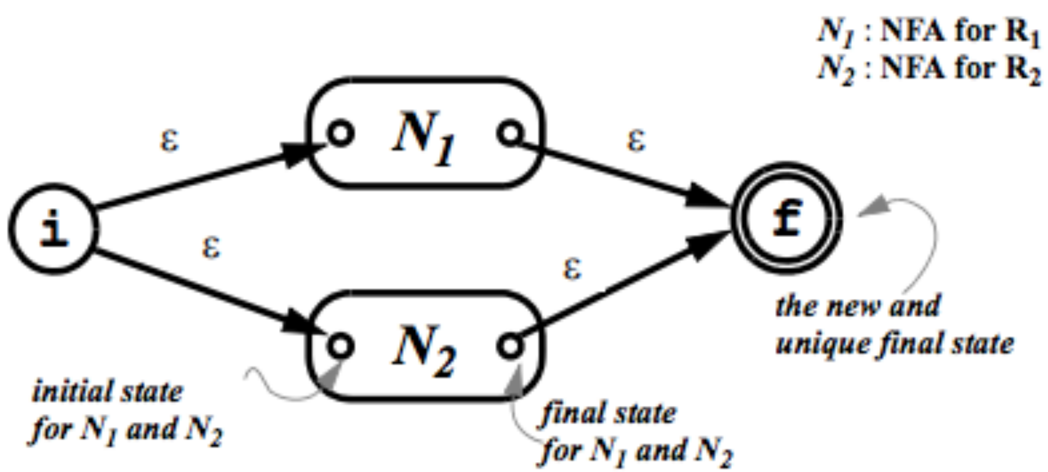


  - Single character RE *a*[单字符]

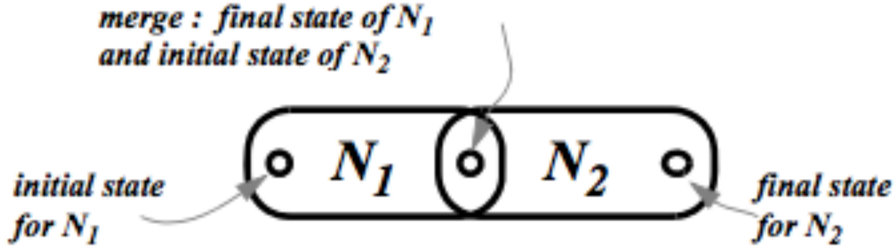


RE → NFA (cont.)

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

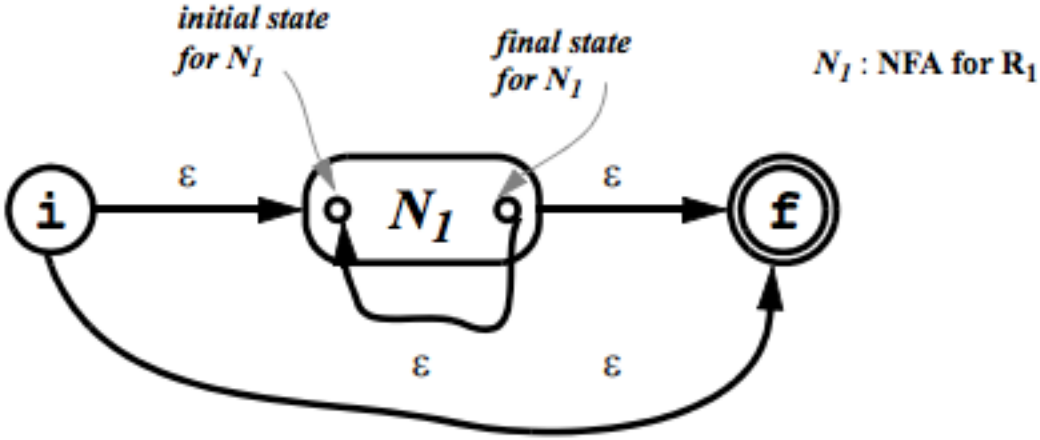


  - $R = R_1R_2$



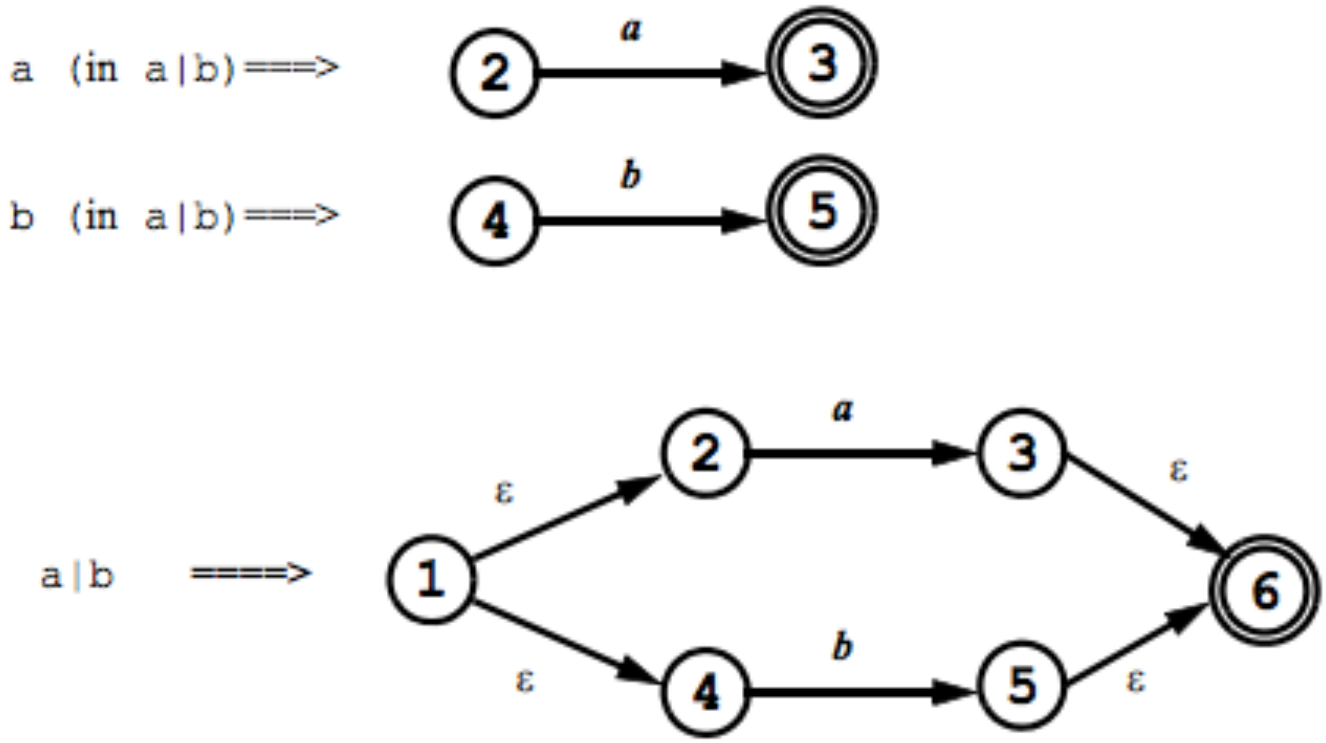
RE → NFA (cont.)

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



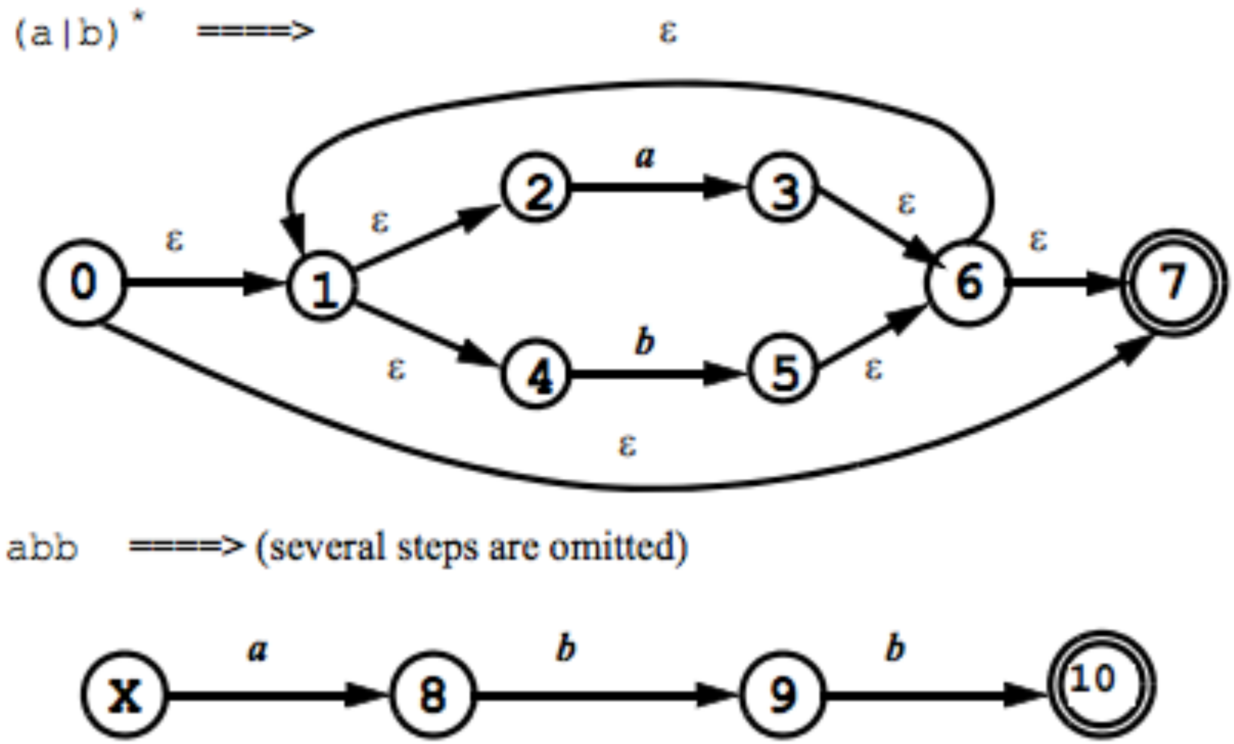
Example

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



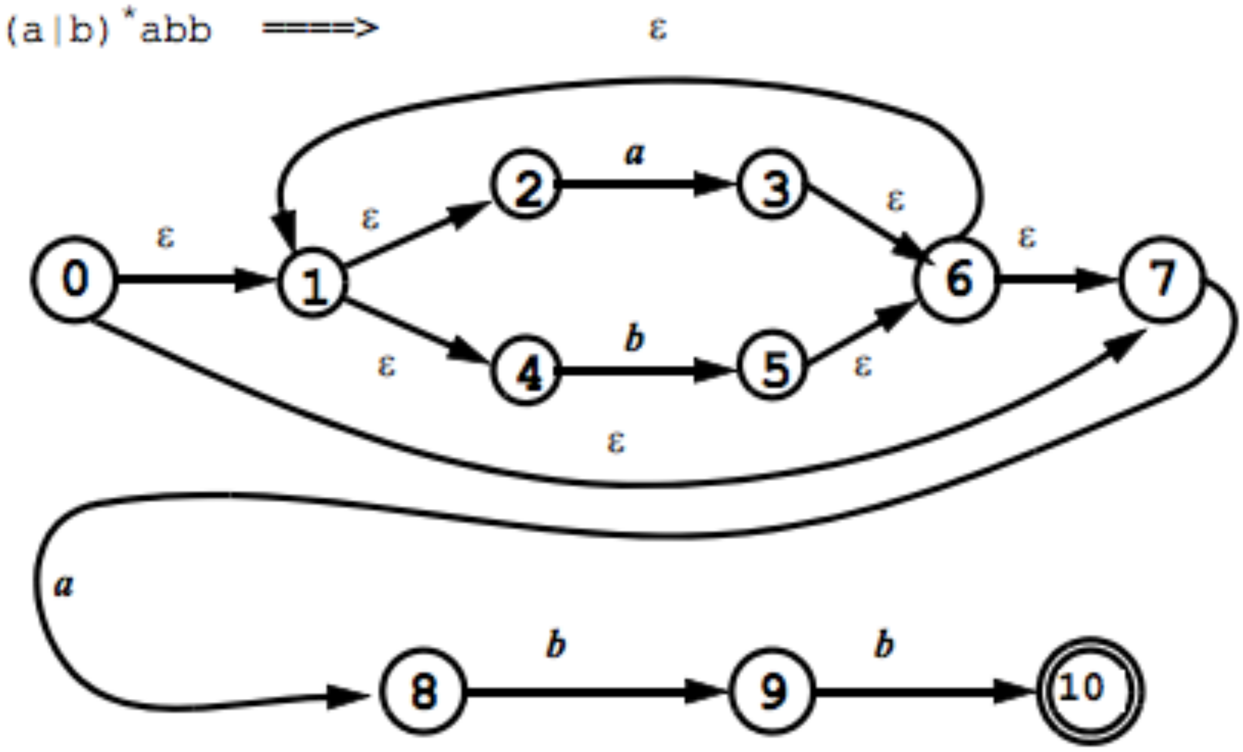
Example (cont.)

- Convert “(a|b)*abb” to NFA



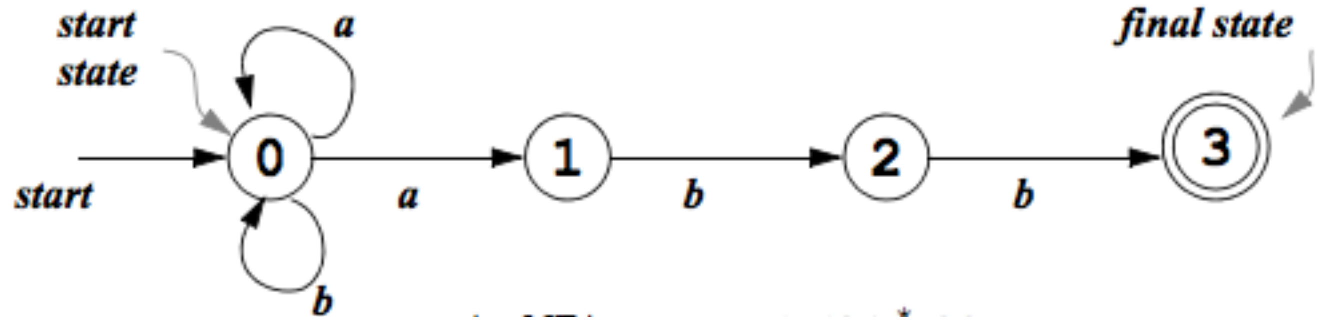
Example (cont.)

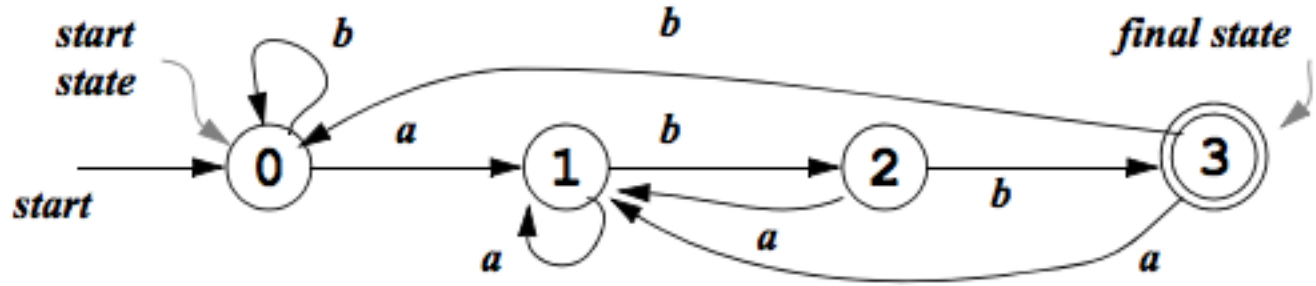
- Convert “(a|b)*abb” to NFA



NFA → DFA: Same[等价]

- NFA and DFA are equivalent





NFA → DFA: Theory[相关理论]

- Question: is L(NFA) ⊆ L(DFA)?
  - Otherwise, conversion would be futile
- Theorem: L(NFA) ≡ L(DFA)
  - Both recognize regular languages L(RE)
  - Will show L(NFA) ⊆ L(DFA) by construction (NFA → DFA)
  - Since L(DFA) ⊆ L(NFA), L(NFA) ≡ L(DFA)
- Resulting DFA consumes more memory than NFA
  - Potentially larger transition table as shown later
- But DFAs are faster to execute
  - For DFAs, number of transitions == length of input
  - For NFAs, number of potential transitions can be larger
- NFA → DFA conversion is done because the speed of DFA far outweigh its extra memory consumption