计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第2讲：词法分析(2)

张献伟

xianweiz.github.io

DCS290, 2/28/2023

Review Questions

- Q1: lexical analysis of "while (i>=1"?

  (keyword, 'while'), (sym, '('), (id, 'i'), (sym, '>='), (num, '1')

- Q2: $\Sigma = \{a\}$, $L_1 = \{aa\}$, $L_2=\{aaa\}$. What are $L_1 \mid L_2$ and L_1L_2?

  $L_3 = L_1 \mid L_2 = \{aa\} \mid \{aaa\} = \{aa, aaa\}$, $L_4 = L_1L_2 = \{aaaaa\}$

- Q3: L_3^2?

  $L_3^2 = L_3L_3 = \{aa, aaa\}\{aa, aaa\} = \{aaaa, aaaaa, aaaaaa\}$

- Q4: describe the meaning of $L_1^* \mid L_2^*$?

  A language composed of '*a*'s of length 2X and 3X, including ε

- Q5: is $(L_1 \mid L_2)^*$ of the same meaning?

  $(L_1 \mid L_2)^* = L_3^* = \{L_3^0, L_3^1, L_3^2, \ldots\} = \{\varepsilon, aa, aaa, aaaa, aaaaa, aaaaaa, \ldots\}$

- Q6: RE of identifiers in C language?

  $(\_letter)(\_letter|digit)^*$

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

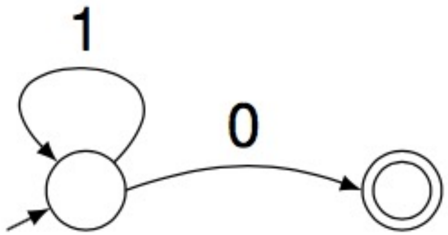
FA: Language

- 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)

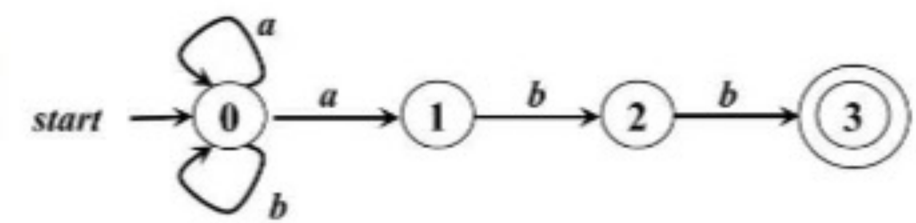
Example

- 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

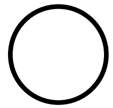
DFA and NFA

- 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 <u>only one path</u> through the state graph

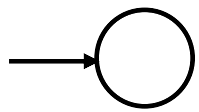
- Nondeterministic Finite Automata (**NFA**): the machine can exist in multiple states at the same time[非确定]
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
  - Can choose which path to take
    - An NFA accepts if <u>some of these paths</u> lead to accepting state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

State Graph

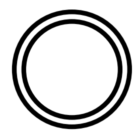
- 5 components （Σ, S, n, F, δ）
  - An input alphabet Σ
  - A set of states S



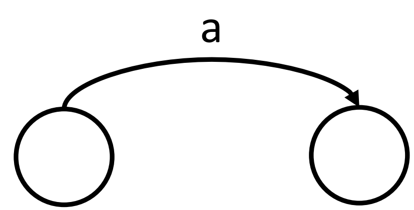
  - A start state $n \in S$



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

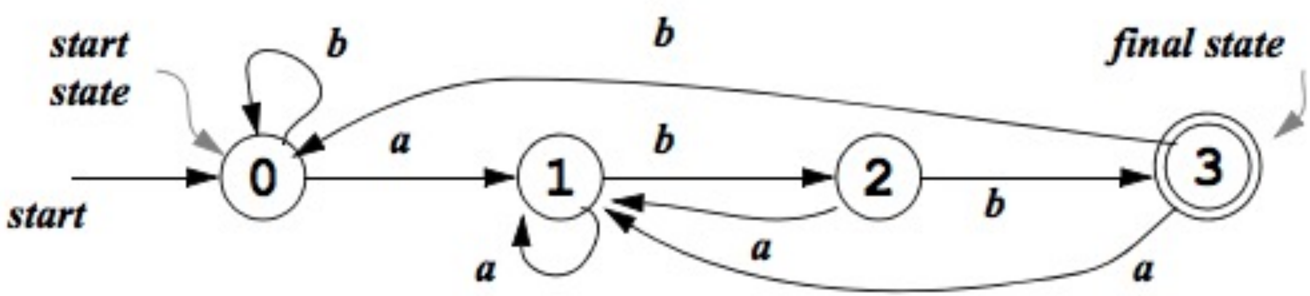


  - A set of transitions δ: $S_a \xrightarrow{\text{input}} S_b$



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

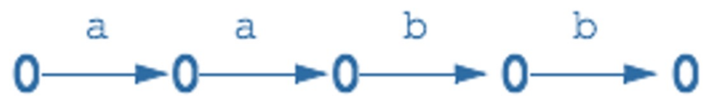


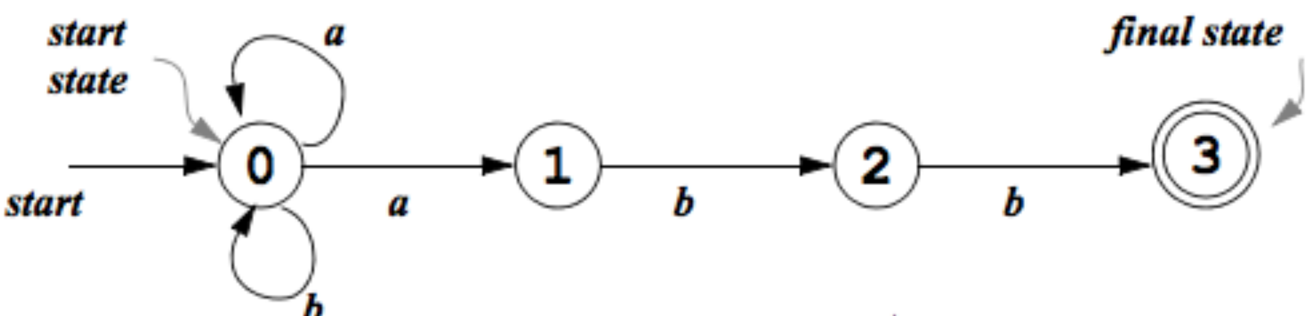
A DFA accepts (a|b)*abb

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:

  - Unsuccessful sequence:

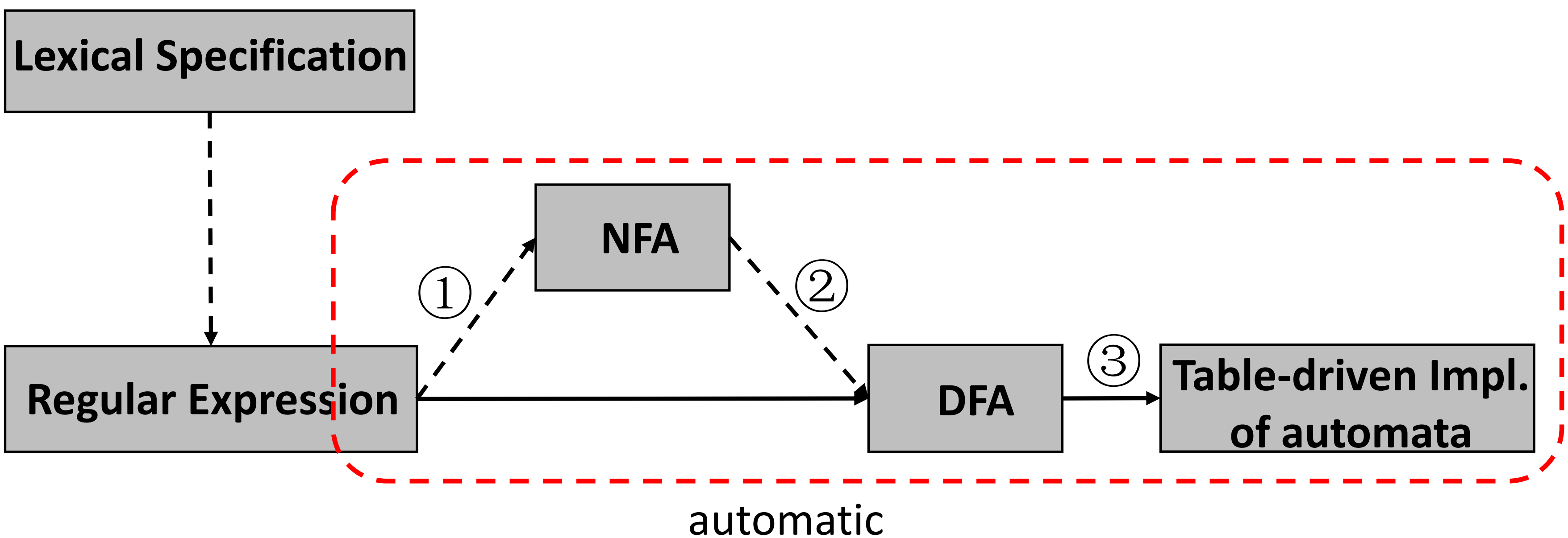




An NFA accepts (a|b)*abb

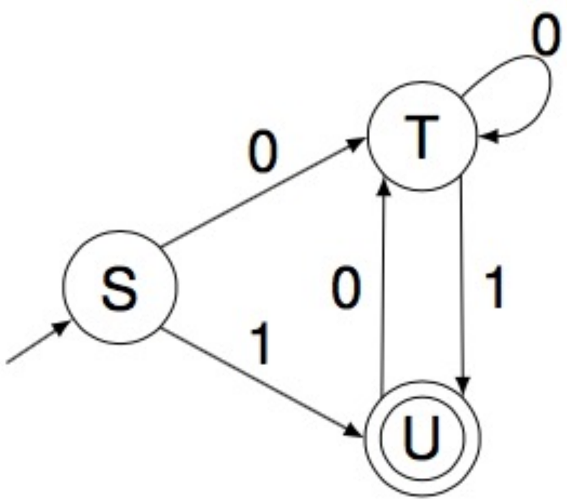
Conversion Flow[转换流程]

- 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

14

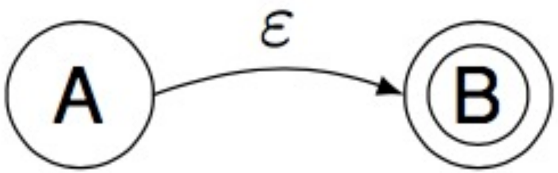
More on Table

- 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

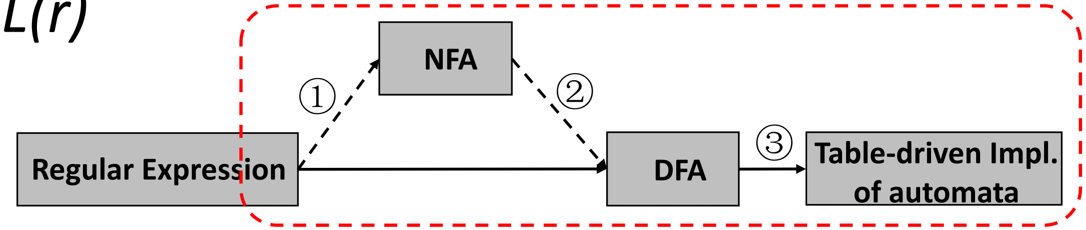
RE → NFA

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



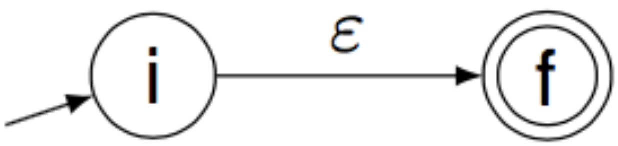
- **M-Y-T algorithm** to convert any RE to an NFA that defines the same language
  - Input: RE *r* over alphabet ∑
  - Output: NFA accepting *L(r)*

McNaughton-Yamada-Thompson

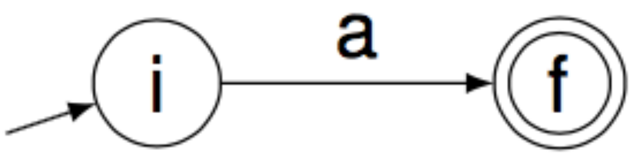


RE → NFA (cont.)

- Step 1: processing atomic REs
  - ε expression[空]

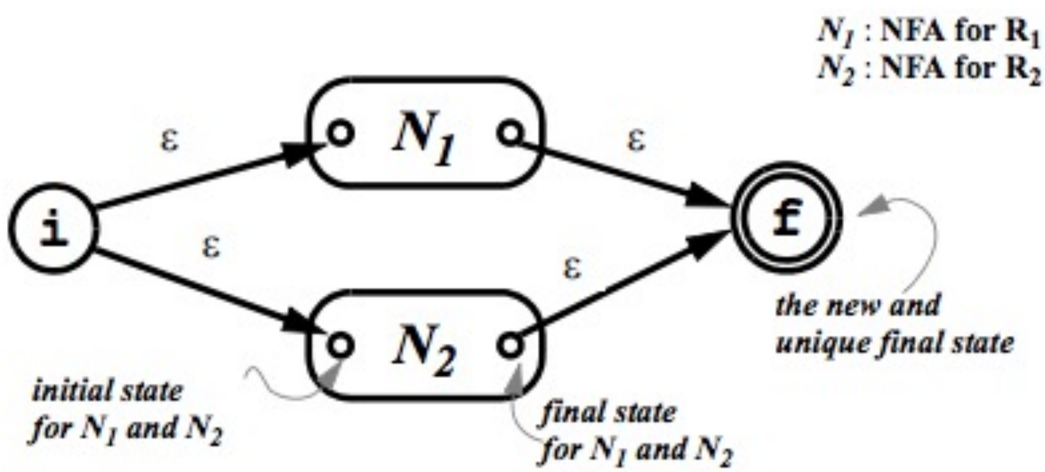


    - *i* is a new state, the start state of NFA
    - *f* is another new state, the accepting state of NFA
  - Single character RE *a*[单字符]

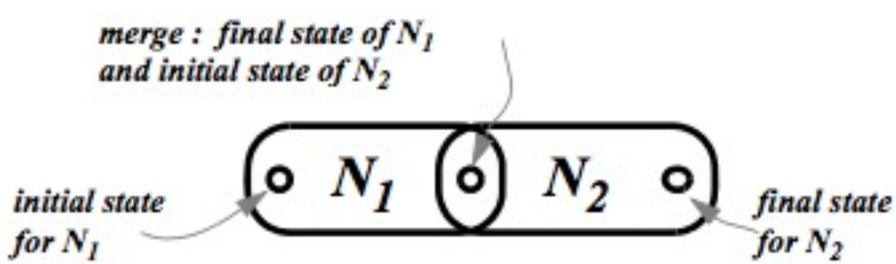


RE → NFA (cont.)

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

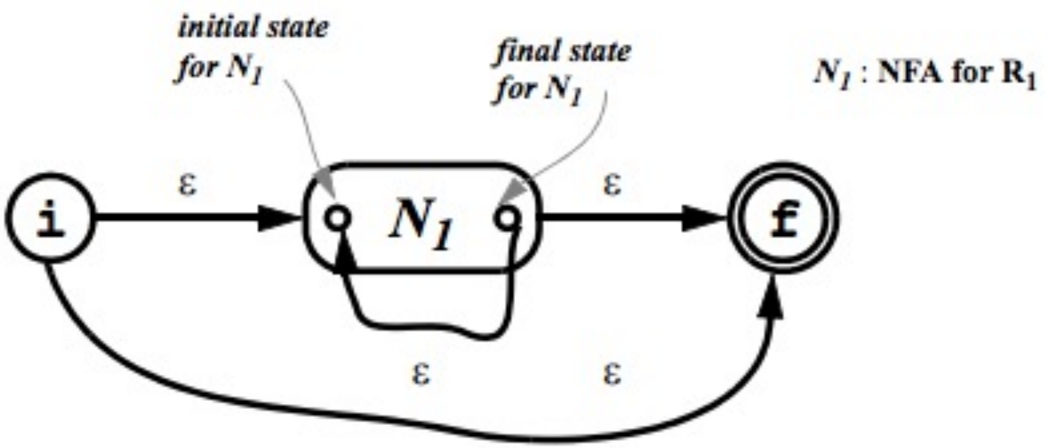


  - $R = R_1R_2$



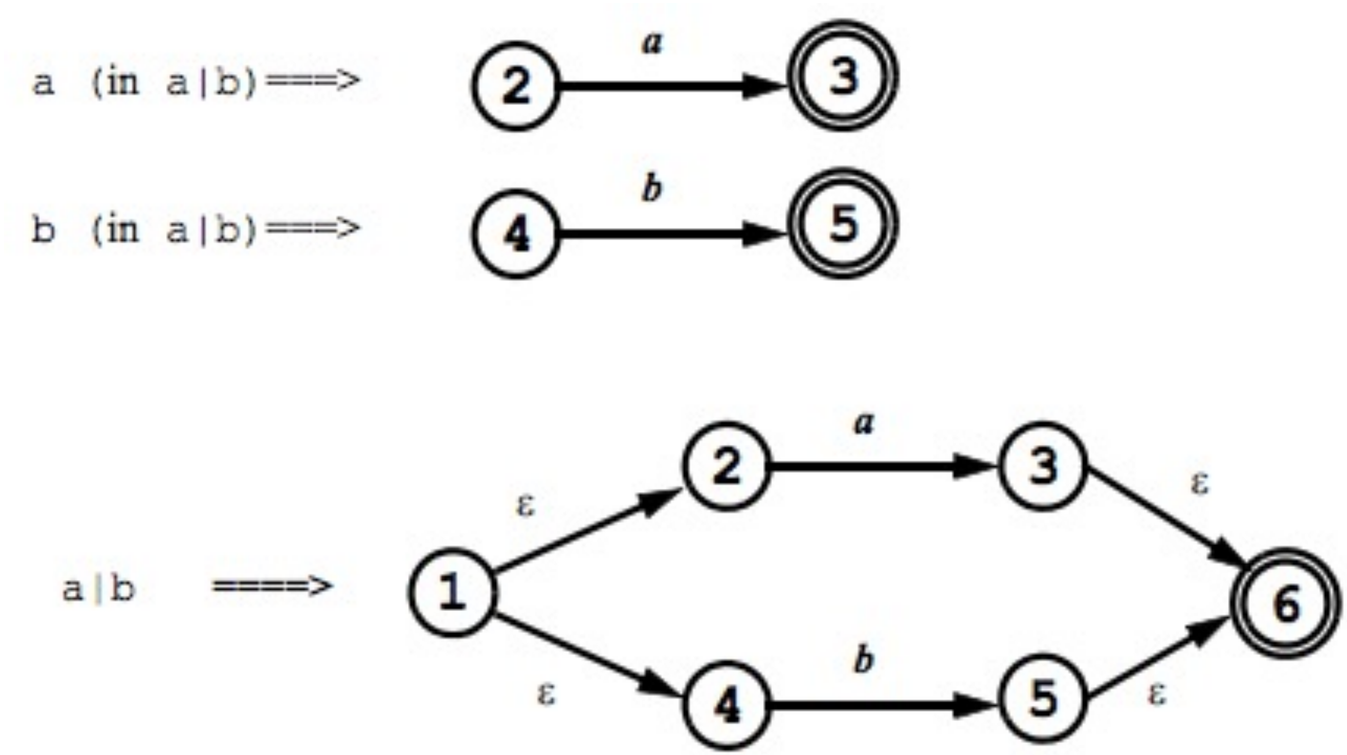
RE → NFA (cont.)

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



Example

- Convert “(a|b)*abb” to NFA



Example (cont.)

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

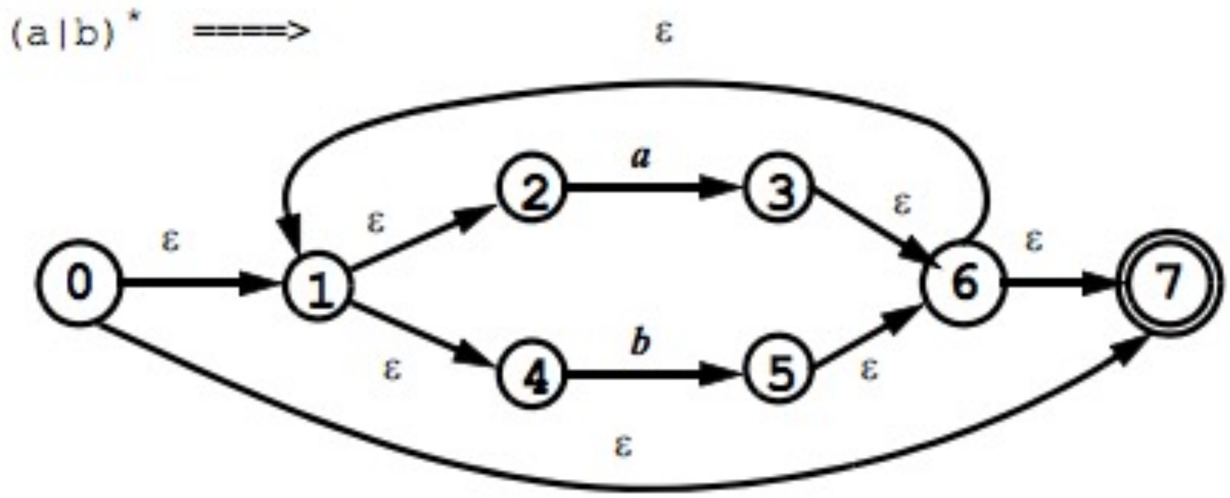
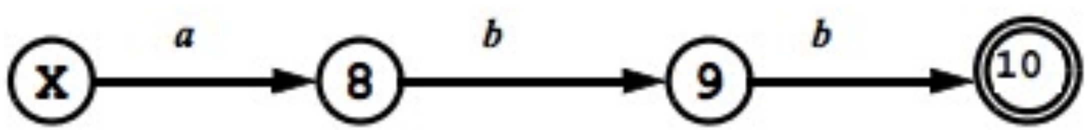
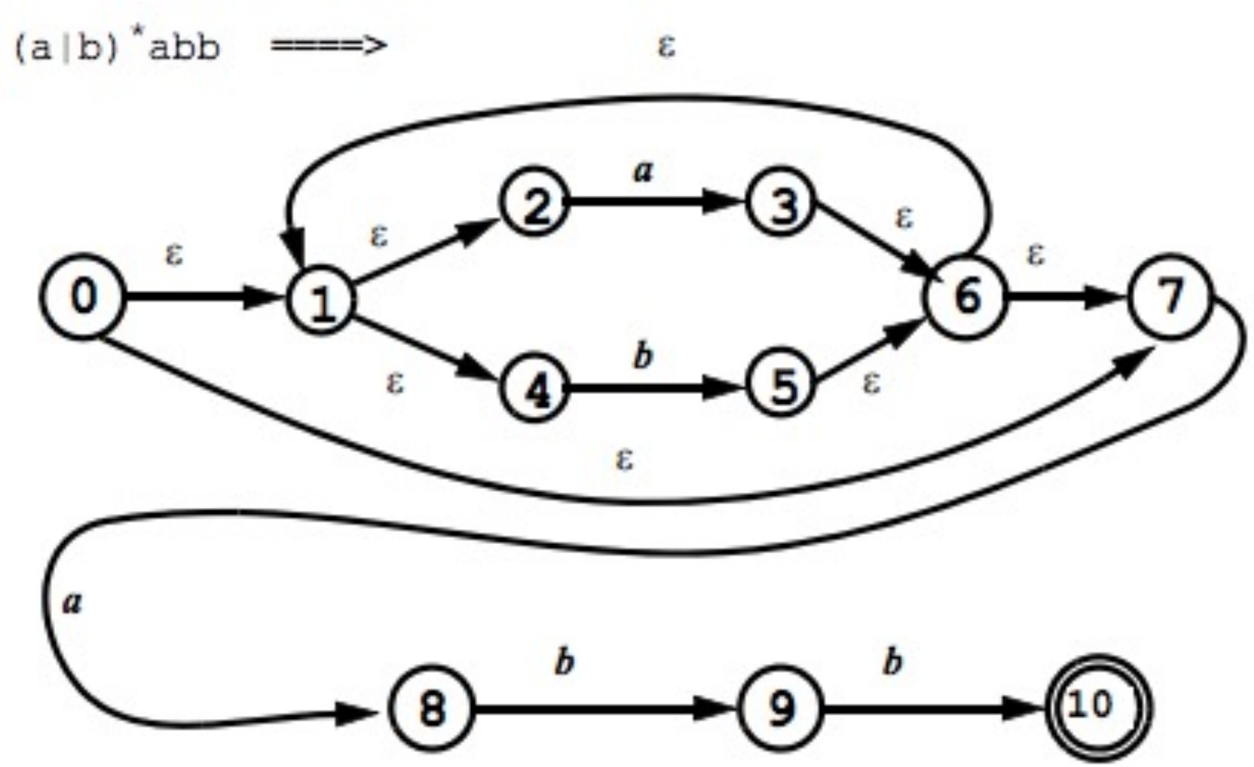


abb ====> (several steps are omitted)



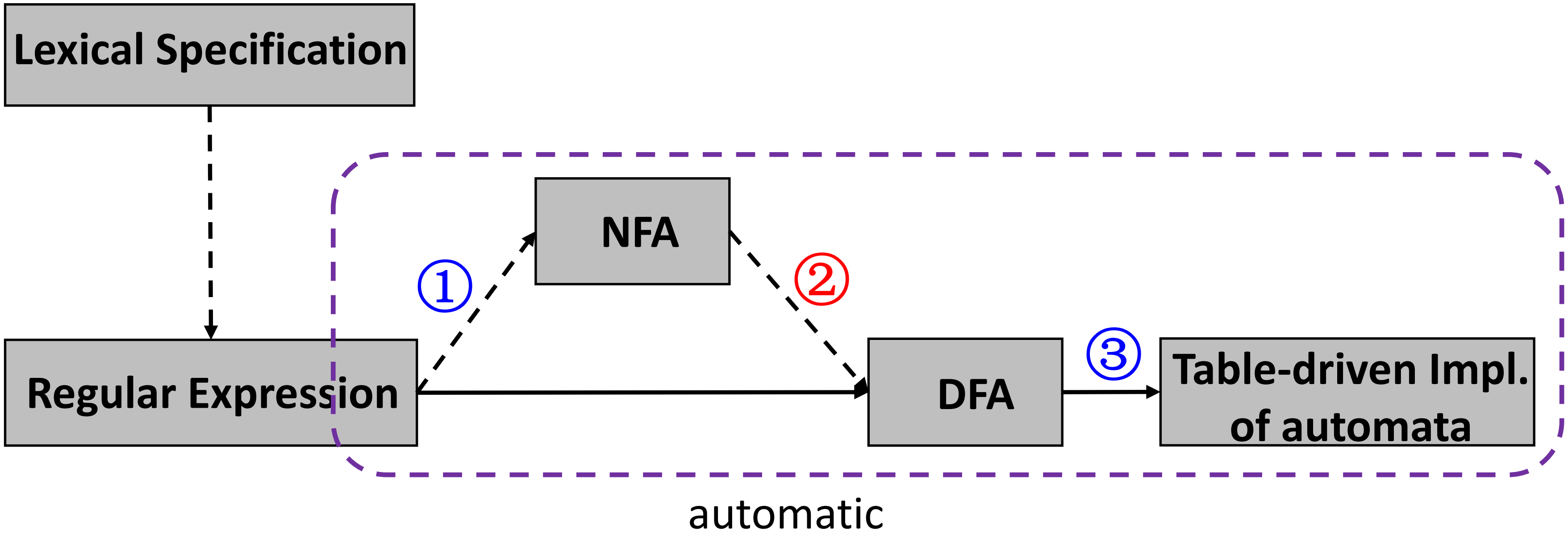
Example (cont.)

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



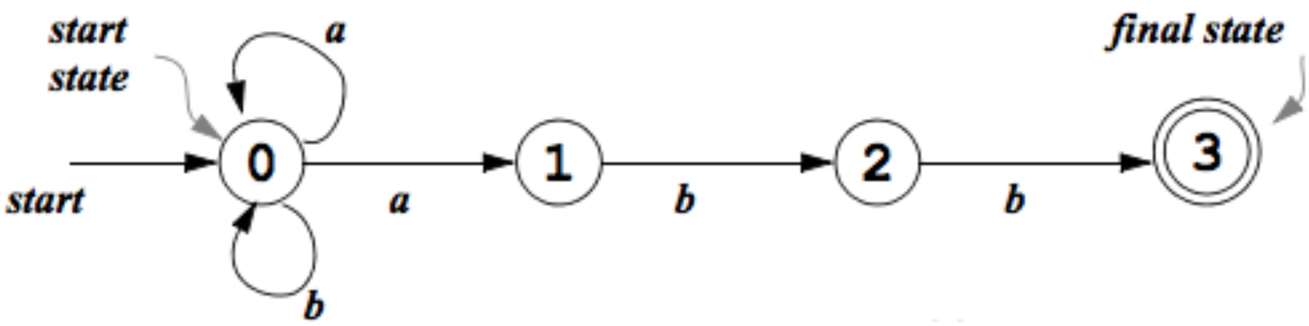
The Conversion Flow

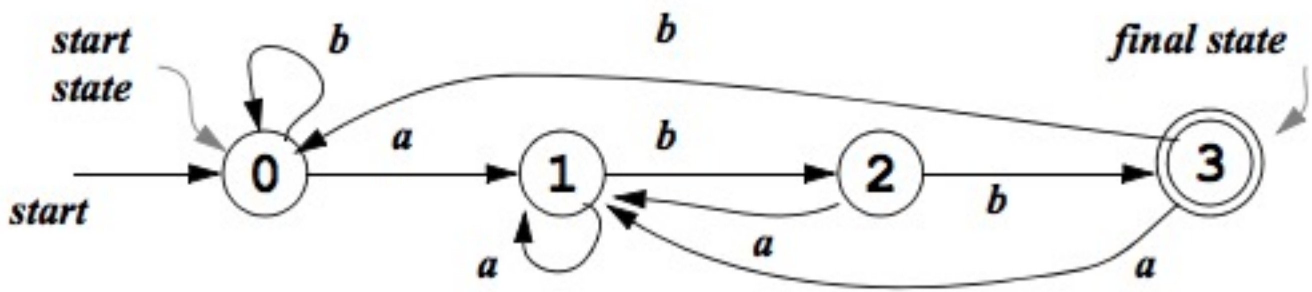
- Outline: RE → NFA → DFA → Table-driven Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs
  - ② Converting NFAs to DFAs



NFA → DFA: Same[等价]

- NFA and DFA are equivalent





To show this we must prove every DFA can be converted into an NFA which accepts the same language, and vice-versa

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)

  Any DFA can be easily changed into NFA
- 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 outweighs its extra memory consumption

NFA → DFA: Idea

- Algorithm to convert[转换算法]
  - Input: an NFA *N*
  - Output: a DFA *D* accepting the same language as *N*

- **Subset construction**[子集构建]
  - Each state of the constructed DFA corresponds to a set of NFA states
    - Hence, the name 'subset construction'
  - After reading input $a_1a_2...a_n$, the DFA is in that state which corresponds to the set of states that the NFA can reach, from its start state, following paths labeled $a_1a_2...a_n$

NFA → DFA: Steps

- The **initial state** of the DFA is the set of all states the NFA can be in without reading any input
- For any state $\{q_i, q_j, \ldots, q_k\}$ of the DFA and any input *a*, the **next state** of the DFA is the set of all states of the NFA that can result as next states if the NFA is in any of the states $q_i, q_j, \ldots, q_k$ when it reads *a*
  - This includes states that can be reached by reading *a* followed by any number of ε-transitions
  - Use this rule to keep adding new states and transitions until it is no longer possible to do so
- The **accepting states** of the DFA are those states that contain an accepting state of the NFA.

https://web.cse.msstate.edu/~hansen/classes/3813spring05/slides/04SubsetConstruction.pdf