中山大學 SUN YAT-SEN UNIVERSITY 计算机学院（软件学院） SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第3讲：词法分析(3)

张献伟

xianweiz.github.io

DCS290, 3/7/2024

中山大學 SUN YAT-SEN UNIVERSITY

Review Questions

- Q1: RE of binary numbers that are multipliers of 2?

  (0|1)*0

- Q2: meaning of (a|b)*bb(a|b)*?

  Strings of a's and b's with consecutive b's

- Q3: usage of RE and FA in lexical analysis?

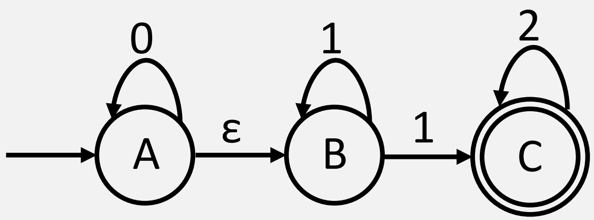
  RE: specify the token class; FA: implement the token recognizer

- Q4: general workflow from RE to implementation?

  RE → NFA → DFA → Table

- Q5: the graph describes NFA or DFA? Why?

  NFA. A: ε-transition, B: 1-transition



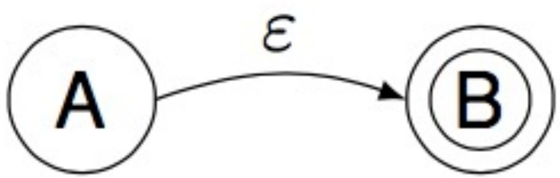
More on Table

- 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

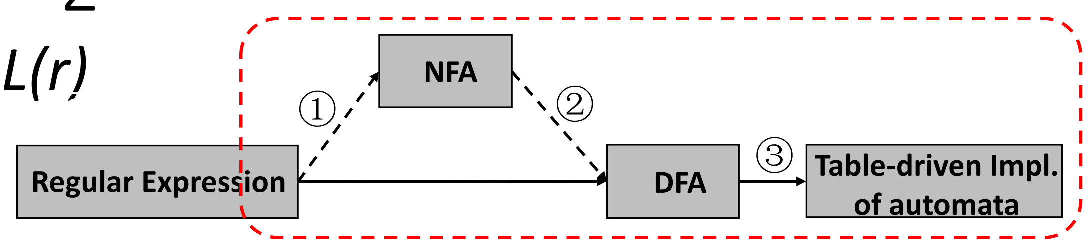
RE → NFA

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



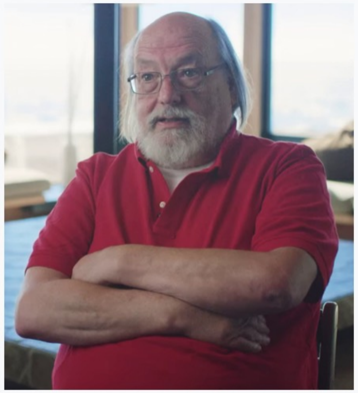
- **M-Y-T algorithm** (Thompson's construction) to convert any RE to an NFA that defines the same language[正则表达式转换到自动机]
  - Input: RE *r* over alphabet ∑
  - Output: NFA accepting *L(r)*

McNaughton-Yamada-Thompson



Thompson

Ken Thompson



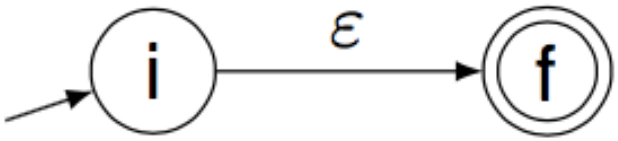
Kenneth Lane Thompson (born February 4, 1943) is an American pioneer of computer science & Computer Chess Development. Thompson worked at Bell Labs for most of his career where he designed and implemented the original Unix operating system. He also invented the B programming language, the direct predecessor to the C programming language, and was one of the creators and early developers of the Plan 9 operating system. Since 2006, Thompson has worked at Google, where he co-developed the Go programming language.

Other notable contributions included his work on regular expressions and early computer text editors QED and ed, the definition of the UTF-8 encoding, and his work on computer chess that included the creation of endgame tablebases and the chess machine Belle. He won the Turing Award in 1983 with his long-term colleague Dennis Ritchie.

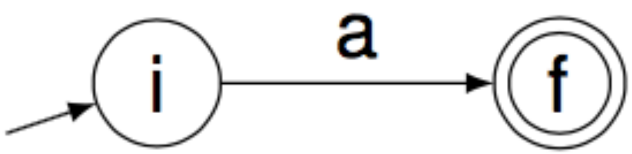
In the 1960s, Thompson also began work on regular expressions. Thompson had developed the CTSS version of the editor QED, which included regular expressions for searching text. QED and Thompson's later editor ed (the standard text editor on Unix) contributed greatly to the eventual popularity of regular expressions, and regular expressions became pervasive in Unix text processing programs. Almost all programs that work with regular expressions today use some variant of Thompson's notation. He also invented Thompson's construction algorithm used for converting regular expressions into nondeterministic finite automata in order to make expression matching faster.[12]

RE → NFA (cont.)

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

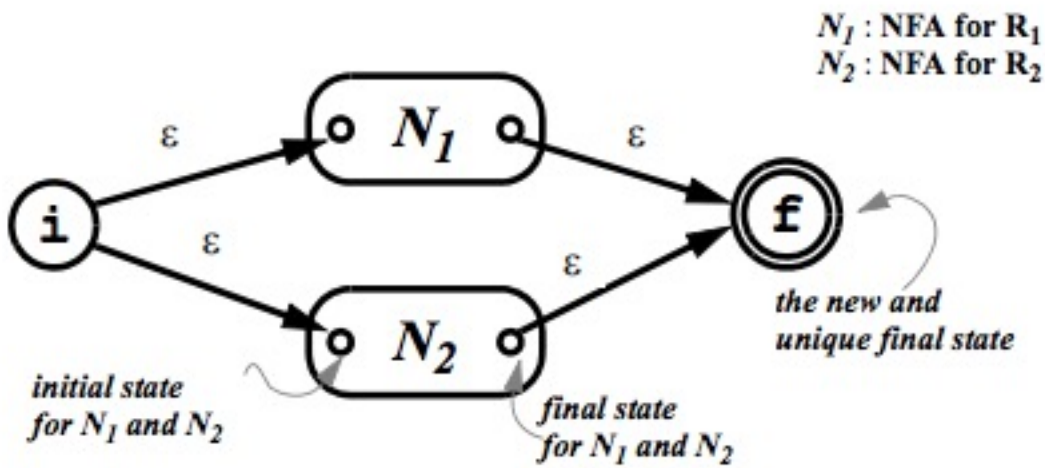


    - *i* is a new state, the start state of NFA
    - *f* is another new state, the accepting state of NFA
  - Single character RE *a*[单字符]

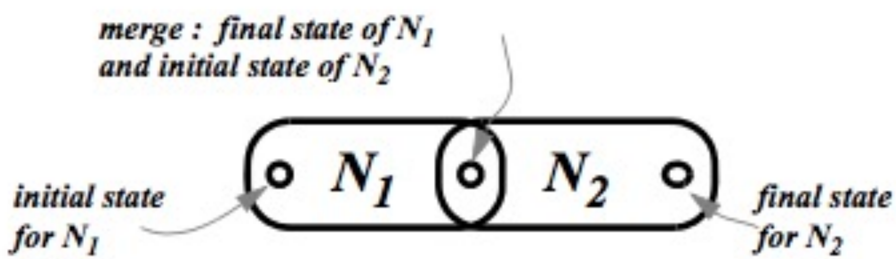


RE → NFA (cont.)

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

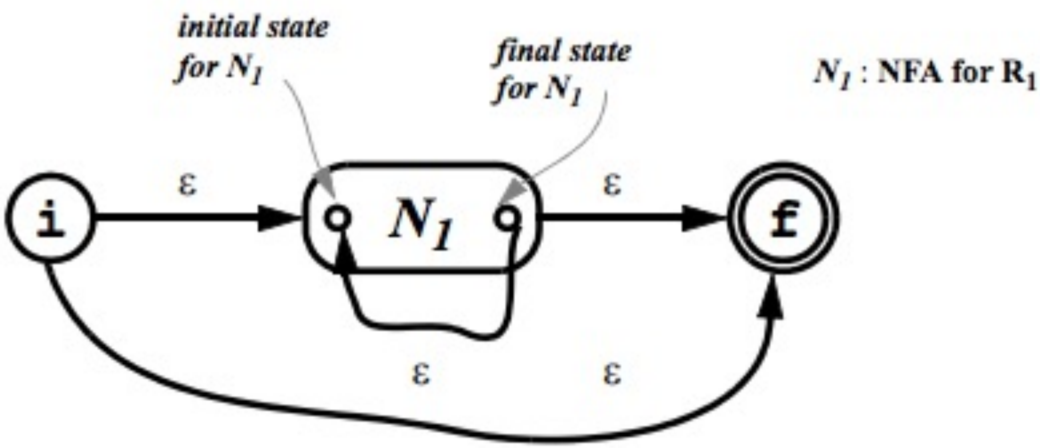


  - $R = R_1R_2$



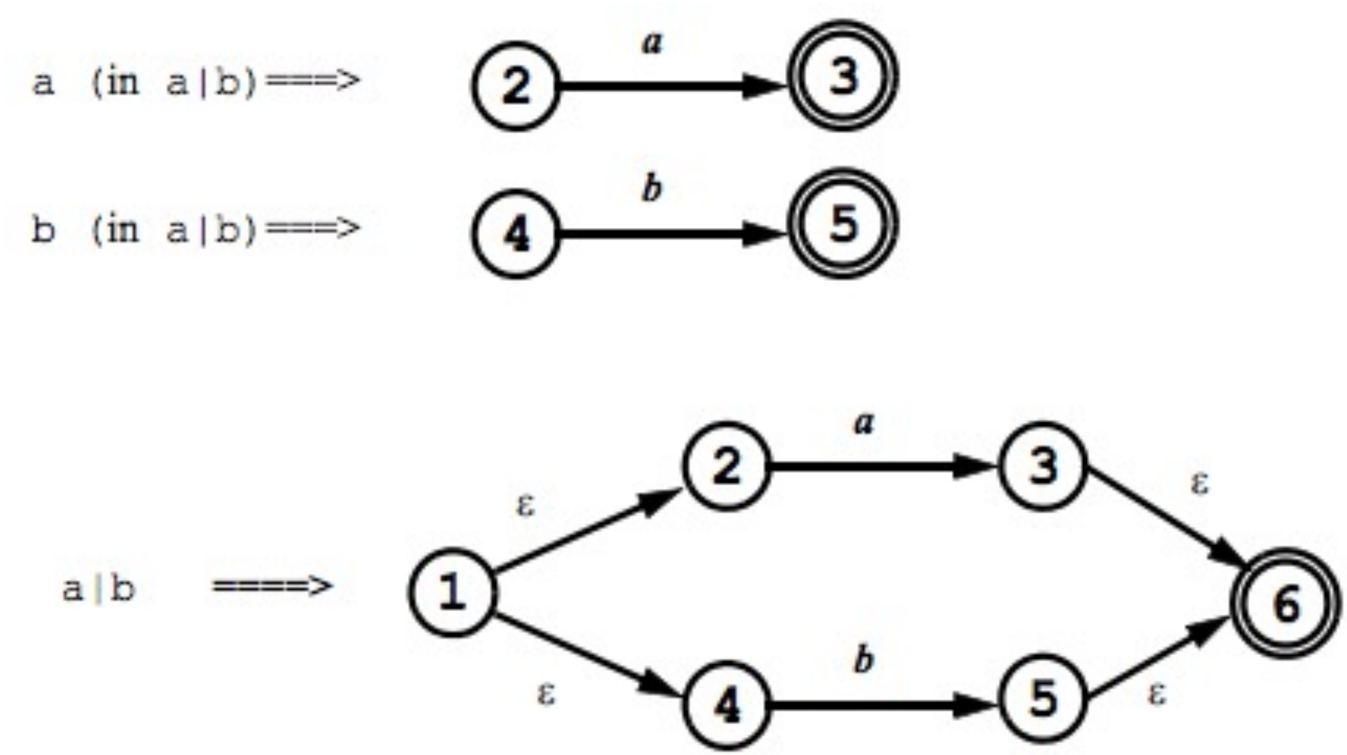
RE → NFA (cont.)

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



Example

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

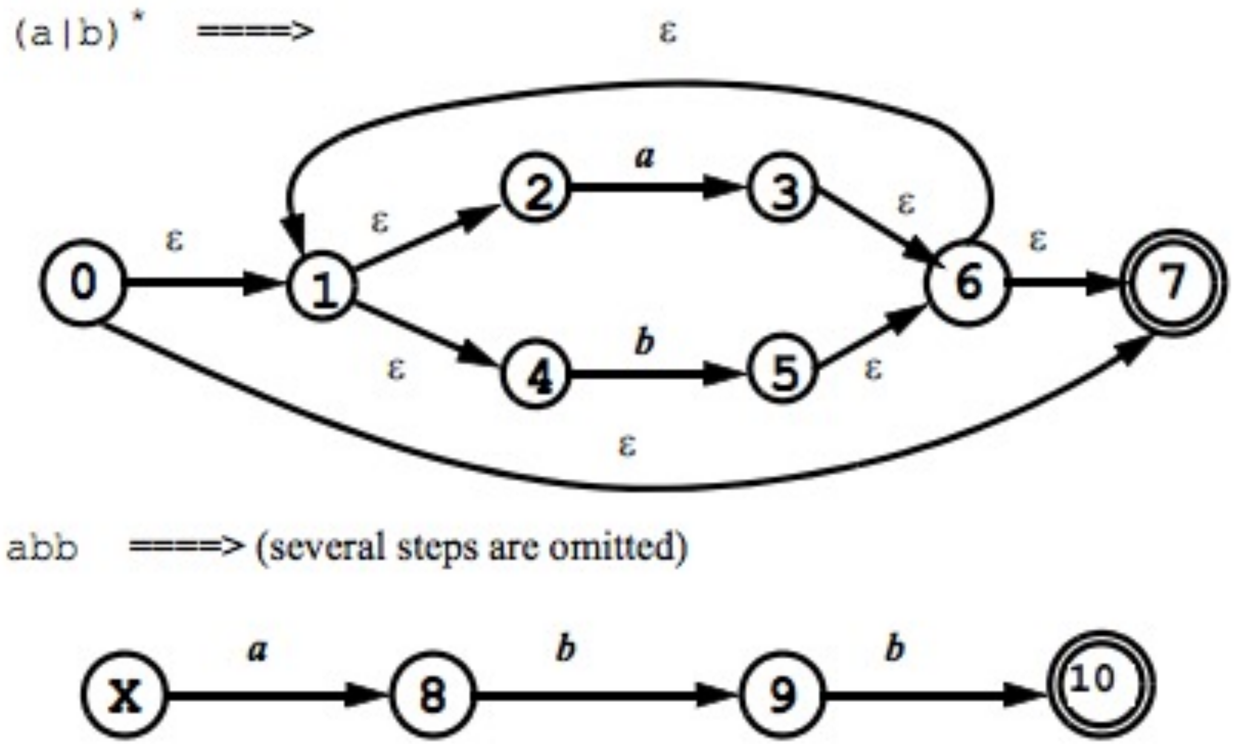


中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

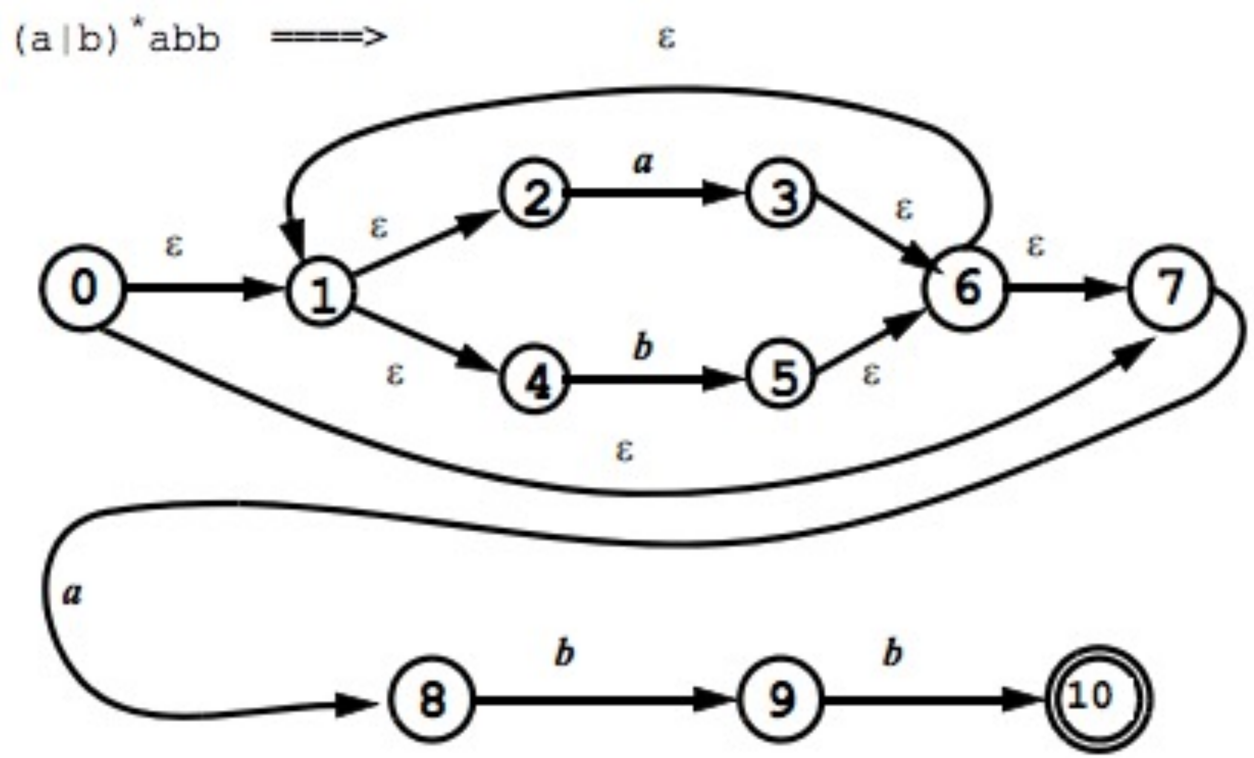
Example (cont.)

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



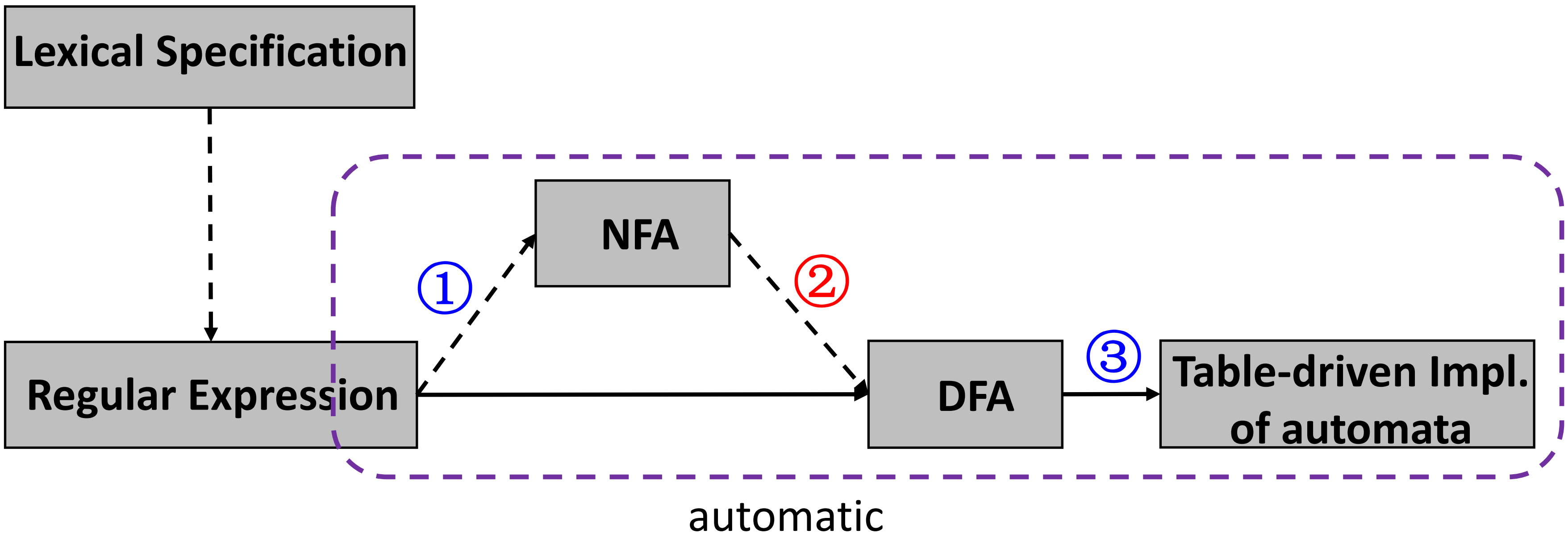
Example (cont.)

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



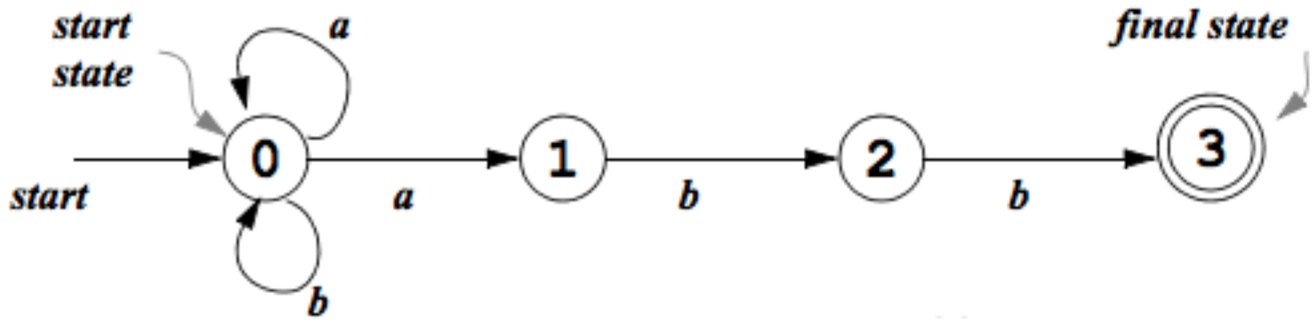
The Conversion Flow[转换流程]

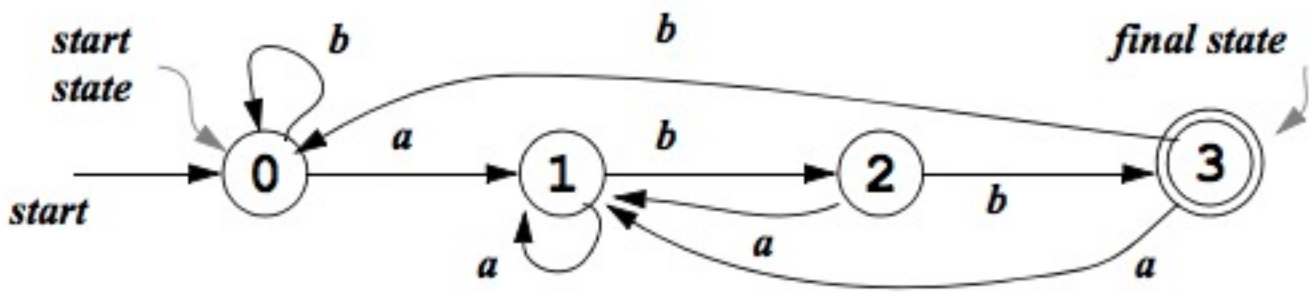
- Outline: RE → NFA → DFA → Table-drive Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs
  - ② Converting NFAs to DFAs



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 (e.g., add ε moves)
- 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 <u>speed</u> of DFA far outweighs its extra <u>memory</u> 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[一个DFA状态对应多个NFA状态]
    - 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[初始状态]
- 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*[下一状态]
  - 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[接收状态]

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

NFA → DFA: Algorithm

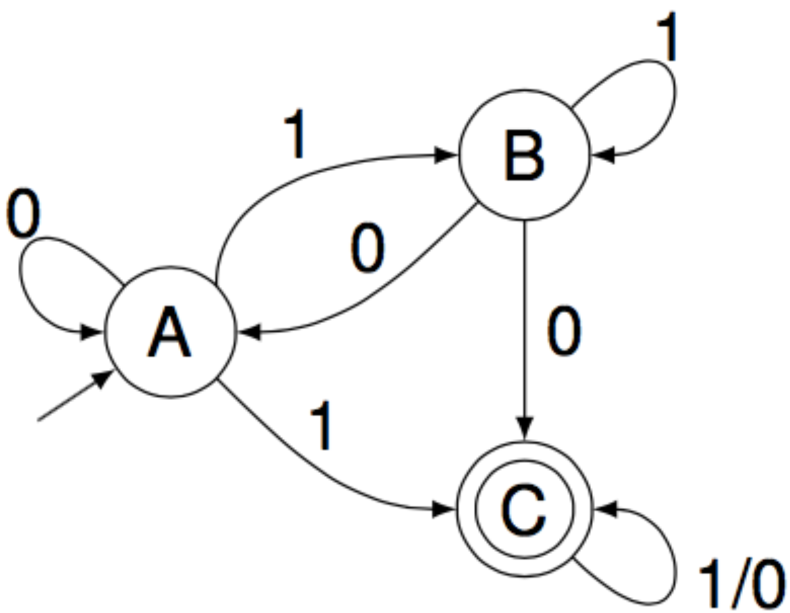
```
Initially, ε-closure(s_0) is the only state in Dstates and it is unmarked
while there is an unmarked state T in Dstates do
    mark T
    for each input symbol a ∈ Σ do
        U := ε-closure(move(T,a))
        if U is not in Dstates then
            add U as an unmarked state to Dstates
        end if
        Dtran[T,a] := U
    end do
end do
```

- Operations on NFA states:
  - ε-closure(s): set of NFA states reachable from NFA state ***s*** on ε-transitions **alone**
  - ε-closure(T): set of NFA states reachable from some NFA state ***s*** in set ***T*** on ε-transitions **alone**; = $\cup_{s \text{ in } T}$ε-closure(s)
  - move(T, a): set of NFA states to which there is a transition on input symbol ***a*** from some state ***s*** in ***T***

NFA → DFA: Example

- Start by constructing ε-closure of the start state[初始状态]
  - ε-closure(A) = A
- Keep getting ε-closure(*move(T, a)*)[更多状态]

  T: A, *a*: 0/1
- Stop, when there are no more new states



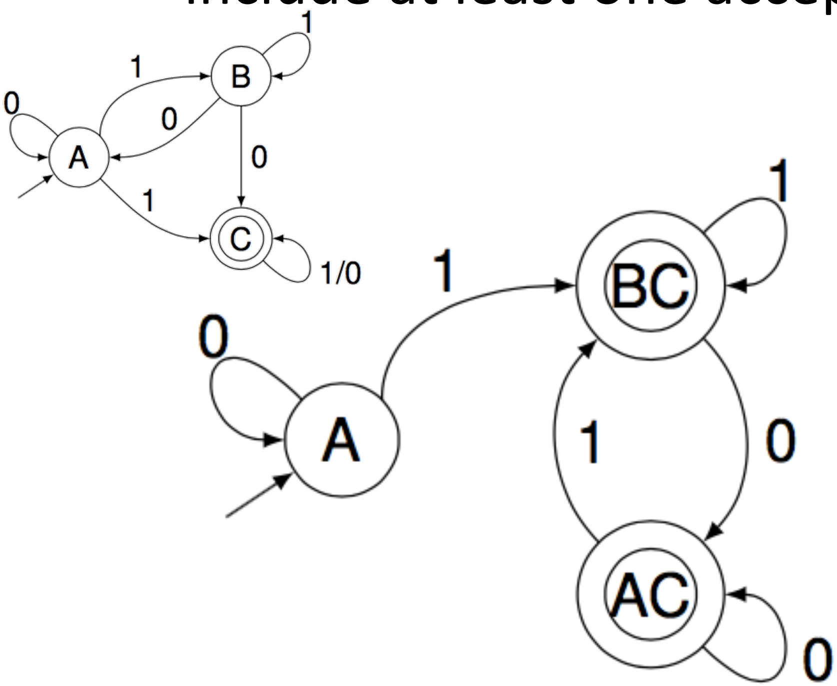
alphabet →

state ↓

	0	1
A	**A**	**BC**
BC	**AC**	**BC**
AC	**AC**	**BC**

NFA → DFA: Example (cont.)

- Mark the final states of the DFA[终止状态]
  - The accepting states of *D* are all those sets of *N*'s states that include at least one accepting state of *N*



alphabet →

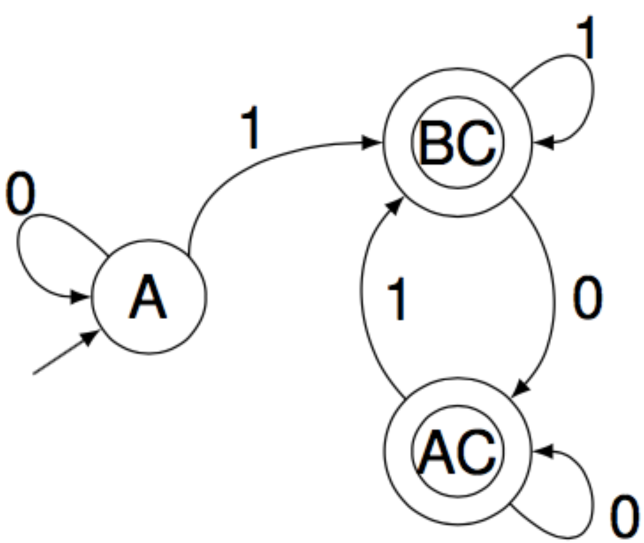
state ↓

	0	1
A	**A**	**BC**
BC	**AC**	**BC**
AC	**AC**	**BC**

- Is the DFA minimal?
  - As few states as possible

NFA → DFA: Minimization[最小化]

- Any DFA can be converted to its minimum-state equivalent DFA
  - Discover sets of equivalent states[存在等价/重复状态]
  - Represent each such set with just one state
- Two states are equivalent if and only if:
  - $\forall \alpha \in \Sigma$, transitions on α lead to equivalent states
  - α-transitions to distinct sets $\Rightarrow$ states must be in distinct sets



Initial: {A}, {BC, AC}

Initial sets:
{non-accepting states}, {accepting states}

For {BC, AC}

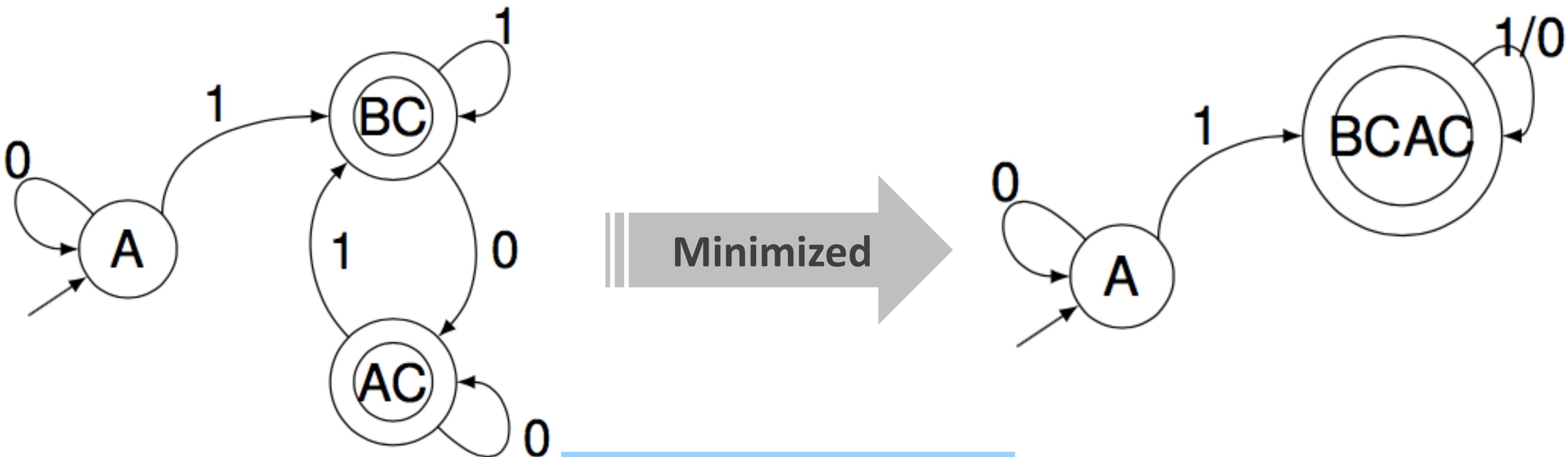
- BC on ‘0’ → AC, AC on ‘0’ → AC
- BC on ‘1’ → BC, AC on ‘1’ → BC
- No way to distinguish BC from AC on any string starting with ‘0’ or ‘1’

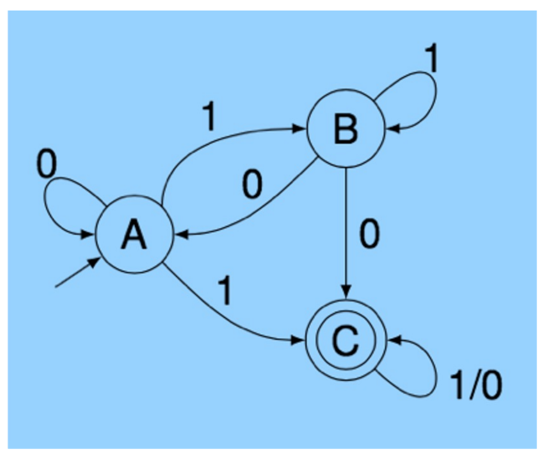
Final: {A}, {BCAC}

https://people.cs.umass.edu/~moss/610-slides/06.pdf

NFA → DFA: Minimization (cont.)

- States *BC* and *AC* do not need differentiation
  - Should be merged into one





	0	1
A	**A**	**BC**
BC	**AC**	**BC**
AC	**AC**	**BC**

Minimization Algorithm

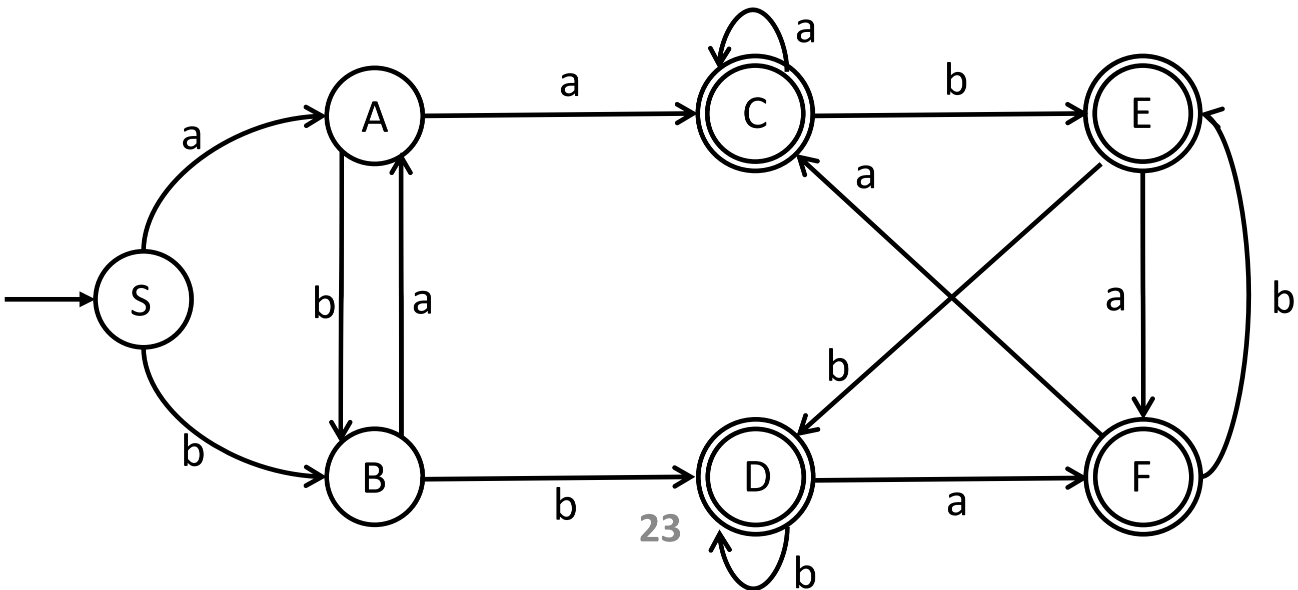
- The algorithm
  - Partitioning the states of a DFA into groups of states that cannot be distinguished (i.e., equivalent)
  - Each groups of states is then merged into a single state of the min-state DFA

```
P <- {F}, {S - F}
while (P is still changing)
  T <- { }
   for each state s ∈ P
      for each α ∈ Σ
         partition s by α into s1 & s2
         T <- T ∪ s1 ∪ s2
   if T ≠ P then
      P <- T
```

- For a DFA （Σ, S, n, F, δ）
  - The initial partition P_0, has two sets and $\{S - F\}$
  - Splitting a set (i.e., partitioning a set by input symbol α)
    - Assume q_a and $q_b \in$ ***s***, and $\delta(q_a, \alpha) = q_x$ and $\delta(q_b, \alpha) = q_y$
    - If q_x and q_y are not in the same set, then ***s*** must be split (i.e., α splits ***s***)
    - One state in the final DFA cannot have two transitions on α

Example

- P0: s_1 = {S, A, B}, s_2 = {C, D, E, F}
- For s_1, further splits into {S}, {A}, {B}
  - a: S --> A ∈ s_1, A --> C ∈ s_2, B --> A ∈ s_1 ⟹ a distincts s_1 => {S, B}, {A}
  - b: S --> B ∈ s_1, A --> B ∈ s_1, B --> D ∈ s_2 ⟹ b distincts s_1 => {S}, {B}, {A}
- For s_2, all states are equivalent
  - a: C --> C ∈ s_2, D --> F ∈ s_2, E --> F ∈ s_2 , F --> C ∈ s_2 ⟹ a doesn't
  - b: C --> E ∈ s_2, D --> D ∈ s_2, E --> D ∈ s_2 , F --> E ∈ s_2 ⟹ b doesn't

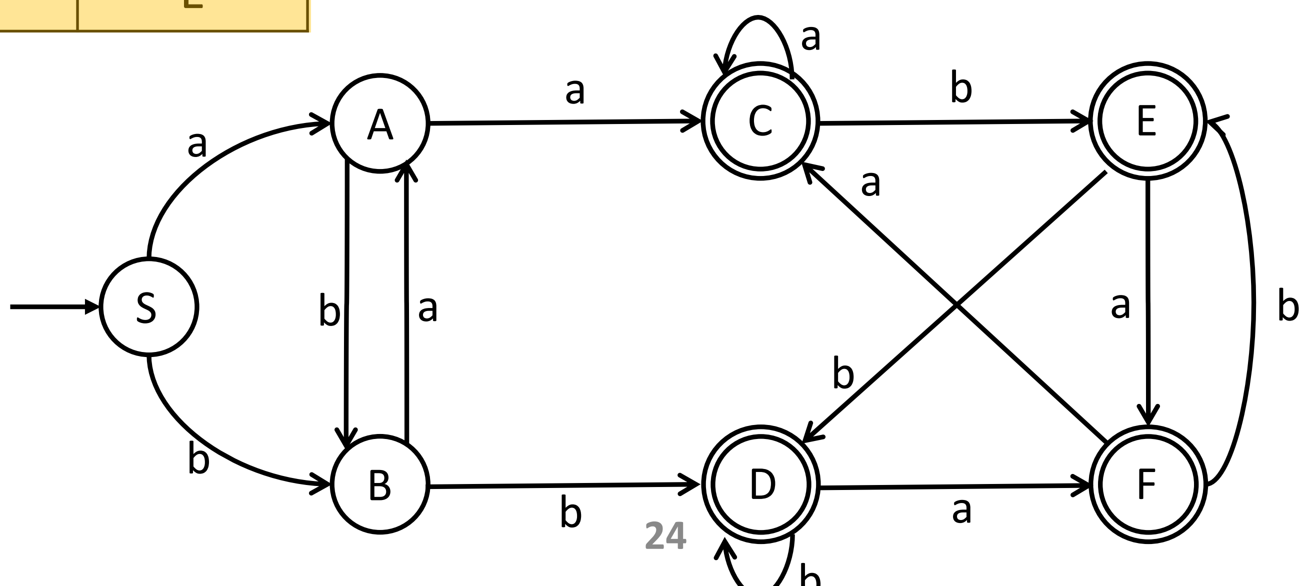


Example (cont.)

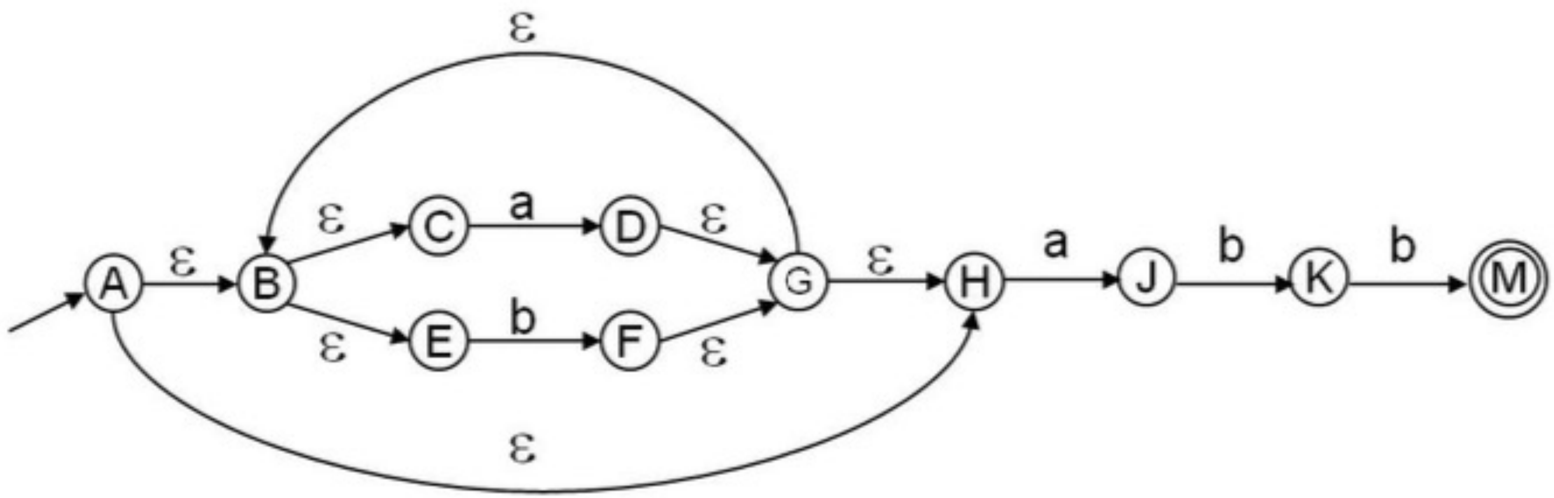
	a	b
S	A	B
A	C	B
B	A	D
C	C	E
D	F	D
E	F	D
F	C	E

	a	b
S	A	B
A	C	B
B	A	D
CF	C	E
DE	F	D

	a	b
S	A	B
A	C	B
B	A	D
CFDE	CF	DE



NFA → DFA: More Example

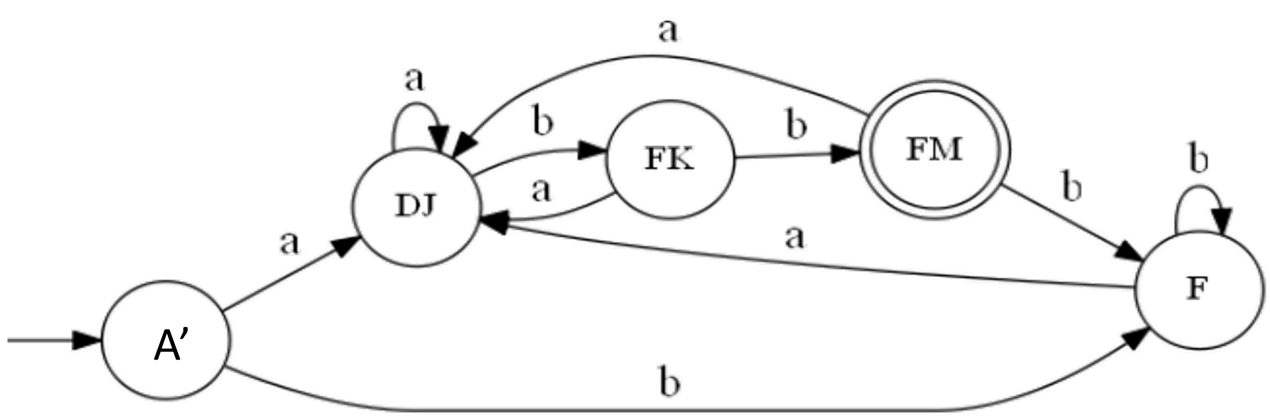


- Start state of the equivalent DFA
  - ε-closure(A) = {A, B, C, E, H} = A’
- ε-closure(move(A’, a)) = ε-closure({D, J}) = {B, C, D, E, H, G, J} = B’
- ε-closure(move(A’, b)) = ε-closure({F}) = {B, C, E , F, G, H} = C’
- … …

NFA → DFA: More Example (cont.)

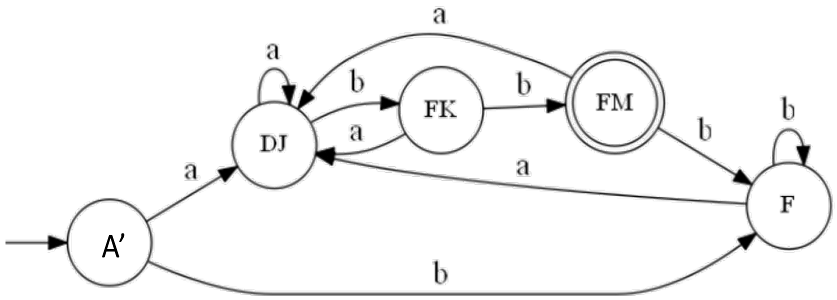
	a	b
A’	DJ	F
DJ	DJ	FK
F	DJ	F
FK	DJ	FM
FM	DJ	F

- Is the DFA minimal?
  - States A’ and F should be merged
- Should we merge states A’ and FM?
  - NO. A’ and FM are in different sets from the very beginning (FM is accepting, A’ is not).



NFA → DFA: More Example (cont.)

- P0: s_1 = {A', DJ, FK, F}, s_2 = {FM}
- For s_1, further splits into {A', DJ, F}, {FK}
  - a: A'--> DJ ∈ s_1, DJ --> DJ ∈ s_1, FK --> DJ ∈ s_1, F --> DJ ∈ s_1 ⟹ a doesn't distinct
  - b: A' --> F ∈ s_1, DJ --> FK ∈ s_1, FK --> FM ∈ s_2, F --> F ∈ s_1 ⟹ b distincts s_1 => s_{11}={A', DJ, F}, s_{12}={FK}
- For s_{11}, further splits into {A', DJ, F}, {FK}
  - a: A'--> DJ ∈ s_{11}, DJ --> DJ ∈ s_{11}, F --> DJ ∈ s_{11} ⟹ a doesn't distinct
  - b: A'--> F ∈ s_{11}, DJ --> FK ∈ s_{12}, F --> DJ ∈ s_{11} ⟹ b distincts s_{11} => s_{111}={A', F}, s_{112}={DJ}
- For s_{111}, impossible to further split
- Final states: S_{111} = {A', F}, S_{112} = {DJ}, S_{12} = {FK}, S_2 = {FM}

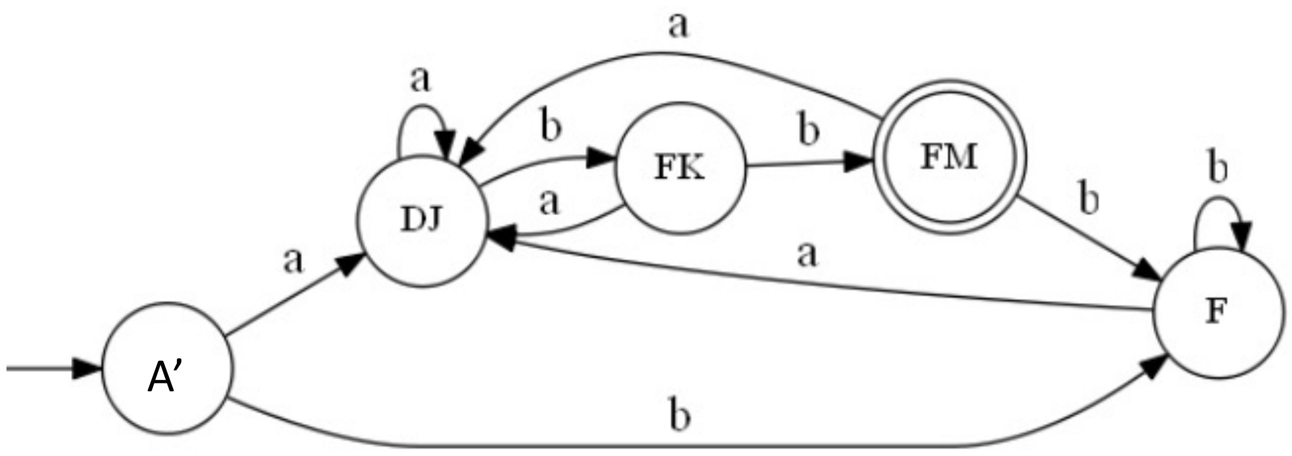


	a	b
A'	DJ	F
DJ	DJ	FK
F	DJ	F
FK	DJ	FM
FM	DJ	F

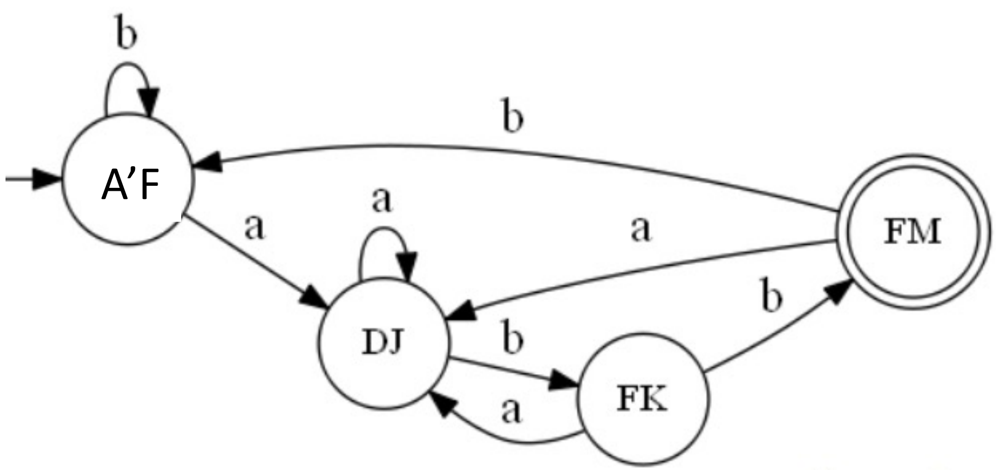
SUN YAT-SEN UNIVERSITY

NFA → DFA: More Example (cont.)

- Original DFA: before merging A’ and F



- Minimized DFA: Do you see the original RE (a|b)*abb



NFA → DFA: Space Complexity[空间复杂度]

- NFA may be in many states at any time

- How many different possible states in DFA?
  - If there are *N* states in NFA, the DFA must be in some subset of those *N* states
  - How many non-empty subsets are there?
    - 2^N-1

- The resulting DFA has $O(2^N)$ space complexity, where *N* is number of original states in NFA
  - For real languages, the NFA and DFA have about same number of states

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ