

# 词法分析

## (3. 自动机理论与词法分析器生成器)

魏恒峰

hfwei@nju.edu.cn

2021 年 11 月 12 日 (周五)

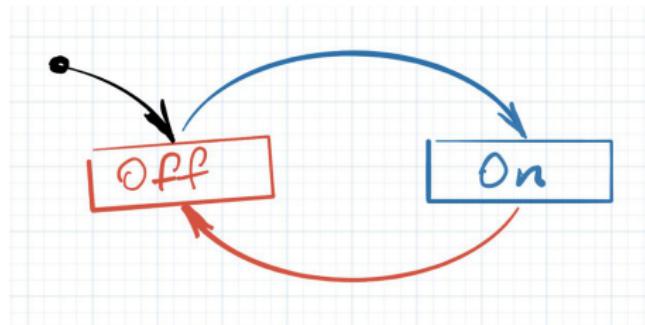




## 自动化词法分析器

# 自动机

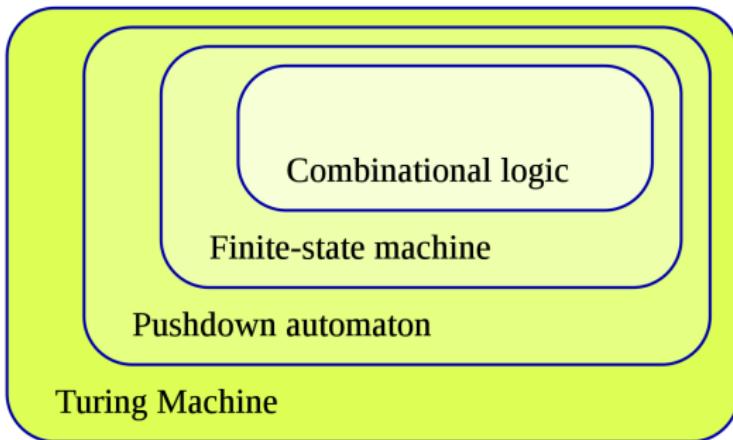
(Automaton; Automata)



“开关”自动机

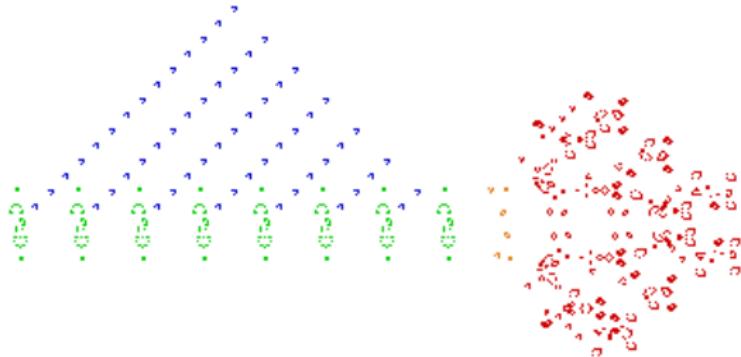
两大要素：状态集  $S$  以及状态转移函数  $\delta$

## Automata theory



根据**表达/计算能力**的强弱，自动机可以分为不同层次。

# 元胞自动机 (Cellular Automaton)



“播种机”、“滑翔机枪”与“滑翔机”

John Horton Conway  
(1937 ~ 2020)

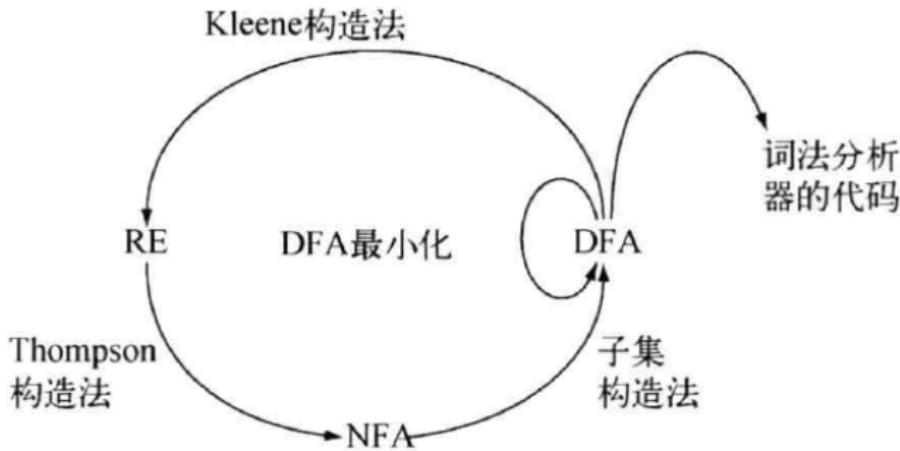
[https://en.wikipedia.org/wiki/File:  
Conways\\_game\\_of\\_life\\_breeder\\_animation.gif](https://en.wikipedia.org/wiki/File:Conways_game_of_life_breeder_animation.gif)



“生命游戏”(Game of Life) 史诗级巨作

<https://www.youtube.com/watch?v=C2vgICfQawE&t=270s>

# 目标: 正则表达式 RE $\Rightarrow$ 词法分析器



终点固然令人向往, 这一路上的风景更是美不胜收

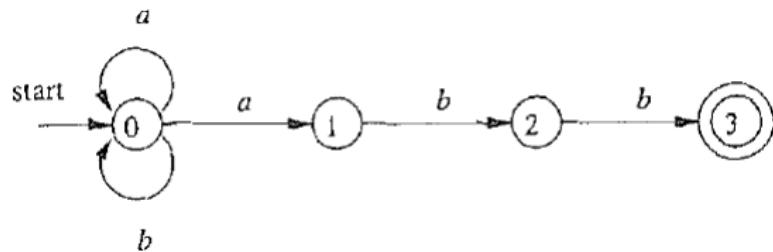
## Definition (NFA (Nondeterministic Finite Automaton))

非确定性有穷自动机  $\mathcal{A}$  是一个五元组  $\mathcal{A} = (\Sigma, S, s_0, \delta, F)$ :

- (1) 字母表  $\Sigma$  ( $\epsilon \notin \Sigma$ )
- (2) **有穷**的状态集合  $S$
- (3) **唯一**的初始状态  $s_0 \in S$
- (4) 状态转移**函数**  $\delta$

$$\delta : S \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^S$$

- (5) 接受状态集合  $F \subseteq S$



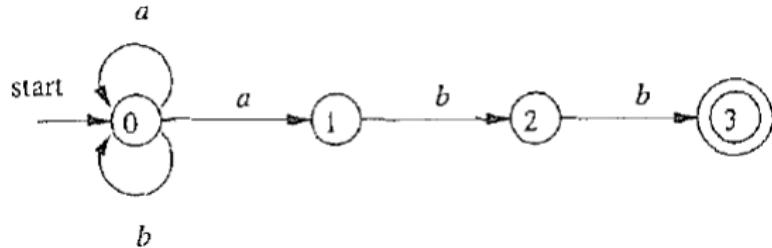
## Definition (NFA (Nondeterministic Finite Automaton))

非确定性有穷自动机  $\mathcal{A}$  是一个五元组  $\mathcal{A} = (\Sigma, S, s_0, \delta, F)$ :

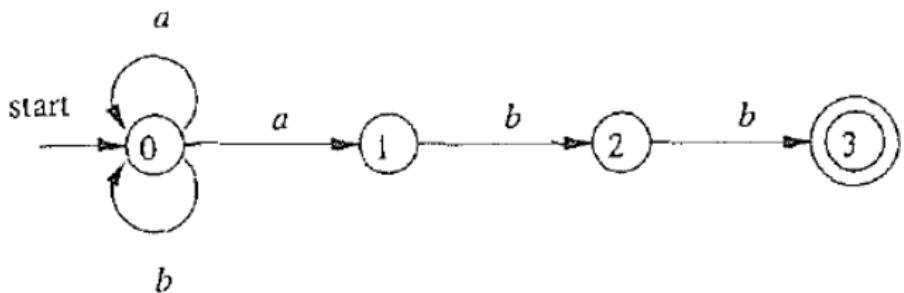
- (1) 字母表  $\Sigma$  ( $\epsilon \notin \Sigma$ )
- (2) **有穷**的状态集合  $S$
- (3) **唯一**的初始状态  $s_0 \in S$
- (4) 状态转移**函数**  $\delta$

$$\delta : S \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^S$$

- (5) 接受状态集合  $F \subseteq S$



**约定:** 所有没有对应出边的字符默认指向一个不存在的“空状态” $\emptyset$



状态	$a$	$b$	$\epsilon$
0	{0, 1}	{0}	$\emptyset$
1	$\emptyset$	{2}	$\emptyset$
2	$\emptyset$	{3}	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$



Michael O. Rabin  
(1931 ~)

### Finite Automata and Their Decision Problems<sup>†</sup>

**Abstract:** Finite automata are considered in this paper as instruments for classifying finite tapes. Each one-tape automaton defines a set of tapes, a two-tape automaton defines a set of pairs of tapes, et cetera. The structure of the defined sets is studied. Various generalizations of the notion of an automaton are introduced and their relation to the classical automata is determined. Some decision problems concerning automata are shown to be solvable by effective algorithms; others turn out to be unsolvable by algorithms.

发表于 1959 年;

1976 年, 共享图灵奖



Dana Scott (1932 ~)

*“which introduced the idea of **nondeterministic machines**, which has proved to be an enormously valuable concept.”*

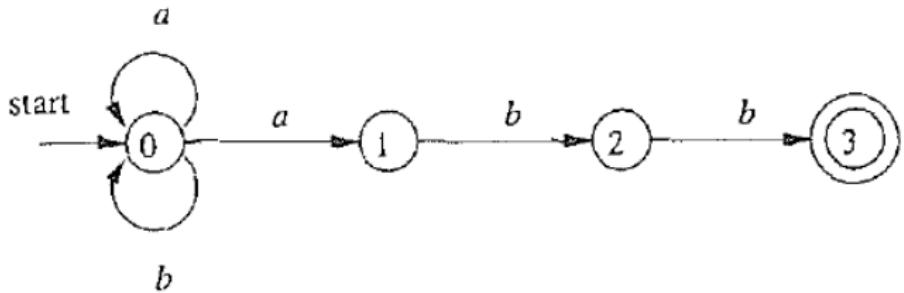
(非确定性) 有穷自动机是一类极其简单的**计算**装置

它可以**识别** (接受/拒绝)  $\Sigma$  上的字符串

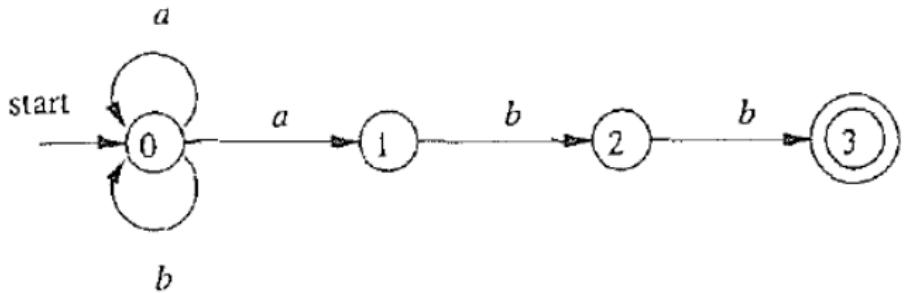
### Definition (接受 (Accept))

(非确定性) 有穷自动机  $\mathcal{A}$  接受字符串  $x$ , 当且仅当**存在**一条从开始状态  $s_0$  到**某个**接受状态  $f \in F$ 、标号为  $x$  的路径。

因此,  $\mathcal{A}$  定义了一种语言  $L(\mathcal{A})$ : 它能接受的所有字符串构成的集合

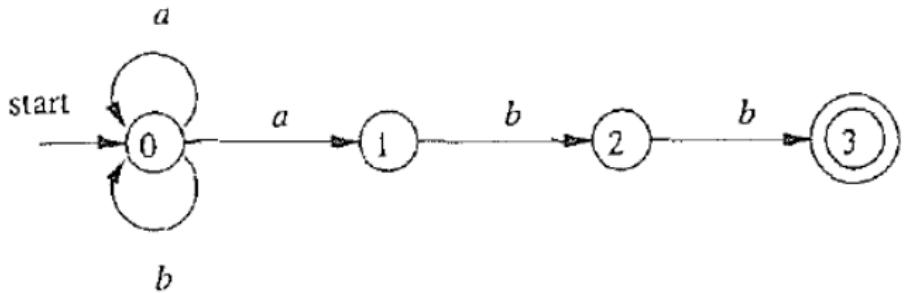


$aabb \in L(\mathcal{A})$        $ababab \notin L(\mathcal{A})$



$$aabbb \in L(\mathcal{A}) \quad ababab \notin L(\mathcal{A})$$

$$L(\mathcal{A}) =$$

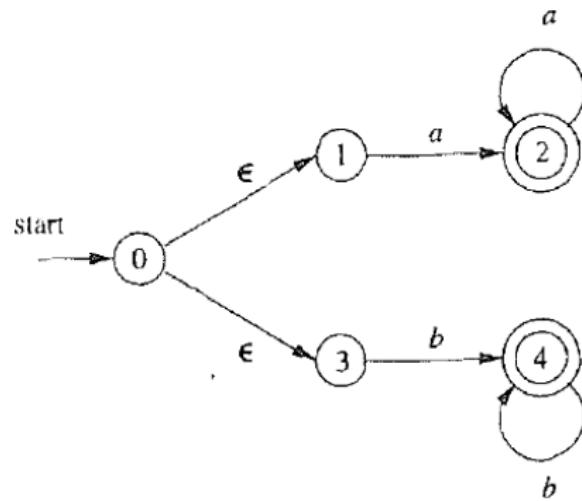


$$aabbb \in L(\mathcal{A}) \quad ababab \notin L(\mathcal{A})$$

$$L(\mathcal{A}) = L((a|b)^*abb)$$

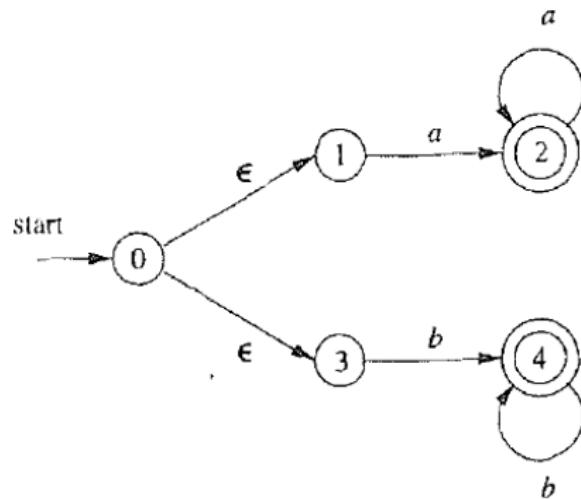
关于自动机  $\mathcal{A}$  的**两个基本问题**:

- ▶ **Membership 问题:** 给定字符串  $x, x \in L(\mathcal{A})?$
- ▶  $L(\mathcal{A})$  究竟是什么?



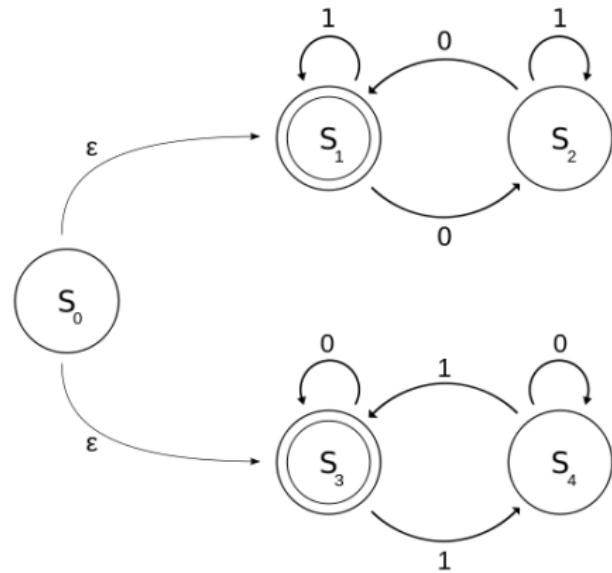
$aaa \in \mathcal{A}$ ?       $aab \in \mathcal{A}$ ?

$L(\mathcal{A}) =$

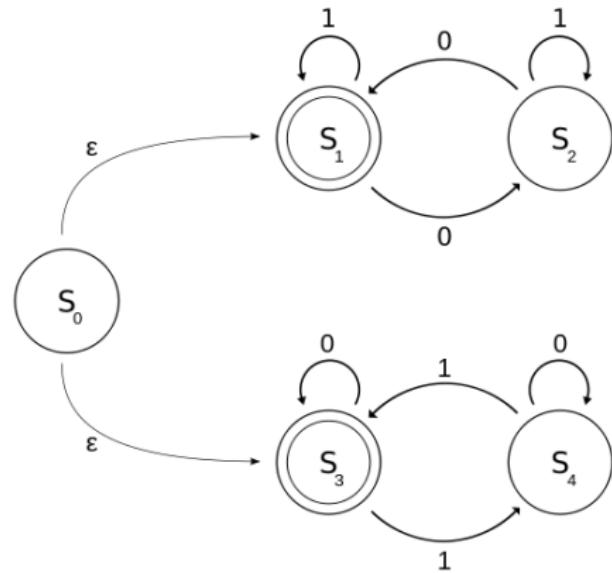


$aaa \in \mathcal{A}$ ?       $aab \in \mathcal{A}$ ?

$$L(\mathcal{A}) = L((aa^*|bb^*))$$



$1011 \in L(\mathcal{A})?$        $0011 \in L(\mathcal{A})?$



$1011 \in L(\mathcal{A})?$        $0011 \in L(\mathcal{A})?$

$L(\mathcal{A}) = \{\text{包含偶数个 1 或偶数个 0 的 01 串}\}$

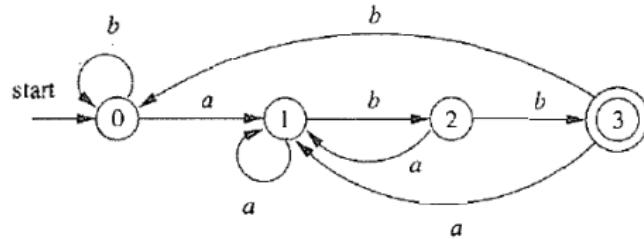
## Definition (DFA (Deterministic Finite Automaton))

确定性有穷自动机  $\mathcal{A}$  是一个五元组  $\mathcal{A} = (\Sigma, S, s_0, \delta, F)$ :

- (1) 字母表  $\Sigma$  ( $\epsilon \notin \Sigma$ )
- (2) **有穷**的状态集合  $S$
- (3) **唯一**的初始状态  $s_0 \in S$
- (4) 状态转移**函数**  $\delta$

$$\delta : S \times \Sigma \rightarrow S$$

- (5) 接受状态集合  $F \subseteq S$



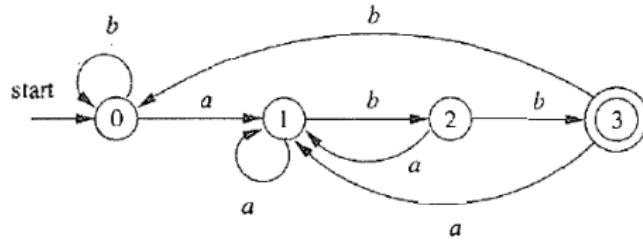
## Definition (DFA (Deterministic Finite Automaton))

确定性有穷自动机  $\mathcal{A}$  是一个五元组  $\mathcal{A} = (\Sigma, S, s_0, \delta, F)$ :

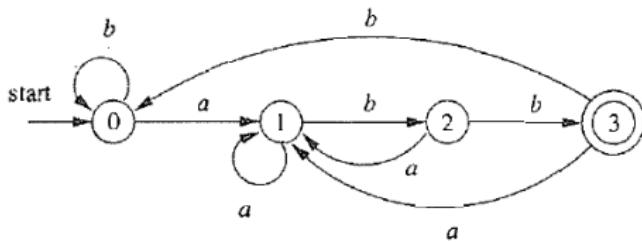
- (1) 字母表  $\Sigma$  ( $\epsilon \notin \Sigma$ )
- (2) **有穷**的状态集合  $S$
- (3) **唯一**的初始状态  $s_0 \in S$
- (4) 状态转移**函数**  $\delta$

$$\delta : S \times \Sigma \rightarrow S$$

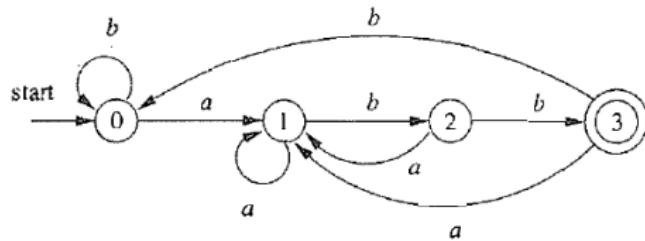
- (5) 接受状态集合  $F \subseteq S$



**约定:** 所有没有对应出边的字符默认指向一个不存在的“死状态”

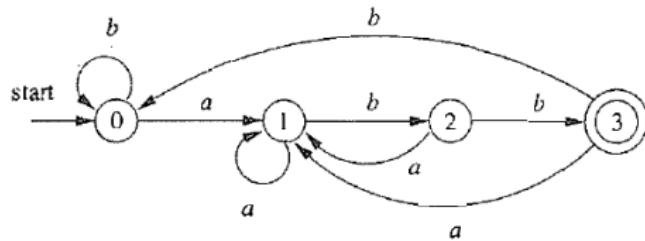


$$aabbb \in L(\mathcal{A}) \quad ababab \notin L(\mathcal{A})$$



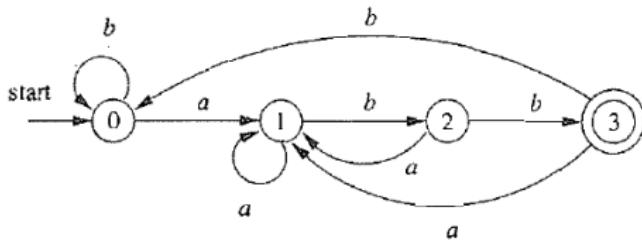
$$aabbb \in L(\mathcal{A}) \quad ababab \notin L(\mathcal{A})$$

$$L(\mathcal{A}) =$$



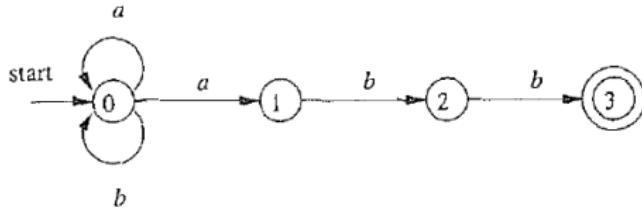
$$aabbb \in L(\mathcal{A}) \quad ababab \notin L(\mathcal{A})$$

$$L(\mathcal{A}) = L((a|b)^*abb)$$



$$aabbb \in L(\mathcal{A}) \quad ababab \notin L(\mathcal{A})$$

$$L(\mathcal{A}) = L((a|b)^*abb)$$



NFA 简洁易于理解, 方便描述语言  $L(\mathcal{A})$

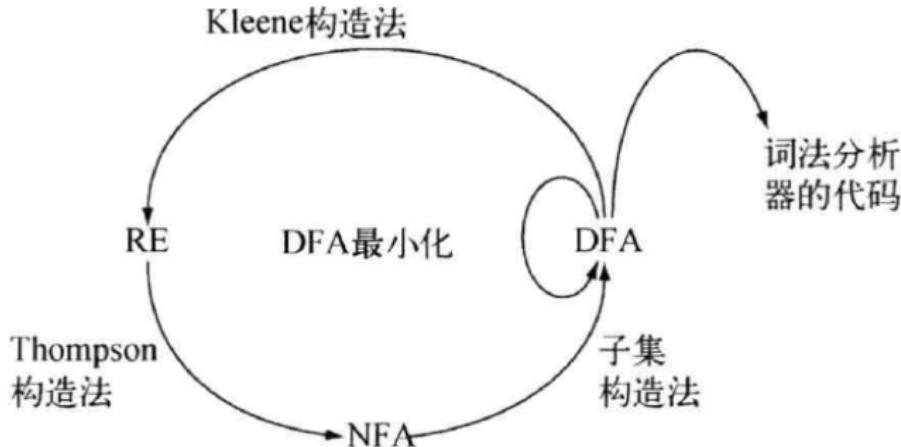
DFA 易于判断  $x \in L(\mathcal{A})$ , 适合产生词法分析器

NFA 简洁易于理解, 方便描述语言  $L(\mathcal{A})$

DFA 易于判断  $x \in L(\mathcal{A})$ , 适合产生词法分析器

用 NFA 描述语言, 用 DFA 实现词法分析器

RE  $\Rightarrow$  NFA  $\Rightarrow$  DFA  $\Rightarrow$  词法分析器



$\text{RE} \implies \text{NFA}$

$$r \implies N(r)$$

要求： $L(N(r)) = L(r)$

# 从 RE 到 NFA: **Thompson** 构造法

## 从 RE 到 NFA: Thompson 构造法



Turing Award, 1983

## Thompson 构造法的基本思想: 按结构归纳

### Definition (正则表达式)

给定字母表  $\Sigma$ ,  $\Sigma$  上的正则表达式由且仅由以下规则定义:

- (1)  $\epsilon$  是正则表达式;
- (2)  $\forall a \in \Sigma, a$  是正则表达式;
- (3) 如果  $s$  是正则表达式, 则  $(s)$  是正则表达式;
- (4) 如果  $s$  与  $t$  是正则表达式, 则  $s|t, st, s^*$  也是正则表达式。

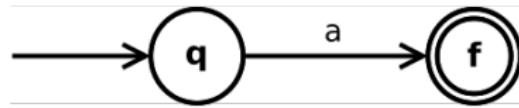
$\epsilon$  是正则表达式。

$\epsilon$  是正则表达式。



$a \in \Sigma$  是正则表达式。

$a \in \Sigma$  是正则表达式。



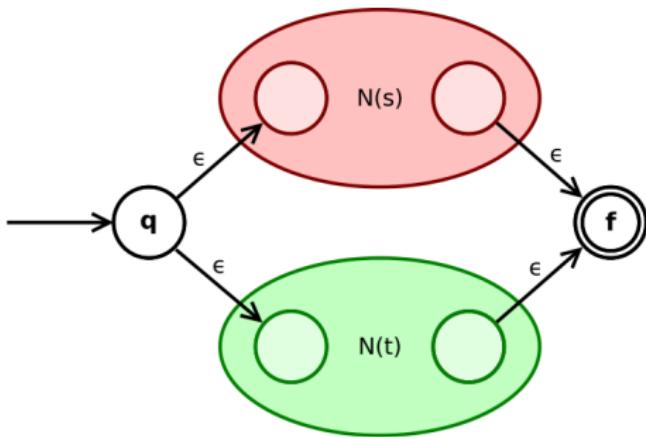
如果  $s$  是正则表达式, 则  $(s)$  是正则表达式;

如果  $s$  是正则表达式, 则  $(s)$  是正则表达式;

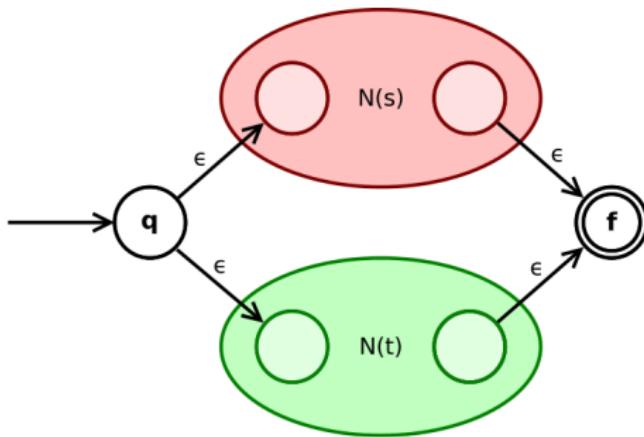
$$N((s)) = N(s).$$

如果  $s, t$  是正则表达式, 则  $s|t$  是正则表达式。

如果  $s, t$  是正则表达式, 则  $s|t$  是正则表达式。

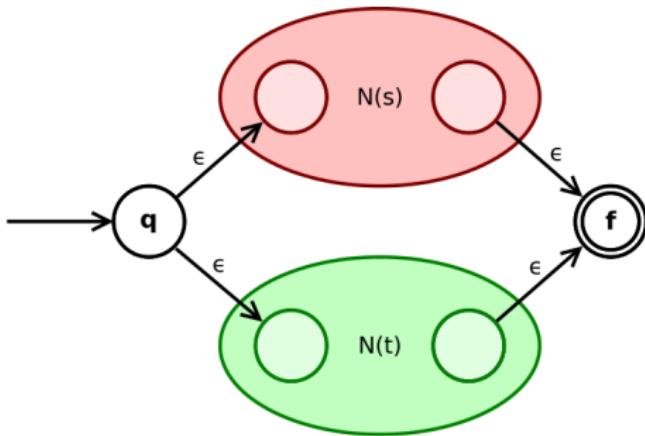


如果  $s, t$  是正则表达式, 则  $s|t$  是正则表达式。



$Q$ : 如果  $N(s)$  或  $N(t)$  的开始状态或接受状态不唯一, 怎么办?

如果  $s, t$  是正则表达式, 则  $s|t$  是正则表达式。

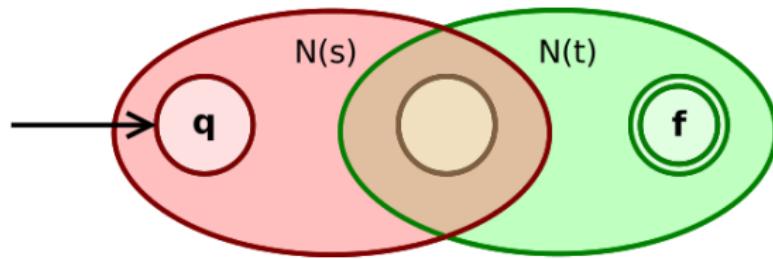


*Q* : 如果  $N(s)$  或  $N(t)$  的开始状态或接受状态不唯一, 怎么办?

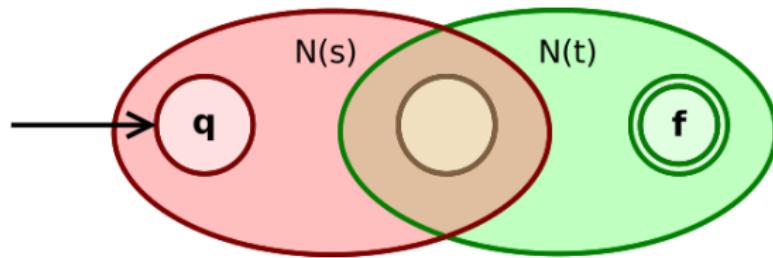
根据**归纳假设**,  $N(s)$  与  $N(t)$  的开始状态与接受状态均**唯一**。

如果  $s, t$  是正则表达式, 则  $st$  是正则表达式。

如果  $s, t$  是正则表达式, 则  $st$  是正则表达式。



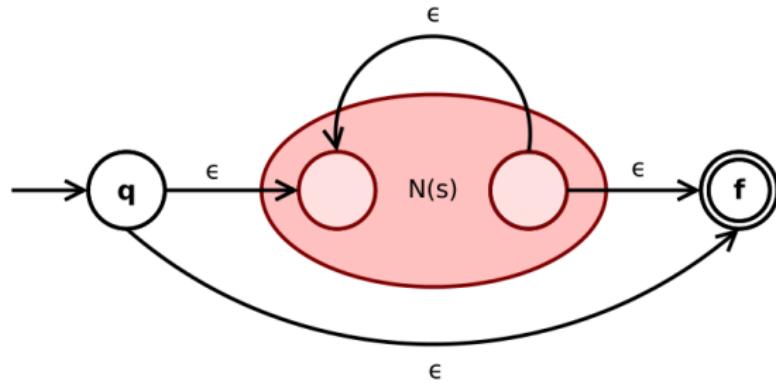
如果  $s, t$  是正则表达式, 则  $st$  是正则表达式。



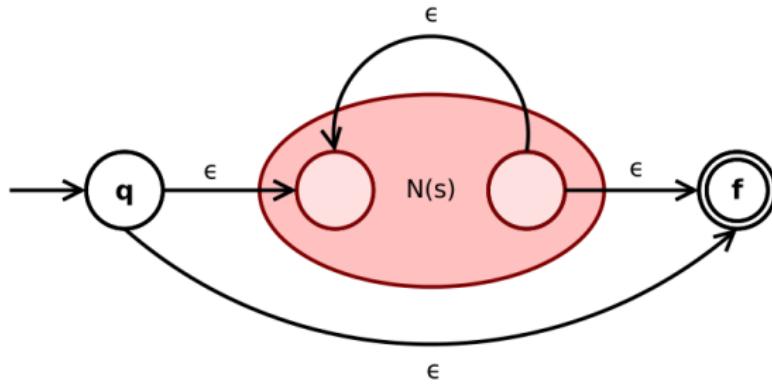
根据**归纳假设**,  $N(s)$  与  $N(t)$  的开始状态与接受状态均**唯一**。

如果  $s$  是正则表达式, 则  $s^*$  是正则表达式。

如果  $s$  是正则表达式，则  $s^*$  是正则表达式。



如果  $s$  是正则表达式，则  $s^*$  是正则表达式。



根据**归纳假设**,  $N(s)$  的开始状态与接受状态**唯一**。

## $N(r)$ 的性质以及 Thompson 构造法复杂度分析

1.  $N(r)$  的开始状态与接受状态均唯一。
2. 开始状态没有人边, 接受状态没有出边。

## $N(r)$ 的性质以及 Thompson 构造法复杂度分析

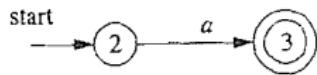
1.  $N(r)$  的开始状态与接受状态均唯一。
2. 开始状态没有人边, 接受状态没有出边。
3.  $N(r)$  的状态数  $|S| \leq 2 \times |r|$ 。  
( $|r| : r$  中运算符与运算分量的总和)

## $N(r)$ 的性质以及 Thompson 构造法复杂度分析

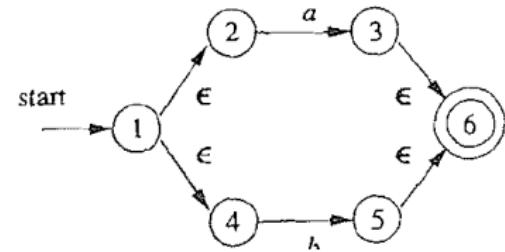
1.  $N(r)$  的开始状态与接受状态均唯一。
2. 开始状态没有入边, 接受状态没有出边。
3.  $N(r)$  的状态数  $|S| \leq 2 \times |r|$ 。  
( $|r| : r$  中运算符与运算分量的总和)
4. 每个状态最多有两个  $\epsilon$ -入边与两个  $\epsilon$ -出边。
5.  $\forall a \in \Sigma$ , 每个状态最多有一个  $a$ -入边与一个  $a$ -出边。

$$r = (a|b)^*abb$$

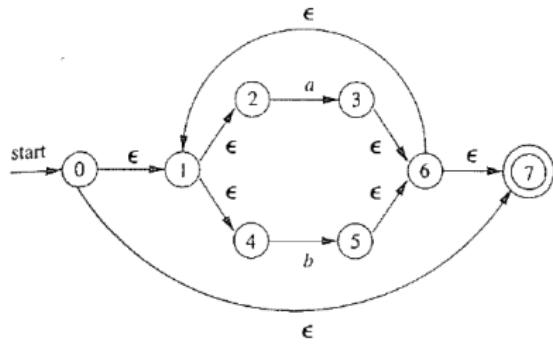
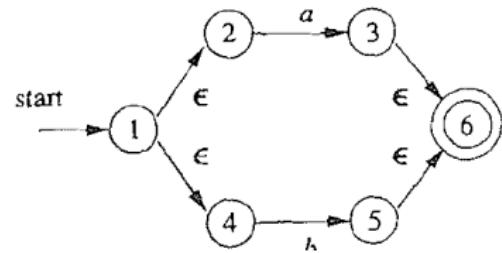
$$r = (a|b)^*abb$$



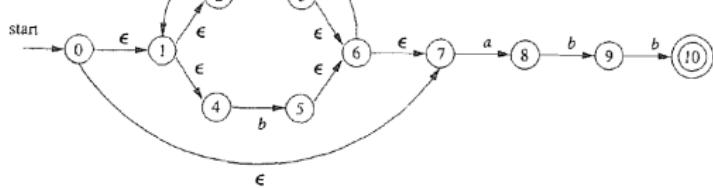
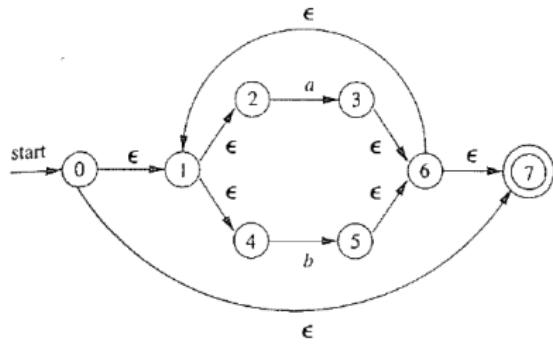
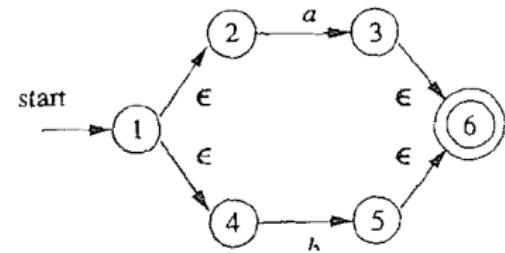
$$r = (a|b)^*abb$$

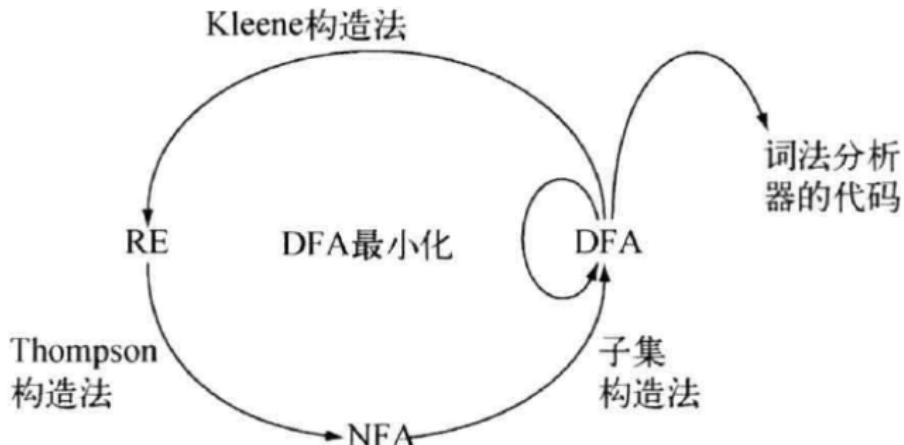


$$r = (a|b)^*abb$$



$$r = (a|b)^*abb$$





$\text{NFA} \implies \text{DFA}$

$$N \implies D$$

要求 :  $L(D) = L(N)$

# 从 NFA 到 DFA 的转换: 子集构造法 (Subset/Powerset Construction)



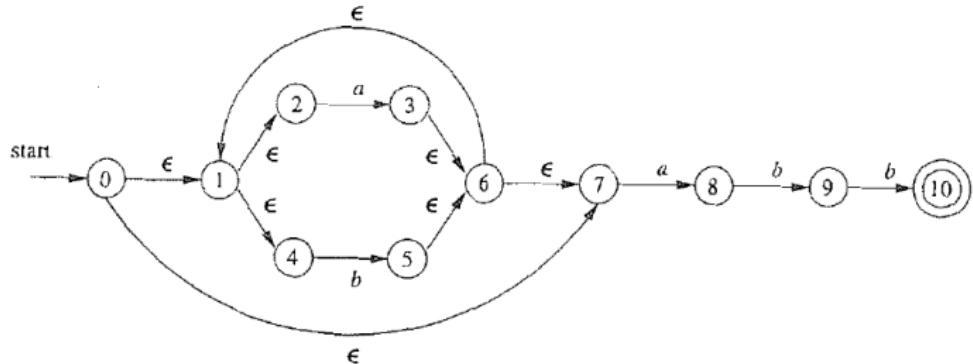
Michael O. Rabin (1931 ~)



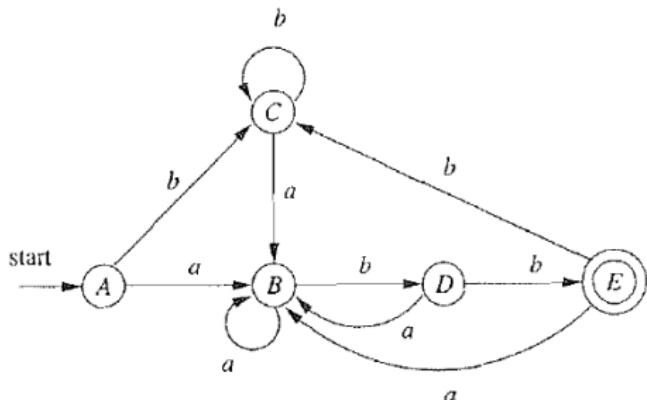
Dana Scott (1932 ~)

思想: 用 DFA 模拟 NFA

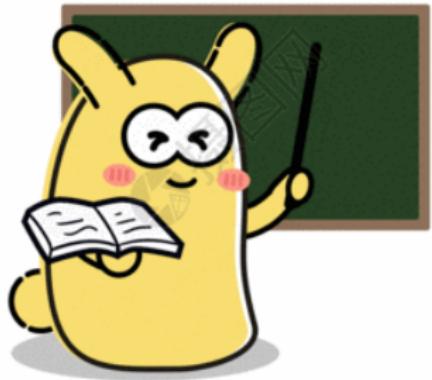
## 用 DFA 模拟 NFA



$$L(\mathcal{A}) = L((a|b)^*abb)$$

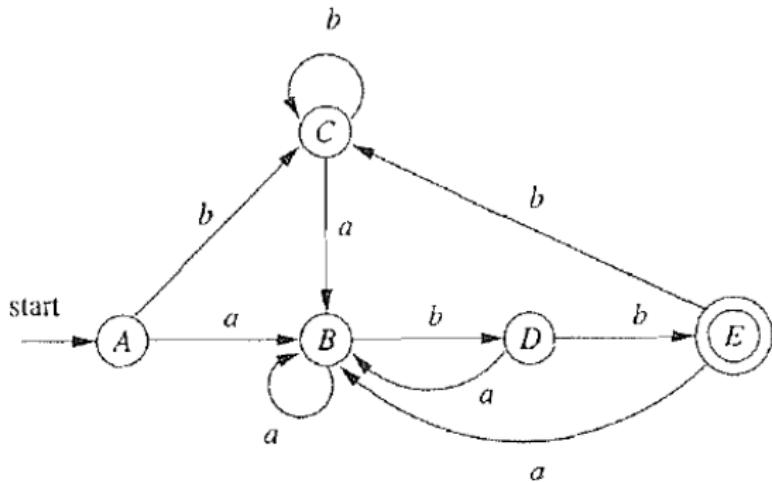


同学们看黑板！！



板书演示算法过程

NFA 状态	DFA 状态	$a$	$b$
$\{0, 1, 2, 4, 7\}$	$A$	$B$	$C$
$\{1, 2, 3, 4, 6, 7, 8\}$	$B$	$B$	$D$
$\{1, 2, 4, 5, 6, 7\}$	$C$	$B$	$C$
$\{1, 2, 4, 5, 6, 7, 9\}$	$D$	$B$	$E$
$\{1, 2, 4, 5, 6, 7, 10\}$	$E$	$B$	$C$



从状态  $s$  开始, 只通过  $\epsilon$ -转移可达的状态集合

$$\epsilon\text{-closure}(s) = \{t \in S_N \mid s \xrightarrow{\epsilon^*} t\}$$

从状态  $s$  开始, 只通过  $\epsilon$ -转移可达的状态集合

$$\epsilon\text{-closure}(s) = \{t \in S_N \mid s \xrightarrow{\epsilon^*} t\}$$

$$\epsilon\text{-closure}(T) = \bigcup_{s \in T} \epsilon\text{-closure}(s)$$

从状态  $s$  开始, 只通过  $\epsilon$ -转移可达的状态集合

$$\epsilon\text{-closure}(s) = \{t \in S_N \mid s \xrightarrow{\epsilon^*} t\}$$

$$\epsilon\text{-closure}(T) = \bigcup_{s \in T} \epsilon\text{-closure}(s)$$

$$\text{move}(T, a) = \bigcup_{s \in T} \delta(s, a)$$

子集构造法 ( $N \Rightarrow D$ ) 的原理:

$\textcolor{blue}{N} : (\Sigma_N, S_N, n_0, \delta_N, F_N)$

$\textcolor{red}{D} : (\Sigma_D, S_D, d_0, \delta_D, F_D)$

子集构造法 ( $N \Rightarrow D$ ) 的原理:

$\textcolor{blue}{N} : (\Sigma_N, S_N, n_0, \delta_N, F_N)$

$\textcolor{red}{D} : (\Sigma_D, S_D, d_0, \delta_D, F_D)$

$$\Sigma_D = \Sigma_N$$

子集构造法 ( $N \Rightarrow D$ ) 的原理:

$\textcolor{blue}{N} : (\Sigma_N, S_N, n_0, \delta_N, F_N)$

$\textcolor{red}{D} : (\Sigma_D, S_D, d_0, \delta_D, F_D)$

$$\Sigma_D = \Sigma_N$$

$$S_D \subseteq 2^{S_N} \quad (\forall s_D \in S_D : \textcolor{blue}{s_D} \subseteq S_N)$$

子集构造法 ( $N \Rightarrow D$ ) 的原理:

$\textcolor{blue}{N} : (\Sigma_N, S_N, n_0, \delta_N, F_N)$

$\textcolor{red}{D} : (\Sigma_D, S_D, d_0, \delta_D, F_D)$

$$\Sigma_D = \Sigma_N$$

$$S_D \subseteq 2^{S_N} \quad (\forall s_D \in S_D : \textcolor{blue}{s_D} \subseteq S_N)$$

初始状态  $d_0 = \epsilon\text{-closure}(n_0)$

子集构造法 ( $N \Rightarrow D$ ) 的原理:

$$\textcolor{blue}{N} : (\Sigma_N, S_N, n_0, \delta_N, F_N)$$

$$\textcolor{red}{D} : (\Sigma_D, S_D, d_0, \delta_D, F_D)$$

$$\Sigma_D = \Sigma_N$$

$$S_D \subseteq 2^{S_N} \quad (\forall s_D \in S_D : \textcolor{blue}{s_D} \subseteq S_N)$$

初始状态  $d_0 = \epsilon\text{-closure}(n_0)$

转移函数  $\forall a \in \Sigma_D : \delta_D(s_D, a) = \epsilon\text{-closure}(\text{move}(s_D, a))$

子集构造法 ( $N \Rightarrow D$ ) 的原理:

$\textcolor{blue}{N} : (\Sigma_N, S_N, n_0, \delta_N, F_N)$

$\textcolor{red}{D} : (\Sigma_D, S_D, d_0, \delta_D, F_D)$

$$\Sigma_D = \Sigma_N$$

$$S_D \subseteq 2^{S_N} \quad (\forall s_D \in S_D : \textcolor{blue}{s_D \subseteq S_N})$$

初始状态  $d_0 = \epsilon\text{-closure}(n_0)$

转移函数  $\forall a \in \Sigma_D : \delta_D(s_D, a) = \epsilon\text{-closure}(\text{move}(s_D, a))$

接受状态集  $F_D = \{s_D \in S_D \mid \exists f \in F_N. f \in s_D\}$

子集构造法 ( $N \Rightarrow D$ ) 的**实现**: 使用**栈**实现 $\epsilon$ -closure( $T$ )

```
将T的所有状态压入stack中;  
将  $\epsilon$ -closure( $T$ ) 初始化为  $T$ ;  
while ( stack 非空) {  
    将栈顶元素  $t$  弹出栈中;  
    for (每个满足如下条件的  $u$ : 从  $t$  出发有一个标号为  $\epsilon$  的转换到达状态  $u$ )  
        if (  $u$  不在  $\epsilon$ -closure( $T$ ) 中) {  
            将  $u$  加入到  $\epsilon$ -closure( $T$ ) 中;  
            将  $u$  压入栈中;  
        }  
    }  
}
```

子集构造法 ( $N \Rightarrow D$ ) 的实现：使用**标记搜索**过程构造**状态集**

```
一开始,  $\epsilon\text{-closure}(s_0)$  是  $Dstates$  中的唯一状态, 且它未加标记;  
while ( 在  $Dstates$  中有一个未标记状态  $T$  ) {  
    给  $T$  加上标记;  
    for ( 每个输入符号  $a$  ) {  
         $U = \epsilon\text{-closure}(\text{move}(T, a))$ ;  
        if (  $U$  不在  $Dstates$  中 )  
            将  $U$  加入到  $Dstates$  中, 且不加标记;  
         $Dtran[T, a] = U$ ;  
    }  
}
```

子集构造法的**复杂度分析**:  
 $(|S_N| = n)$

$$|S_D| = \Theta(2^n)$$

最坏情况下,  $|S_D| = \Omega(2^n)$

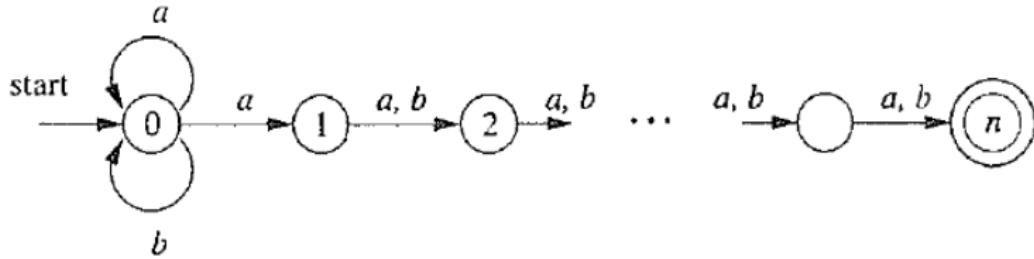
“长度为  $m \geq n$  个字符的  $a, b$  串, 且倒数第  $n$  个字符是  $a$ ”

“长度为  $m \geq n$  个字符的  $a, b$  串, 且倒数第  $n$  个字符是  $a$ ”

$$L_n = (a|b)^* a (a|b)^{\textcolor{red}{n-1}}$$

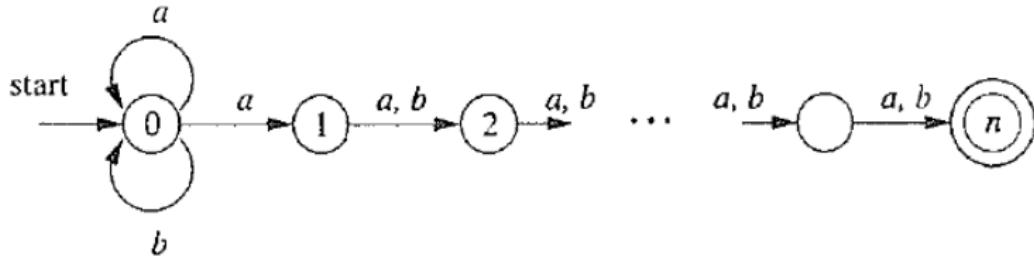
“长度为  $m \geq n$  个字符的  $a, b$  串, 且倒数第  $n$  个字符是  $a$ ”

$$L_n = (a|b)^* a (a|b)^{n-1}$$



“长度为  $m \geq n$  个字符的  $a, b$  串，且倒数第  $n$  个字符是  $a$ ”

$$L_n = (a|b)^* a (a|b)^{n-1}$$



作业:  $n = 3$

闭包 (Closure):  $f$ -closure( $T$ )

闭包 (Closure):  $f$ -closure( $T$ )

$\epsilon$ -closure( $T$ )

闭包 (Closure):  $f$ -closure( $T$ )

$\epsilon$ -closure( $T$ )

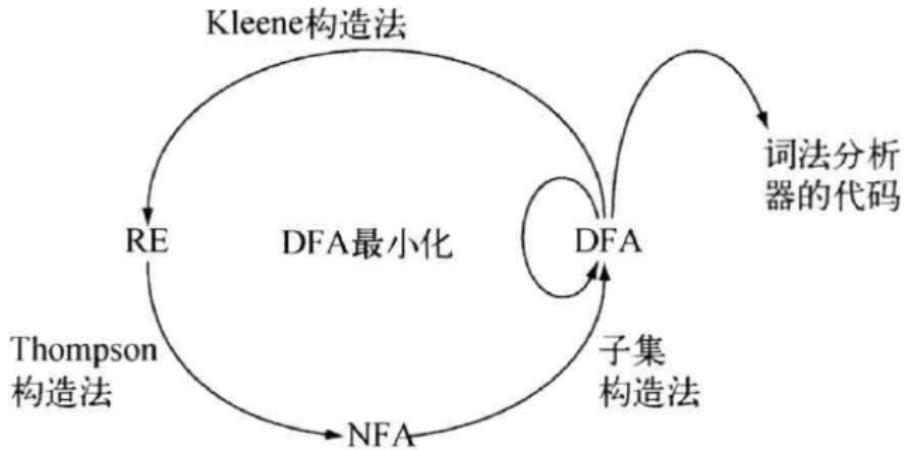
$T \implies f(T) \implies f(f(T)) \implies f(f(f(T))) \implies \dots$

闭包 (Closure):  $f$ -closure( $T$ )

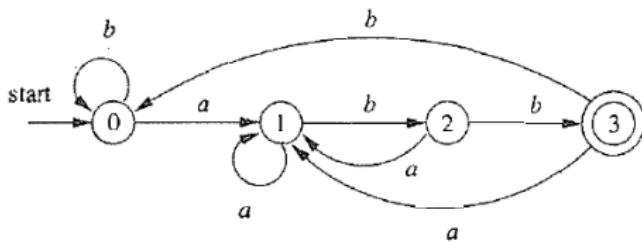
$\epsilon$ -closure( $T$ )

$T \implies f(T) \implies f(f(T)) \implies f(f(f(T))) \implies \dots$

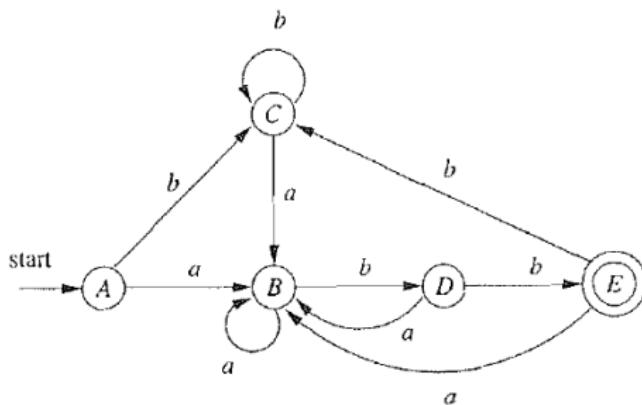
直到找到  $x$  使得  $f(x) = x$  ( $x$  称为  $f$  的**不动点**)



## DFA 最小化



$$L(\mathcal{A}) = L((a|b)^*abb)$$



子集构造法得到的 DFA

## DFA 最小化算法基本思想: 等价的状态可以合并

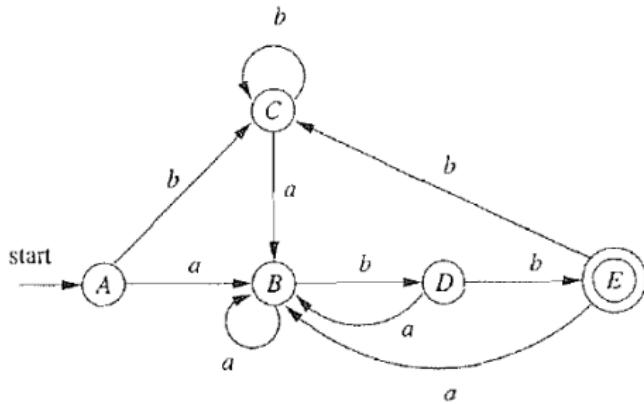


John Hopcroft (1939 ~)

“With Robert E. Tarjan, for fundamental achievements in the design and analysis of **algorithms and data structures**”

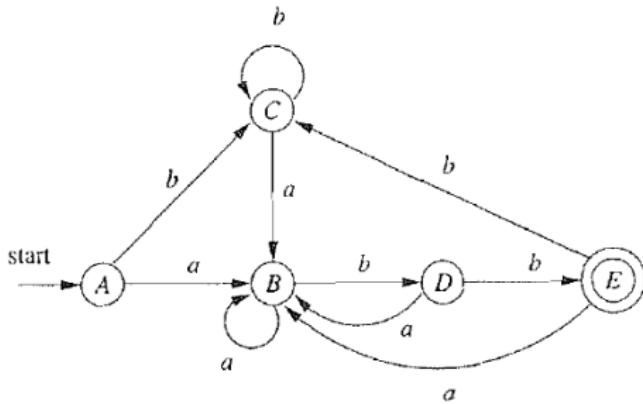
— Turing Award, 1986

## 如何定义等价状态?



$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \implies (s' = t').$$

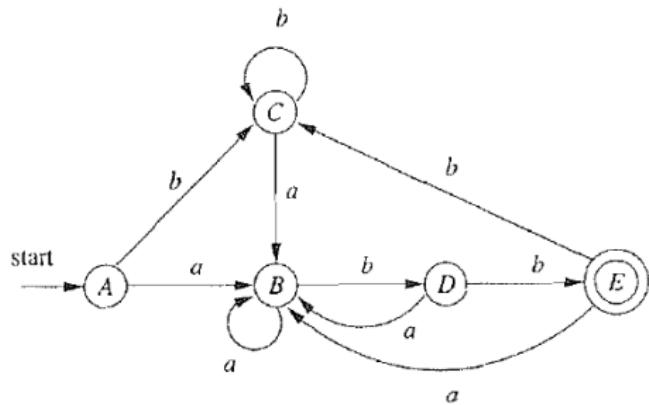
## 如何定义等价状态?



$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \implies (s' = t').$$

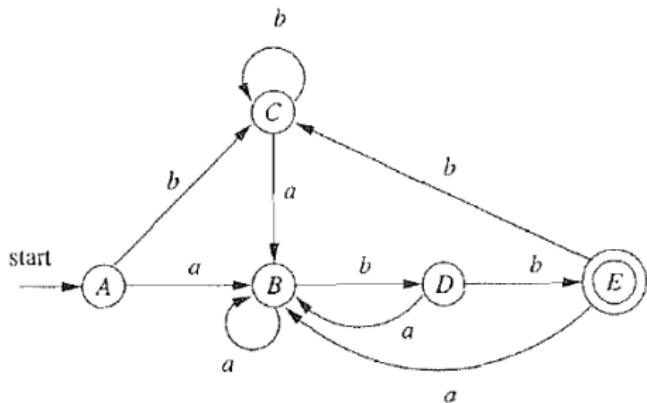
但是,这个定义是错误的

$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \implies (s' = t').$$



$$A \sim C \sim E$$

$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \implies (s' = t').$$

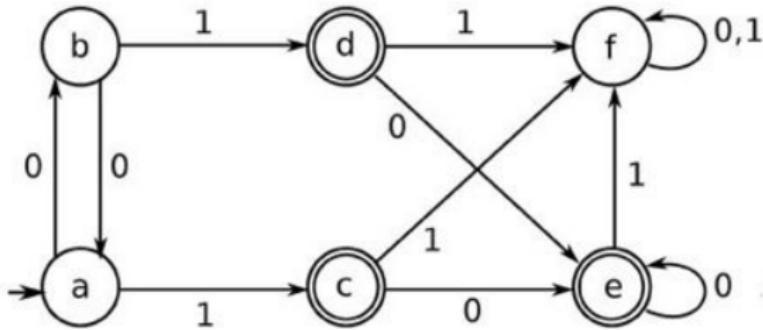


$$A \sim C \sim E$$

但是，接受状态与非接受状态必定不等价

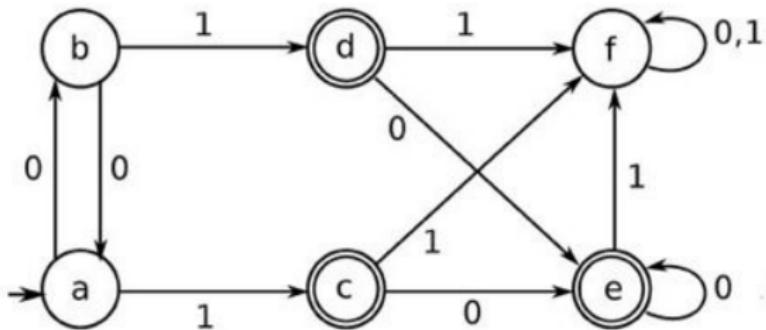
空串  $\epsilon$  区分了这两类状态

$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \implies (s' = t').$$

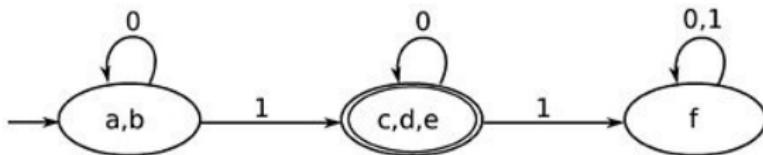


$$c \sim d \sim e \quad a \not\sim b$$

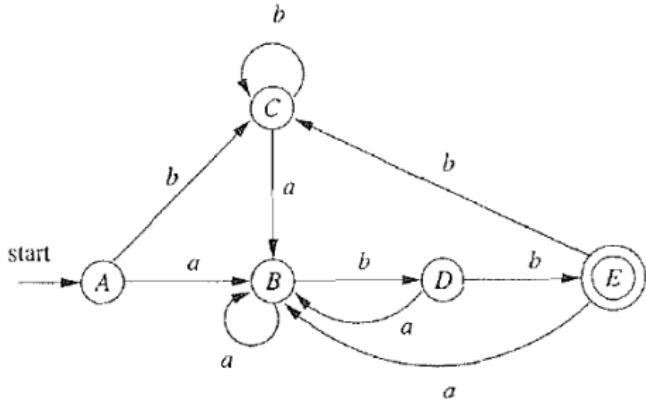
$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \Rightarrow (s' = t').$$



$$c \sim d \sim e \quad a \not\sim b$$

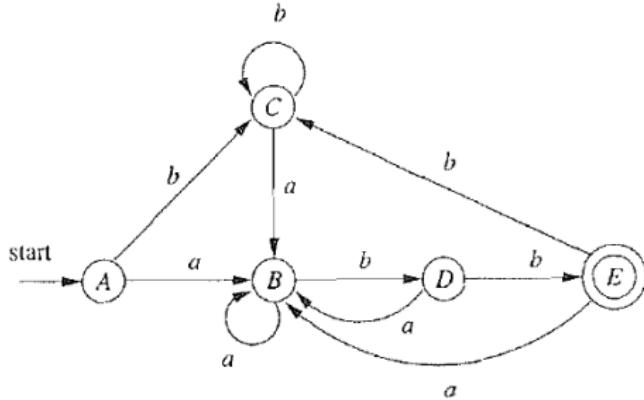


## 如何定义等价状态?



$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \Rightarrow (s' \sim t').$$

## 如何定义等价状态?



$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \Rightarrow (s' \sim t').$$

$$s \not\sim t \iff \exists a \in \Sigma. (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \wedge (s' \not\sim t')$$

$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \Rightarrow (s' \sim t').$$

基于该定义, 不断**合并**等价的状态, 直到无法合并为止

$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \Rightarrow (s' \sim t').$$

基于该定义, 不断**合并**等价的状态, 直到无法合并为止

但是, 这是一个递归定义, 从哪里开始呢?

$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \Rightarrow (s' \sim t').$$

基于该定义, 不断**合并**等价的状态, 直到无法合并为止

但是, 这是一个递归定义, 从哪里开始呢?

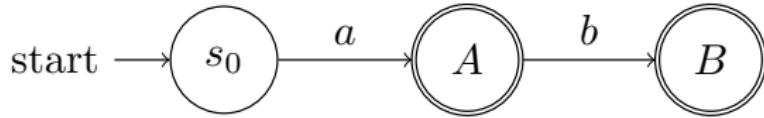
$Q$ : 所有接受状态都是等价的吗?

$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \Rightarrow (s' \sim t').$$

基于该定义, 不断**合并**等价的状态, 直到无法合并为止

但是, 这是一个递归定义, 从哪里开始呢?

Q : 所有接受状态都是等价的吗?



$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \implies (s' \sim t').$$

缺少基础情况, 不知从何下手

$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \implies (s' \sim t').$$

缺少基础情况, 不知从何下手

$$s \not\sim t \iff \exists a \in \Sigma. (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \wedge (s' \not\sim t')$$

$$s \sim t \iff \forall a \in \Sigma. \left( (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \right) \implies (s' \sim t').$$

缺少基础情况, 不知从何下手

$$s \not\sim t \iff \exists a \in \Sigma. (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \wedge (s' \not\sim t')$$

**划分, 而非合并**

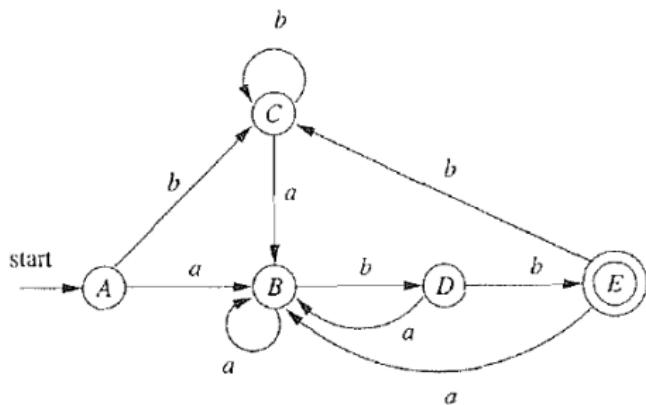
$$s \not\sim t \iff \exists a \in \Sigma. (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \wedge (s' \not\sim t')$$

$$s \not\sim t \iff \exists a \in \Sigma. (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \wedge (s' \not\sim t')$$

从哪里开始呢？

$$s \not\sim t \iff \exists a \in \Sigma. (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \wedge (s' \not\sim t')$$

从哪里开始呢?

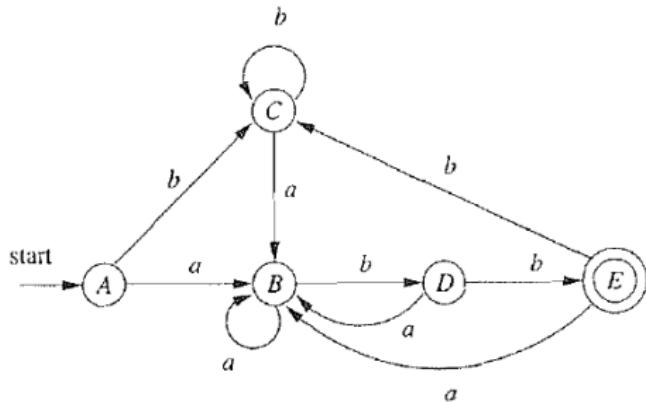


$$\Pi = \{F, S \setminus F\}$$

接受状态与非接受状态必定不等价

$$s \not\sim t \iff \exists a \in \Sigma. (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \wedge (s' \not\sim t')$$

从哪里开始呢?

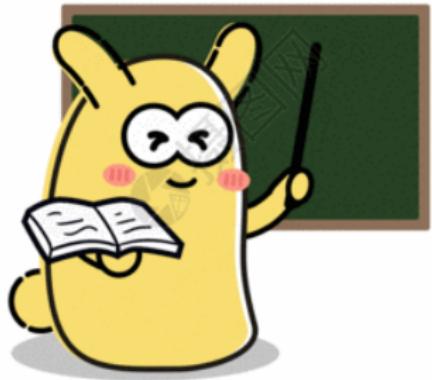


$$\Pi = \{F, S \setminus F\}$$

接受状态与非接受状态必定不等价

空串  $\epsilon$  区分了这两类状态

同学们看黑板！！



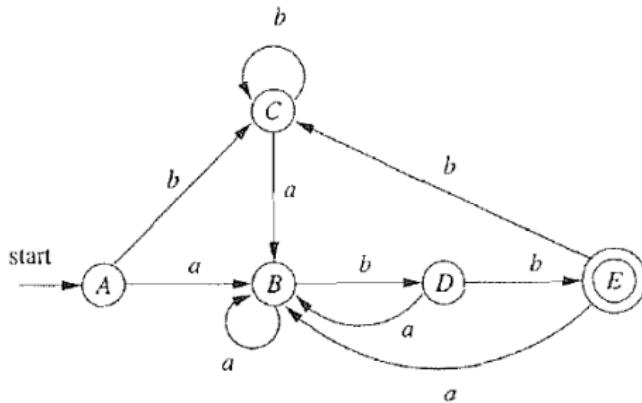
板书演示算法过程

$$\Pi_0 = \{\{A, B, C, D\}, \{E\}\}$$

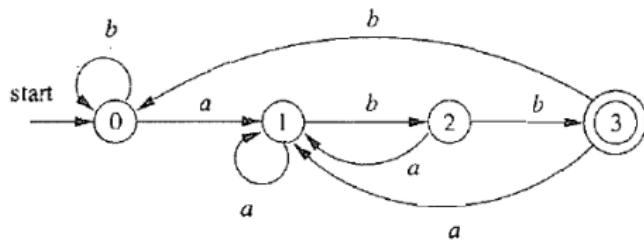
$$\Pi_1 = \{\{A, B, C\}, \{D\}, \{E\}\}$$

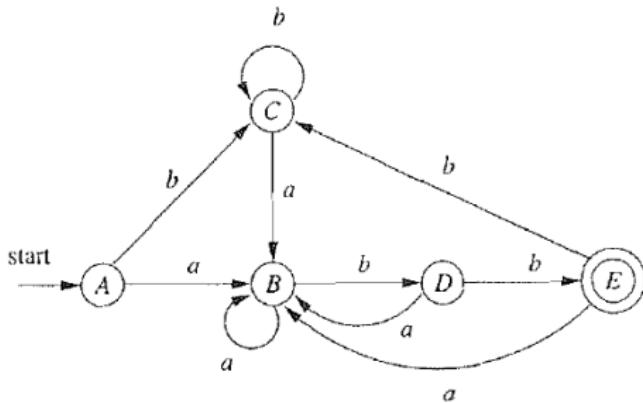
$$\Pi_2 = \{\{A, C\}, \{B\}, \{D\}, \{E\}\}$$

$$\Pi_3 = \Pi_2 = \text{Pi}_{\text{final}}$$

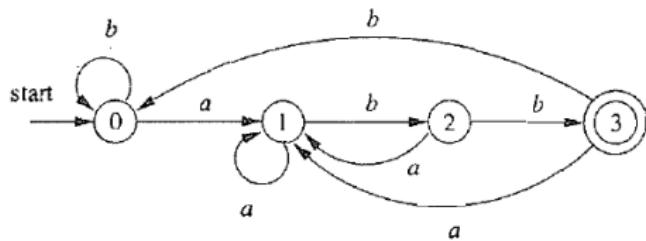


合并等价状态  $A \sim C$





合并等价状态  $A \sim C$



Q : 合并后是否一定还是 DFA? 初始状态、接受状态是哪些?

## DFA 最小化等价状态划分方法

$$\Pi = \{F, S \setminus F\}$$

```
最初, 令  $\Pi_{\text{new}} = \Pi$ ;
for ( $\Pi$ 中的每个组  $G$ ) {
    将 $G$ 分划为更小的组, 使得两个状态 $s$ 和 $t$ 在同一小组中当且仅当对于所有的
        的输入符号 $a$ , 状态 $s$ 和 $t$ 在 $a$ 上的转换都到达 $\Pi$ 中的同一组;
    /* 在最坏情况下, 每个状态各自组成一个组*/
    在 $\Pi_{\text{new}}$ 中将 $G$ 替换为对 $G$ 进行分划得到的那些小组;
}
```

直到再也无法**划分**为止 (不动点!)

然后, 将同一等价类里的状态**合并**

# 如何分析 DFA 最小化算法的复杂度?

如何分析 DFA 最小化算法的**复杂度**?

为什么 DFA 最小化算法是**正确的**?

如何分析 DFA 最小化算法的**复杂度**?

为什么 DFA 最小化算法是**正确的**?

最小化 DFA 是**唯一**的吗?

如何分析 DFA 最小化算法的**复杂度**?

为什么 DFA 最小化算法是**正确的**?

最小化 DFA 是**唯一的**吗?

DFA Minimization @ wiki

## Definition (字符串 $s$ 区分状态 $s$ 与 $t$ )

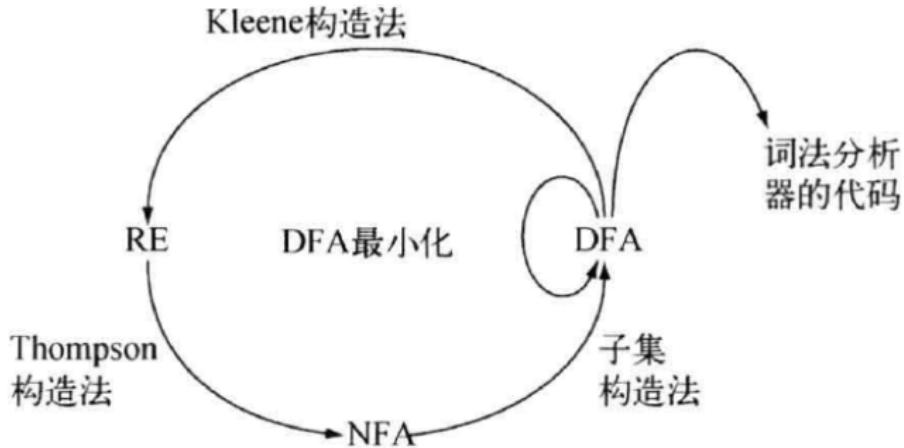
如果分别从  $s$  与  $t$  出发, 沿着标号为  $s$  的路径到达的两个状态中只有一个接受状态, 则称  $s$  **区分**了状态  $s$  与  $t$ 。

### Definition (字符串 $s$ 区分状态 $s$ 与 $t$ )

如果分别从  $s$  与  $t$  出发, 沿着标号为  $s$  的路径到达的两个状态中只有一个接受状态, 则称  $s$  **区分**了状态  $s$  与  $t$ 。

### Definition (可区分的 (Distinguishable))

如果存在某个能区分状态  $s$  与  $t$  的字符串, 则称  $s$  与  $t$  是**可区分的**。



DFA  $\Rightarrow$  词法分析器

a  
abb  
 $a^* b^+$

{模式  $p_1$  的动作  $A_1$ }  
{模式  $p_2$  的动作  $A_2$ }  
{模式  $p_3$  的动作  $A_3$ }

最前优先匹配:  $abb$

最长优先匹配:  $aabbb$

根据正则表达式构造相应的 NFA

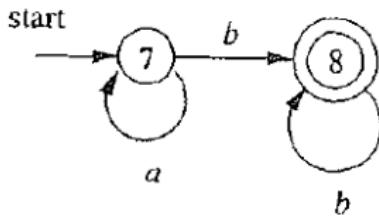
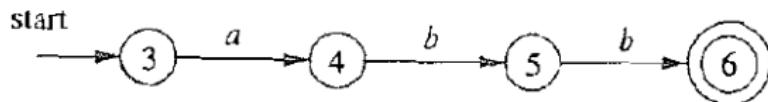
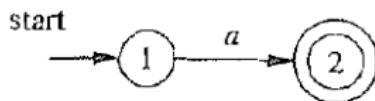
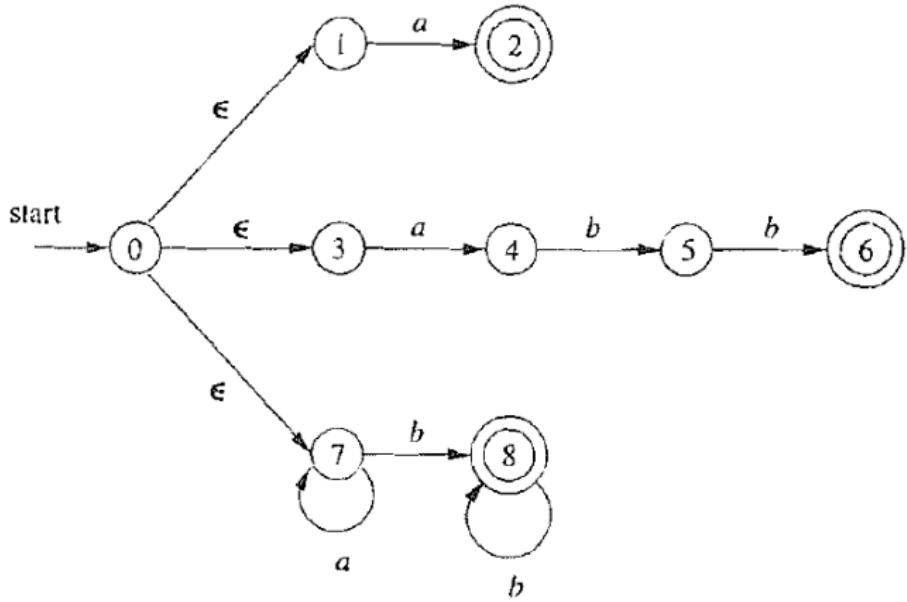
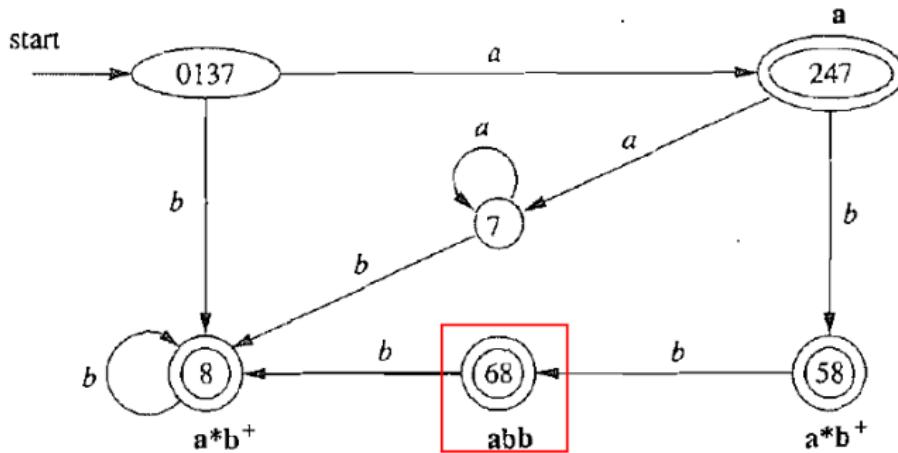


图 3-51  $a$ 、 $abb$  和  $a^* b^+$  的 NFA

合并这三个 NFA, 构造  $a|abb|a^*b^+$  对应的 NFA



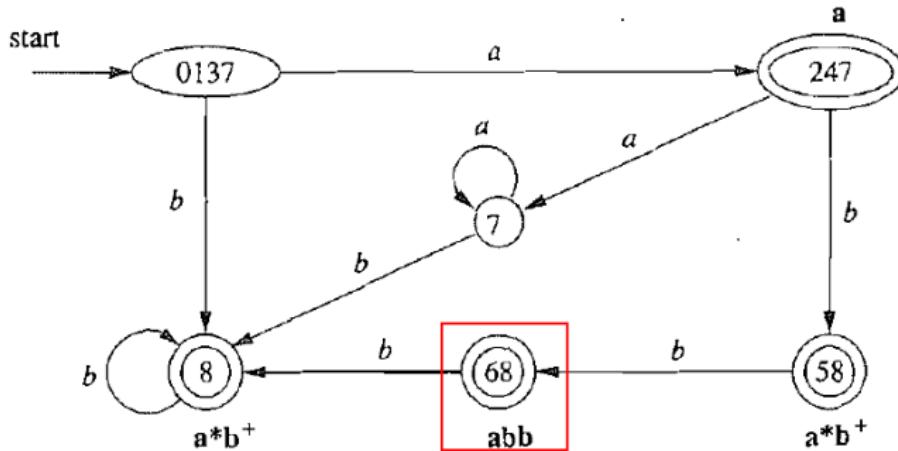
使用子集构造法将 NFA 转化为等价的 DFA (并消除“死状态” $\emptyset$ )



注意: 要保留各个 NFA 的接受状态信息, 并采用**最前优先匹配原则**

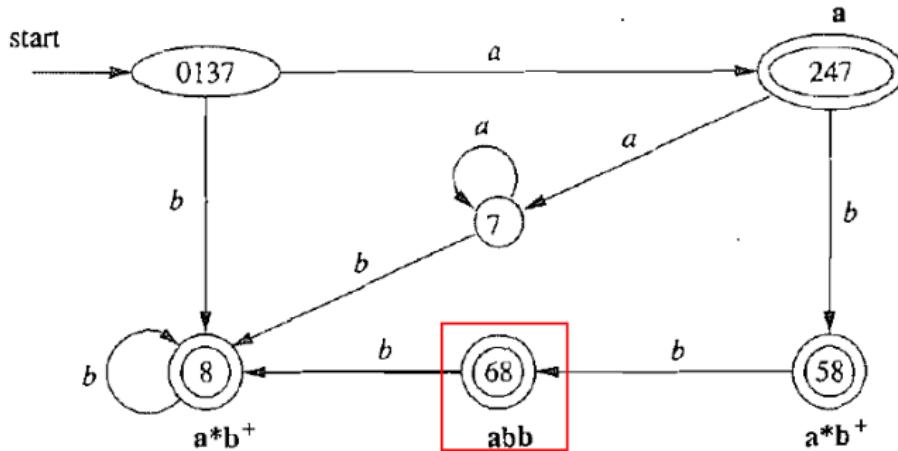
模拟运行该 DFA, 直到无法继续为止 (输入结束或状态无转移);

假设此时状态为  $s$



模拟运行该 DFA, 直到无法继续为止 (输入结束或状态无转移);

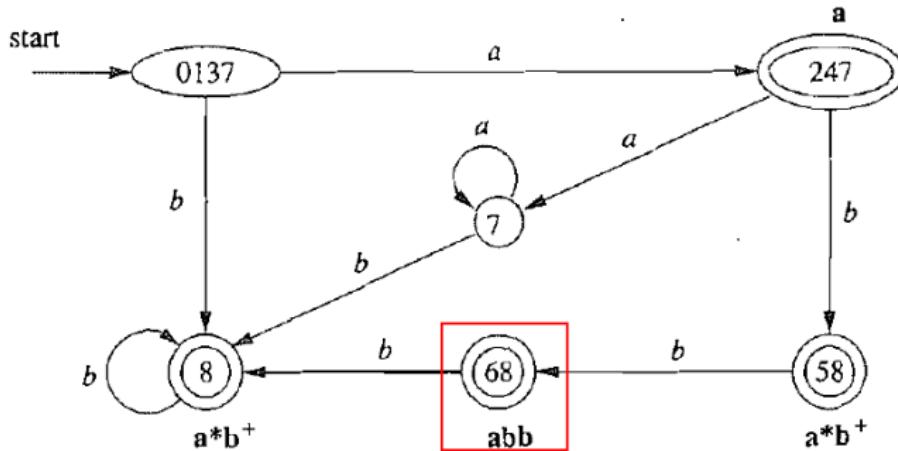
假设此时状态为  $s$



若  $s$  为接受状态, 则识别成功;

模拟运行该 DFA, 直到无法继续为止 (输入结束或状态无转移);

假设此时状态为  $s$

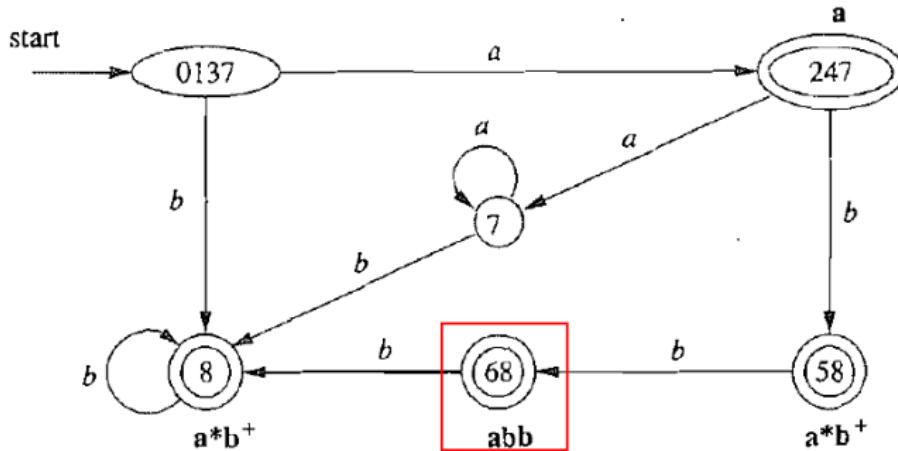


若  $s$  为接受状态, 则识别成功;

否则, 回溯 (包括状态与输入流) 至最近一次经过的接受状态, 识别成功;

模拟运行该 DFA, 直到无法继续为止 (输入结束或状态无转移);

假设此时状态为  $s$



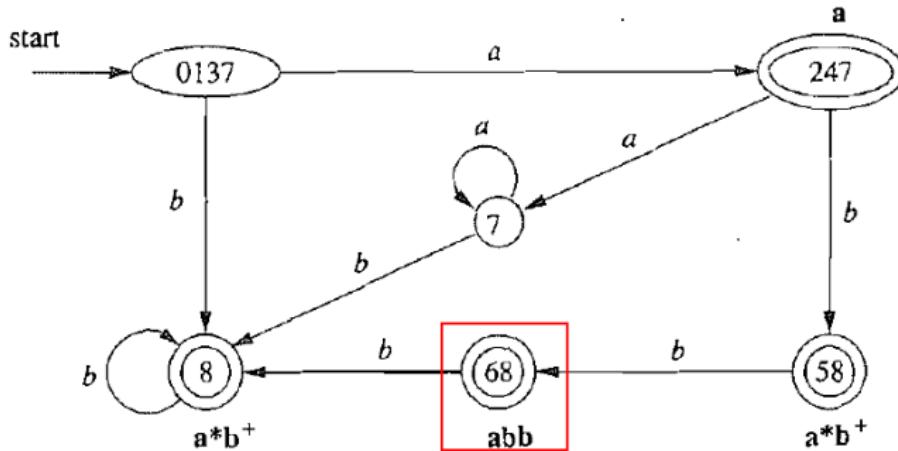
若  $s$  为接受状态, 则识别成功;

否则, 回溯 (包括状态与输入流) 至最近一次经过的接受状态, 识别成功;

若没有经过任何接受状态, 则报错 (忽略第一个字符)

模拟运行该 DFA, 直到无法继续为止 (输入结束或状态无转移);

假设此时状态为  $s$

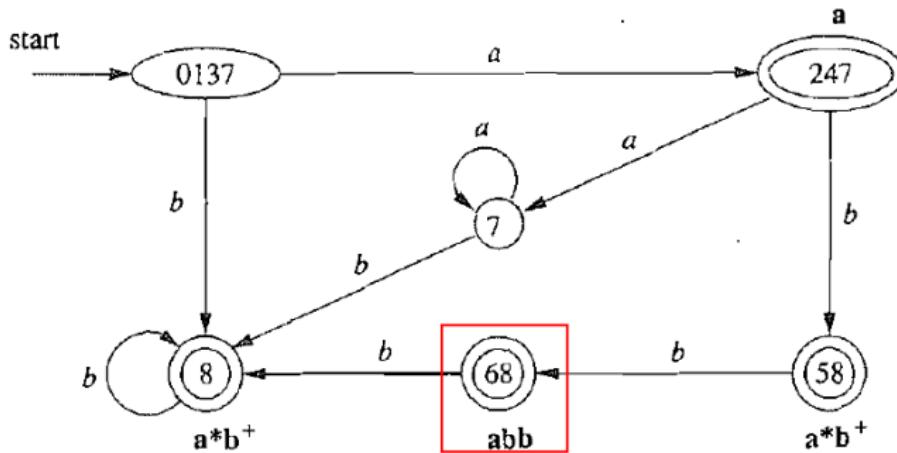


若  $s$  为接受状态, 则识别成功;

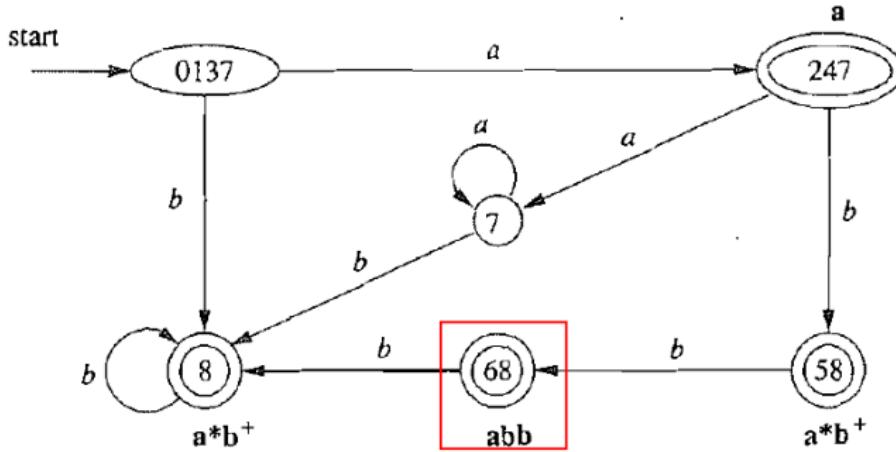
否则, 回溯 (包括状态与输入流) 至最近一次经过的接受状态, 识别成功;

若没有经过任何接受状态, 则报错 (忽略第一个字符)

无论成功还是失败, 都从初始状态开始继续识别下一个词法单元

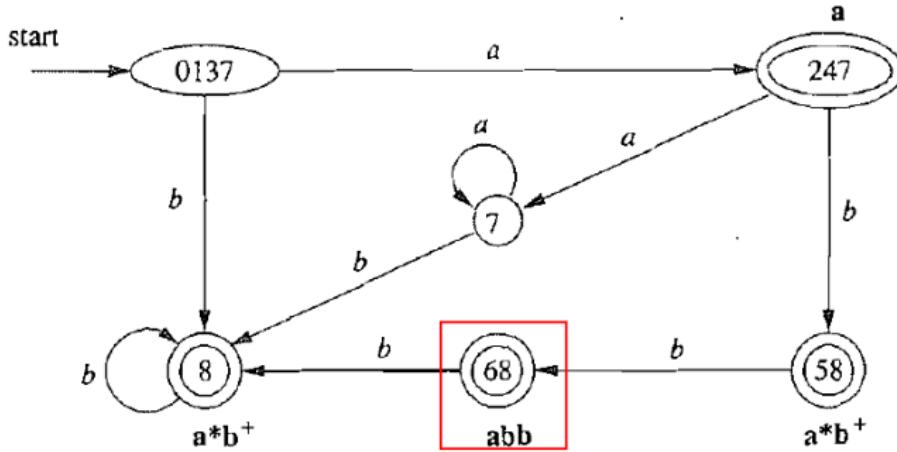


$x = a$       输入结束; 接受; 识别出  $a$



$x = a$       输入结束; 接受; 识别出  $a$

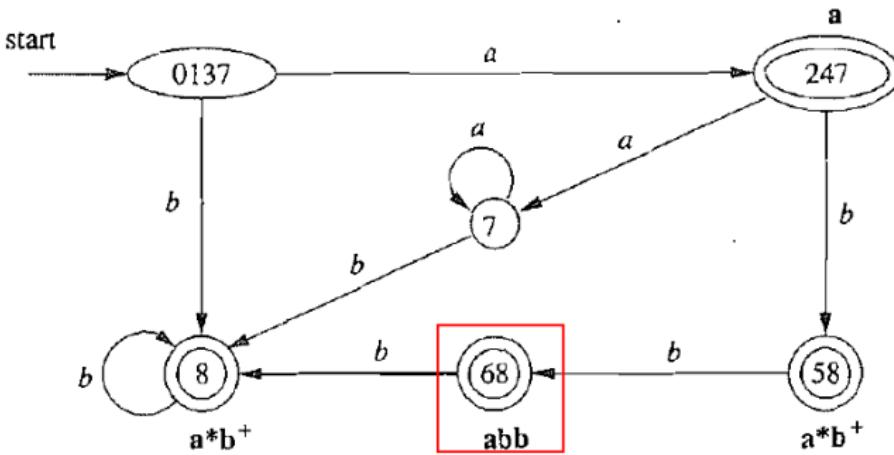
$x = abba$       状态无转移; 回溯成功; 识别出  $abb$



$x = a$       输入结束; 接受; 识别出  $a$

$x = abba$       状态无转移; 回溯成功; 识别出  $abb$

$x = aaaa$       输入结束; 回溯成功; 识别出  $a$



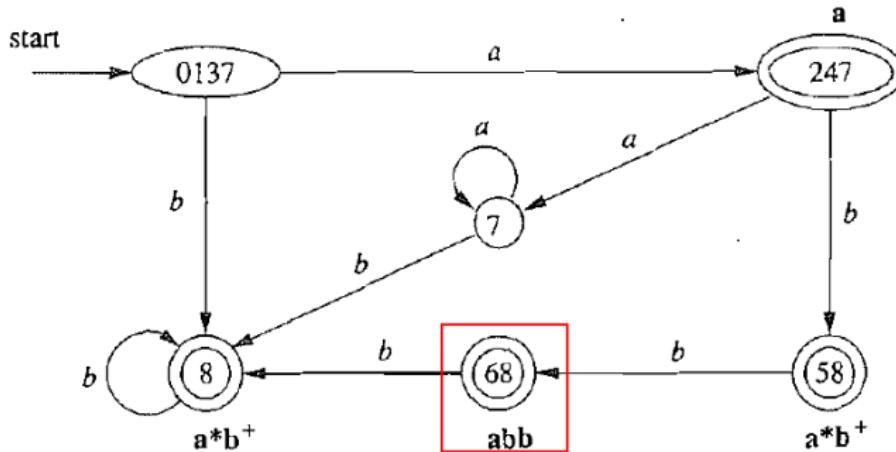
$x = a$       输入结束; 接受; 识别出  $a$

$x = abba$       状态无转移; 回溯成功; 识别出  $abb$

$x = aaaa$       输入结束; 回溯成功; 识别出  $a$

$x = cabb$       状态无转移; 回溯失败; 报错  $c$

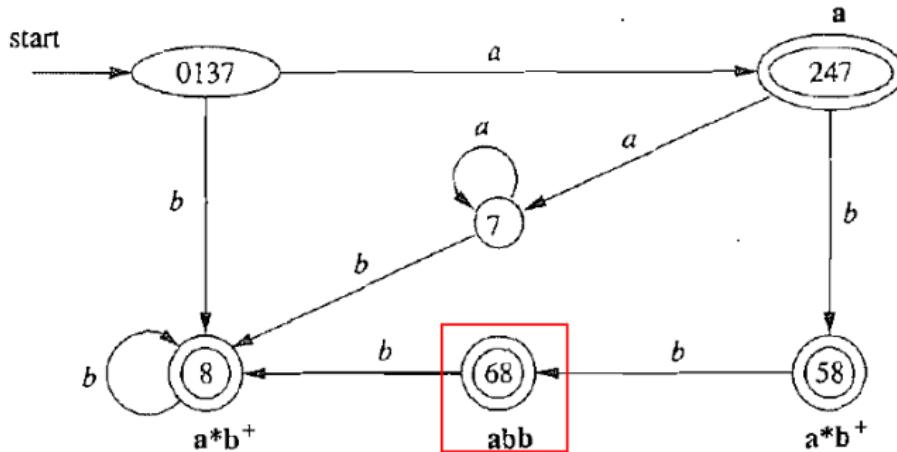
## 特定于词法分析器的 DFA 最小化方法



初始划分需要考虑不同的词法单元

$$\Pi_0 = \{\{0137, 7\}, \{247\}, \{8, 58\}, \{68\}, \{\emptyset\}\}$$

## 特定于词法分析器的 DFA 最小化方法



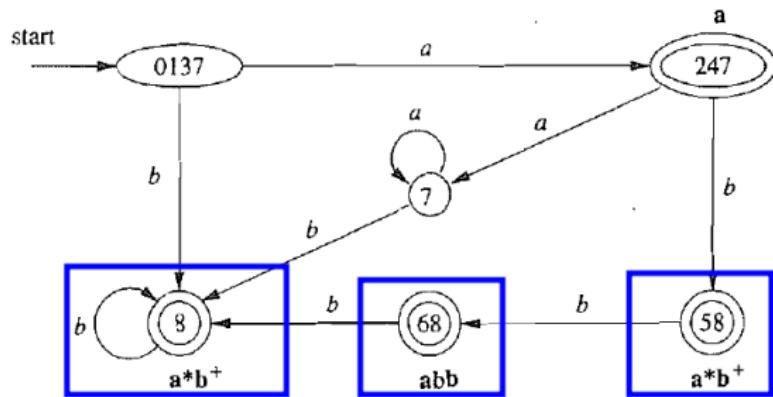
初始划分需要考虑不同的词法单元

$$\Pi_0 = \{\{0137, 7\}, \{247\}, \{8, 58\}, \{68\}, \{\emptyset\}\}$$

$$\Pi_1 = \{\{0137\}, \{7\}, \{247\}, \{8\}, \{58\}, \{68\}\}$$

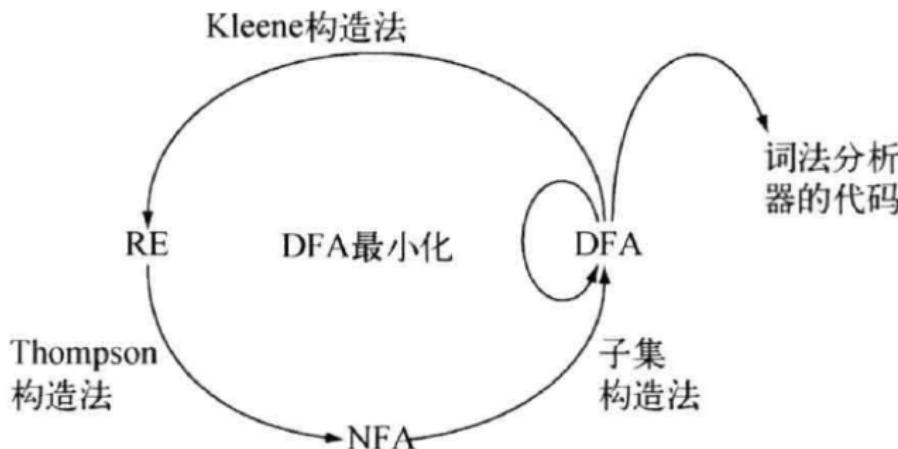
## 特殊处理“死状态 $\emptyset$ ”

$$\text{move}(T, a) = \emptyset \implies \delta(T, a) = \emptyset$$



$$\forall a \in \Sigma. \delta(\emptyset, a) = \emptyset$$

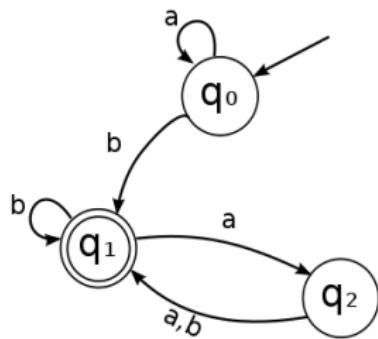
需要判断是否进入死状态，避免词法分析器徒劳消耗输入流

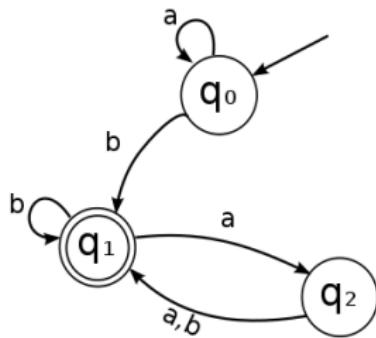


DFA  $\Rightarrow$  RE

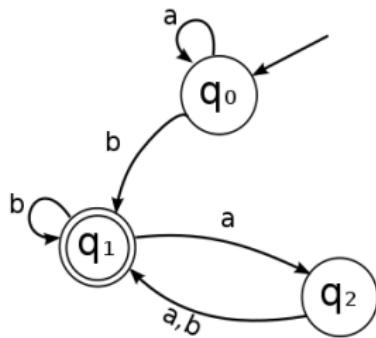
$$D \implies r$$

要求： $L(r) = L(D)$



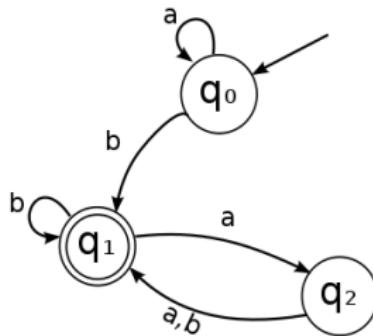


$$L(D) = \{x \mid \exists f \in F_D. s_0 \xrightarrow{x} f\}$$



$$L(D) = \{x \mid \exists f \in F_D. s_0 \xrightarrow{x} f\}$$

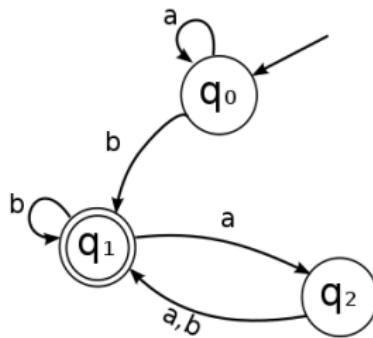
$$r = |_{x \in L(D)} x$$



$$L(D) = \{x \mid \exists f \in F_D. s_0 \xrightarrow{x} f\}$$

$$r = |_{x \in L(D)} x$$

字符串  $x$  对应于有向图中的路径

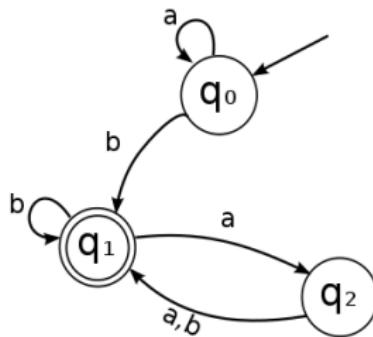


$$L(D) = \{x \mid \exists f \in F_D. s_0 \xrightarrow{x} f\}$$

$$r = |_{x \in L(D)} x$$

字符串  $x$  对应于有向图中的路径

求有向图中所有 (从初始状态到接受状态的) 路径



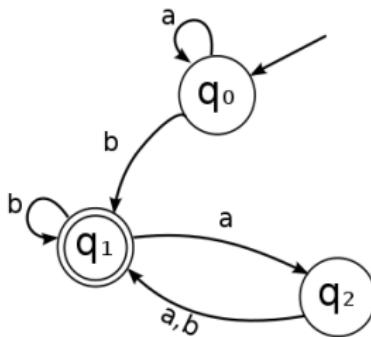
$$L(D) = \{x \mid \exists f \in F_D. s_0 \xrightarrow{x} f\}$$

$$r = |_{x \in L(D)} x$$

字符串  $x$  对应于有向图中的路径

求有向图中所有 (从初始状态到接受状态的) 路径

但是, 如果有向图中含有环, 则存在无穷多条路径



$$L(D) = \{x \mid \exists f \in F_D. s_0 \xrightarrow{x} f\}$$

$$r = |_{x \in L(D)} x$$

字符串  $x$  对应于有向图中的路径

求有向图中所有 (从初始状态到接受状态的) 路径

但是, 如果有向图中含有环, 则存在无穷多条路径

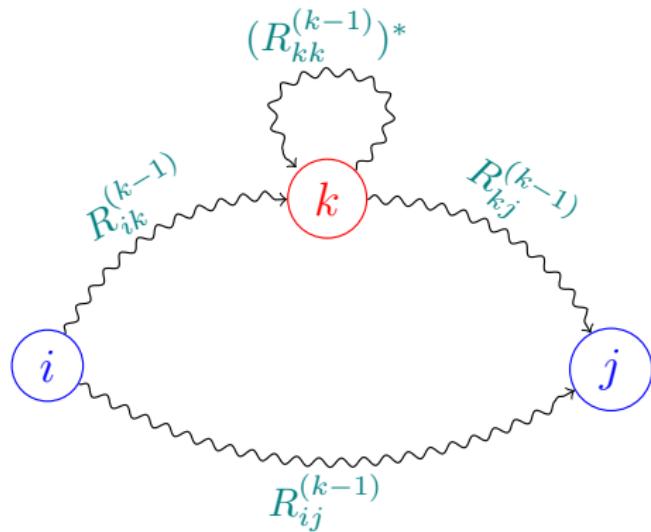
不要怕, 我们有 Kleene 闭包

假设有向图中节点编号为 0(初始状态)到  $n - 1$

$R_{ij}^k$  : 从节点  $i$  到节点  $j$ 、且**中间节点编号不大于  $k$** 的所有路径

假设有向图中节点编号为 0(初始状态)到  $n - 1$

$R_{ij}^k$  : 从节点  $i$  到节点  $j$ 、且**中间节点编号不大于  $k$** 的所有路径



$$R_{ij}^k = R_{ik}^{k-1} (R_{kk}^{(k-1)})^* R_{kj}^{k-1} \mid R_{ij}^{(k-1)}$$

# $Q$ : 如何初始化?

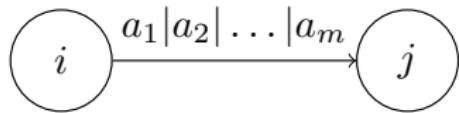
$R_{ij}^{-1}$  :

$Q$  : 如何初始化?

$R_{ij}^{-1}$  : 从节点  $i$  到节点  $j$ 、且**不经过中间节点**的所有路径

$Q$  : 如何初始化?

$R_{ij}^{-1}$  : 从节点  $i$  到节点  $j$ 、且**不经过中间节点**的所有路径



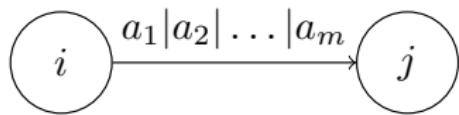
$$R_{ij}^{-1} = a_1 | a_2 | \dots | a_m$$



$$R_{ij}^{-1} = \emptyset \text{ (“无路可走”)}$$

## $Q$ : 如何初始化?

$R_{ij}^{-1}$  : 从节点  $i$  到节点  $j$ 、且**不经过中间节点**的所有路径

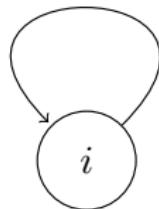


$$R_{ij}^{-1} = a_1 | a_2 | \dots | a_m$$

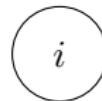


$$R_{ij}^{-1} = \emptyset \text{ (“无路可走”)}$$

$$a_1 | a_2 | \dots | a_m$$



$$R_{ii}^{-1} = a_1 | a_2 | \dots | a_m | \epsilon$$



$$R_{ii}^{-1} = \epsilon \text{ (“无需走路”)}$$

关于  $\emptyset$  (注意: 它不是正则表达式) 的规定

$$\emptyset r = r\emptyset = \emptyset$$

$$\emptyset|r = r$$

$Q : r$  的最终结果是什么?

求有向图中所有 (从初始状态到接受状态的) 路径

$Q : r$  的最终结果是什么?

求有向图中所有 (从初始状态到接受状态的) 路径

$$r = |_{s_j \in F_D} R_{0j}^{|S_D|-1}$$

```

for i = 0 to |D|-1
    for j = 0 to |D|-1
         $R_{ij}^{-1} = \{a \mid \delta(d_i, a) = d_j\}$ 
        if (i = j) then
             $R_{ij}^{-1} = R_{ij}^{-1} \cup \{\epsilon\}$ 

```

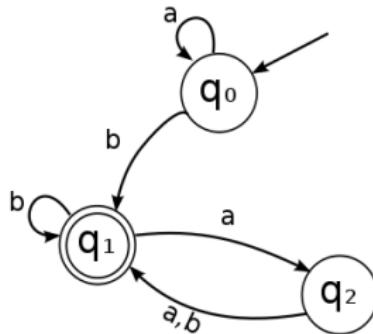
```

for k = 0 to |D|-1
    for i = 0 to |D|-1
        for j = 0 to |D|-1
             $R_{ij}^k = R_{ik}^{k-1} (R_{kk}^{k-1})^* R_{kj}^{k-1} \cup R_{ij}^{k-1}$ 

```

$$L = \bigcup_{s_j \in D_A} R_{0j}^{|D|-1}$$

$|D|$ : 状态数 ( $|S_D|$ );       $D_A$ : 接受状态集合 ( $F_D$ )



$$R_{00}^{-1} = a \mid \varepsilon$$

$$R_{01}^{-1} = b$$

$$R_{02}^{-1} = \emptyset$$

$$R_{10}^{-1} = \emptyset$$

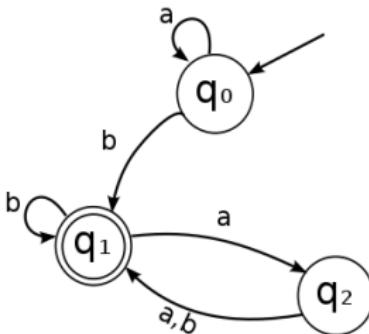
$$R_{11}^{-1} = b \mid \varepsilon$$

$$R_{12}^{-1} = a$$

$$R_{20}^{-1} = \emptyset$$

$$R_{21}^{-1} = a \mid b$$

$$R_{22}^{-1} = \varepsilon$$



### Step 0

$$R_{00}^0 = R_{00}^{-1} (R_{00}^{-1})^* R_{00}^{-1} | R_{00}^{-1} = (a | \epsilon) (a | \epsilon)^* (a | \epsilon) | a | \epsilon = a^*$$

$$R_{01}^0 = R_{00}^{-1} (R_{00}^{-1})^* R_{01}^{-1} | R_{01}^{-1} = (a | \epsilon) (a | \epsilon)^* b | b = a^* b$$

$$R_{02}^0 = R_{00}^{-1} (R_{00}^{-1})^* R_{02}^{-1} | R_{02}^{-1} = (a | \epsilon) (a | \epsilon)^* \emptyset | \emptyset = \emptyset$$

$$R_{10}^0 = R_{10}^{-1} (R_{00}^{-1})^* R_{00}^{-1} | R_{10}^{-1} = \emptyset (a | \epsilon)^* (a | \epsilon) | \emptyset = \emptyset$$

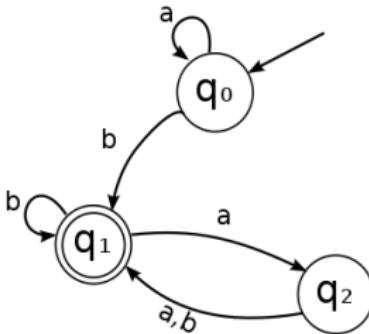
$$R_{11}^0 = R_{10}^{-1} (R_{00}^{-1})^* R_{01}^{-1} | R_{11}^{-1} = \emptyset (a | \epsilon)^* b | b | \epsilon = b | \epsilon$$

$$R_{12}^0 = R_{10}^{-1} (R_{00}^{-1})^* R_{02}^{-1} | R_{12}^{-1} = \emptyset (a | \epsilon)^* \emptyset | a = a$$

$$R_{20}^0 = R_{20}^{-1} (R_{00}^{-1})^* R_{00}^{-1} | R_{20}^{-1} = \emptyset (a | \epsilon)^* (a | \epsilon) | \emptyset = \emptyset$$

$$R_{21}^0 = R_{20}^{-1} (R_{00}^{-1})^* R_{01}^{-1} | R_{21}^{-1} = \emptyset (a | \epsilon)^* b | a | b = a | b$$

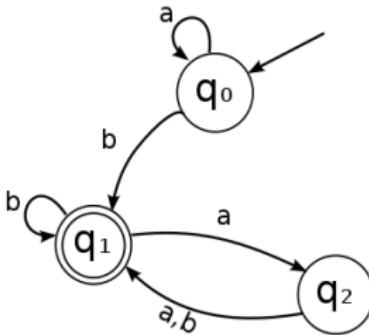
$$R_{22}^0 = R_{20}^{-1} (R_{00}^{-1})^* R_{02}^{-1} | R_{22}^{-1} = \emptyset (a | \epsilon)^* \emptyset | \epsilon = \epsilon$$



### Step 1

$R_{00}^1 = R_{01}^0 (R_{11}^0)^* R_{10}^0   R_{00}^0$	$= a^* b \quad (b   \varepsilon)^* \emptyset \quad   a^* \quad = a^*$
$R_{01}^1 = R_{01}^0 (R_{11}^0)^* R_{11}^0   R_{01}^0$	$= a^* b \quad (b   \varepsilon)^* (b   \varepsilon)   a^* b \quad = a^* b^* b$
$R_{02}^1 = R_{01}^0 (R_{11}^0)^* R_{12}^0   R_{02}^0$	$= a^* b \quad (b   \varepsilon)^* a \quad   \emptyset \quad = a^* b^* ba$
$R_{10}^1 = R_{11}^0 (R_{11}^0)^* R_{10}^0   R_{10}^0$	$= (b   \varepsilon) (b   \varepsilon)^* \emptyset \quad   \emptyset \quad = \emptyset$
$R_{11}^1 = R_{11}^0 (R_{11}^0)^* R_{11}^0   R_{11}^0$	$= (b   \varepsilon) (b   \varepsilon)^* (b   \varepsilon)   b   \varepsilon \quad = b^*$
$R_{12}^1 = R_{11}^0 (R_{11}^0)^* R_{12}^0   R_{12}^0$	$= (b   \varepsilon) (b   \varepsilon)^* a \quad   a \quad = b^* a$
$R_{20}^1 = R_{21}^0 (R_{11}^0)^* R_{10}^0   R_{20}^0$	$= (a   b) (b   \varepsilon)^* \emptyset \quad   \emptyset \quad = \emptyset$
$R_{21}^1 = R_{21}^0 (R_{11}^0)^* R_{11}^0   R_{21}^0$	$= (a   b) (b   \varepsilon)^* (b   \varepsilon)   a   b \quad = (a   b) b^*$
$R_{22}^1 = R_{21}^0 (R_{11}^0)^* R_{12}^0   R_{22}^0$	$= (a   b) (b   \varepsilon)^* a \quad   \varepsilon \quad = (a   b) b^* a   \varepsilon$

$$a^* b(a(a|b)|b)^*$$

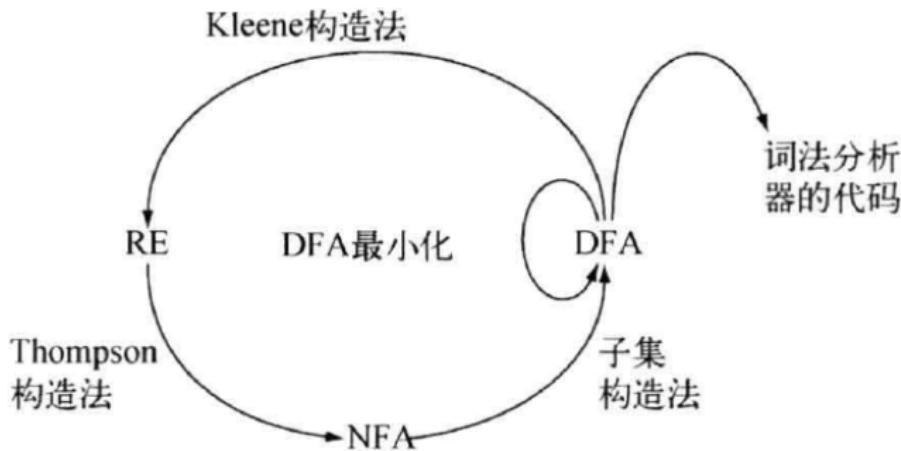


Step 2

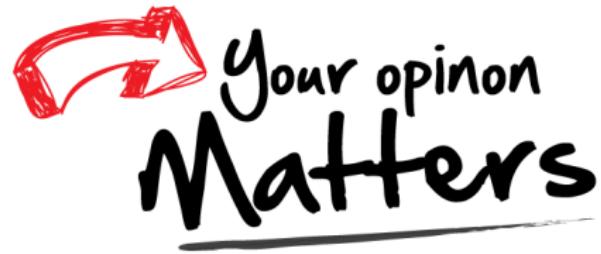
$R_{00}^2$	$= R_{02}^1 (R_{22}^1)^* R_{20}^1   R_{00}^1$	$= a^* b^* ba$	$((a b)b^* a   \epsilon)^* \emptyset$	$  a^*$	$= a^*$
$R_{01}^2$	$= R_{02}^1 (R_{22}^1)^* R_{21}^1   R_{01}^1$	$= a^* b^* ba$	$((a b)b^* a   \epsilon)^* (a b)b^*$	$  a^* b^* b$	$= a^* b(a(a b) b)^*$
$R_{02}^2$	$= R_{02}^1 (R_{22}^1)^* R_{22}^1   R_{02}^1$	$= a^* b^* ba$	$((a b)b^* a   \epsilon)^* ((a b)b^* a   \epsilon)   a^* b^* ba$		$= a^* b^* b(a(a b)b^*)^* a$
$R_{10}^2$	$= R_{12}^1 (R_{22}^1)^* R_{20}^1   R_{10}^1$	$= b^* a$	$((a b)b^* a   \epsilon)^* \emptyset$	$  \emptyset$	$= \emptyset$
$R_{11}^2$	$= R_{12}^1 (R_{22}^1)^* R_{21}^1   R_{11}^1$	$= b^* a$	$((a b)b^* a   \epsilon)^* (a b)b^*$	$  b^*$	$= (a(a b) b)^*$
$R_{12}^2$	$= R_{12}^1 (R_{22}^1)^* R_{22}^1   R_{12}^1$	$= b^* a$	$((a b)b^* a   \epsilon)^* ((a b)b^* a   \epsilon)   b^* a$		$= (a(a b) b)^* a$
$R_{20}^2$	$= R_{22}^1 (R_{22}^1)^* R_{20}^1   R_{20}^1$	$= ((a b)b^* a   \epsilon) ((a b)b^* a   \epsilon)^* \emptyset$	$  \emptyset$		$= \emptyset$
$R_{21}^2$	$= R_{22}^1 (R_{22}^1)^* R_{21}^1   R_{21}^1$	$= ((a b)b^* a   \epsilon) ((a b)b^* a   \epsilon)^* (a b)b^*$	$  (a b)b^*$		$= (a b)(a(a b) b)^*$
$R_{22}^2$	$= R_{22}^1 (R_{22}^1)^* R_{22}^1   R_{22}^1$	$= ((a b)b^* a   \epsilon) ((a b)b^* a   \epsilon)^* ((a b)b^* a   \epsilon)   (a b)b^* a   \epsilon$			$= ((a b)b^* a)^*$



# 目标: 正则表达式 RE $\Rightarrow$ 词法分析器



# Thank You!



Office 926

hfwei@nju.edu.cn