

词法分析

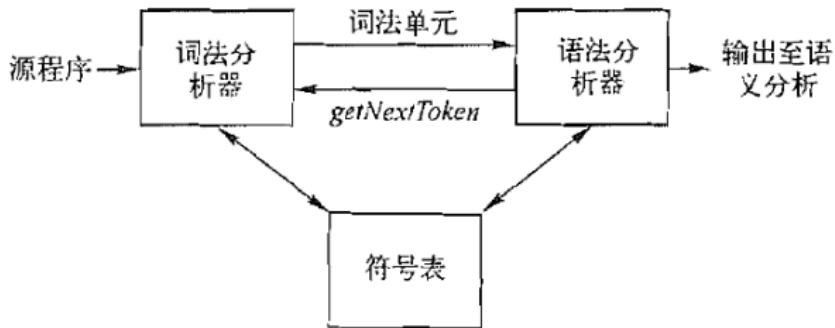
魏恒峰

hfwei@nju.edu.cn

2020 年 11 月 10 日



输入: 程序文本/字符串 s & 词法单元 (token) 的规约



输出: 词法单元流

token : <token-class, attribute-value>

词法单元	非正式描述	词素示例
if	字符 i, f	if
else	字符 e, l, s, e	else
comparison	< 或 > 或 <= 或 >= 或 == 或 !=	<=, !=
id	字母开头的字母 / 数字串	pi, score, D2
number	任何数字常量	3.14159, 0, 6.02e23
literal	在两个 "之间, 除" 以外的任何字符	"core dumped"

token : <token-class, attribute-value>

词法单元	非正式描述	词素示例
if	字符 i, f	if
else	字符 e, l, s, e	else
comparison	< 或 > 或 <= 或 >= 或 == 或 !=	<=, !=
id	字母开头的字母 / 数字串	pi, score, D2
number	任何数字常量	3.14159, 0, 6.02e23
literal	在两个 "之间, 除" 以外的任何字符	"core dumped"

int/if 关键词

ws 空格、制表符、换行符

comment “//” 开头的一行注释或者 “/* */” 包围的多行注释

```
int main(void)
{
    printf("hello, world\n");
}
```

```
int main(void)
{
    printf("hello, world\n");
}
```

int ws main/id LP void RP ws

LB ws

ws id LP literal RP SC ws

RB

```
int main(void)
{
    printf("hello, world\n");
}
```

int ws main/id LP void RP ws
LB ws
ws id LP literal RP SC ws
RB

本质上, 这就是一个**字符串 (匹配/识别) 算法**

词法分析器的三种设计方法



手写词法分析器



词法分析器的生成器



自动化词法分析器

生产环境下的编译器 (如 gcc) 通常选择手写词法分析器



手写词法分析器

识别字符串 s 中符合某种词法单元模式的所有词素

```
if ab42>=42  
      xyz =3.14  
else xyz = 2.718
```

ws if else id integer real relop assign

识别字符串 s 中符合某种词法单元模式的所有词素

```
if ab42>=42  
      xyz =3.14  
else xyz = 2.718
```

ws if else id integer real relop assign

识别字符串 s 中符合某种词法单元模式的**开头第一个词素**

识别字符串 s 中符合某种词法单元模式的所有词素

```
if ab42>=42  
      xyz =3.14  
else xyz = 2.718
```

ws if else id integer real relop assign

识别字符串 s 中符合某种词法单元模式的**开头第一个词素**

识别字符串 s 中符合**特定词法单元模式**的**开头第一个词素**

识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

- 4) public int line = 1;
- 5) private char peek = ' ';
- 6) private Hashtable words = new Hashtable();

line: 行号, 用于调试

peek: 下一个向前看字符 (Lookahead)

words: 从词素到词法单元**标识符或关键词**的映射表

识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

ws: blank tab newline

识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

ws: blank tab newline

```
12)     public Token scan() throws IOException {  
13)         for( ; ; peek = (char)System.in.read() ) {  
14)             if( peek == ' ' || peek == '\t' ) continue;  
15)             else if( peek == '\n' ) line = line + 1;  
16)             else break;  
17)     }
```

识别空白部分, 但不做处理

识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

ws: blank tab newline

```
12)     public Token scan() throws IOException {  
13)         for( ; ; peek = (char)System.in.read() ) {  
14)             if( peek == ' ' || peek == '\t' ) continue;  
15)             else if( peek == '\n' ) line = line + 1;  
16)             else break;  
17)     }
```

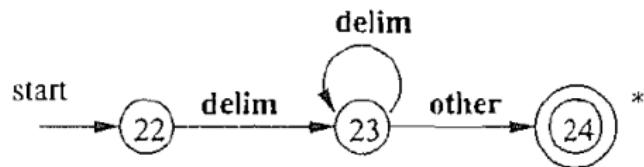
识别空白部分, 但不做处理

```
peek = next input character;  
while (peek != null) {  
    // if peek is not a ws, break  
    peek = next input character  
}
```

Q : 这样写, 可不可以?

识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

ws: blank tab newline



用于识别**空白符**的状态转移图

识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

num: 整数 (允许以 0 开头)

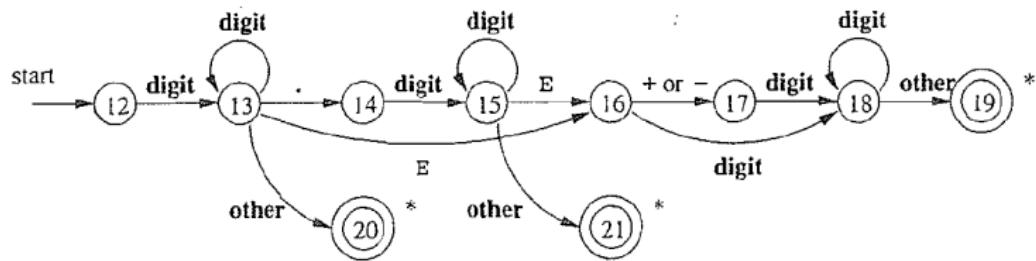
识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

num: 整数 (允许以 0 开头)

```
18)     if( Character.isDigit(peek) ) {  
19)         int v = 0;  
20)         do {  
21)             v = 10*v + Character.digit(peek, 10);  
22)             peek = (char)System.in.read();  
23)         } while( Character.isDigit(peek) );  
24)         return new Num(v);  
25)     }
```

识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

num: 整数 (允许以 0 开头)



用于识别**带科学计数法的浮点数**的状态转移图

识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

id: 字母开头的字母/数字串

识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

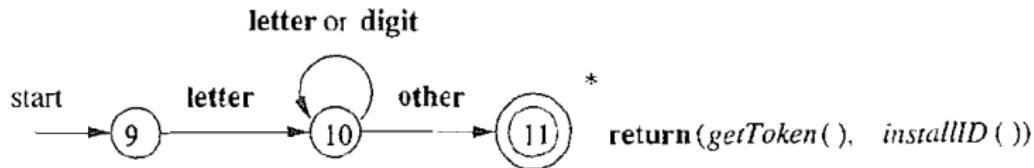
id: 字母开头的字母/数字串

```
26)     if( Character.isLetter(peek) ) {  
27)         StringBuffer b = new StringBuffer();  
28)         do {  
29)             b.append(peek);  
30)             peek = (char)System.in.read();  
31)         } while( Character.isLetterOrDigit(peek) );  
32)         String s = b.toString();  
33)         Word w = (Word)words.get(s);  
34)         if( w != null ) return w;  
35)         w = new Word(Tag.ID, s);  
36)         words.put(s, w);  
37)     return w;  
38) }
```

识别词素、判断是否是预留的关键字、保存该标识符

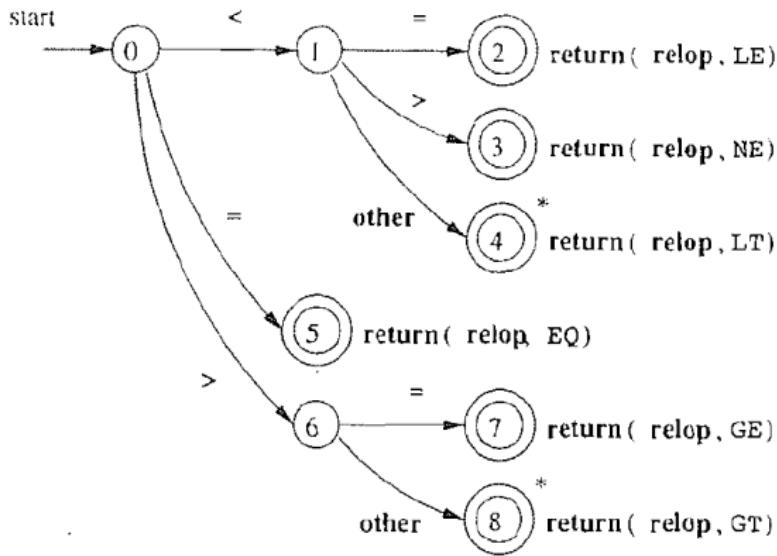
识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

id: 字母开头的字母/数字串



识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

relop: < > <= >= == <>

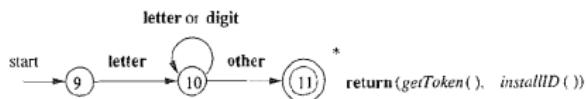
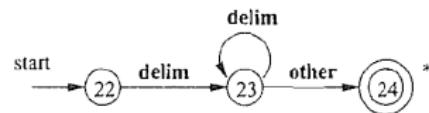
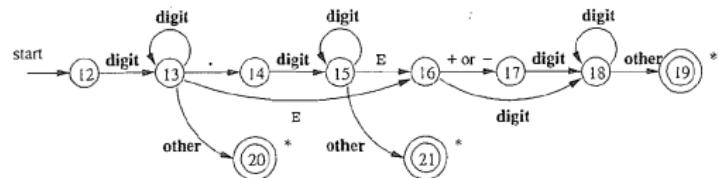
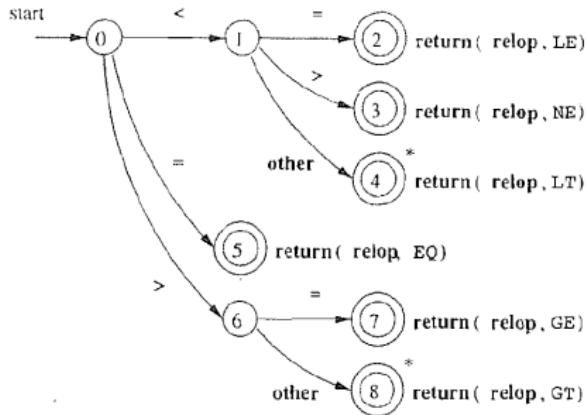


识别字符串 s 中符合**特定词法单元模式**的开头第一个词素

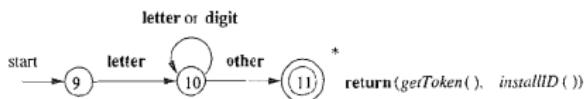
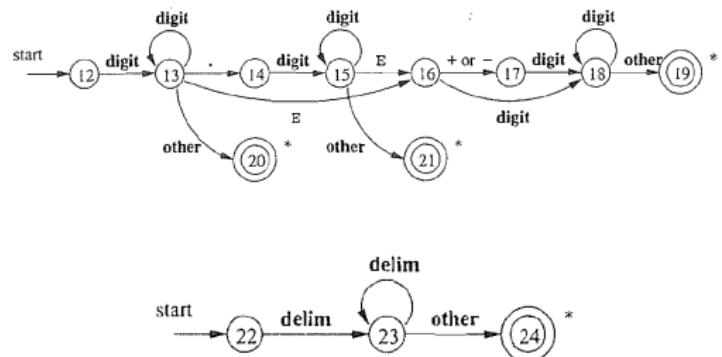
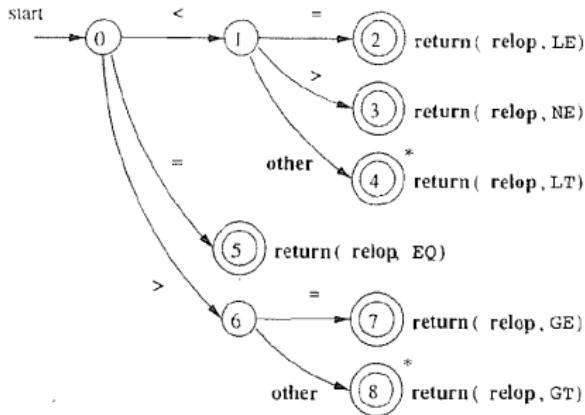
```
39)      Token t = new Token(peek);  
40)      peek = ' ';  
41)      return t;
```

出现**词法错误**, 直接报告异常字符

识别字符串 s 中符合某种词法单元模式的开头第一个词素 (SCAN())



识别字符串 s 中符合某种词法单元模式的开头第一个词素 (SCAN())



关键点：根据下一个字符即可判定词法单元的类型

否则，报告该字符有误，并忽略该字符

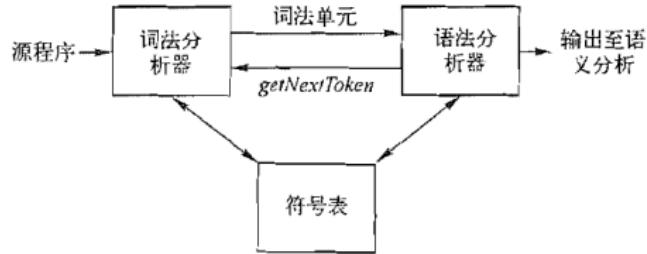
```
1) package lexer;           //文件 Lexer.java
2) import java.io.*; import java.util.*;
3) public class Lexer {
4)     public int line = 1;
5)     private char peek = ' ';
6)     private Hashtable words = new Hashtable();
7)     void reserve(Word t) { words.put(t.lexeme, t); }
8)     public Lexer() {
9)         reserve( new Word(Tag.TRUE, "true") );
10)        reserve( new Word(Tag.FALSE, "false") );
11)    }
12)    public Token scan() throws IOException {
13)        for( ; ; peek = (char)System.in.read() ) {
14)            if( peek == ' ' || peek == '\t' ) continue;
15)            else if( peek == '\n' ) line = line + 1;
16)            else break;
17)    }
}
```

```
18)     if( Character.isDigit(peek) ) {
19)         int v = 0;
20)         do {
21)             v = 10*v + Character.digit(peek, 10);
22)             peek = (char)System.in.read();
23)         } while( Character.isDigit(peek) );
24)         return new Num(v);
25)     }
26)     if( Character.isLetter(peek) ) {
27)         StringBuffer b = new StringBuffer();
28)         do {
29)             b.append(peek);
30)             peek = (char)System.in.read();
31)         } while( Character.isLetterOrDigit(peek) );
32)         String s = b.toString();
33)         Word w = (Word)words.get(s);
34)         if( w != null ) return w;
35)         w = new Word(Tag.ID, s);
36)         words.put(s, w);
37)         return w;
38)     }
39)     Token t = new Token(peek);
40)     peek = ' ';
41)     return t;
42) }
43) }
```

识别字符串 s 中符合某种词法单元模式的**所有词素**

外层**循环**调用 `SCAN()`

或者，由语法分析器**按需**调用 `SCAN()`



```
12)     public Token scan() throws IOException {
13)         for( ; ; peek = (char)System.in.read() ) {
14)             if( peek == ' ' || peek == '\t' ) continue;
15)             else if( peek == '\n' ) line = line + 1;
16)             else break;
17)     }
```

```
12)     public Token scan() throws IOException {  
13)         for( ; ; peek = (char)System.in.read() ) {  
14)             if( peek == ' ' || peek == '\t' ) continue;  
15)             else if( peek == '\n' ) line = line + 1;  
16)             else break;  
17)     }
```

```
peek = next input character;  
while (peek != null) {  
    // if peek is not a ws, break  
    peek = next input character  
}
```

```
12)     public Token scan() throws IOException {  
13)         for( ; ; peek = (char)System.in.read() ) {  
14)             if( peek == ' ' || peek == '\t' ) continue;  
15)             else if( peek == '\n' ) line = line + 1;  
16)             else break;  
17)     }
```

```
peek = next input character;  
while (peek != null) {  
    // if peek is not a ws, break  
    peek = next input character  
}
```

char peek = ' ': 下一个向前看字符

```
12)     public Token scan() throws IOException {  
13)         for( ; ; peek = (char)System.in.read() ) {  
14)             if( peek == ' ' || peek == '\t' ) continue;  
15)             else if( peek == '\n' ) line = line + 1;  
16)             else break;  
17)     }
```

```
peek = next input character;  
while (peek != null) {  
    // if peek is not a ws, break  
    peek = next input character  
}
```

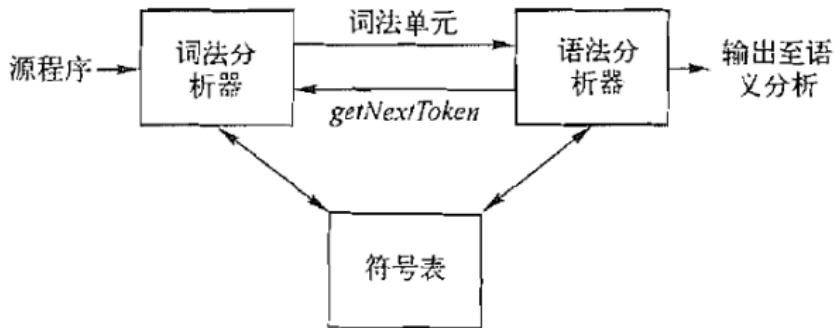
char peek = ' ': 下一个向前看字符

循环不变式:

当SCAN() 返回一个词法单元时,
peek 是空白符或者是当前词素后的第一个字符

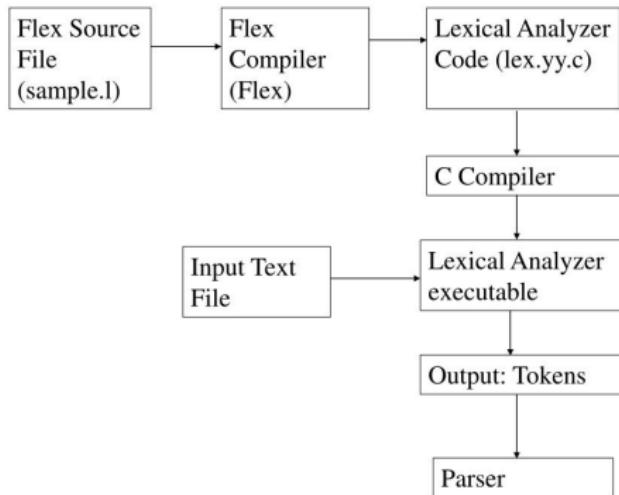


输入: 程序文本/字符串 s & 词法单元的规约



输出: 词法单元流

输入: 词法单元的规约

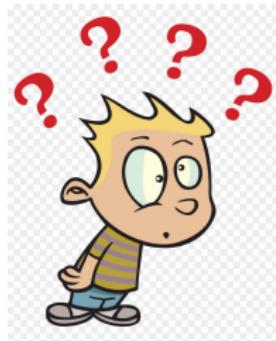


输出: 词法分析器

词法单元的规约

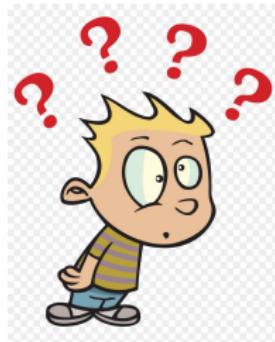
词法单元	非正式描述	词素示例
if	字符 i, f	if
else	字符 e, l, s, e	else
comparison	< 或 > 或 <= 或 >= 或 == 或 !=	<=, !=
id	字母开头的字母 / 数字串	pi, score, D2
number	任何数字常量	3.14159, 0, 6.02e23
literal	在两个 "之间, 除" 以外的任何字符	"core dumped"

词法单元的规约



词法单元	非正式描述	词素示例
if	字符 i, f	if
else	字符 e, l, s, e	else
comparison	< 或 > 或 <= 或 >= 或 == 或 !=	<=, !=
id	字母开头的字母 / 数字串	pi, score, D2
number	任何数字常量	3.14159, 0, 6.02e23
literal	在两个 "之间, 除" 以外的任何字符	"core dumped"

词法单元的规约



词法单元	非正式描述	词素示例
if	字符 i, f	if
else	字符 e, l, s, e	else
comparison	< 或 > 或 <= 或 >= 或 == 或 !=	<=, !=
id	字母开头的字母 / 数字串	pi, score, D2
number	任何数字常量	3.14159, 0, 6.02e23
literal	在两个 "之间, 除" 以外的任何字符	"core dumped"

我们需要词法单元的**形式化**规约

id: 字母开头的字母/数字串

id 定义了一个集合, 我们称之为**语言 (Language)**

它使用了字母与数字等符号集合, 我们称之为**字母表 (Alphabet)**

该语言中的每个元素 (即, 标识符) 称为**串 (String)**

Definition (字母表)

字母表 Σ 是一个有限的符号集合。



Definition (串)

字母表 Σ 上的**串** (s) 是由 Σ 中符号构成的一个**有穷**序列。

ϵ

空串 : $|\epsilon| = 0$

Definition (串上的“连接”运算)

$x = \text{dog}, y = \text{house} \quad xy = \text{doghouse}$

$$s\epsilon = \epsilon s = s$$

Definition (串上的“连接”运算)

$$x = \text{dog}, y = \text{house} \quad xy = \text{doghouse}$$

$$s\epsilon = \epsilon s = s$$

Definition (串上的“指数”运算)

$$s^0 \triangleq \epsilon$$

$$s^i \triangleq ss^{i-1}, i > 0$$

Definition (语言)

语言是给定字母表 Σ 上一个任意的可数的串集合。

\emptyset

$\{\epsilon\}$

Definition (语言)

语言是给定字母表 Σ 上一个任意的可数的串集合。

\emptyset

$\{\epsilon\}$

id : $\{a, b, c, a1, a2, \dots\}$

ws : {blank, tab, newline}

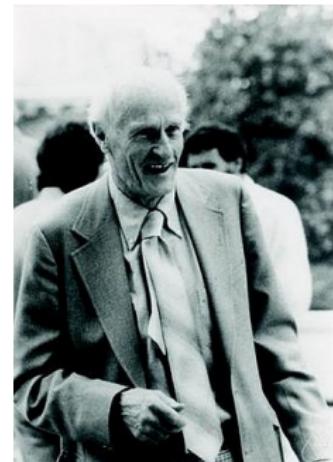
if : $\{if\}$

语言是串的集合

因此, 我们可以通过集合操作构造新的语言。

运算	定义和表示
L 和 M 的并	$L \cup M \doteq \{s \mid s \text{ 属于 } L \text{ 或者 } s \text{ 属于 } M\}$
L 和 M 的连接	$LM = \{st \mid s \text{ 属于 } L \text{ 且 } t \text{ 属于 } M\}$
L 的 Kleene 闭包	$L^* = \bigcup_{i=0}^{\infty} L^i$
L 的正闭包	$L^+ = \bigcup_{i=1}^{\infty} L^i$

L^* 允许我们构造无穷集合



Stephen Kleene
(1909 ~ 1994)

$$L = \{A, B, \dots, Z, a, b, \dots, z\}$$

$$D = \{0, 1, \dots, 9\}$$

运算	定义和表示
L 和 M 的并	$L \cup M \doteq \{s \mid s \text{ 属于 } L \text{ 或者 } s \text{ 属于 } M\}$
L 和 M 的连接	$LM = \{st \mid s \text{ 属于 } L \text{ 且 } t \text{ 属于 } M\}$
L 的 Kleene 闭包	$L^* = \bigcup_{i=0}^{\infty} L^i$
L 的正闭包	$L^+ = \bigcup_{i=1}^{\infty} L^i$

$$L = \{A, B, \dots, Z, a, b, \dots, z\}$$

$$D = \{0, 1, \dots, 9\}$$

运算	定义和表示
L 和 M 的并	$L \cup M \doteq \{s \mid s \text{ 属于 } L \text{ 或者 } s \text{ 属于 } M\}$
L 和 M 的连接	$LM = \{st \mid s \text{ 属于 } L \text{ 且 } t \text{ 属于 } M\}$
L 的 Kleene 闭包	$L^* = \bigcup_{i=0}^{\infty} L^i$
L 的正闭包	$L^+ = \bigcup_{i=1}^{\infty} L^i$

$$L \cup D \quad LD \quad L^4 \quad L^* \quad D^+$$

$$L(L \cup D)^*$$

$$L = \{A, B, \dots, Z, a, b, \dots, z\}$$

$$D = \{0, 1, \dots, 9\}$$

运算	定义和表示
L 和 M 的并	$L \cup M \doteq \{s \mid s \text{ 属于 } L \text{ 或者 } s \text{ 属于 } M\}$
L 和 M 的连接	$LM = \{st \mid s \text{ 属于 } L \text{ 且 } t \text{ 属于 } M\}$
L 的 Kleene 闭包	$L^* = \bigcup_{i=0}^{\infty} L^i$
L 的正闭包	$L^+ = \bigcup_{i=1}^{\infty} L^i$

$$L \cup D \quad LD \quad L^4 \quad L^* \quad D^+$$

$$L(L \cup D)^* \quad \text{标识符}$$

id : $L(L \cup D)^*$

如何更简洁地描述该 **id** 语言?

id : $L(L \cup D)^*$

如何更简洁地描述该 **id** 语言?



下面向大家隆重介绍简洁、优雅、强大的**正则表达式**

每个正则表达式 r 对应一个正则语言 $L(r)$



正则表达式是语法, 正则语言是语义

Definition (正则表达式)

给定字母表 Σ , Σ 上的正则表达式由且仅由以下规则定义:

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

Definition (正则表达式)

给定字母表 Σ , Σ 上的正则表达式由且仅由以下规则定义:

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

运算优先级: () $\succ *$ \succ 连接 \succ |

$$(a)|((b)^*(c)) \equiv a|b^*c$$

Definition (正则表达式)

给定字母表 Σ , Σ 上的正则表达式由且仅由以下规则定义:

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

运算优先级: $() \succ * \succ \text{连接} \succ |$

$$(a)|((b)^*(c)) \equiv a|b^*c$$

结构 (Structure)

每个正则表达式 r 对应一个正则语言 $L(r)$

Definition (正则表达式对应的正则语言)

$$L(\epsilon) = \{\epsilon\} \quad (1)$$

$$L(a) = \{a\}, \forall a \in \Sigma \quad (2)$$

$$L((r)) = L(r) \quad (3)$$

$$L(r|s) = L(r) \cup L(s) \quad L(rs) = L(r)L(s) \quad L(r^*) = (L(r))^* \quad (4)$$

$$\Sigma = \{a, b\}$$

$$L(a|b) = \{a, b\}$$

$$\Sigma = \{a, b\}$$

$$L(a|b) = \{a, b\}$$

$$L((a|b)(a|b))$$

$$\Sigma = \{a, b\}$$

$$L(a|b) = \{a, b\}$$

$$L((a|b)(a|b))$$

$$L(a^*)$$

$$\Sigma = \{a, b\}$$

$$L(a|b) = \{a, b\}$$

$$L((a|b)(a|b))$$

$$L(a^*)$$

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

$$\Sigma = \{a, b\}$$

$$L(a|b) = \{a, b\}$$

$$L((a|b)(a|b))$$

$$L(a^*)$$

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

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



表达式	匹配	例子
c	单个非运算符字符 c	a
\c	字符 c 的字面值	*
"s"	串 s 的字面值	"**"
.	除换行符以外的任何字符	a.*b
^	一行的开始	^abc
\$	行的结尾	abc\$
[s]	字符串 s 中的任何一个字符	[abc]
[^s]	不在串 s 中的任何一个字符	[^abc]
r*	和 r 匹配的零个或多个串连接成的串	a*
r+	和 r 匹配的一个或多个串连接成的串	a+
r?	零个或一个 r	a?
r{m,n}	最少 m 个, 最多 n 个 r 的重复出现	a{1,5}
r ₁ r ₂	r ₁ 后加上 r ₂	ab
r ₁ r ₂	r ₁ 或 r ₂	a b
(r)	与 r 相同	(a b)
r ₁ /r ₂	后面跟有 r ₂ 时的 r ₁	abc/123

正则定义与简记法

Vim	Java	ASCII	Description
	\p{ASCII}	[\x00-\x7F]	ASCII characters
	\p{Alnum}	[A-Za-z0-9]	Alphanumeric characters
\w	\w	[A-Za-z0-9_]	Alphanumeric characters plus "_"
\W	\W	[^A-Za-z0-9_]	Non-word characters
\a	\p{Alpha}	[A-Za-z]	Alphabetic characters
\s	\p{Blank}	[\t]	Space and tab
\< \>	\b	(?=<\W) (?=\w) (?=<\w) (?=\W)	Word boundaries
	\B	(?=<\W) (?=\W) (?=<\w) (?=\w)	Non-word boundaries
	\p{Cntrl}	[\x00-\x1F\x7F]	Control characters
\d	\p{Digit} or \d	[0-9]	Digits
\D	\D	[^0-9]	Non-digits
	\p{Graph}	[\x21-\x7E]	Visible characters
\l	\p{Lower}	[a-z]	Lowercase letters
\p	\p{Print}	[\x20-\x7E]	Visible characters and the space character
	\p{Punct}	[!"#\$%&'()*+,./:;<=>?@\\^_`{ }~-]	Punctuation characters
\s	\p{Space} or \s	[\t\r\n\v\f]	Whitespace characters
\S	\S	[^ \t\r\n\v\f]	Non-whitespace characters
\u	\p{Upper}	[A-Z]	Uppercase letters
\x	\p{XDigit}	[A-Fa-f0-9]	Hexadecimal digits

老虎不发威
当我是Hello Kitty

C 语言中的标识符?

C 语言中的标识符?

REGULAR EXPRESSION

```
: / ^[a-zA-Z_]( [a-zA-Z\d])* $
```

TEST STRING

```
_setlibpath
__setlibpath
hello123
hello123world
123hello
```

整数部分 . 小数部分 E/e 指数部分

整数部分 . 小数部分 E/e 指数部分

REGULAR EXPRESSION

6 matches, 225 steps (~1ms)

`^ / \d+(\.\d+)?([Ee][+-]?\d+)?`

/ gm

TEST STRING

```
42 // The Answer to the Ultimate Question of Life, The Universe, and
Everything
2.99792458e8    // speed of light in vacuum
6.62606957E-34 // Planck constant
1.602176565e-19 // elementary charge
3.1415          // pi
2.718           // e|
```

C 语言中单行注释对应的正则表达式?

C 语言中单行注释对应的正则表达式？

REGULAR EXPRESSION

```
:// \/\ /[^\\n]*\\n?
```

TEST STRING

```
// this is a comment
int main () { // this is a comment
} // this is a comment
```

C 语言中**多行注释**对应的正则表达式?

C 语言中多行注释对应的正则表达式？

REGULAR EXPRESSION

4 matches, 422 steps (~1ms)

```
/* / \/* ([^/*]|\/*+[^/*]) * \*/
```

/ gm

TEST STRING

```
/* this is a multi-line comment */
/* // this is a nested multi-line comment */
/* this is a multi-line comment which really
spans multiple lines */
int main () { // this is a comment
} // this is a comment
/*
    this is a multi-line comment
    with * and / in it
*/
```

$$\left(0|(1(01^*0)^*1)\right)^*$$



<https://regex101.com/r/ED4qgC/1>

Flex 程序的结构 (.l 文件)

声明部分: 直接拷贝到 .c 文件中

声明部分

% %

转换规则: 正则表达式 {动作}

转换规则

辅助函数: 动作中使用的辅助函数

% %

辅助函数

```
%{  
    /* definitions of manifest constants  
    LT, LE, EQ, NE, GT, GE,  
    IF, THEN, ELSE, ID, NUMBER, RELOP */  
}  
  
/* regular definitions */  
delim      [ \t\n]  
ws         {delim}+  
letter     [A-Za-z]  
digit      [0-9]  
id          {letter}({letter}|{digit})*  
number     {digit}+(\.{digit}+)?(E[+-]?)?{digit}+)?  
  
%%
```

%%

```
{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}       {yyval = (int) installID(); return(ID);}
{number}   {yyval = (int) installNum(); return(NUMBER);}
"<"       {yyval = LT; return(RELOP);}
"<="      {yyval = LE; return(RELOP);}
"="       {yyval = EQ; return(RELOP);}
"<>"     {yyval = NE; return(RELOP);}
">"      {yyval = GT; return(RELOP);}
">="      {yyval = GE; return(RELOP);}

%%
```

```
%%

int installID() {/* function to install the lexeme, whose
                  first character is pointed to by yytext,
                  and whose length is yylen, into the
                  symbol table and return a pointer
                  thereto */
}

int installNum() /* similar to installID, but puts numerical
                  constants into a separate table */
}
```

```
8 /* fb1-1 just like unix wc (wordcount) */
9 %{
10 int chars = 0;
11 int words = 0;
12 int lines = 0;
13 %}
14
15 %%
16
17 [^ \t\n\r\f\v]+ { words++; chars += strlen(yytext); }
18 \n { chars++; lines++; }
19 . { chars++; }
20
21 %%
22
23 int main() {
24     yylex();
25     printf("%8d%8d%8d\n", lines, words, chars);
26 }
```

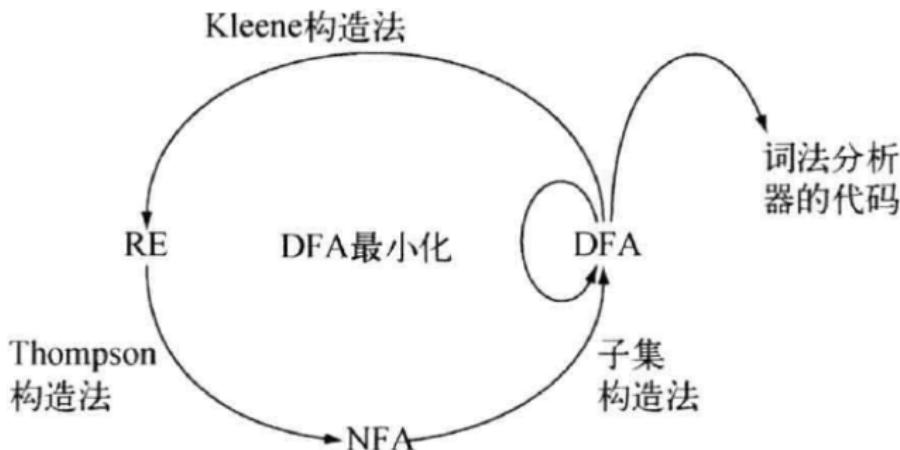
两大冲突解决规则

最前优先匹配: 关键字

最长优先匹配: “ $>=$ ”, “ifhappy”



目标: 正则表达式 RE \Rightarrow 词法分析器



$$RE \Rightarrow \epsilon\text{-NFA} \Rightarrow NFA$$

终点固然令人向往, 这一路上的风景更是美不胜收

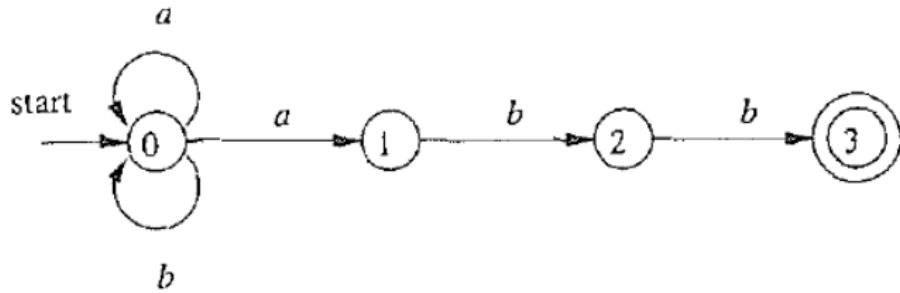
Definition (NFA (Nondeterministic Finite Automaton))

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 Σ ($\epsilon \notin \Sigma$)
- (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** δ

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

- (5) 接受状态集合 $F \subseteq S$





Michael O. Rabin
(1931 ~)

Finite Automata and Their Decision Problems[†]

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.”*

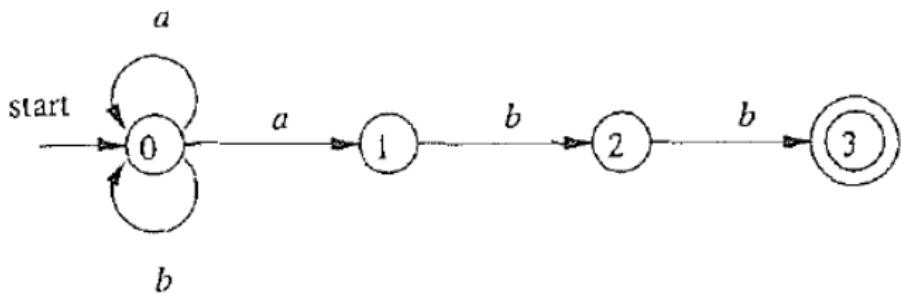
(非确定性) 有穷自动机是一类极其简单的**计算**装置

它可以**识别** (接受/拒绝) Σ 上的字符串

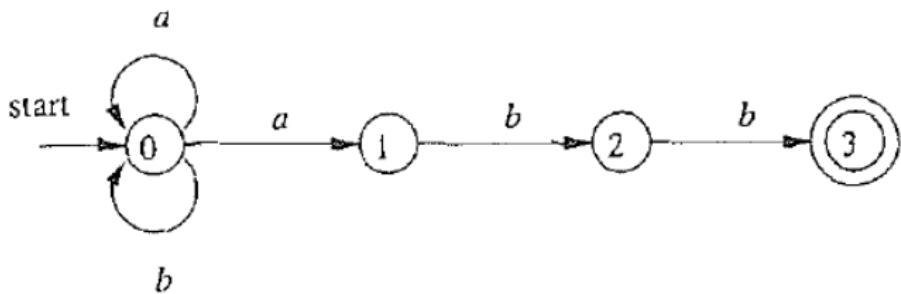
Definition (接受 (Accept))

(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



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



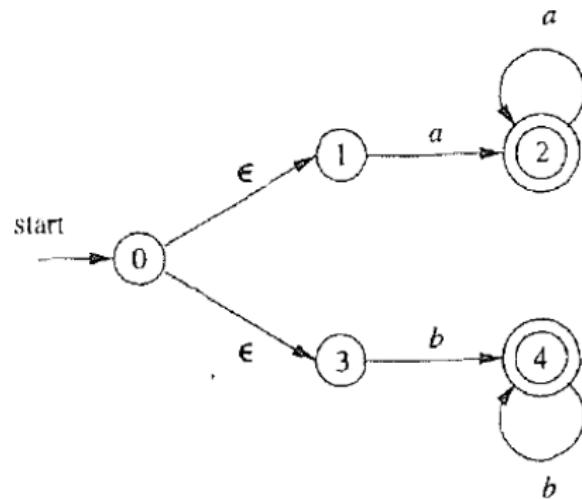
$$aab \in L(\mathcal{A}) \quad ababab \notin L(\mathcal{A})$$

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

关于语言与自动机 \mathcal{A} 的两个最基本的问题:

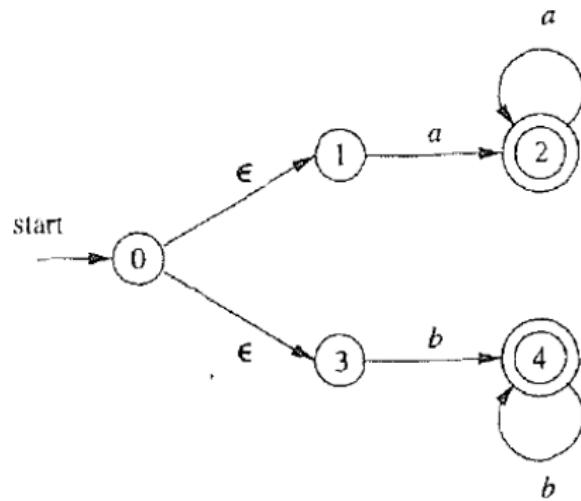
给定字符串 $x, x \in L(\mathcal{A})$?

$L(\mathcal{A})$ 究竟是什么?



$aaa \in \mathcal{A}?$

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



$aaa \in \mathcal{A}?$

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

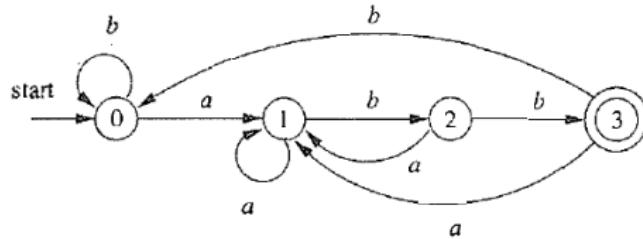
Definition (DFA (Deterministic Finite Automaton))

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 Σ ($\epsilon \notin \Sigma$)
- (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** δ

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

- (5) 接受状态集合 $F \subseteq S$



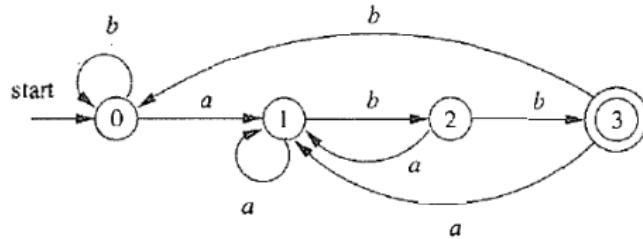
Definition (DFA (Deterministic Finite Automaton))

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

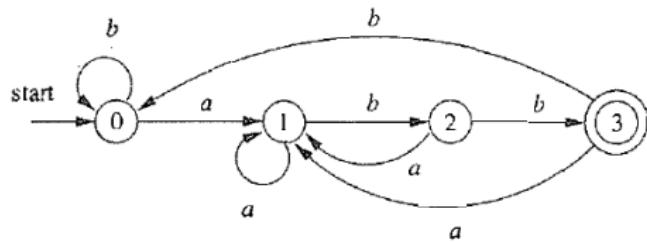
- (1) 字母表 Σ ($\epsilon \notin \Sigma$)
- (2) **有穷**的状态集合 S
- (3) **唯一**的初始状态 $s_0 \in S$
- (4) 状态转移**函数** δ

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

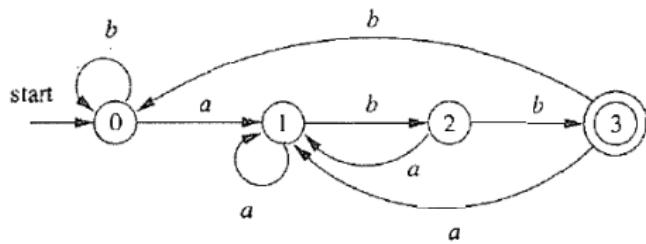
- (5) 接受状态集合 $F \subseteq S$



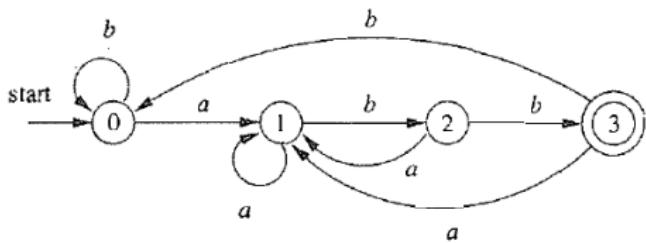
约定: 所有没有对应出边的字符默认指向一个不存在的“死状态”



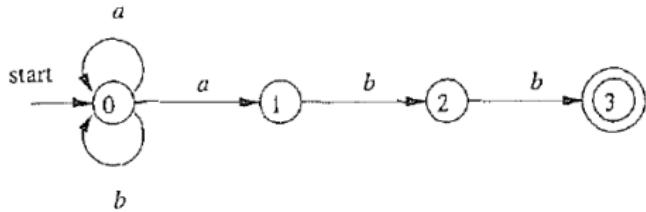
$$L(\mathcal{A}) =$$



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



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



NFA 与 DFA 的**优缺点**比较:

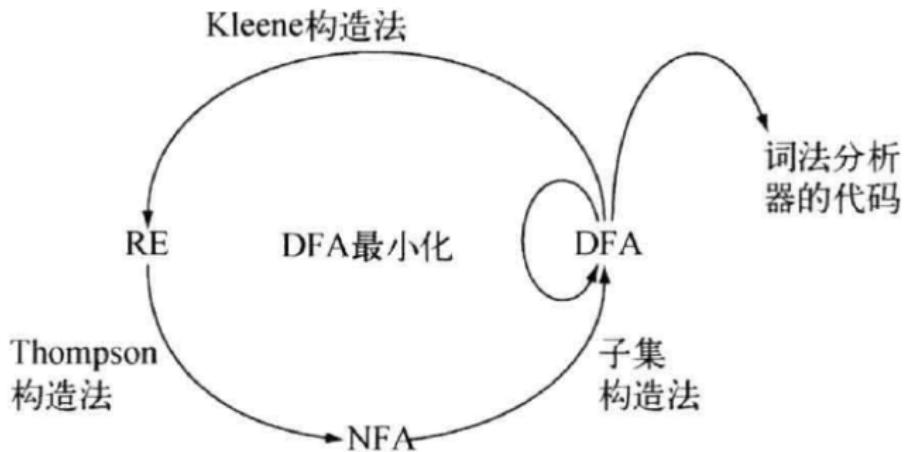
$x \in L(\mathcal{A})$: DFA 易于实现; NFA 不易实现

$L(\mathcal{A})$: NFA 简洁易于理解; DFA 状态多转移多

取长补短:

用 NFA 描述, 用 NFA 实现;

从 NFA 到 DFA 的转化自动完成



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 Σ , Σ 上的正则表达式**由且仅由**以下规则定义:

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

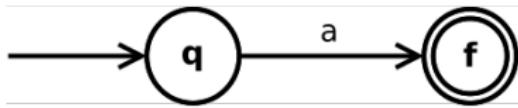
ϵ 是正则表达式。

ϵ 是正则表达式。



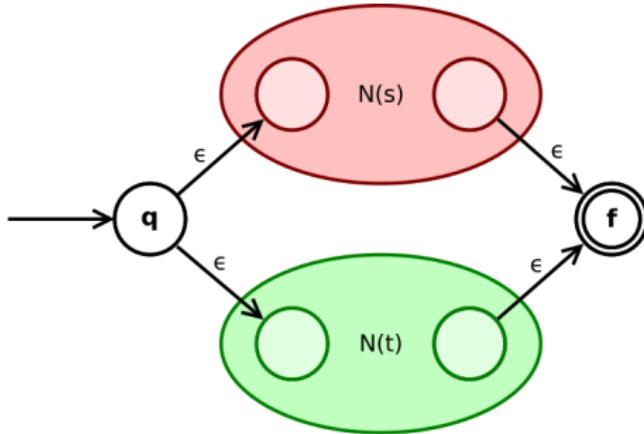
$a \in \Sigma$ 是正则表达式。

$a \in \Sigma$ 是正则表达式。



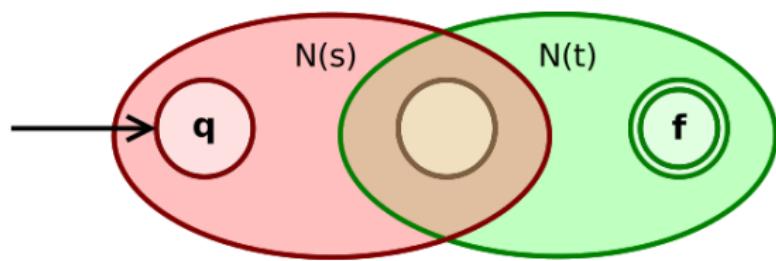
如果 s, t 是正则表达式, 则 $s|t$ 是正则表达式。

如果 s, t 是正则表达式, 则 $s|t$ 是正则表达式。



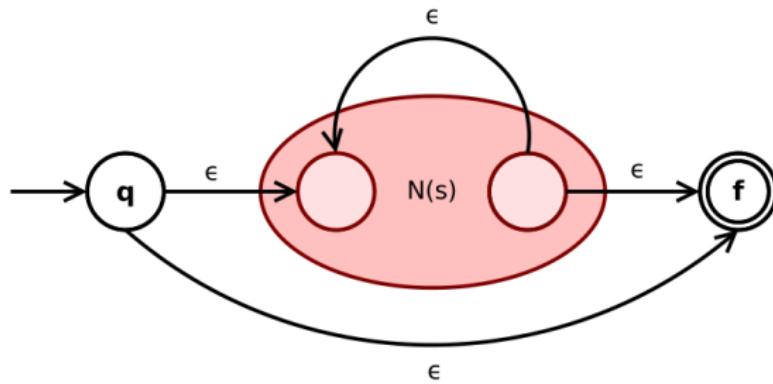
如果 s, t 是正则表达式, 则 st 是正则表达式。

如果 s, t 是正则表达式, 则 st 是正则表达式。



如果 s , 则 s^* 是正则表达式。

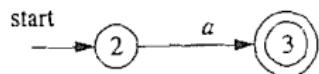
如果 s , 则 s^* 是正则表达式。



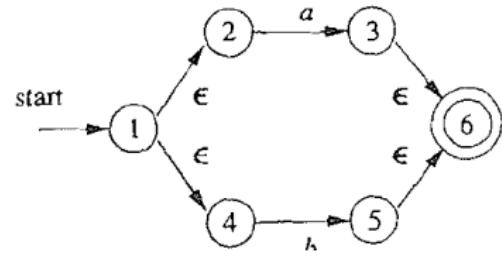
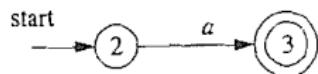
Thompson 构造法得到的 NFA 的**性质**以及**复杂度分析**

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

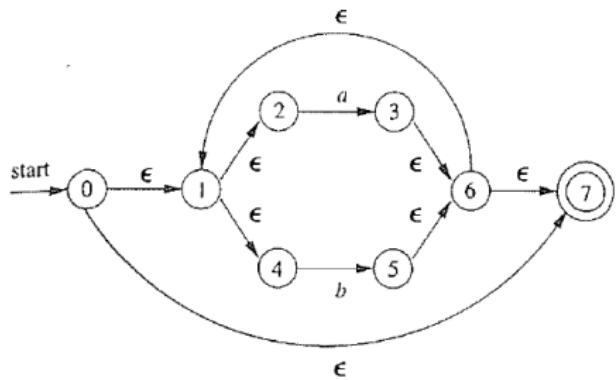
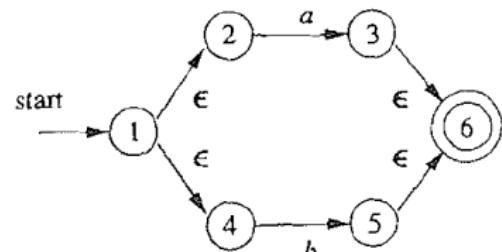
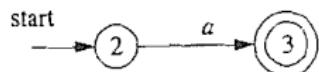
$$r = (a|b)^*abb$$



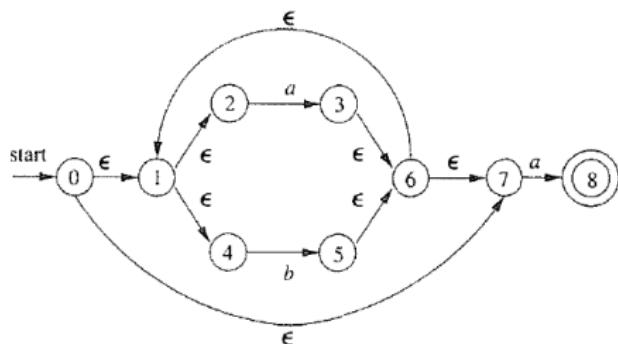
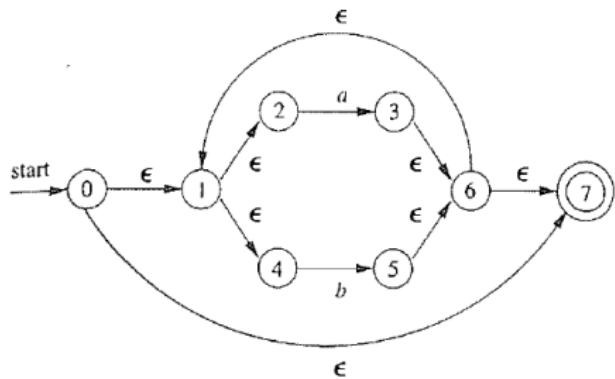
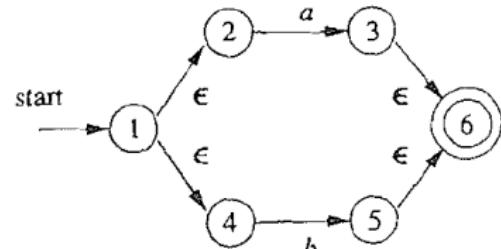
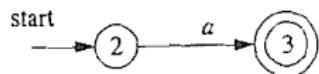
$$r = (a|b)^*abb$$



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



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



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



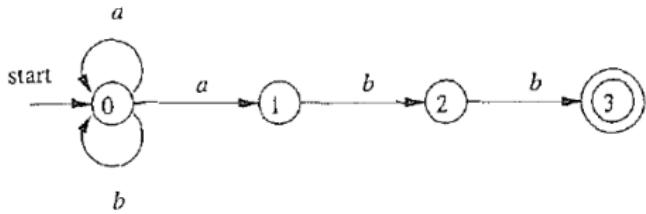
Michael O. Rabin (1931 ~)



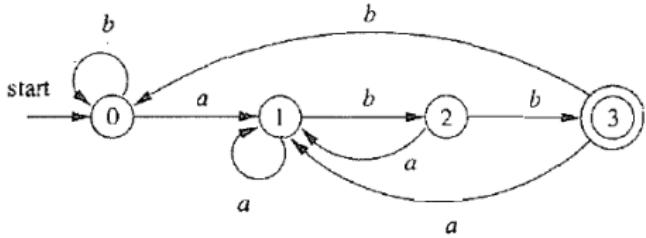
Dana Scott (1932 ~)

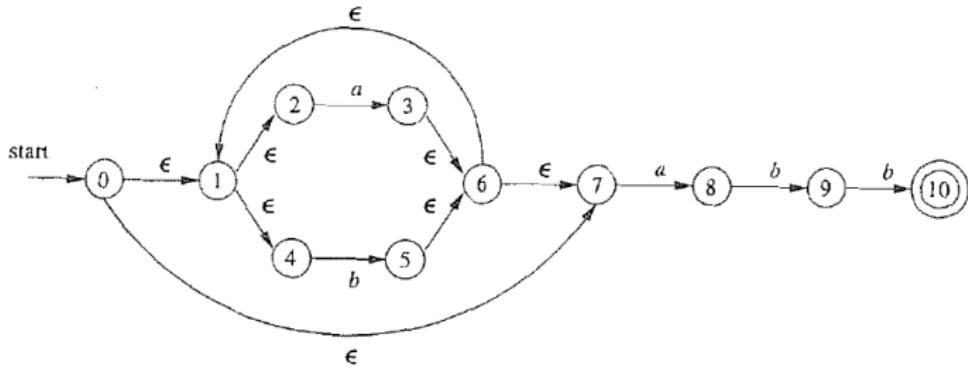
思想: 用 DFA 模拟 NFA

用 DFA 模拟 NFA

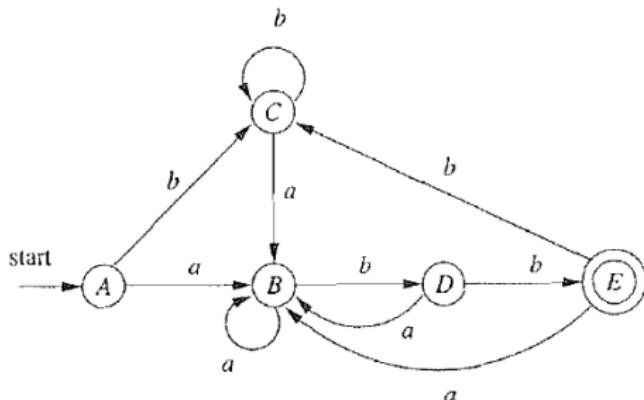


$$L(\mathcal{A}) = L((a|b)^*abb)$$

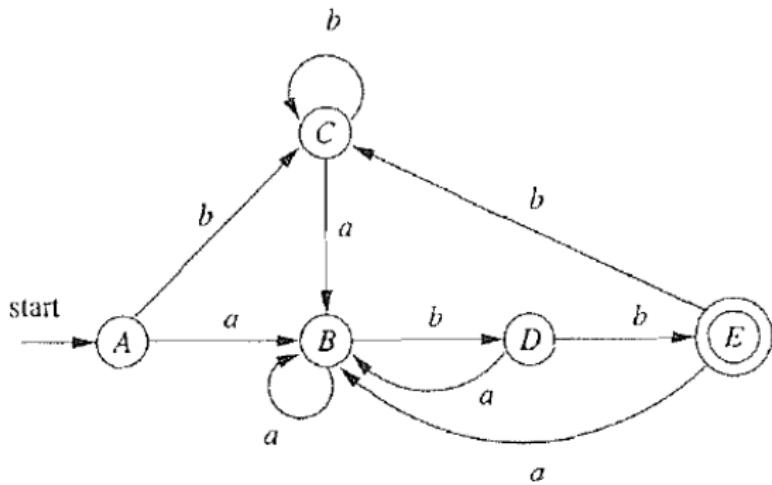




$$L(\mathcal{A}) = L((a|b)^*abb)$$



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



ϵ -closure(s)

ϵ -closure(s)

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

ϵ -closure(s)

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

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

子集构造法 ($\mathcal{N} \Rightarrow \mathcal{D}$) 实现时采用**标记搜索**过程

```
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  $a$  ) {  
         $U = \epsilon\text{-closure}(move(T, a))$ ;  
        if (  $U$  不在  $Dstates$  中 )  
            将  $U$  加入到  $Dstates$  中, 且不加标记;  
         $Dtran[T, a] = U$ ;  
    }  
}
```

接受状态集 $F_{\mathcal{D}} = \{s \in S_{\mathcal{D}} \mid \exists f \in F_{\mathcal{N}}. f \in s\}$

子集构造法的**复杂度分析**:
 $(|S_{\mathcal{N}}| = n)$

$$\Theta(2^n)$$

子集构造法的**复杂度分析**:
 $(|S_{\mathcal{N}}| = n)$

$$\Theta(2^n)$$

最坏情况下, 复杂度为 $\Omega(2^n)$

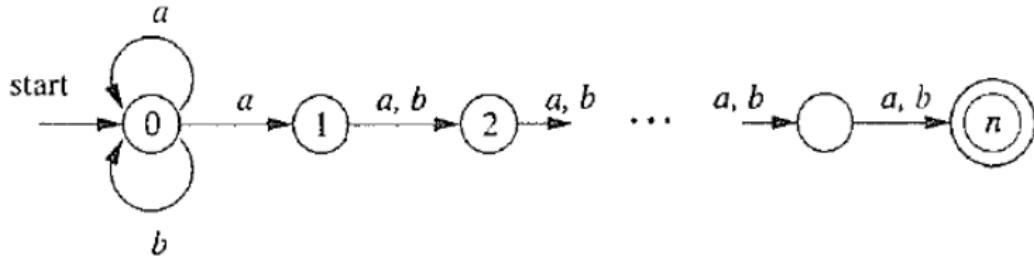
“长度为 $m \geq n$ 个字符的 a, b 串, 且倒数第 n 个字符是 a ”

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

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

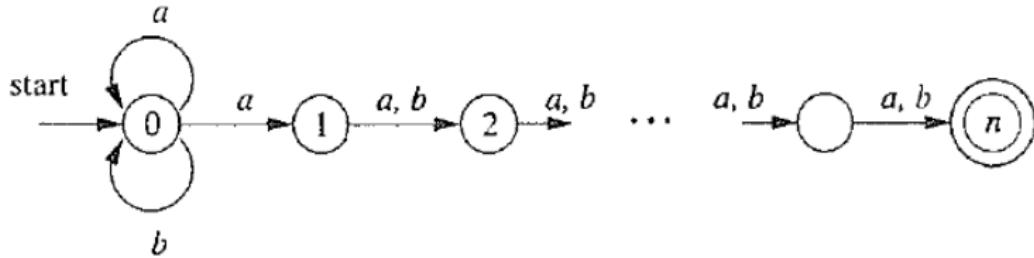
“长度为 $m \geq n$ 个字符的 a, b 串，且倒数第 n 个字符是 a ”

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

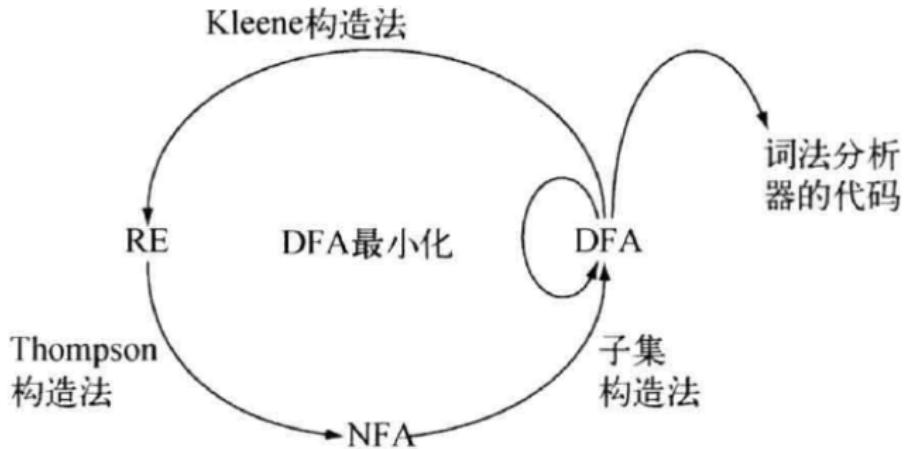


“长度为 $m \geq n$ 个字符的 a, b 串，且倒数第 n 个字符是 a ”

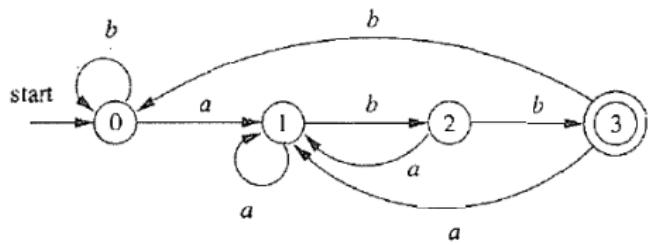
$$L_n = (a|b)^* a (a|b)^{n-1}$$



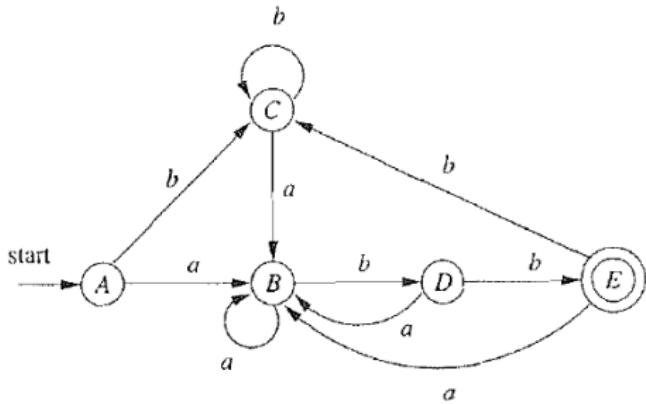
作业: $m = n = 3$



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

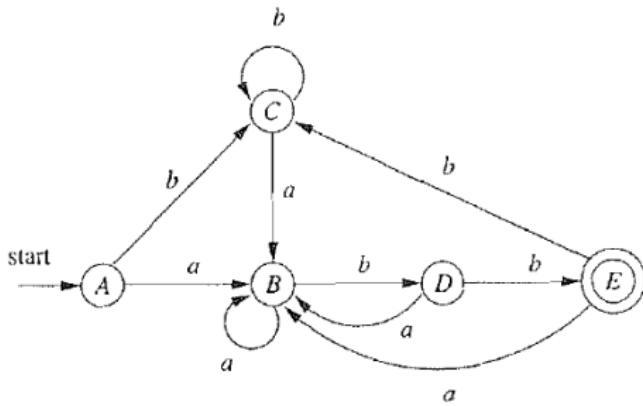


John Hopcroft (1939 ~)

*“for fundamental achievements in the design and analysis
of **algorithms and data structures**”*

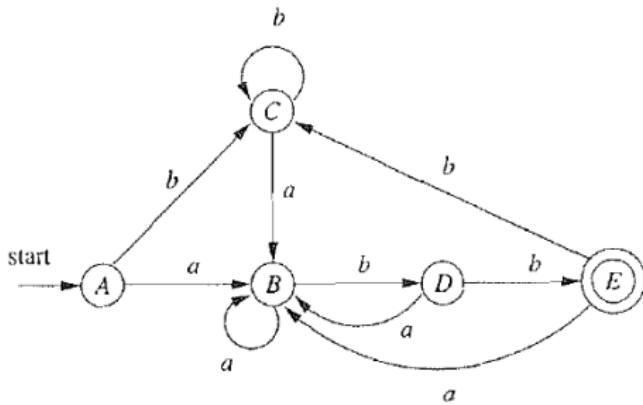
— Turing Award, 1986

如何定义等价状态?



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

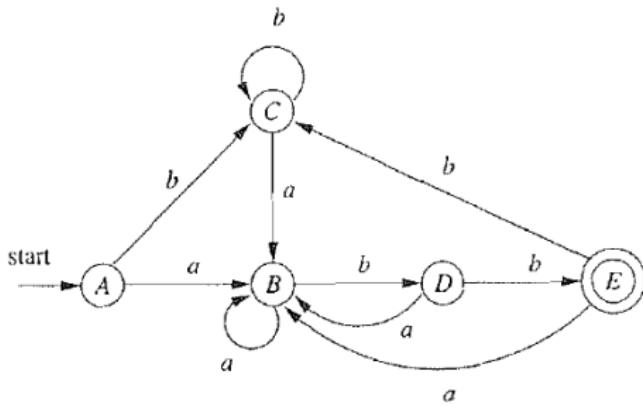
如何定义等价状态?



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

从哪里开始?

如何定义等价状态?



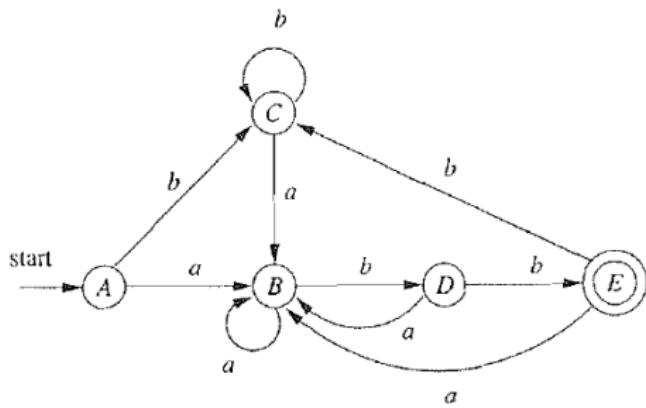
$$s \sim t \iff \forall a \in \Sigma. (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \wedge (s' \sim t').$$

从哪里开始?

$$\forall f, f' \in F. f \sim f'.$$

$$s \sim t \iff \forall a \in \Sigma. (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \wedge (\textcolor{red}{s' \sim t'}).$$

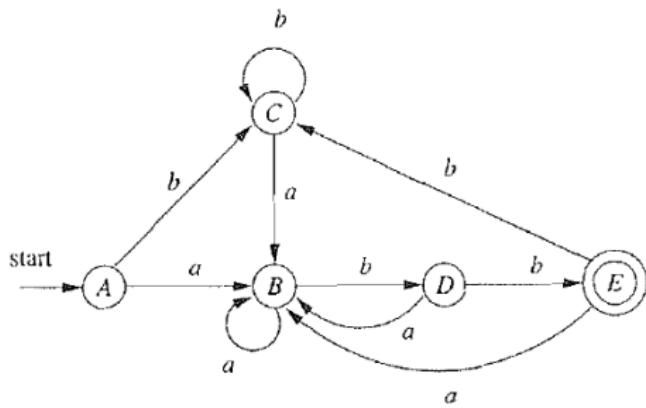
划分操作



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

$$s \sim t \iff \forall a \in \Sigma. (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \wedge (\textcolor{red}{s' \sim t'}).$$

划分操作



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

DFA 最小化等价状态划分方法

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

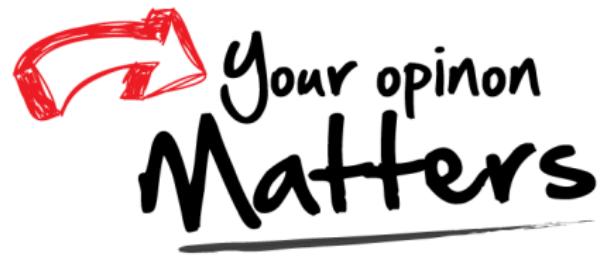
```
最初, 令  $\Pi_{\text{new}} = \Pi$ ;
for ( $\Pi$  中的每个组  $G$ ) {
    将  $G$  分划为更小的组, 使得两个状态  $s$  和  $t$  在同一小组中当且仅当对于所有
        的输入符号  $a$ , 状态  $s$  和  $t$  在  $a$  上的转换都到达  $\Pi$  中的同一组;
    /* 在最坏情况下, 每个状态各自组成一个组 */
    在  $\Pi_{\text{new}}$  中将  $G$  替换为对  $G$  进行分划得到的那些小组;
}
```

直到再也无法划分为止

特定于词法分析器的最小化方法

最小化的唯一性?

Thank You!



Office 926

hfwei@nju.edu.cn