

词法分析

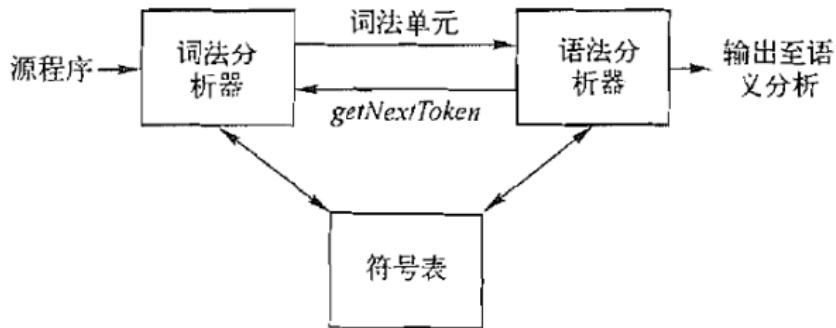
魏恒峰

hfwei@nju.edu.cn

2020 年 11 月 15 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



master

[gcc / gcc / c-family / c-lex.c](#)



iains Objective-C/C++ : Improve '@' keyword locations. ...

19 contributors



+7

1435 lines (1278 sloc) | 38.8 KB

master

[gcc / libcpp / lex.c](#)



urnathan cpplib: EOF in pragmas ...

25 contributors



4364 lines (3825 sloc) | 119 KB

我们着重介绍词法分析原理, 具体方案与 *GCC* 有差异

识别字符串 s 中符合某种词法单元模式的所有词素

```
if ab42>=3.14
    xyz :=2.99792458E8
else
    xyz:= 2.718
    abc := 1024
```

ws if else id integer real sci rellop 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>=3.14
    xyz :=2.99792458E8
else
    xyz:= 2.718
    abc := 1024
```

ws if else id integer real sci rellop assign ($:=$)

识别字符串 s 中符合某种词法单元模式的前缀词素

识别字符串 s 中符合某种词法单元模式的所有词素

```
if ab42>=3.14
    xyz :=2.99792458E8
else
    xyz:= 2.718
abc := 1024
```

ws if else id integer real sci rellop assign ($:=$)

识别字符串 s 中符合某种词法单元模式的前缀词素

识别字符串 s 中符合特定词法单元模式的前缀词素

识别字符串 s 中符合**特定词法单元模式**的前缀词素

分支: 先判断属于哪一类, 然后进入特定词法单元的前缀词素匹配流程

识别字符串 s 中符合某种词法单元模式的**前缀词素**

循环: 返回当前识别出来的词法单元与词素, 继续识别下一个前缀词素

识别字符串 s 中符合某种词法单元模式的所有词素

先: ws if else id integer

然后: relop

最后: real sci

留给大家: assign (:=)

识别字符串 s 中符合**特定词法单元模式**的前缀词素

- 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

words.put("if", if)

words.put("else", else)

识别字符串 s 中符合**特定词法单元模式**的前缀词素

ws: blank tab newline

识别字符串 s 中符合**特定词法单元模式**的前缀词素

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，并忽略之

识别字符串 s 中符合**特定词法单元模式**的前缀词素

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)     }
```

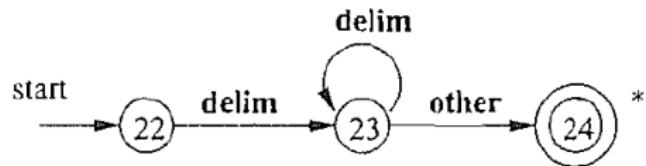
识别空白部分，并忽略之

```
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**的前缀词素

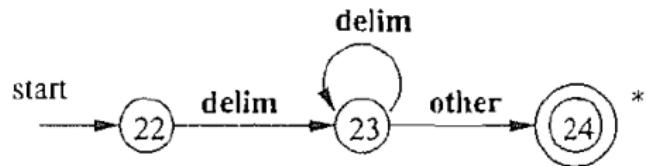
ws: blank tab newline



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**的前缀词素

ws: blank tab newline

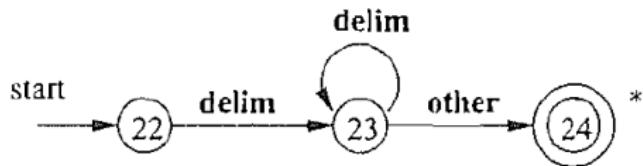


用于识别**空白符**的状态转移图

*: 识别出的空白符**不包含**当前 peek 指向的字符

识别字符串 s 中符合**特定词法单元模式**的前缀词素

ws: blank tab newline



用于识别**空白符**的状态转移图

*: 识别出的空白符**不包含**当前 peek 指向的字符

22: 碰到 other 怎么办?

识别字符串 s 中符合**特定词法单元模式**的前缀词素

integer: 整数 (简化处理, 允许以 0 开头)

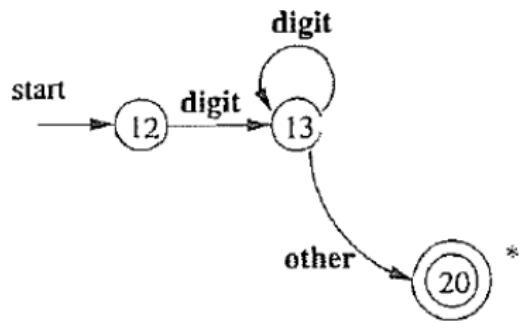
识别字符串 s 中符合**特定词法单元模式**的前缀词素

integer: 整数 (简化处理, 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**的前缀词素

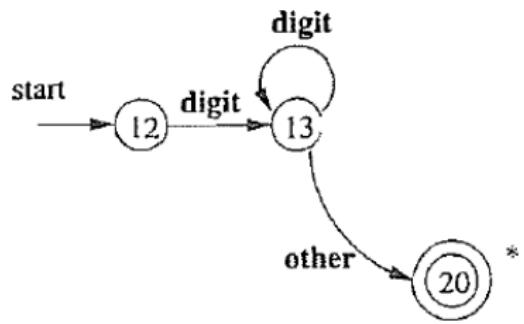
integer: 整数 (简化处理, 允许以 0 开头)



用于识别**integer**的状态转移图

识别字符串 s 中符合**特定词法单元模式**的前缀词素

integer: 整数 (简化处理, 允许以 0 开头)



用于识别**integer**的状态转移图

12: 碰到 other 怎么办?

识别字符串 s 中符合**特定词法单元模式**的前缀词素

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**的前缀词素

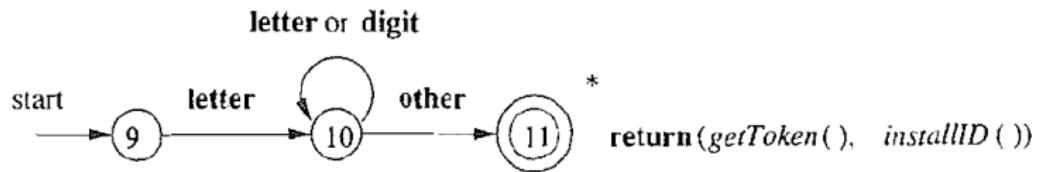
id: 字母开头的字母/数字串

```
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或已识别的标识符、保存该标识符

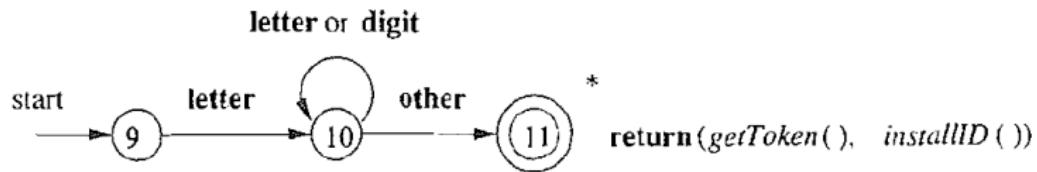
识别字符串 s 中符合**特定词法单元模式**的前缀词素

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**的前缀词素

id: 字母开头的字母/数字串



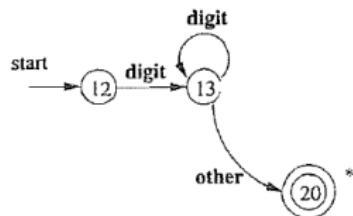
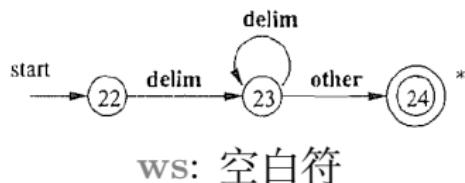
9 : 碰到 other 怎么办?

识别字符串 s 中符合**特定词法单元模式**的前缀词素

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

错误处理模块：出现**词法错误**，直接报告异常字符

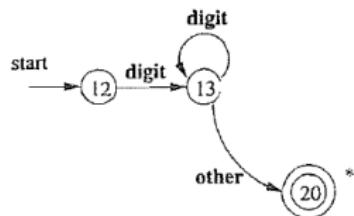
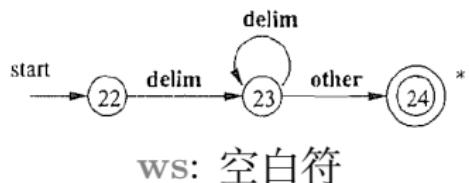
识别字符串 s 中符合某种词法单元模式的前缀词素 (SCAN())



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

错误处理模块

识别字符串 s 中符合某种词法单元模式的前缀词素 (SCAN())



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

错误处理模块

integer: 整数

关键点: 合并 22, 12, 9, 根据下一个字符即可判定词法单元的类型

否则, 调用错误处理模块 (对应 other), 报告该字符有误, 并忽略该字符

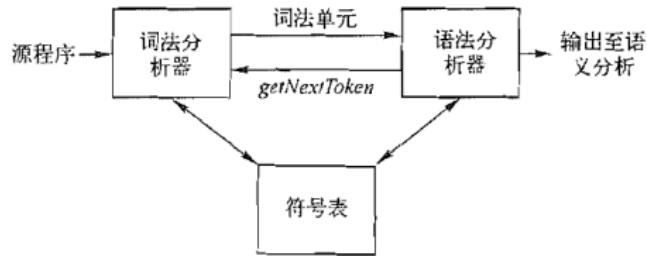
```
1) package lexer;           //文件 Lexer.java
2) import java.io.*; import java.util.*;
3) public class Lexer {
4)     public int line = 1;
5)     private char peek = ' ';
6)     private Hashtable words = new Hashtable();
7)     void reserve(Word t) { words.put(t.lexeme, t); }
8)     public Lexer() {
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)     }
```

```
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

考慮例子“123abc”

```
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

考慮例子“123abc”

$\langle \text{number}, 123 \rangle$ $\langle \text{id}, abc \rangle$

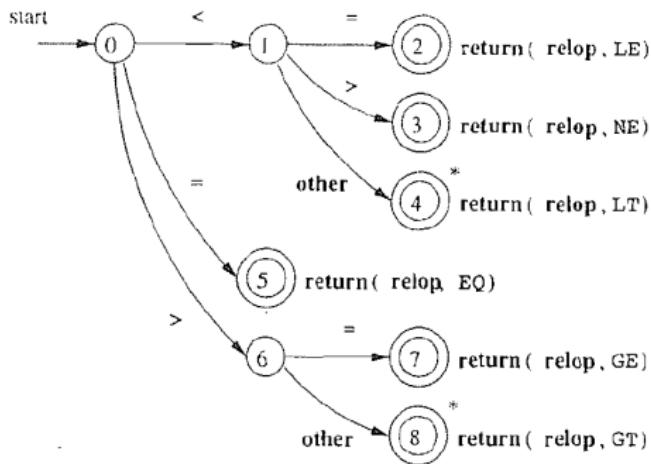
识别字符串 s 中符合**特定词法单元模式的前缀词素**

relop¹: < > <= >= = <>

¹此处,= 是判断是否相等的关系运算符。请考虑,如果 = 表示赋值, == 表示相等判断,该如何设计词法分析器?

识别字符串 s 中符合**特定词法单元模式的前缀词素**

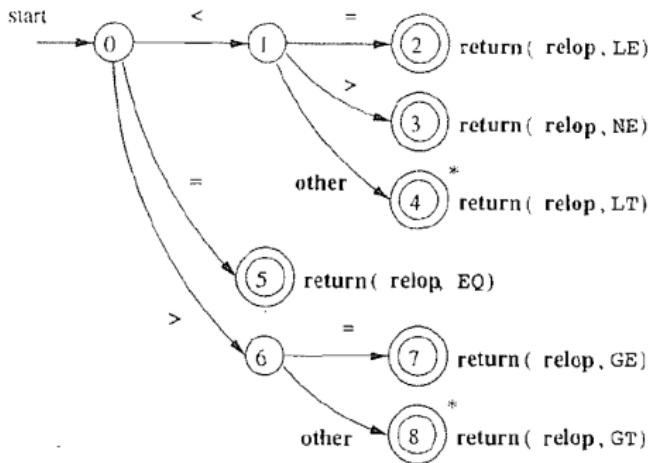
relop¹: < > <= >= = <>



¹此处,= 是判断是否相等的关系运算符。请考虑,如果 = 表示赋值, == 表示相等判断,该如何设计词法分析器?

识别字符串 s 中符合**特定词法单元模式的前缀词素**

relop¹: < > <= >= = <>

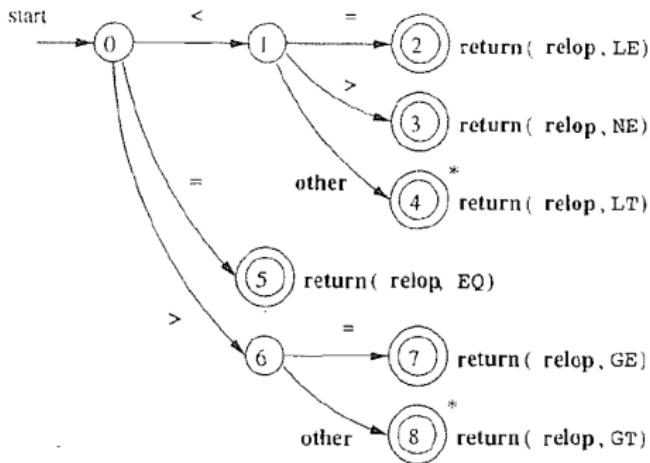


“最长优先原则”：例如，识别出 $<=$ ，而不是 $<$ 与 $=$

¹ 此处, $=$ 是判断是否相等的关系运算符。请考虑, 如果 $=$ 表示赋值, $==$ 表示相等判断, 该如何设计词法分析器?

识别字符串 s 中符合**特定词法单元模式的前缀词素**

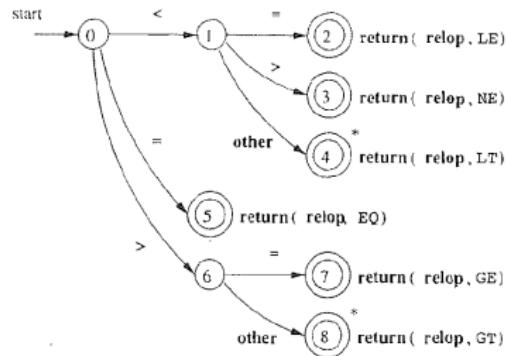
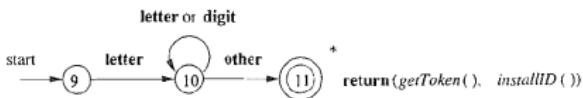
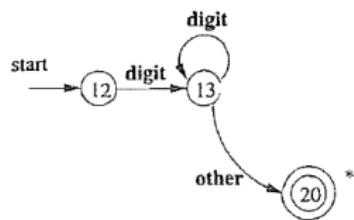
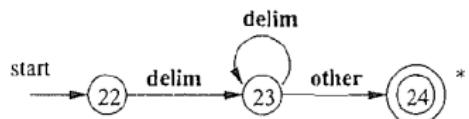
relop¹: < > <= >= = <>



“最长优先原则”：例如，识别出 $<=$ ，而不是 $<$ 与 $=$
0：碰到 other 怎么办？

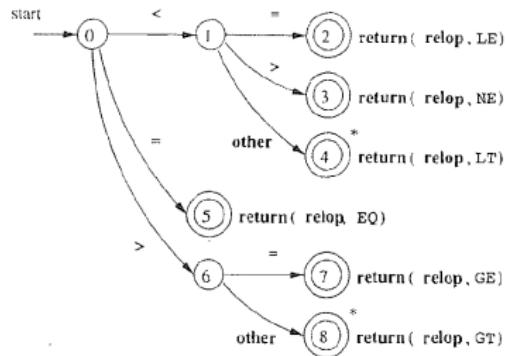
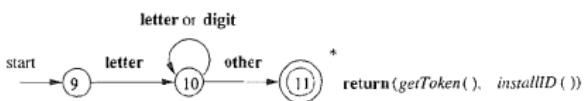
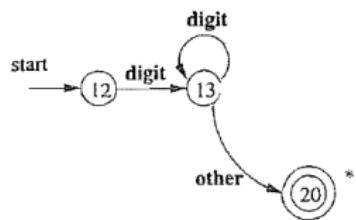
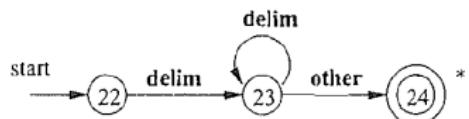
¹此处， $=$ 是判断是否相等的关系运算符。请考虑，如果 $=$ 表示赋值， $==$ 表示相等判断，该如何设计词法分析器？

识别字符串 s 中符合某种词法单元模式的前缀词素 (SCAN())



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

识别字符串 s 中符合某种词法单元模式的前缀词素 (SCAN())



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

关键点: 合并 22, 12, 9, 0, 根据**下一个字符**即可判定词法单元的类型

否则, 调用错误处理模块 (对应 other), 报告**该字符有误**, 并忽略该字符

但是，词法分析器的设计并没有这么容易



ws if else id integer relop

根据下一个字符即可判定词法单元的类型

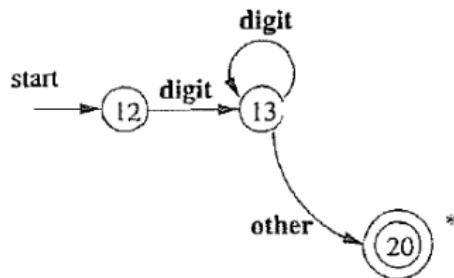
每个状态转移图的每个状态要么是接受状态, 要么带有other 边

ws if else id integer relop

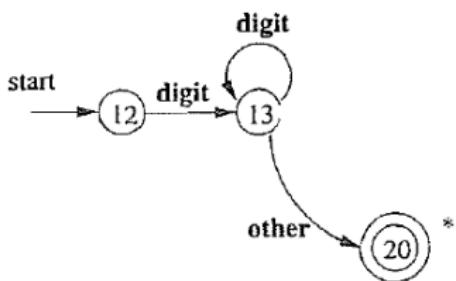
根据下一个字符即可判定词法单元的类型

每个状态转移图的每个状态要么是接受状态, 要么带有other 边

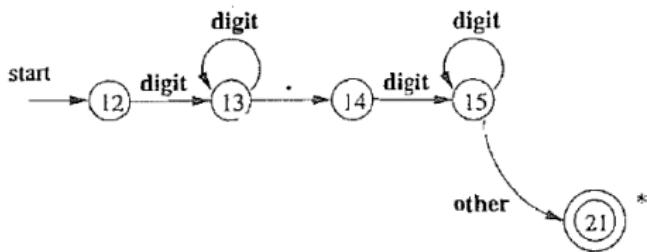
如何同时识别 real 与 sci?



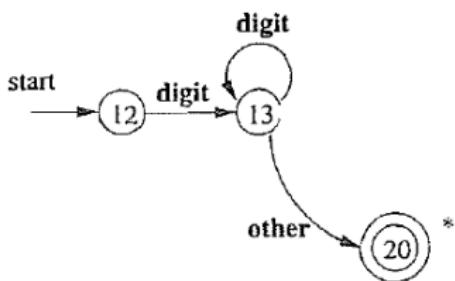
integer: 整数



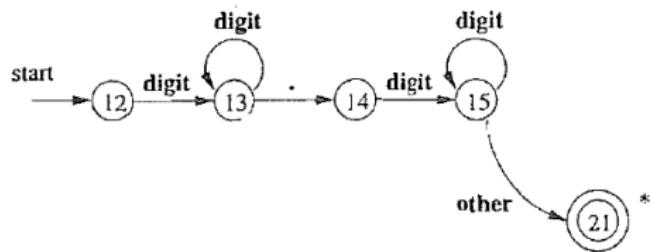
integer: 整数



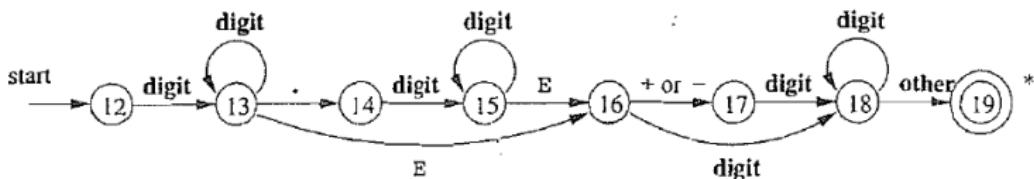
real: 浮点数 (无科学计数法)
(不识别 2.)



integer: 整数



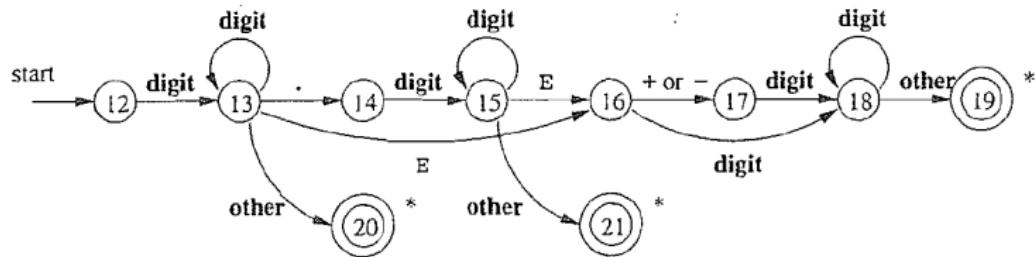
real: 浮点数 (无科学计数法)
(不识别 2.)



sci: 带科学计数法的浮点数
(2.99792458E8 3E8)

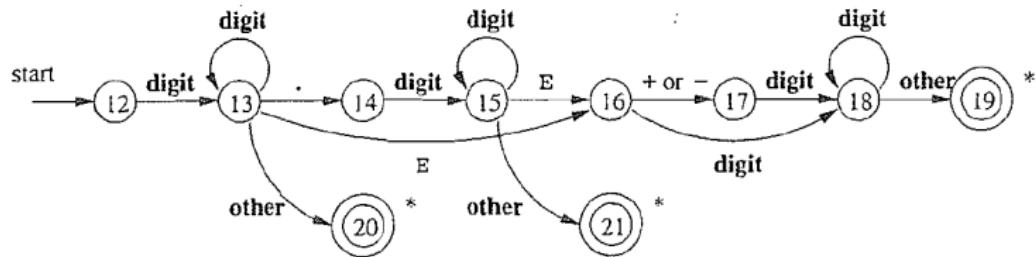
number: 整数部分 [. 可选的小数部分] [E[可选的 + -] 可选的指数部分]

number: 整数部分 [. 可选的小数部分] [E [可选的 + -] 可选的指数部分]



19, 20, 21 : 代表了不同的数字类型

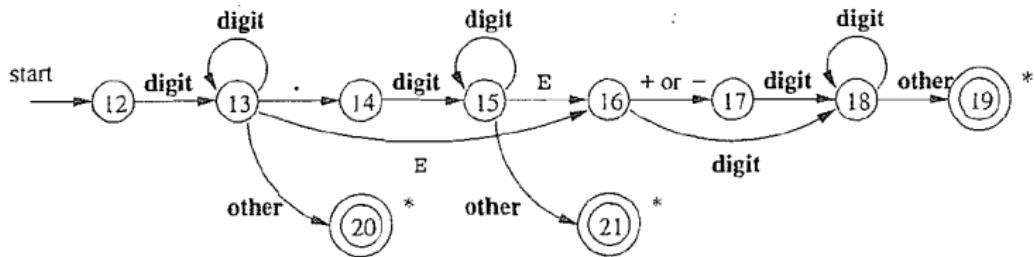
number: 整数部分 [. 可选的小数部分] [E[可选的 + -] 可选的指数部分]



19, 20, 21 : 代表了不同的数字类型

12 : 碰到 other 怎么办?

number: 整数部分 [. 可选的小数部分] [E[可选的 + -] 可选的指数部分]

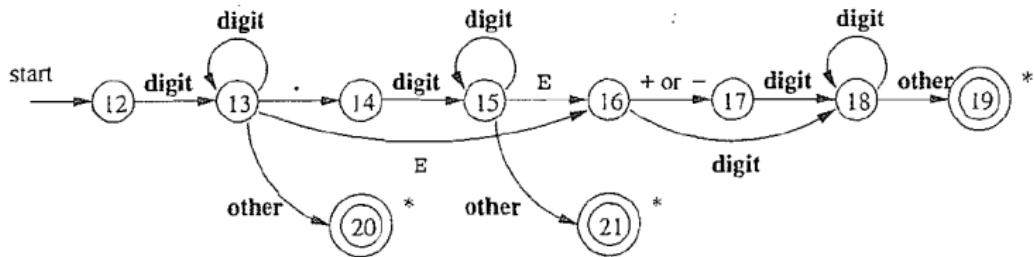


19, 20, 21 : 代表了不同的数字类型

12 : 碰到 other 怎么办?

(尝试其它词法单元或进入错误处理模块)

number: 整数部分 [. 可选的小数部分] [E [可选的 + -] 可选的指数部分]



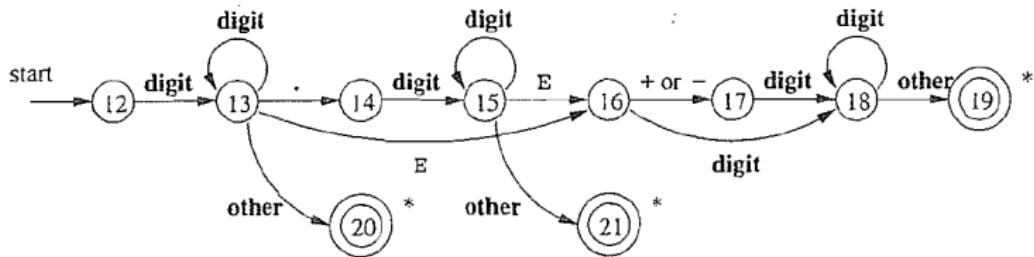
19, 20, 21 : 代表了不同的数字类型

12 : 碰到 other 怎么办?

(尝试其它词法单元或进入错误处理模块)

14, 16, 17 : 碰到 other 怎么办?

number: 整数部分 [. 可选的小数部分] [E [可选的 + -] 可选的指数部分]



19, 20, 21 : 代表了不同的数字类型

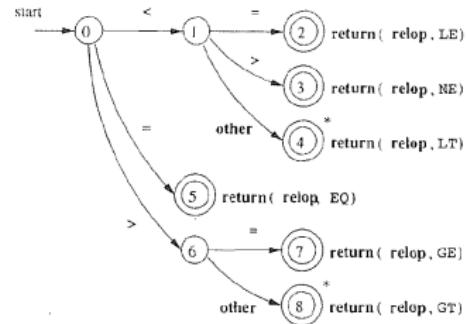
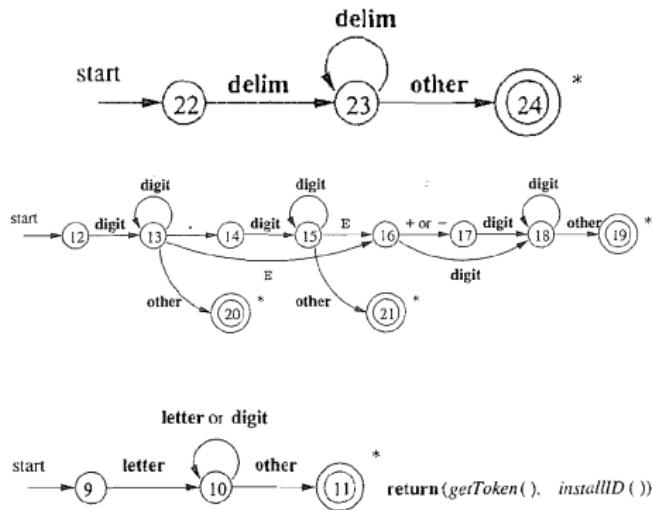
12 : 碰到 other 怎么办?

(尝试其它词法单元或进入错误处理模块)

14, 16, 17 : 碰到 other 怎么办?

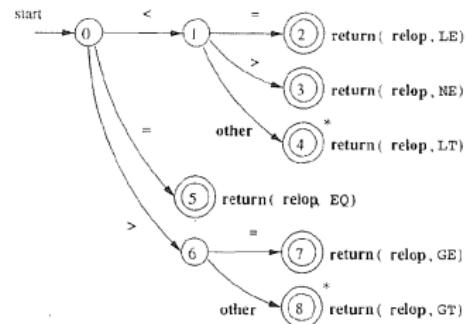
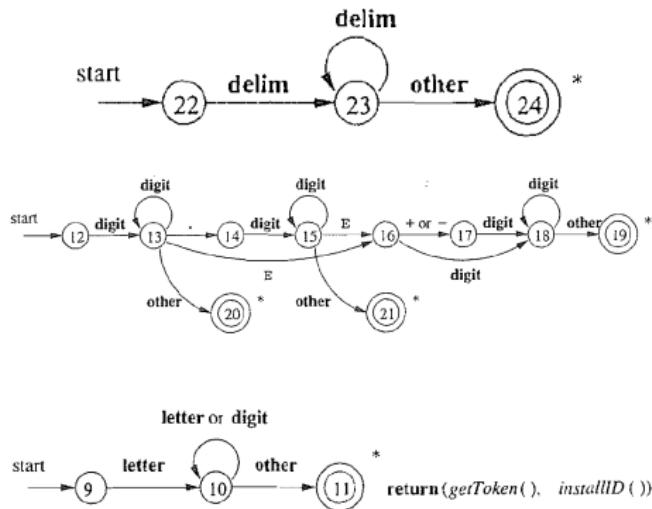
(回退, 寻找**最长匹配**)

识别字符串 s 中符合某种词法单元模式的前缀词素 (SCAN())



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

识别字符串 s 中符合某种词法单元模式的前缀词素 (SCAN())



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

关键点: 合并 22, 12, 9, 0, 根据下一个字符即可判定词法单元的类型
否则, 调用错误处理模块 (对应 other), 报告该字符有误, 忽略该字符。

注意, 在sci中, 有时需要回退, 寻找最长匹配。

3) 有一个更好的方法，也是我们将在下面各节中采用的方法，就是将所有的状态转换图合并为一个图。我们允许合并后的状态转换图尽量读取输入，直到不存在下一个状态为止；然后像上面的2中讨论的那样取最长的和某个模式匹配的最长词素。

1.2345E+a

3) 有一个更好的方法，也是我们将在下面各节中采用的方法，就是将所有的状态转换图合并为一个图。我们允许合并后的状态转换图尽量读取输入，直到不存在下一个状态为止；然后像上面的2中讨论的那样取最长的和某个模式匹配的最长词素。

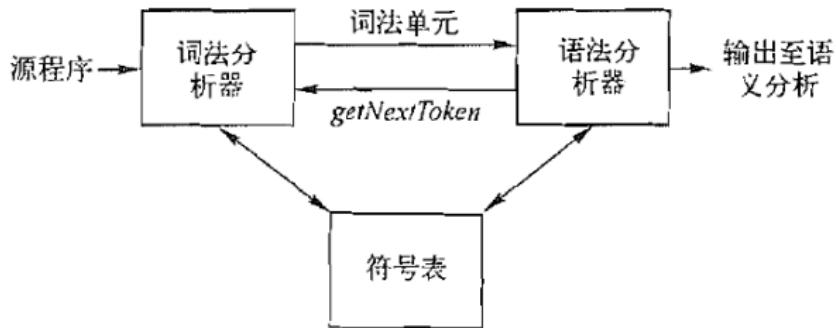
1.2345E+a

1.2345 E + a



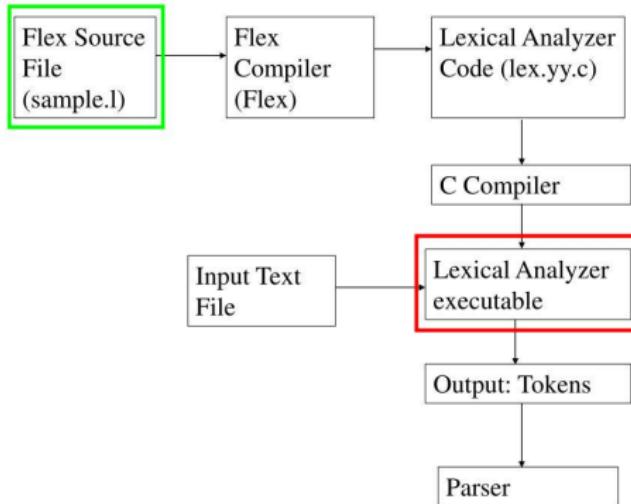
词法分析器的生成器 (Fast *Lexical* Analyzer Generator)

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

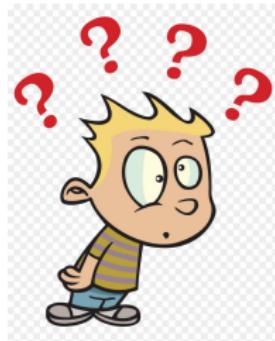


输出: 词法分析器

词法单元的规约

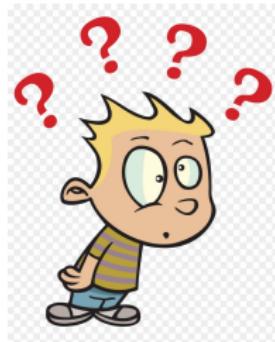
词法单元	非正式描述	词素示例
if	字符 i, f	if
else	字符 e, l, s, e	else
comparison	< 或 > 或 <= 或 >= 或 == 或 !=	<=, !=
id	字母开头的字母 / 数字串	pi, score, D2
number	任何数字常量	3.14159, 0, 6.02e23
literal	在两个 "之间, 除" 以外的任何字符	"core dumped"

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}

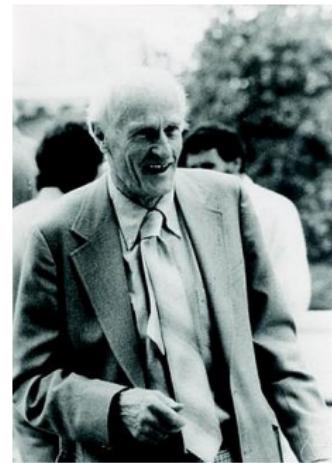
if : $\{if\}$

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。

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

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

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

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
```

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 v1 ▾

```
// | \/\|/.*
```

TEST STRING

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

<https://regex101.com/r/ED4qgC/2>

如何满足 L1 关于多行注释的要求？

一种是使用 “`/*`” 以及 “`*/`” 进行多行注释，在这种情况下，在 “`/*`” 与 之后最先遇到的 “`*/`” 之间的所有字符都被视作注释内容。需要注意的是， “`/*`” 与 “`*/`” 是 不允许嵌套的：即在任意一对 “`/*`” 和 “`*/`” 之间不能再包含成对的 “`/*`” 和 “`*/`” ，否则编译器需要进行报错。

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



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

REGULAR EXPRESSION v1 ▾

```
// ^((0|(1(01*0)*1))*$
```

TEST STRING

0

1

10

11

100

101

110

111

1000

1001

1010

1011

1100

1101

1110

1111

10000

10001

10010

10011

10100

10101

10110



REGULAR EXPRESSION v1 ▾

```
// ^((0|(1(01*0)*1))*$
```

TEST STRING

```
0  
1  
10  
11  
100  
101  
110  
111  
1000  
1001  
1010  
1011  
1100  
1101  
1110  
1111  
10000  
10001  
10010  
10011  
10100  
10101  
10110
```

3 的倍数 (二进制表示)



Flex 程序的结构 (.l 文件)

声明部分: 声明变量、正则表达式定义

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

```
1  %{
2      #include "stdio.h"
3  %}
4
5  if          "if"
6  else        "else"
7  ws          [ \t\n\r]+
8  digit       [0-9]
9  letter      [A-Za-z]
10 integer    {digit}+
11 id          ({letter})({letter}|{digit})*
12
13 %%
14
15 {ws}        { printf("whitespace\n"); }
16 {if}         { printf("if\n"); }
17 {else}       { printf("else\n"); }
18 {id}         { printf("id: %s\n", yytext); }
19 {integer}    { printf("integer: %s\n", yytext); }
20 .
21             { printf("Mysterious character %s\n", yytext); }
22 %%
```

```
flex lexical.l
```

```
cc lex.yy.c -lfl -o lexical.out
```

```
./lexical-out
```

Flex 中两大**冲突解决**规则

最前优先匹配: 关键字 *vs.* 标识符

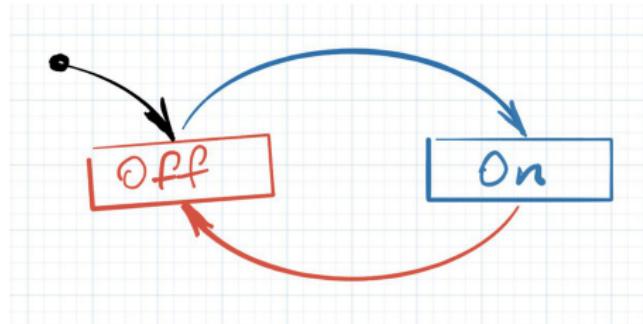
最长优先匹配: >=, ifhappy, thenext, 1.23



自动化词法分析器

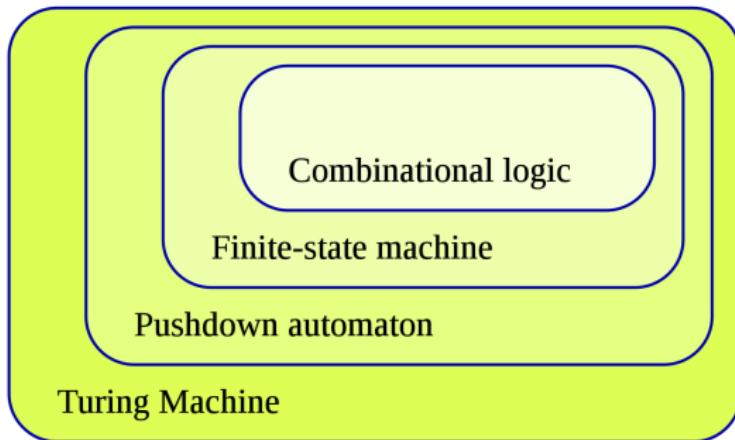
自动机

(Automaton; Automata)



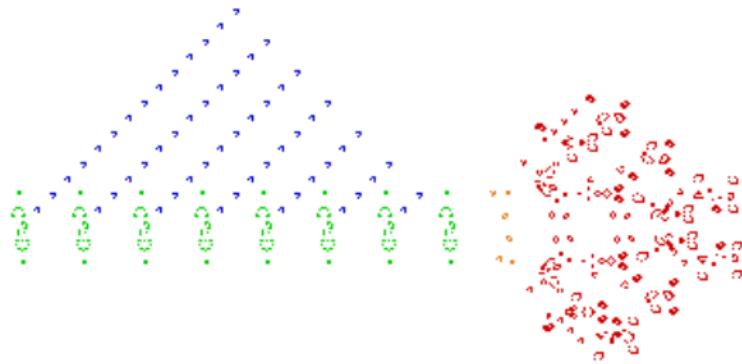
两大要素: 状态集 S 以及状态转移函数 δ

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”



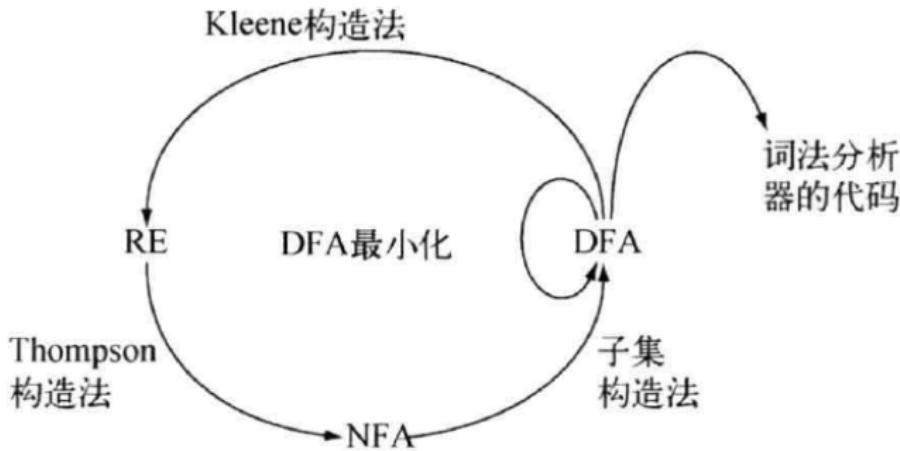
John Horton Conway
(1937 ~ 2020)



“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

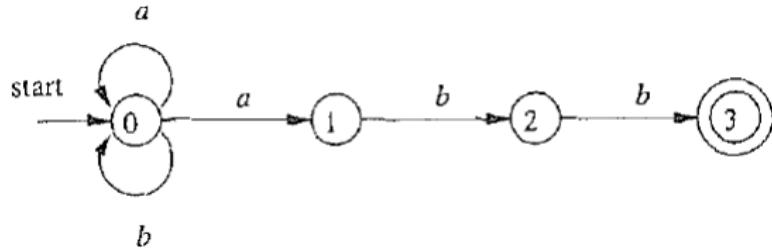
Definition (NFA (Nondeterministic Finite Automaton))

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 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 $F \subseteq S$



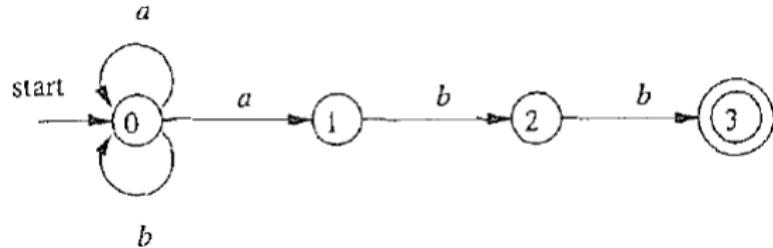
Definition (NFA (Nondeterministic Finite Automaton))

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 $F \subseteq S$



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



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.”*

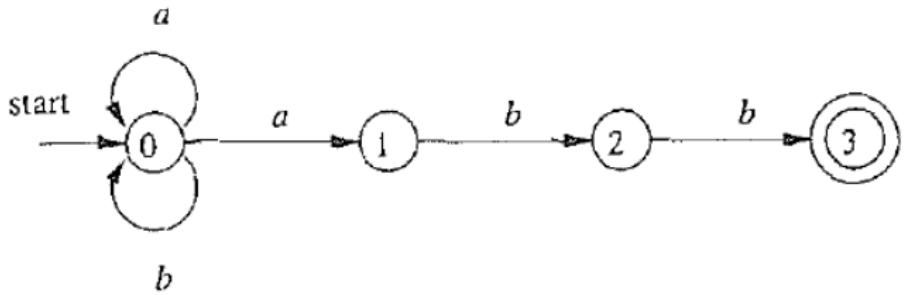
(非确定性) 有穷自动机是一类极其简单的**计算**装置

它可以**识别** (接受/拒绝) Σ 上的字符串

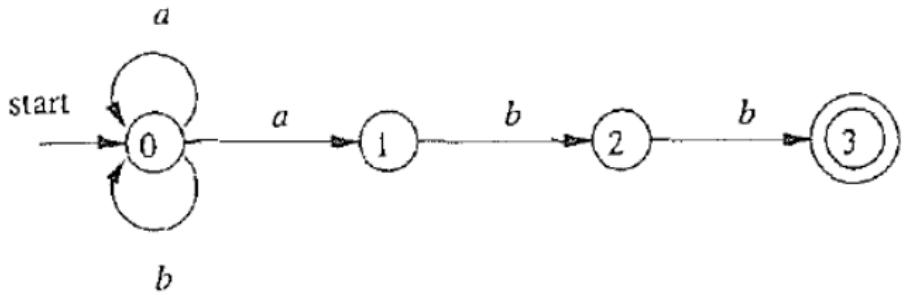
Definition (接受 (Accept))

(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

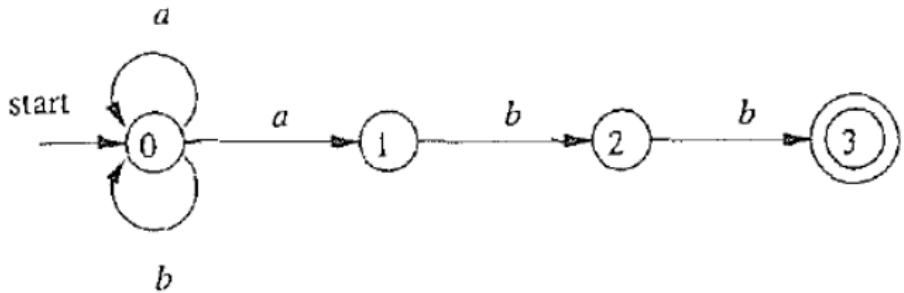


$aabb \in L(\mathcal{A})$ $ababab \notin L(\mathcal{A})$



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

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

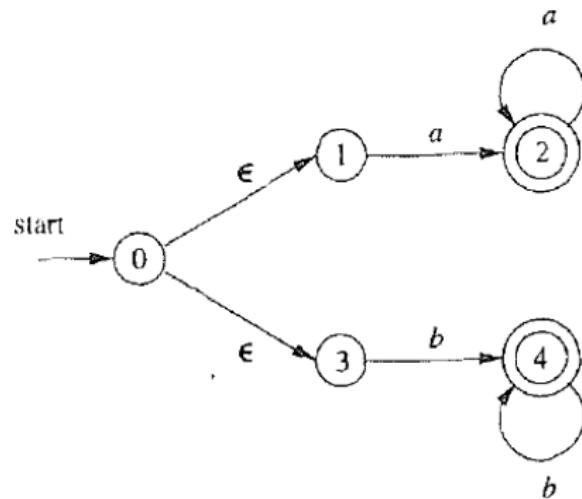


$$aabbb \in L(\mathcal{A}) \quad ababab \notin L(\mathcal{A})$$

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

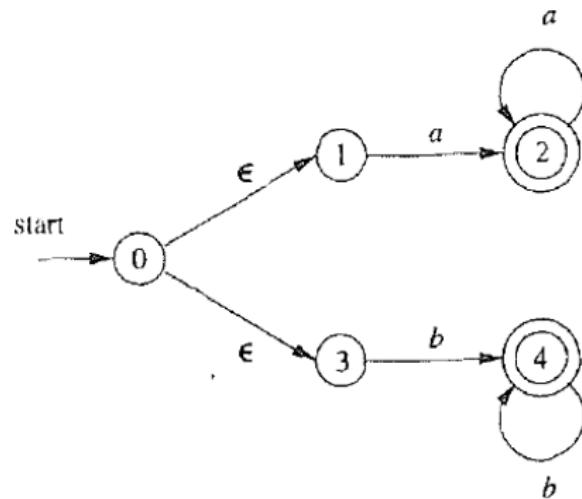
关于自动机 \mathcal{A} 的**两个基本问题**:

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



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

$L(\mathcal{A}) =$



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

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

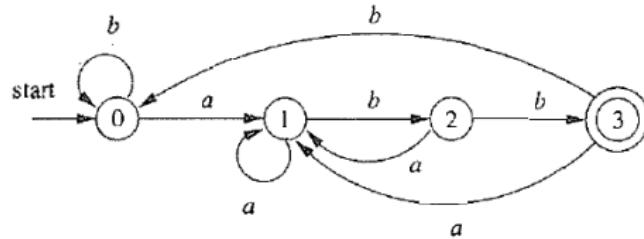
Definition (DFA (Deterministic Finite Automaton))

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 $F \subseteq S$



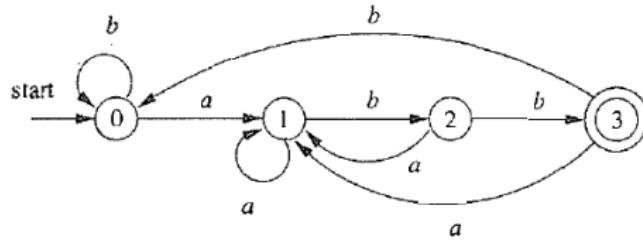
Definition (DFA (Deterministic Finite Automaton))

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

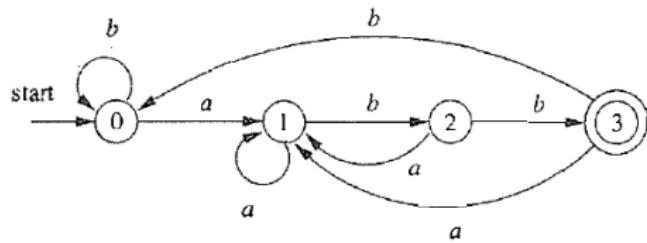
- (1) 字母表 Σ ($\epsilon \notin \Sigma$)
- (2) **有穷**的状态集合 S
- (3) **唯一**的初始状态 $s_0 \in S$
- (4) 状态转移**函数** δ

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

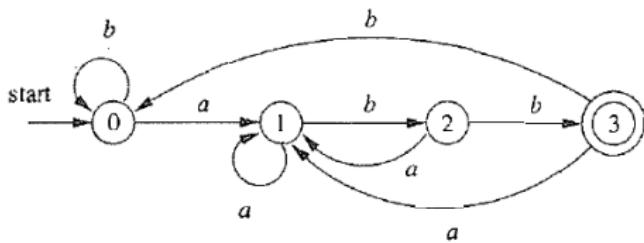
- (5) 接受状态集合 $F \subseteq S$



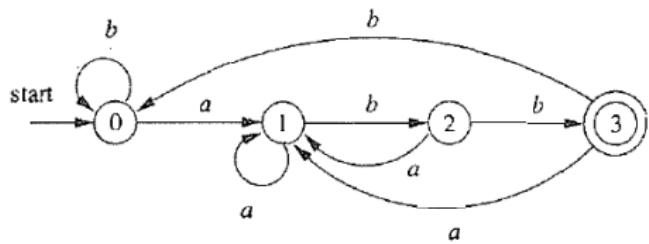
约定: 所有没有对应出边的字符默认指向一个不存在的“死状态”



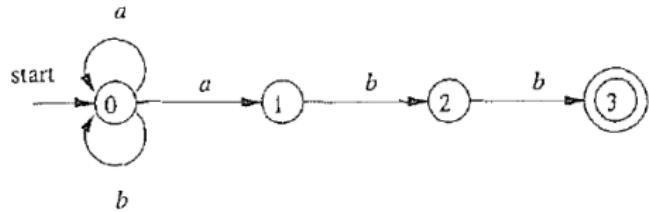
$$L(\mathcal{A}) =$$



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



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



NFA 与 DFA 的**优缺点**比较:

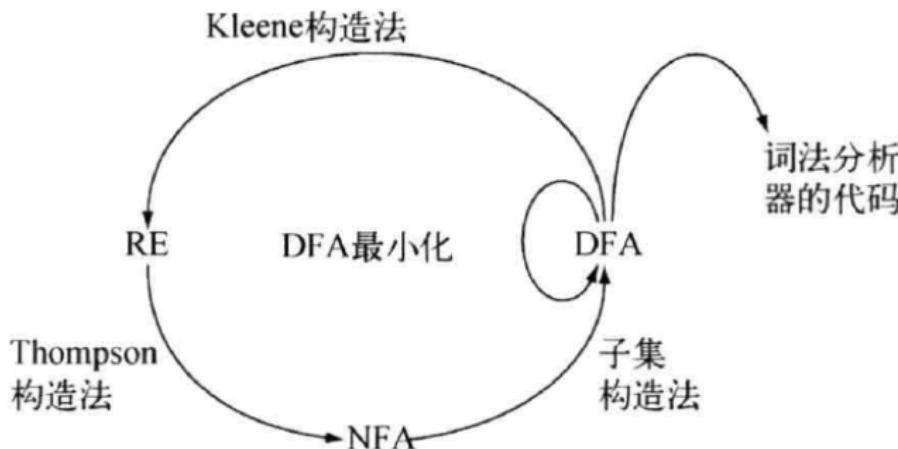
$x \in L(\mathcal{A})$: DFA 易于实现; NFA 不易实现

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

取长补短:

用 NFA 描述, 用 DFA 实现;

从 NFA 到 DFA 的转化自动完成



RE \Rightarrow 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 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 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。

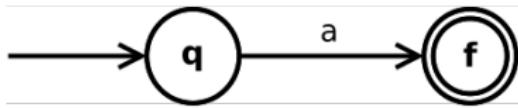
ϵ 是正则表达式。

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。



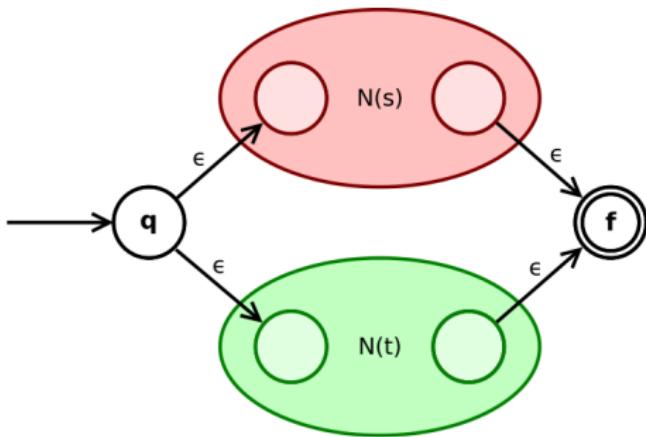
如果 s 是正则表达式, 则 (s) 是正则表达式;

如果 s 是正则表达式, 则 (s) 是正则表达式;

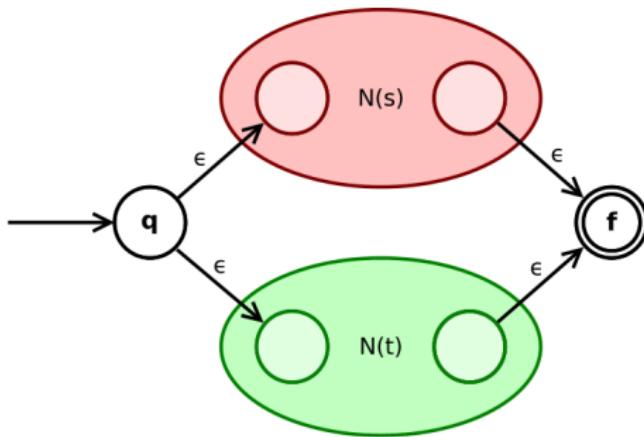
$$N((s)) = N(s).$$

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。

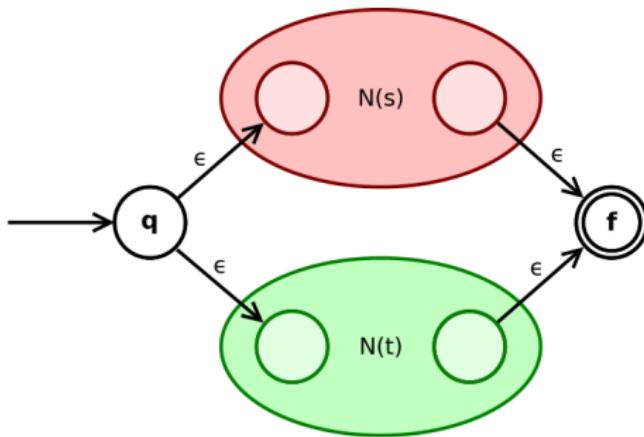


如果 s, t 是正则表达式, 则 $s|t$ 是正则表达式。



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。

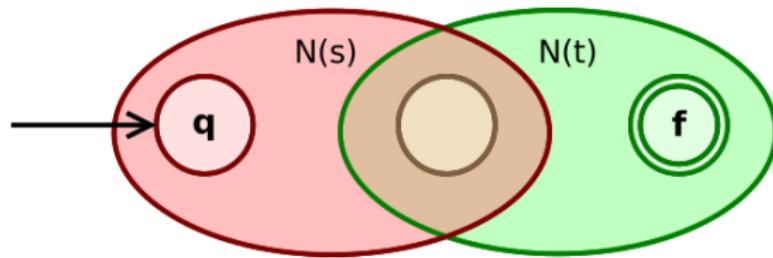


Q : 如果 $N(s)$ 或 $N(t)$ 的开始状态或接受状态不唯一, 怎么办?

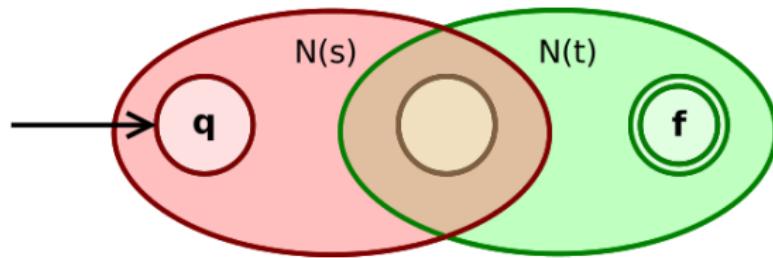
根据**归纳假设**, $N(s)$ 与 $N(t)$ 的开始状态与接受状态均**唯一**。

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。



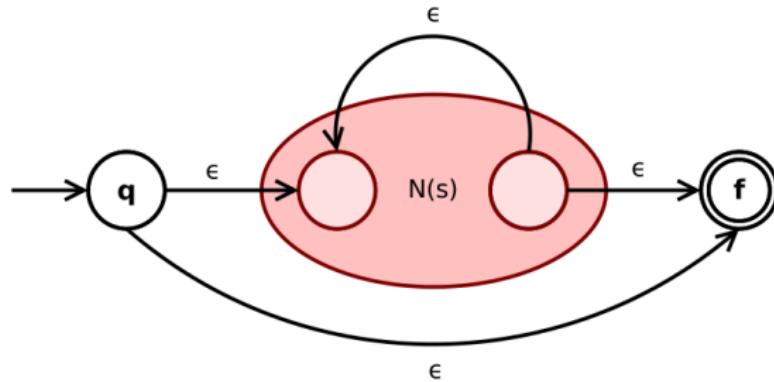
如果 s, t 是正则表达式, 则 st 是正则表达式。



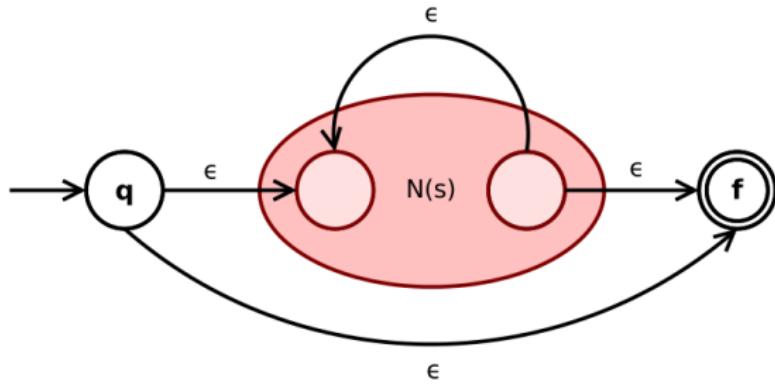
根据**归纳假设**, $N(s)$ 与 $N(t)$ 的开始状态与接受状态均**唯一**。

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。



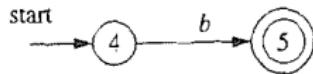
根据**归纳假设**, $N(s)$ 的开始状态与接受状态**唯一**。

$N(r)$ 的性质以及 Thompson 构造法复杂度分析

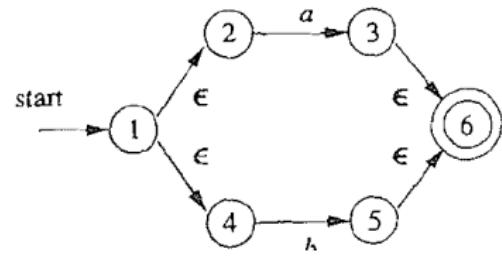
1. $N(r)$ 的开始状态与接受状态均唯一。
2. 开始状态没有入边, 接受状态没有出边。
3. 每个状态最多有两个 ϵ -入边与两个 ϵ 出边。
4. $\forall a \in \Sigma$, 每个状态最多有一个 a -入边与一个 a -出边。
5. $N(r)$ 的状态数 $|S| \leq 2 \times |r|$ 。
($|r| : r$ 中运算符与运算分量的总和)

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

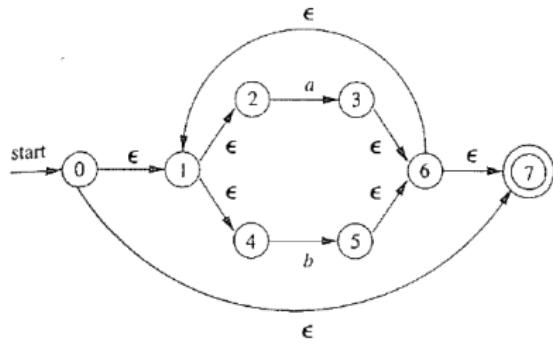
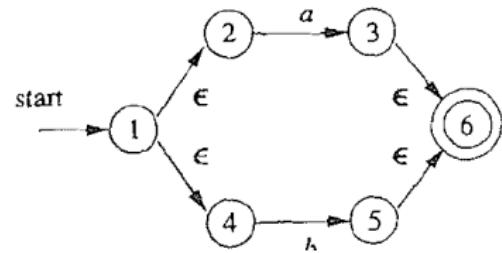
$$r = (a|b)^*abb$$



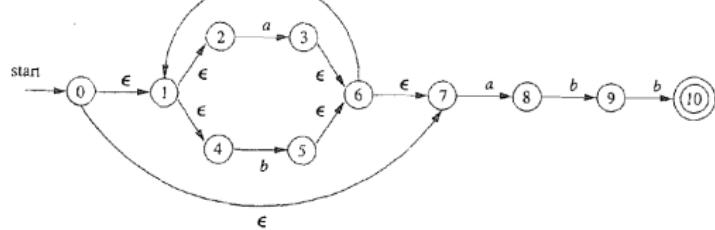
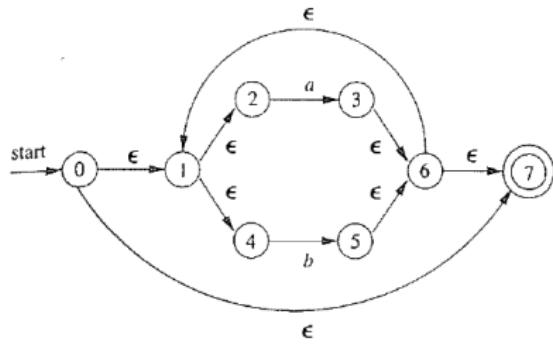
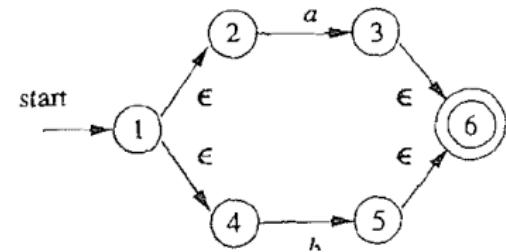
$$r = (a|b)^*abb$$

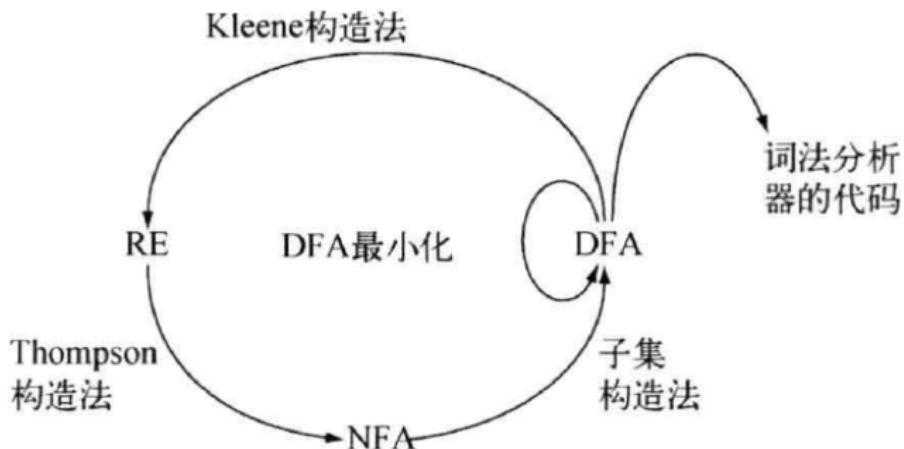


$$r = (a|b)^*abb$$



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





NFA \Rightarrow 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)



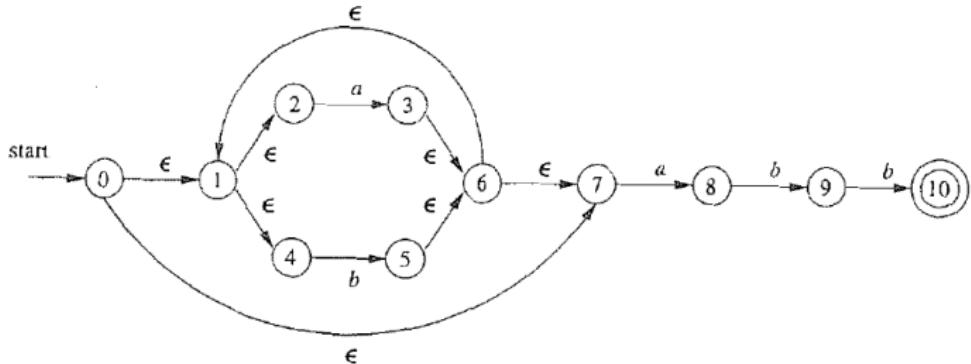
Michael O. Rabin (1931 ~)



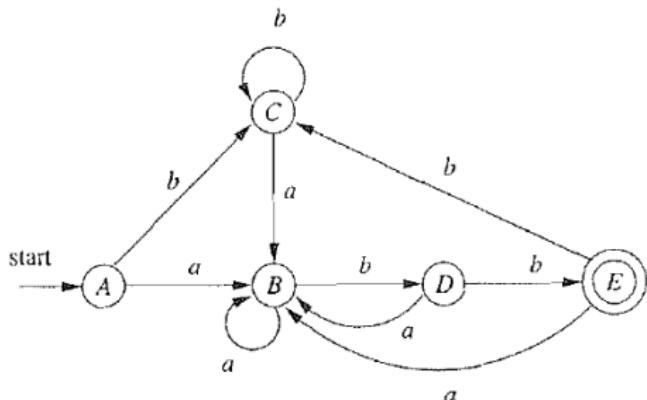
Dana Scott (1932 ~)

思想: 用 DFA 模拟 NFA

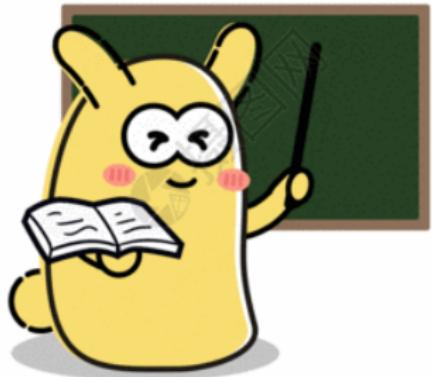
用 DFA 模拟 NFA



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

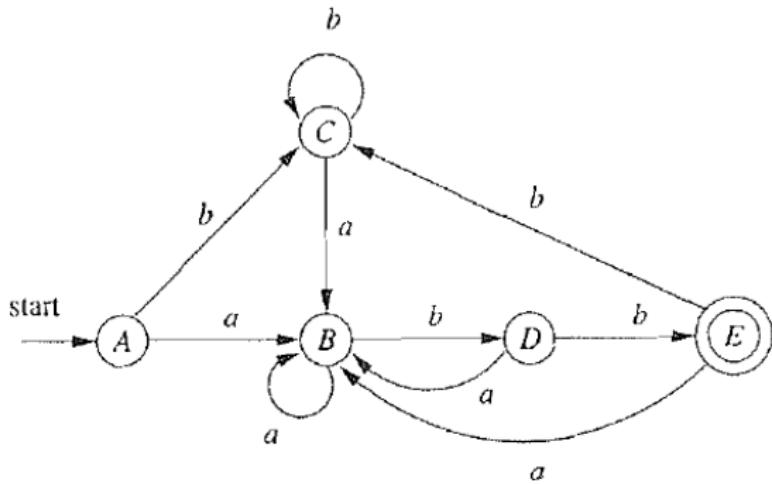


同学们看黑板！！



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 ϵ -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 ϵ -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 ϵ -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 $d_0 = \epsilon\text{-closure}(s_0)$

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 $d_0 = \epsilon\text{-closure}(s_0)$

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 $d_0 = \epsilon\text{-closure}(s_0)$

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 $F_D = \{s \in S_D \mid \exists f \in F_N. f \in s\}$

子集构造法 ($N \Rightarrow D$) 的**实现**: 使用**栈**实现 ϵ -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, $|S_D| = \Omega(2^n)$

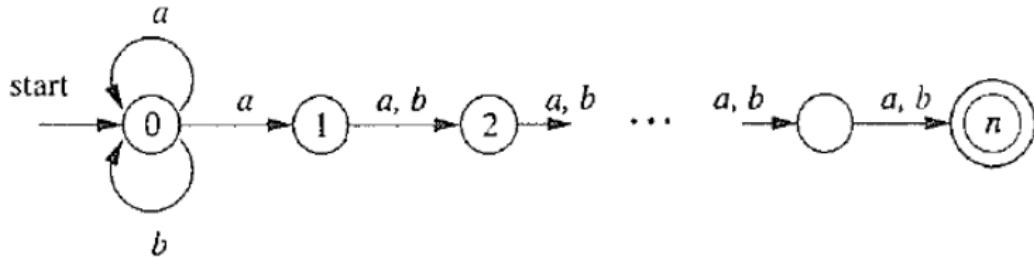
“长度为 $m \geq n$ 个字符的 a, b 串，且倒数第 n 个字符是 a ”

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

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

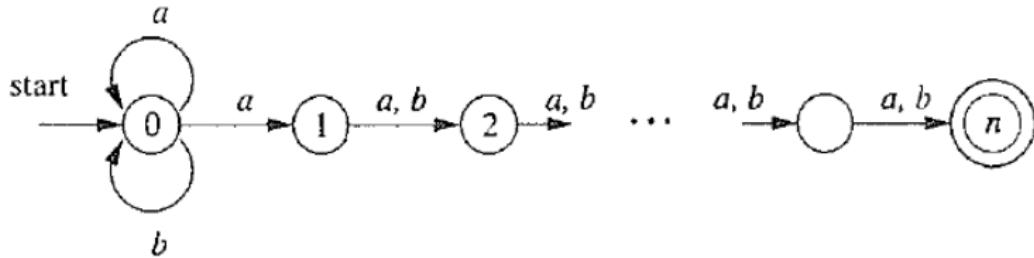
“长度为 $m \geq n$ 个字符的 a, b 串，且倒数第 n 个字符是 a ”

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



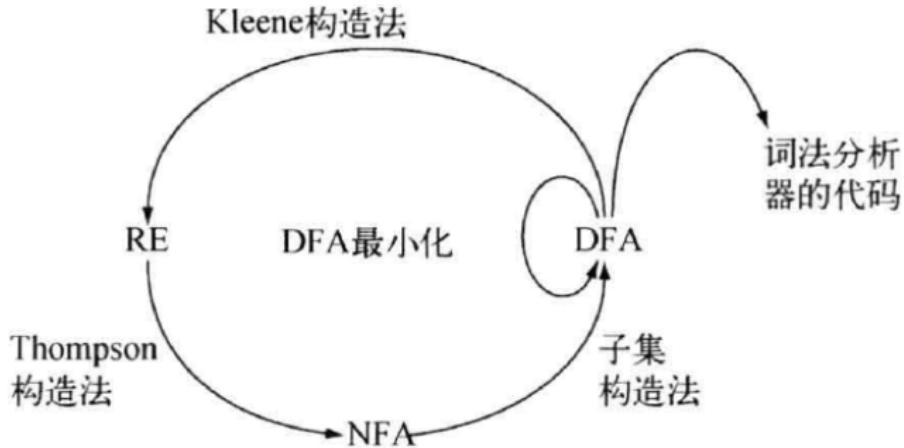
“长度为 $m \geq n$ 个字符的 a, b 串，且倒数第 n 个字符是 a ”

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

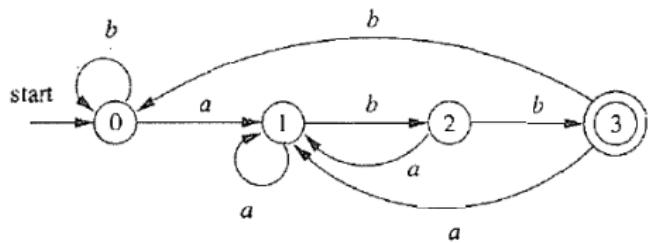


练习 (非作业): $m = n = 3$

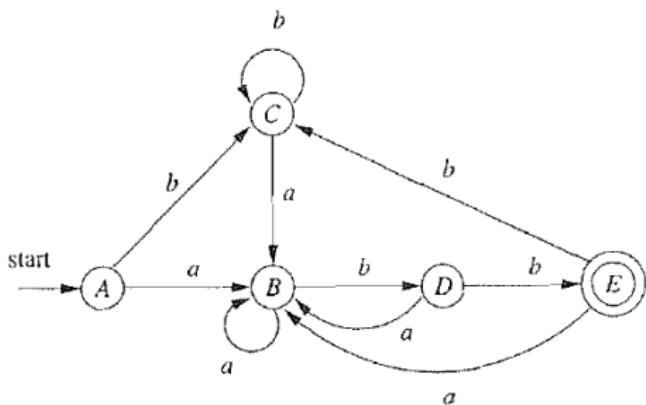
闭包 (Closure) 与不动点 (Fixed Point)



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

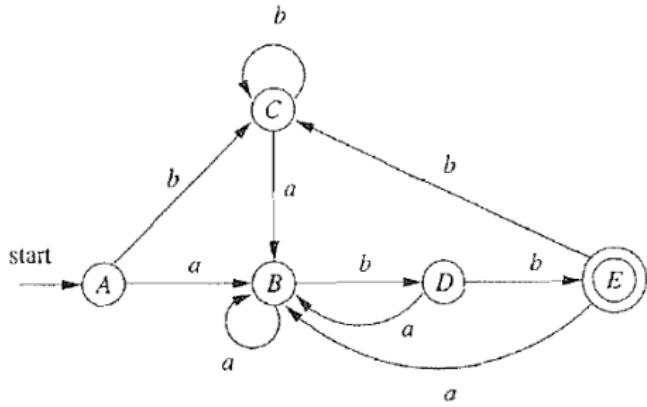


John Hopcroft (1939 ~)

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

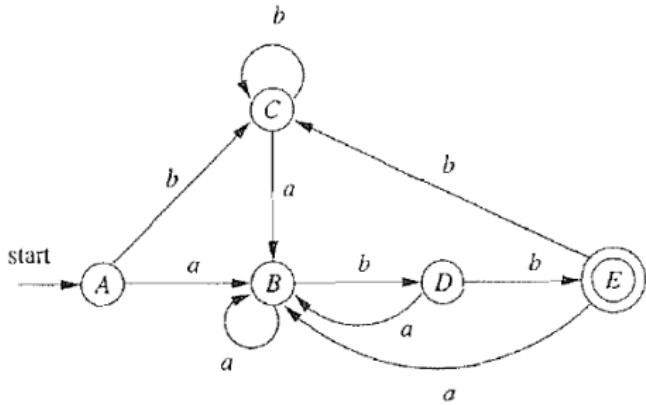
— Turing Award, 1986

如何定义等价状态?



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

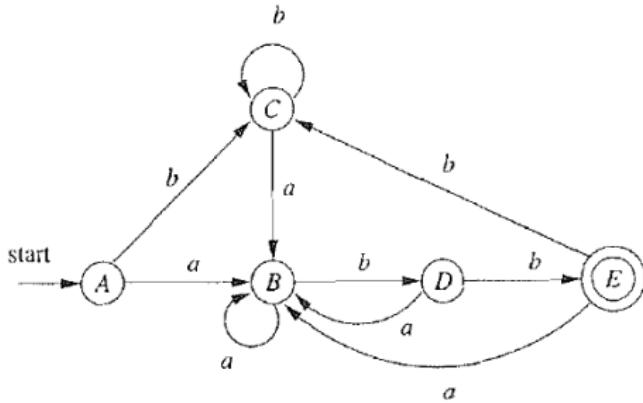
如何定义等价状态?



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

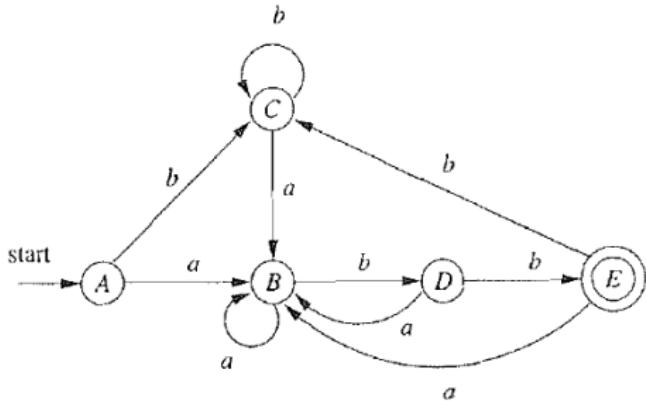
但是,这个定义过于严格

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?



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

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

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

基于该定义, 不断**合并**等价的状态, 直到无法合并为止

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

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？

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

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？

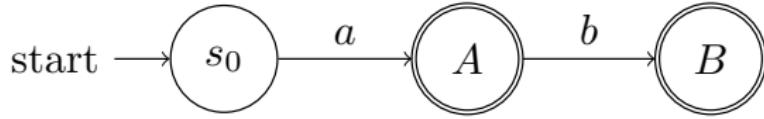
Q：所有接受状态都是等价的吗？

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

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?



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

缺少基础情况, 不知从何下手

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

缺少基础情况, 不知从何下手

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

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

缺少基础情况, 不知从何下手

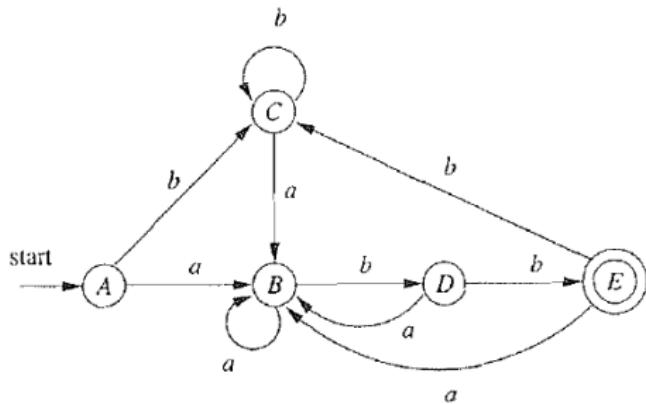
$$s \not\sim t \iff \exists a \in \Sigma. (s \xrightarrow{a} s') \wedge (t \xrightarrow{a} t') \wedge (s' \not\sim t')$$

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?

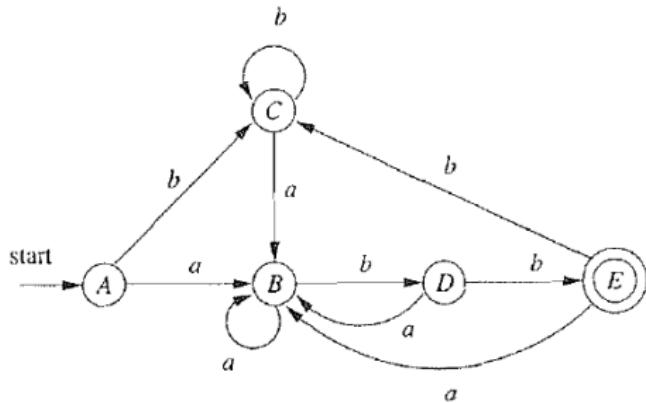


$$\Pi = \{F, S \setminus F\}$$

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?

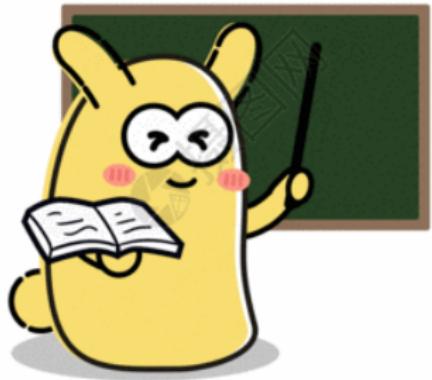


$$\Pi = \{F, S \setminus F\}$$

接受状态与非接受状态必定不等价

空串 ϵ 区分了这两类状态

同学们看黑板！！



板书演示算法过程

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 (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?

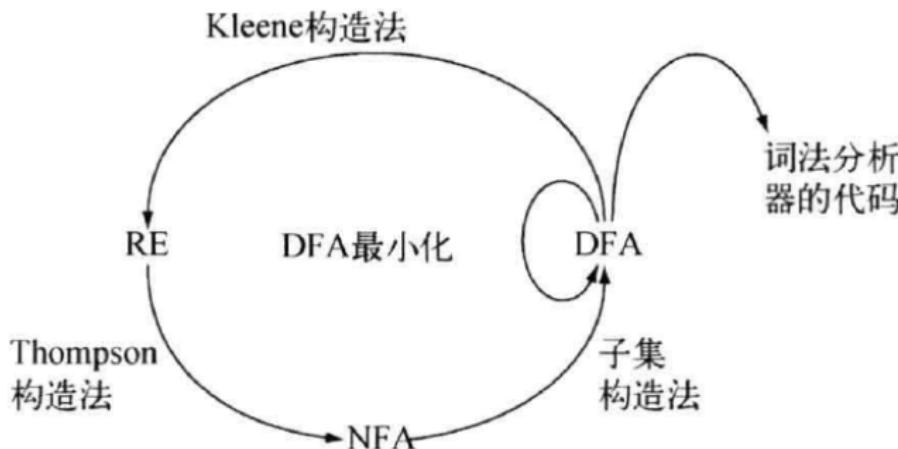
如何分析 DFA 最小化算法的**复杂度**?

为什么 DFA 最小化算法是**正确的**?

最小化 DFA 是**唯一的**吗?



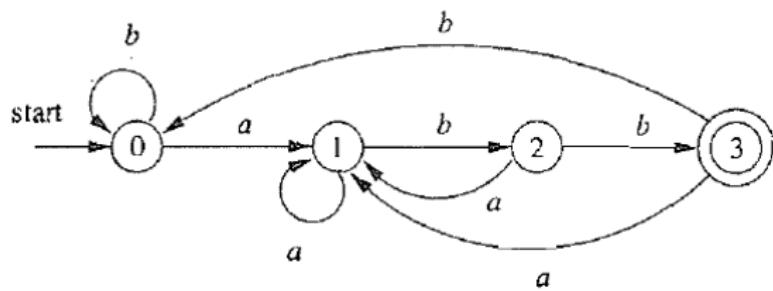
可选报告 (1)

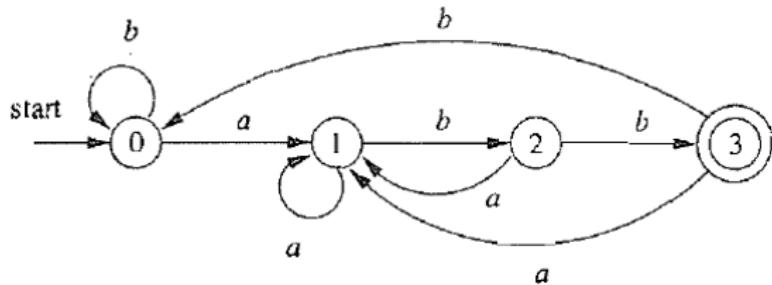


DFA \Rightarrow RE

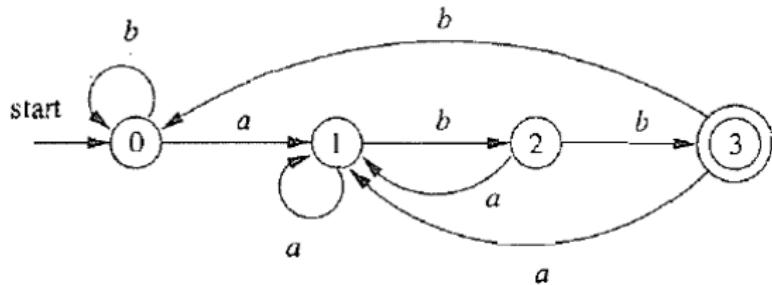
$$D \implies r$$

要求： $L(r) = L(D)$



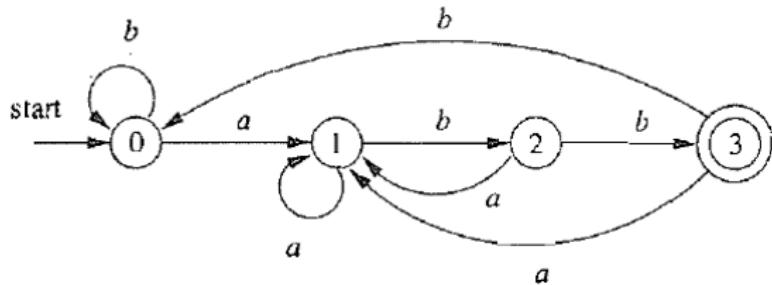


$$L(D) = \{x \mid \exists f \in F_D. d_0 \xrightarrow{x} f\}$$



$$L(D) = \{x \mid \exists f \in F_D. d_0 \xrightarrow{x} f\}$$

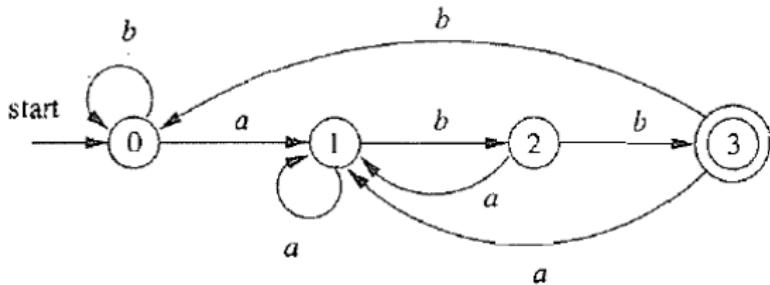
$$r = |_{x \in L(D)} x$$



$$L(D) = \{x \mid \exists f \in F_D. d_0 \xrightarrow{x} f\}$$

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

字符串 x 对应于有向图中的路径

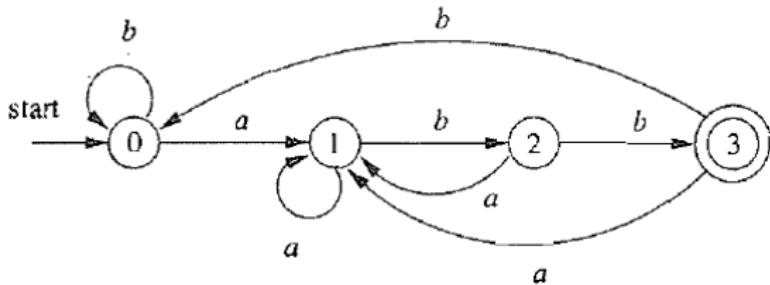


$$L(D) = \{x \mid \exists f \in F_D. d_0 \xrightarrow{x} f\}$$

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

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



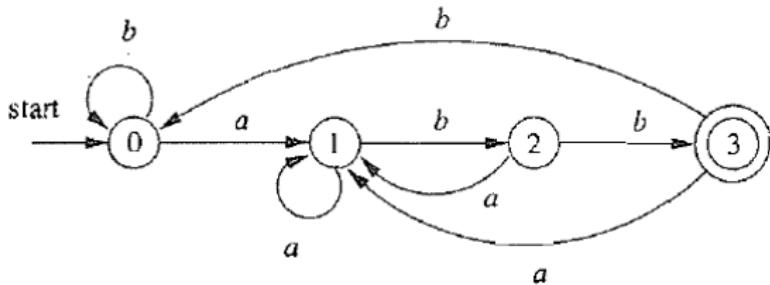
$$L(D) = \{x \mid \exists f \in F_D. d_0 \xrightarrow{x} f\}$$

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

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



$$L(D) = \{x \mid \exists f \in F_D. d_0 \xrightarrow{x} f\}$$

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

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

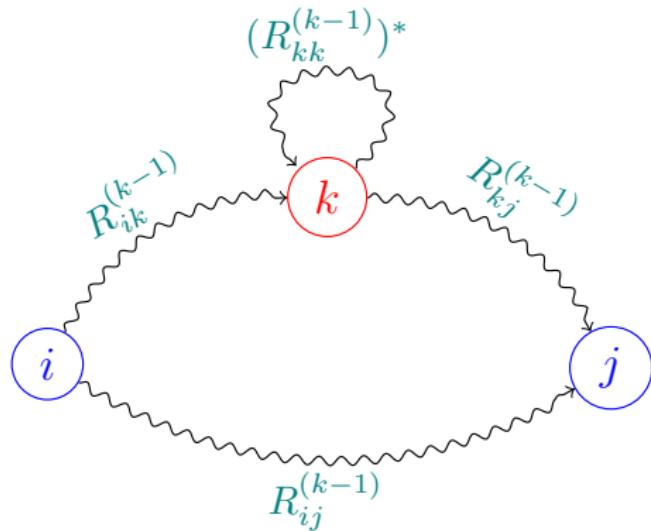
不要怕, 我们有 Kleene 闭包

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



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

```

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
             $R_{ij}^{-1} = R_{ij}^{-1} \cup \{\epsilon\}$ 

```

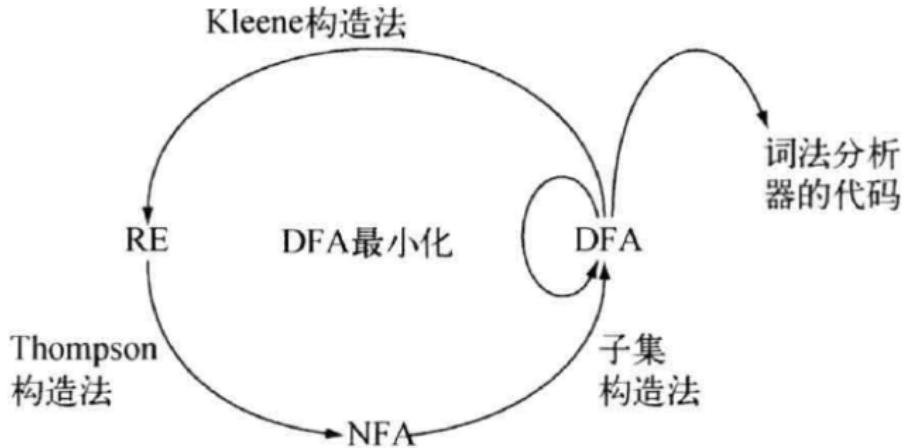
```

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

```

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

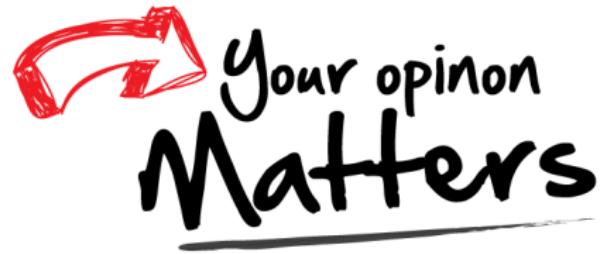
$|D|$: 状态数; D_A : 接受状态集合



DFA \Rightarrow 词法分析器

特定于词法分析器的最小化方法

Thank You!



Office 926

hfwei@nju.edu.cn