第三章 词法分析

许畅

南京大学计算机系

2018年春季

内容

- 词法分析器的作用
- 词法单元的规约(正则表达式)
- 词法单元的识别(状态转换图)
- 词法分析器生成工具及设计
- 有穷自动机

词法分析器的作用

- 读入字符流，组成词素，输出**词法单元**序列
- 过滤空白、换行、制表符、注释等
- 将词素添加到符号表中
- 在逻辑上独立于语法分析，但是通常和语法分析器处于同一趟中

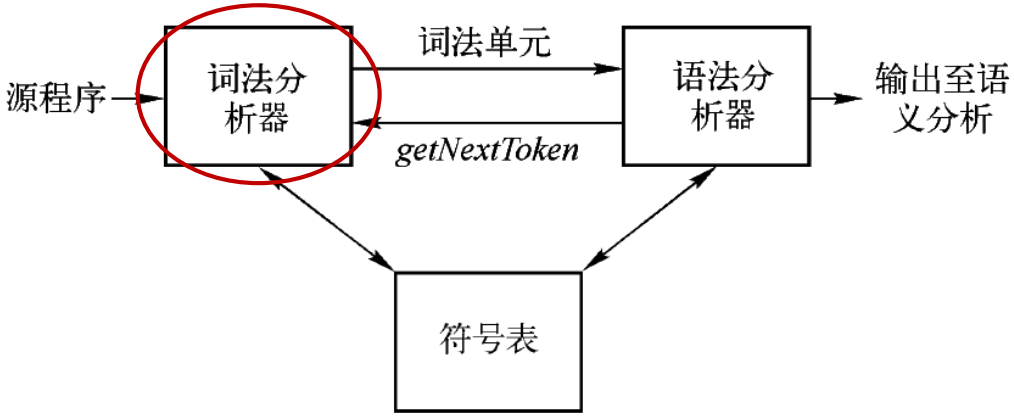


图 3-1　词法分析器与语法分析器之间的交互

为什么要设立独立的词法分析器?

- 简化编译器的设计
  - 词法分析器可以首先完成一些简单的处理工作
- 提高编译器效率
  - 相对于语法分析，词法分析过程简单，可高效实现(下推自动机 vs. 有穷自动机)
- 增强编译器的可移植性

词法单元、模式、词素

- **词法单元 (Token)**
  - <词法单元名、属性值 (可选) >
  - 单元名是表示词法单位种类的抽象符号，语法分析器通过单元名即可确定词法单元序列的结构
  - 属性值通常用于语义分析之后的阶段
- **模式 (Pattern)**
  - 描述了一类词法单元的词素可能具有的形式
- **词素 (Lexeme)**
  - 源程序中的字符序列
  - 它和某个词法单元的模式匹配，被词法分析器识别为该词法单元的实例

词法单元、模式、词素 (例子)

- printf("Total = %d\n", score);
  - printf, score和标识符 (id) 的模式匹配
  - "Total = % d\n"和literal的模式匹配

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”

图 3-2　词法单元的例子

词法单元的属性

- 一个模式匹配多个词素时，必须通过属性来传递附加的信息
  - 属性值将被用于语义分析、代码生成等阶段
- 不同的目的需要不同的属性
  - 属性值通常是一个结构化数据
- 如词法单元id的属性
  - 词素、类型、第一次出现的位置、...

内容

- 词法分析器的作用
- 词法单元的规约 (正则表达式)
- 词法单元的识别 (状态转换图)
- 词法分析器生成工具及设计
- 有穷自动机

词法单元的规约

- **正则表达式**可以高效、简洁地描述处理词法单元时用到的模式类型
- 内容
  - 串和语言
  - 语言上的运算
  - 正则表达式和正则定义
  - 正则表达式的扩展

串和语言 (1)

- **字母表** (Alphabet)：一个有穷的符号集合
  - 符号典型例子：字母、数字、标点符号
  - 例子：{0,1}, ASCII, Unicode
  - 在理论上，我们可以把任意的有限集合看作字母表
- 字母表上的**串** (String) 是该表中符号的有穷序列
  - 串*s*的长度，即 |*s*|，是指*s*中符号出现的次数
  - 空串：长度为0的串，ε
- **语言** (Language) 是某个给定字母表上的串的可数集合

串和语言 (2)

- 和串有关的术语 (以banana为例)
  - **前缀**：从串的尾部删除0个或多个符号后得到的串 (ban、banana、ε)
  - **后缀**：从串的开始处删除0个或多个符号后得到的串 (nana、banana、ε)
  - **子串**：删除串的某个前缀和某个后缀得到的串 (banana、nan、ε)
  - **真前缀、真后缀、真子串**：既不等于原串，也不等于空串的前缀、后缀、子串 (前面例子的红色部分)
  - 子序列：从原串中删除0个或者多个符号后得到的串 (baan)

串和语言 (3)

- 串的运算
  - **连接** (Concatenation)：x和y的连接是把y附加到x的后面而形成的串，记作xy
    - x = dog，y = house，xy = doghouse
  - **指数运算 (幂运算)**：$s^0 = \varepsilon$，$s^1 = s$，$s^i = s^{i-1}s$
    - x = dog，$x^0 = \varepsilon$，x^1 = dog，x^3 = dogdogdog

串和语言 (4)

- 语言的运算
  - 语言是某个给定字母表上的串的可数集合

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

图 3-6　语言上的运算的定义

串和语言 (5)

- 例子
  - L = {A, B , ..., Z, a, b, ..., z}
  - D = {0, 1, ..., 9}
  - $L \cup D$ = {A, B, ..., Z, a, b, ..., z, 0, 1, ..., 9}
  - LD：520个长度为2的串的集合 (字母 + 数字)
  - L^4：所有由四个字母构成的串的集合
  - L^*：所有字母构成的集合，包括ε
  - $L(L \cup D)^*$：？
  - D^+：？

正则表达式

字母表Σ上的**正则表达式**的定义

- 基本部分
  - ε是一个正则表达式，$L(\varepsilon) = \{\varepsilon\}$
  - 如果a是Σ上的一个符号，那么a是正则表达式，$L(a) = \{a\}$
- 归纳步骤
  - 选择：$(r)|(s)$，$L((r)|(s)) = L(r) \cup L(s)$
  - 连接：$(r)(s)$，$L((r)(s)) = L(r)L(s)$
  - 闭包：$(r)^*$，$L((r)^*) = (L(r))^*$
  - 括号：(r)，$L((r)) = L(r)$
- 运算的优先级：* > 连接 > |，如(a) | ((b)*(c)) = a | b*c
- **正则集合**：可以用一个正则表达式定义的语言

正则表达式的例子

- $\Sigma = \{a, b\}$
- $L(a \mid b) = \{a, b\}$
- $L((a \mid b)\ (a \mid b)) = \{aa, ab, ba, bb\}$
- $L(a^*) = \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$
- $L((a \mid b)^*) = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$
- $L(a \mid a^*b) = \{a, b, ab, aab, aaab, \ldots\}$

正则定义 (1)

- 为了书写方便，可以给正则表达式命名，且可以通过名字使用正则表达式
- 正则定义是如下形式的定义序列

  $d_1 \rightarrow r_1$

  $d_2 \rightarrow r_2$

  ...

  $d_n \rightarrow r_n$
- 其中
  - d_i不在Σ中，且各不相同
  - 每个r_i是字母表$\Sigma \cup \{d_1, d_2, ..., d_{i-1}\}$上的正则表达式；这保证了不会出现递归定义

正则定义的例子

- C语言标识符的正则定义
  - *letter_* → A | B | … | Z | a | b | … | z | _
  - *digit* → 0 | 1 | … | 9
  - *id* → *letter_* (*letter_* | *digit*)*
- *id*对应的正则表达式为
  - (A | B | … | Z | a | b | … | z | _) ((A | B | … | Z | a | b | … | z | _) | (0 | 1 | … | 9))*

正则表达式的扩展

- 基本运算符：选择、连接、Kleene闭包
- 扩展的运算符
  - 一个或多个实例：单目后缀$^+$
    - r^+等价于$r\,r^*$
  - 零个或一个实例：?
    - $r?$等价于$\varepsilon \mid r$
  - 字符类
    - $[a_1a_2...a_n]$等价于$a_1 \mid a_2 \mid ... \mid a_n$
    - [a–e]等价于a | b | c | d | e
- 使正则表达式更简洁，但不会使其描述能力增强

内容

- 词法分析器的作用
- 词法单元的规约(正则表达式)
- 词法单元的识别(状态转换图)
- 词法分析器生成工具及设计
- 有穷自动机

词法单元的识别

- 词法分析器要求能够检查输入字符串，在其前缀中找出和某个模式匹配的词素
- 首先通过正则定义来描述各种词法单元的模式
- 定义$ws \rightarrow (\text{blank} \mid \text{tab} \mid \text{newline})^{+}$来消除空白
  - 当词法分析器识别出这个模式时，不返回词法单元，继续识别其它模式

```
digit   →  [0-9]
digits  →  digit+
number  →  digits (. digits)? ( E [+-]? digits )?
letter  →  [A-Za-z]
id      →  letter ( letter | digit )*
if      →  if
then    →  then
else    →  else
relop   →  < | > | <= | >= | = | <>
```

图 3-11　例 3.8 中词法单元的模

词素	词法单元名字	属性值
Any *ws*	–	–
if	if	–
then	then	–
else	else	–
Any *id*	id	指向符号表条目的指针
Any *number*	number	指向符号表条目的指针
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

图 3-12　词法单元、它们的模式以及属性值

状态转换图

- 词法分析器的重要组件之一
- **状态转换图 (Transition diagram)**
  - **状态 (State)**：表示在识别词素时可能出现的情况
    - 状态看作是已处理部分的总结
    - 某些状态为**接受状态**或**最终状态**，表明已找到词素
    - 加上*的接受状态表示最后读入的符号不在词素中
    - **开始状态 (初始状态)**：用Start边表示
  - **边 (Edge)**：从一个状态指向另一个状态
    - 边的标号是一个或多个符号
    - 当前状态为s，下一个输入符号为a，就沿着从s离开，标号为a的边到达下一个状态

状态转换图的例子

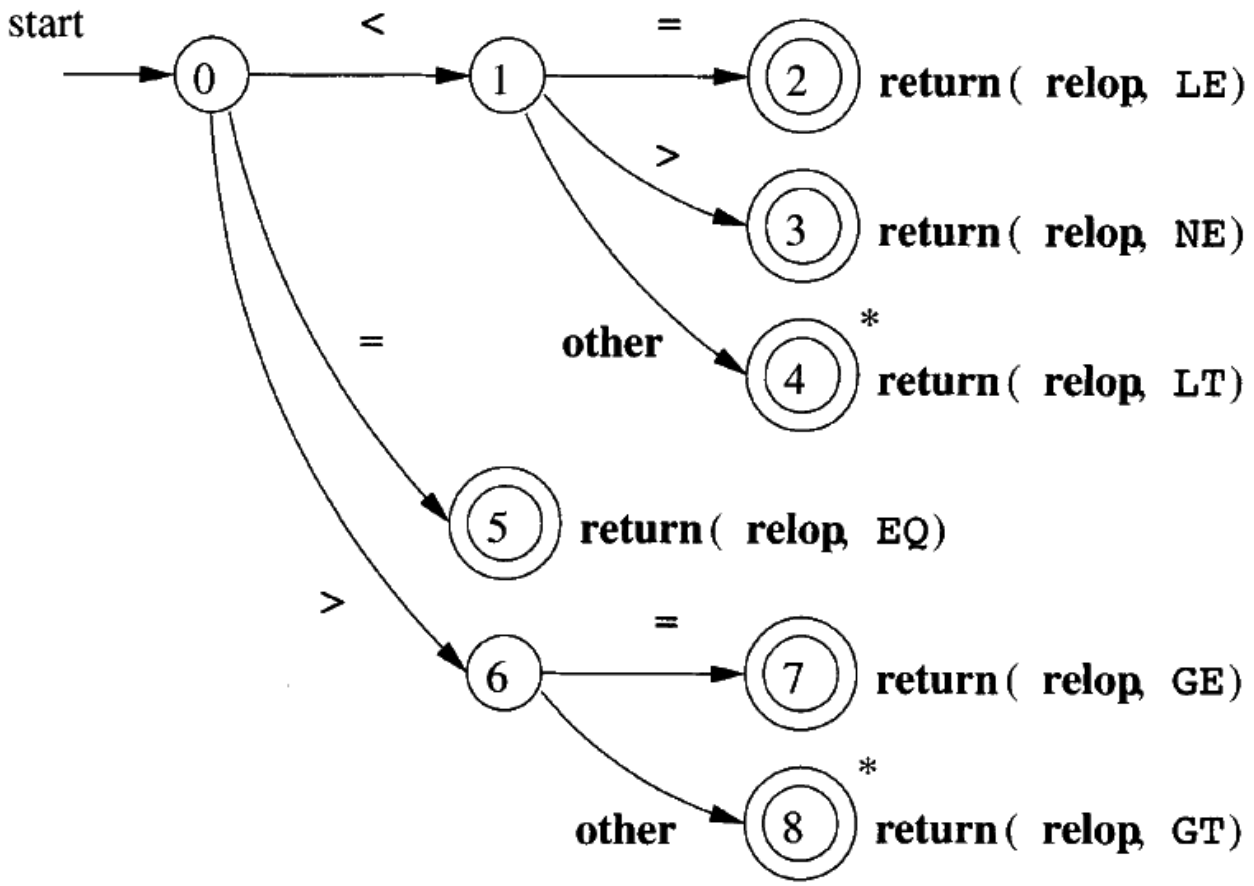


图 3-13　词法单元 **relop** 的状态转换图

保留字和标识符的识别

- 在很多时候，保留字也符合标识符的模式
  - 识别标识符的状态转换图也会识别保留字
- 解决方法
  - 在符号表中先填保留字，并指明它们不是普通标识符
  - 为保留字建立独立的、高优先级的状态转换图

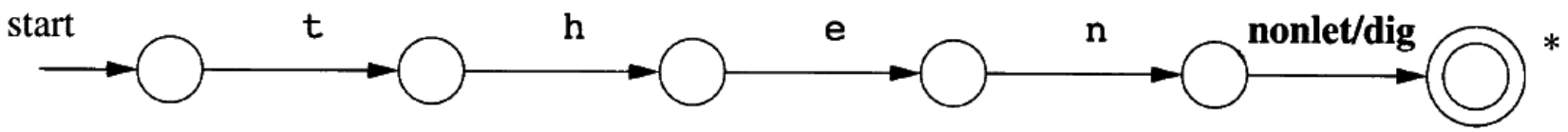


图 3-15　假想的关键字 then 的状态转换图

其它的状态转换图

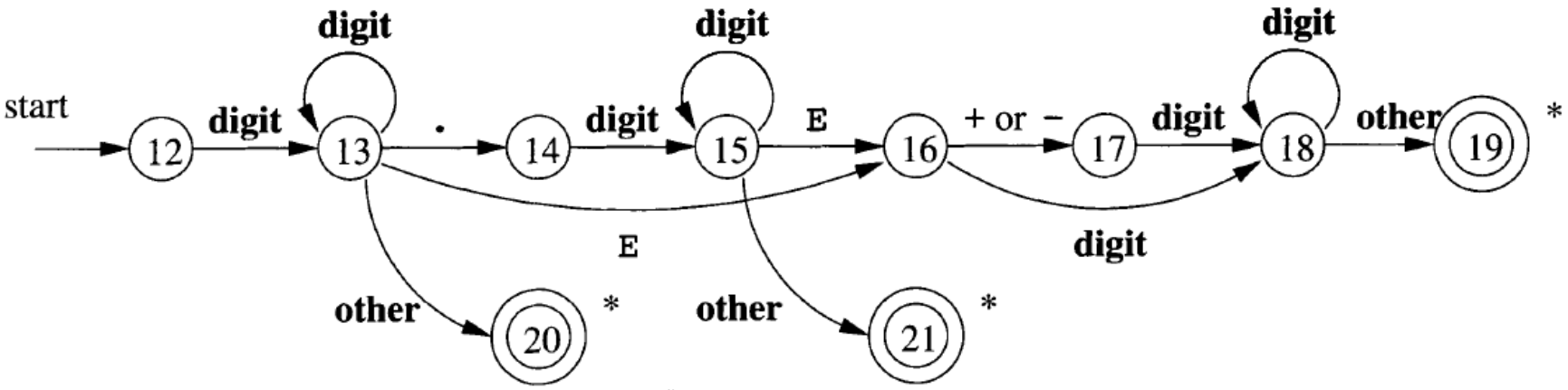


图 3-16　无符号数字的状态转换图

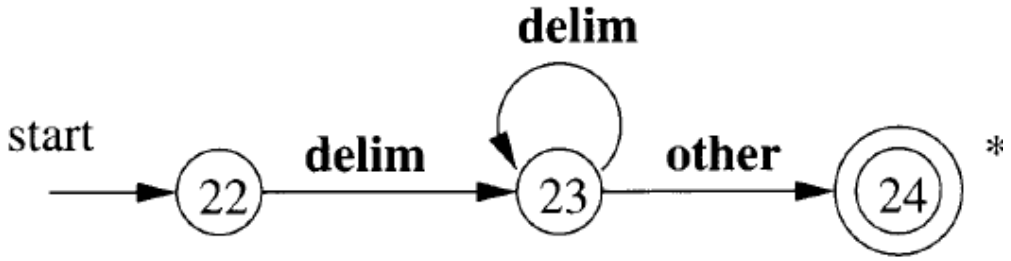


图 3-17　空白符的状态转换图

词法分析器的体系结构

- 从转换图构造词法分析器的方法
  - 变量State记录当前状态
  - 一个switch根据State的值转到相应的代码
  - 每个状态对应于一段代码
    - 这段代码根据读入的符号，确定下一个状态
    - 如果找不到相应的边，则调用fail()进行错误恢复
  - 进入某个接受状态时，返回相应的词法单元
    - 注意状态有*标记时，需要回退forward指针
- 实际是模拟转换图的运行

Relop对应的代码概要

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

内容

- 词法分析器的作用
- 词法单元的规约 (正则表达式)
- 词法单元的识别 (状态转换图)
- 词法分析器生成工具及设计
- 有穷自动机

词法分析工具Lex/Flex

- Lex/Flex是一个有用的词法分析器生成工具
- 通常和YACC/Bison一起使用，生成编译器的前端

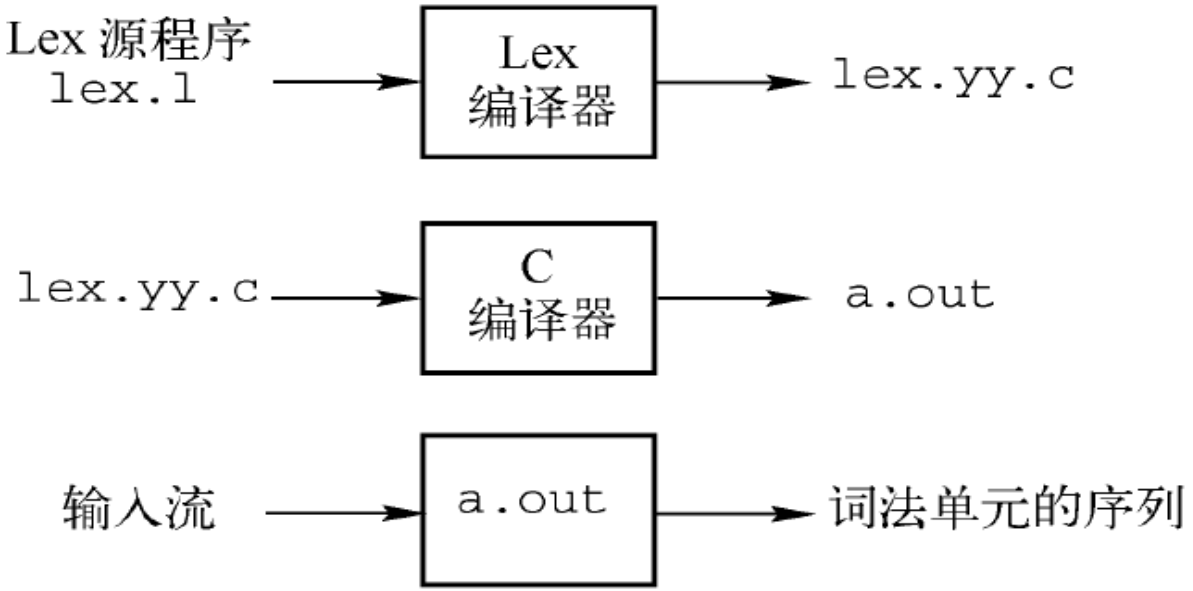


图 3-22　用 Lex 创建一个词法分析器

Lex源程序的结构

- **声明部分**
  - 常量：表示常数的标识符
  - 正则定义
- **转换规则**
  - 模式{动作}
    - 模式是正则表达式
    - 动作表示识别到相应模式时应采取的处理方式
    - 处理方式通常用是C语言代码表示
- **辅助函数**
  - 各个动作中使用的函数

声明部分
%%
转换规则
%%
辅助函数

Lex程序的形式

Lex程序的例子 (1)

```
%{
    /* 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}+)?

%%
```

%{和}%之间的内容一般被直接拷贝到lex.yy.c中；
这里的内容就是一段注释；
LT、LE等的值在Yacc源程序中定义

正则定义

分隔声明部分和转换规则部分

Lex程序的例子 (2)

```
%%

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

%%
```

没有返回，表示继续识别其它的词法单元

把识别到的标识符加入标识符表

识别到数字常量，加入常量表

Lex程序的例子 (3)

- Lex处理源程序时，辅助函数被直接拷贝到lex.yy.c中
- 辅助函数可在规则中直接调用

```
%%

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

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Lex中的冲突解决方法

- **冲突**：多个输入前缀与某个模式相匹配，或者一个前缀与多个模式匹配
- Lex解决冲突的方法
  - 多个前缀可能匹配时，选择最长的前缀
    - 比如，词法分析器把<=当作一个词法单元识别
  - 某个前缀和多个模式匹配时，选择列在前面的模式
    - 如果保留字的规则在标识符的规则之前，词法分析器将识别出保留字

内容

- 词法分析器的作用
- 词法单元的规约 (正则表达式)
- 词法单元的识别 (状态转换图)
- 词法分析器生成工具及设计
- 有穷自动机

有穷自动机

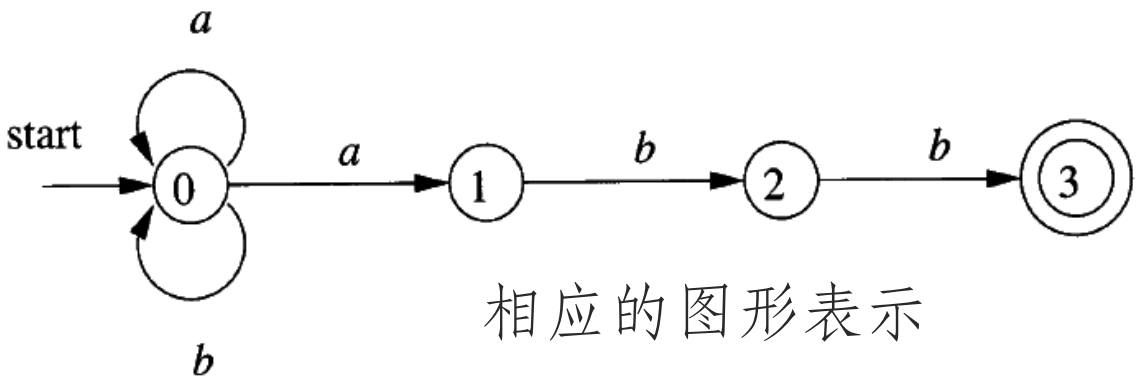
- 本质上和状态转换图相同，但有穷自动机只回答Yes/No
  - 分为两类
    - **不确定的有穷自动机** (Nondeterministic Finite Automata/NFA)：边上的标号没有限制，一个符号可出现在离开同一个状态的多条边上，ε可以做标号
    - **确定的有穷自动机** (Deterministic Finite Automata/DFA)：对于每个状态以及每个符号，有且只有一条边(或最多只有一条边)
- 两种自动机都识别正则语言
  - 对于每个可以用正则表达式描述的语言，均可用某个NFA或DFA来识别；反之亦然

不确定的有穷自动机

- NFA的定义
  - 一个有穷的状态集合S
  - 一个输入符号集合Σ (Input alphabet)
  - **转换函数** (Transition function) 对于每个状态和$\Sigma \cup \{\varepsilon\}$中的符号，给出相应的后继状态集合
  - S中的某个状态s_0被指定为**开始状态/初始状态** (有些定义中可以有多个开始状态)
  - S的一个子集F被指定为**接受状态**集合

NFA的例子

- 状态集合$S = \{0, 1, 2, 3\}$
- 开始状态s_0：0
- 接受状态集合F：{3}
- 转换函数
  - (0, a) → {0, 1}　　(0, b) → {0}
  - (1, b) → {2}　　(2, b) → {3}



相应的图形表示

转换表 (Transition table) 表示法

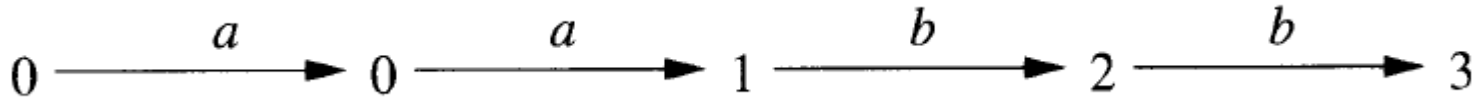
- 用二维表表示NFA的转换函数
  - 每行对应于一个状态
  - 每列对应于一个输入符号或者ε
  - 每个条目表示对应的后继状态集合

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

转换表表示法

输入字符串的接受

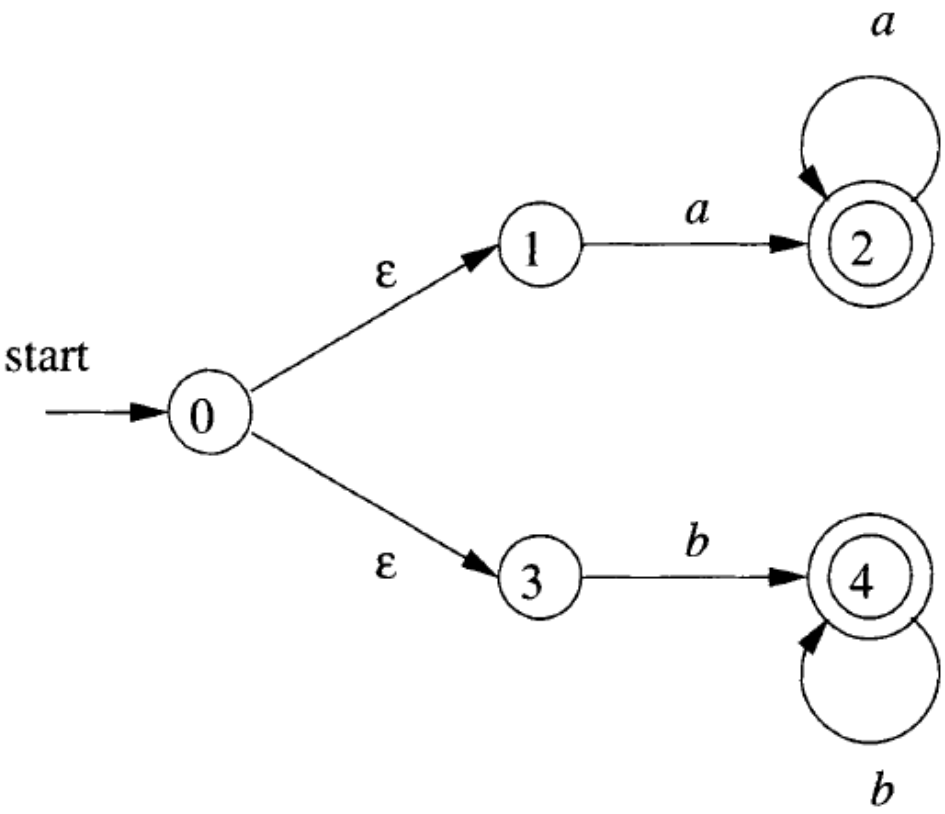
- 一个NFA接受输入字符串x
  - 当且仅当对应的转换图中存在一条从开始状态到某个接受状态的路径，且该路径各条边上的标号按顺序组成x(不含ε标号)
- 前面的NFA接受aabb，因为



- **NFA接受的语言**：从开始状态到达接受状态的所有路径的标号串的集合
  - 即该NFA接受的所有符号串的集合

NFA和相应语言的例子

- 相应的语言：$L(aa^* \mid bb^*)$



确定的有穷自动机 (DFA)

- 一个NFA被称为DFA，如果
  - 没有标号为ε的转换，并且
  - 对于每个状态s和每个输入符号a，有且仅有一条标号为a的离开s的边
- 可以高效判断一个串能否被一个DFA接受
- 每个NFA都有一个等价的DFA
  - 即它们接受相同的语言

DFA的模拟运行

- 假设输入符号是字符
- *nextChar*读入下一个字符
- *move*给出了离开状态*s*且标号为*c*的边的目标状态

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s在F中 ) return "yes";
else return "no";
```

DFA的例子

- 假设输入为ababb，那么进入的状态序列为
  - 0, 1, 2, 1, 2, 3，返回yes

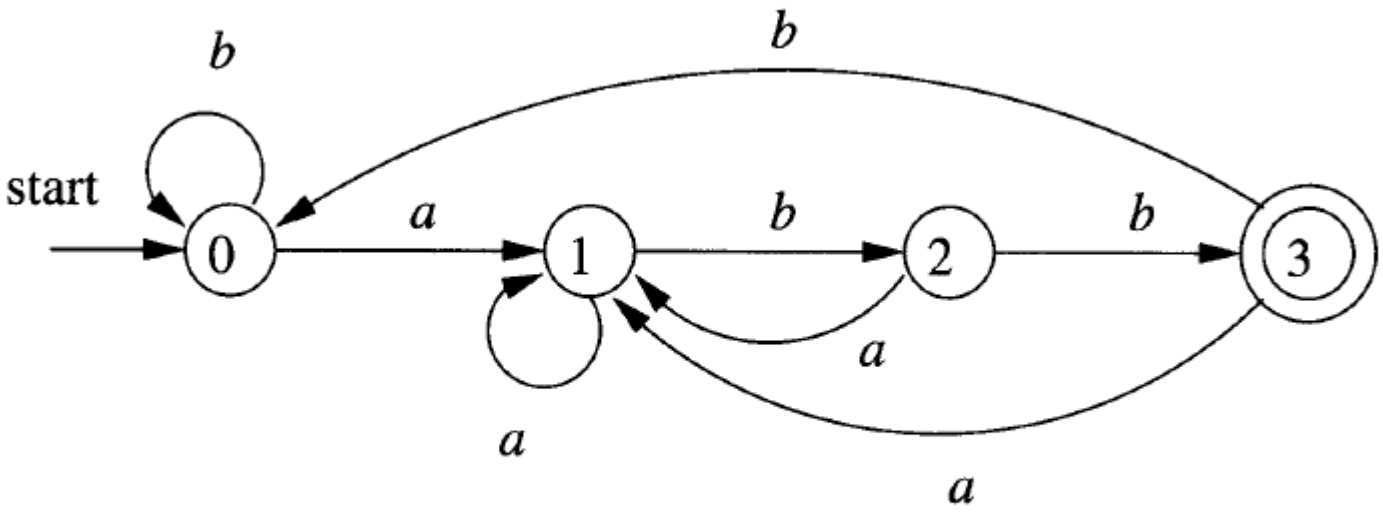


图 3-28　接受(a|b)*abb 的 DFA

从正则表达式到自动机的转换

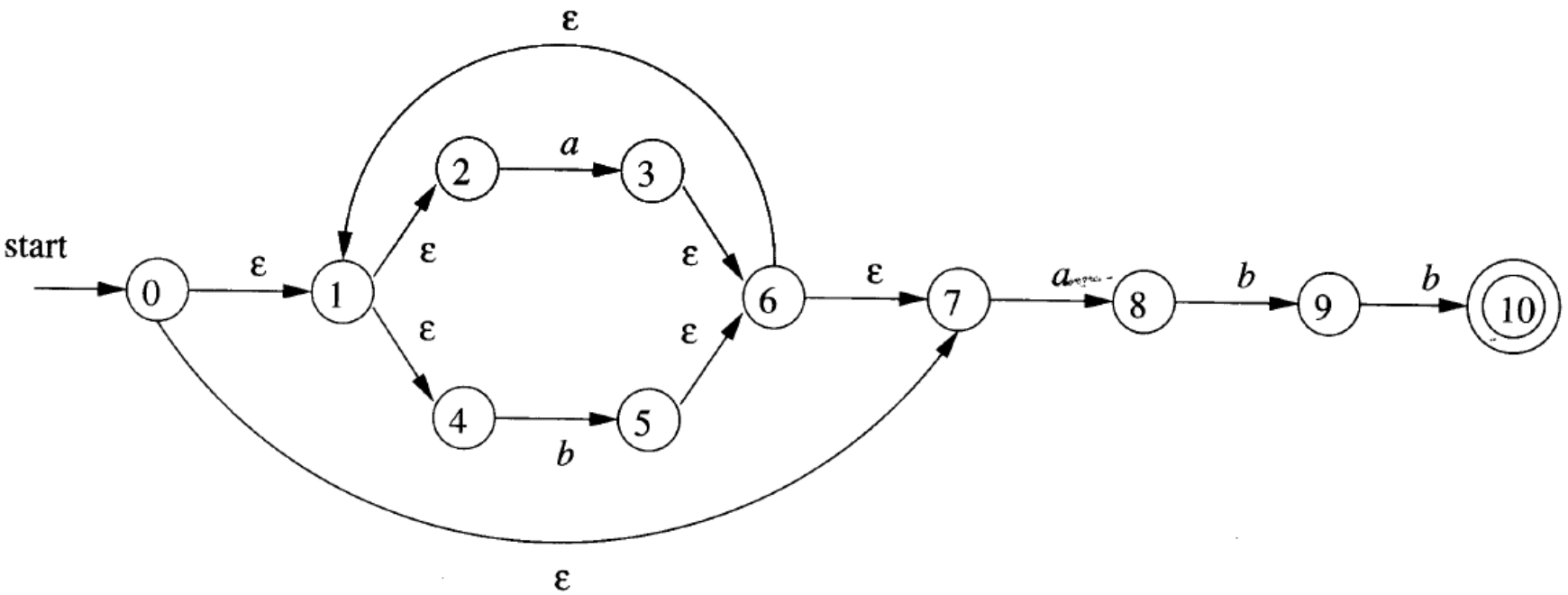
- 正则表达式可简洁、精确地描述词法单元的模式
- 模拟DFA的执行可高效地进行模式匹配
- 将正则表达式转换为DFA的步骤
  - 正则表达式到NFA
  - NFA到DFA

NFA到DFA (子集构造法) (1)

- 基本思想
  - 构造得到的DFA的每个状态和NFA的状态子集对应
  - DFA读入$a_1, a_2, \ldots, a_n$后到达的状态对应于从NFA开始状态出发沿着$a_1, a_2, \ldots, a_n$可能到达的状态集合
  - 在算法中“并行地模拟”NFA在遇到一个给定输入串时可能执行的所有动作

NFA例子

- 下面的NFA能够接受串babb
  - 考虑从开始状态出发，沿着标号分别为b，ba，bab，babb能到达的所有可能状态的集合



NFA到DFA (子集构造法) (2)

- 理论上，最坏情况下DFA的状态个数会是NFA状态个数的指数多个
  - 但对于大部分应用，NFA和相应的DFA的状态数量大致相同

NFA到DFA (子集构造法) (3)

- 算法中使用到的基本操作
  - ε−closure(s)：从NFA状态s开始，只通过ε转换能到达的NFA状态集合
  - ε−closure(T)：从T中某个状态s开始，只通过ε转换能到达的NFA状态集合
  - move(T, a)：从T中某个状态s出发，通过一个标号为a的转换能到达的NFA状态集合

NFA到DFA (子集构造法) (4)

- 计算ε–closure(T)的算法
  - 实际上是一个图搜索过程 (只考虑ε标号边)

```
将T的所有状态压入stack中;
将 ε-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ε的转换到达状态u)
                if ( u不在 ε-closure(T)中) {
                        将u加入到ε-closure(T)中;
                        将u压入栈中;
                }
}
```

NFA到DFA(子集构造法)(5)

- 整个算法实际是一个搜索过程
  - *Dstates*中的一个状态未加标记表示还没有搜索过它的各个后继

```
一开始, ε-closure(s0)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 = ε-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;
        }
}
```

子集构造法的例子 (1)

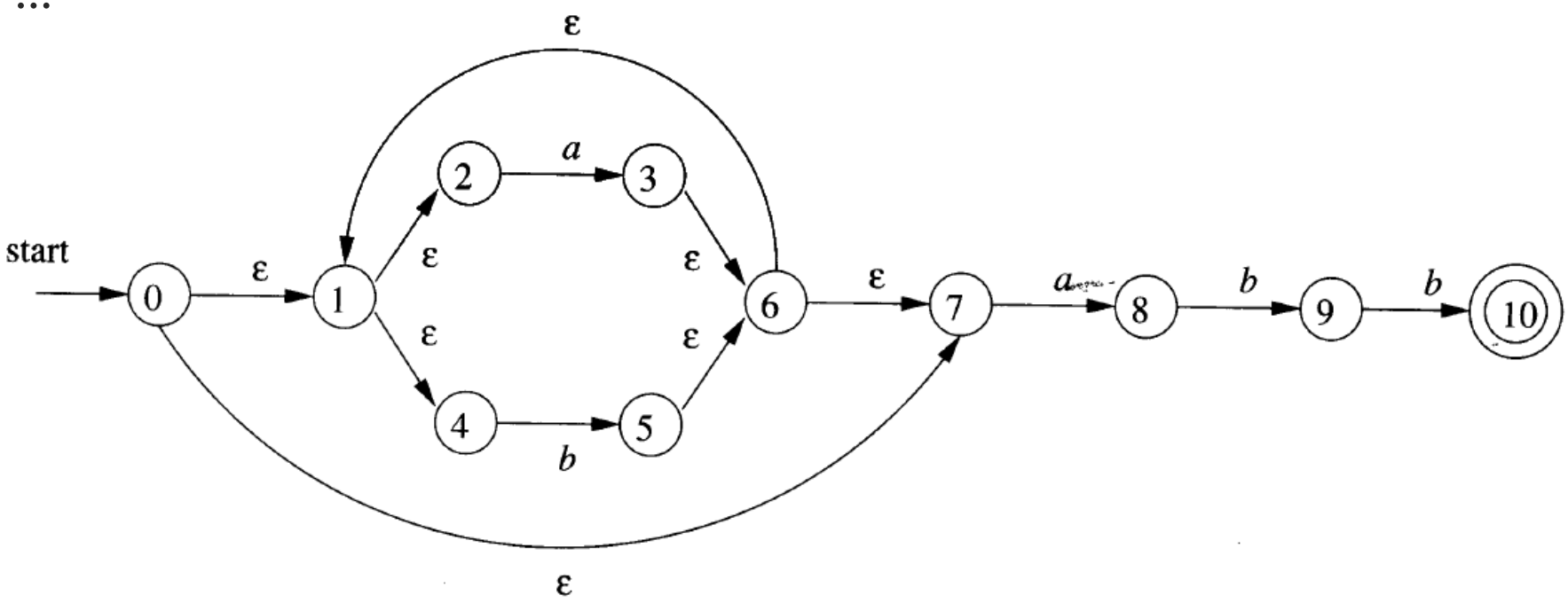
A: ε–closure(0) = {0, 1, 2, 4, 7}

B: Dtran[A, a] = ε–closure({3, 8}) = {1, 2, 3, 4, 6, 7, 8}

C: Dtran[A, b] = ε–closure({5}) = {1, 2, 4, 5, 6, 7}

D: Dtran[B, b] = ε–closure({5, 9}) = {1, 2, 4, 5, 6, 7, 9}

…



子集构造法的例子 (2)

- 开始状态：A；接受状态：E

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

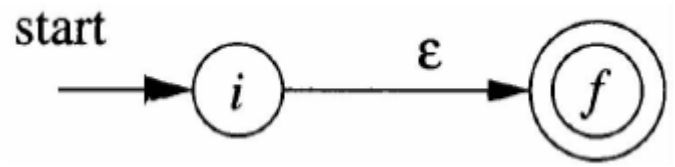
图 3-35　DFA D 的转换表 $Dtran$

正则表达式到NFA

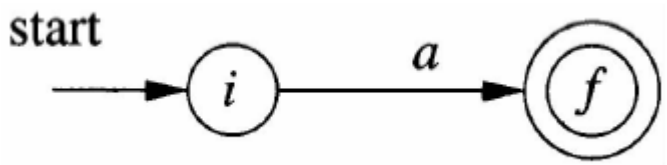
- 基本思想
  - 根据正则表达式的递归定义，按照正则表达式的结构递归地构造出相应的NFA
  - 算法分成两个部分
    - 基本规则处理ε和单符号的情况
    - 对于每个正则表达式的运算，建立组合相应NFA的方法

转换算法 (1)

- 基本规则部分
  - 表达式ε

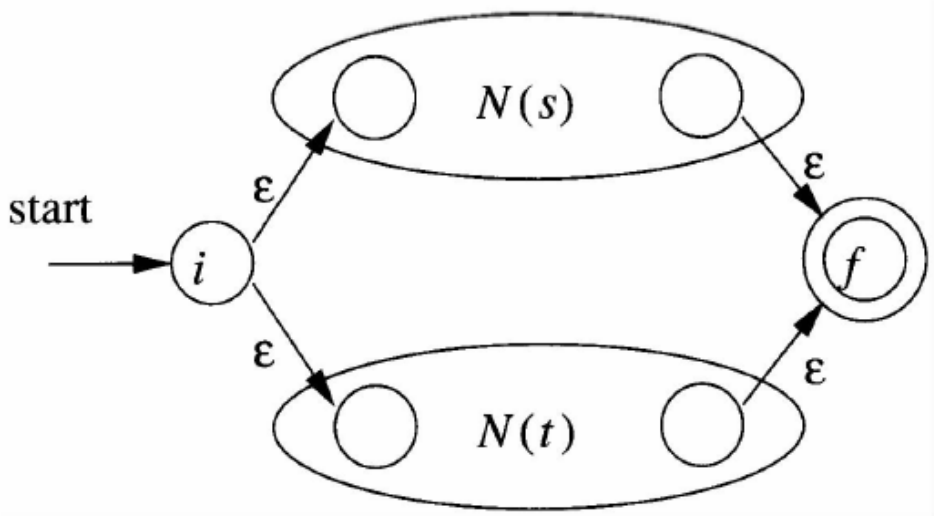


  - 表达式a

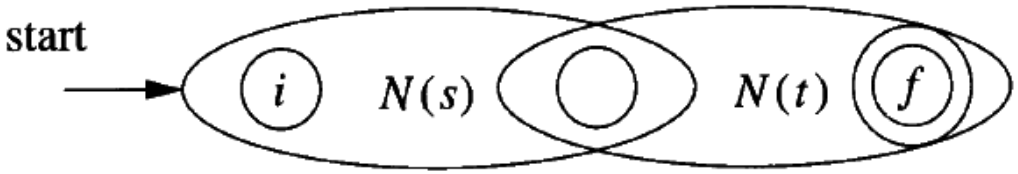


转换算法 (2)

- 归纳部分
  - $s \mid t$

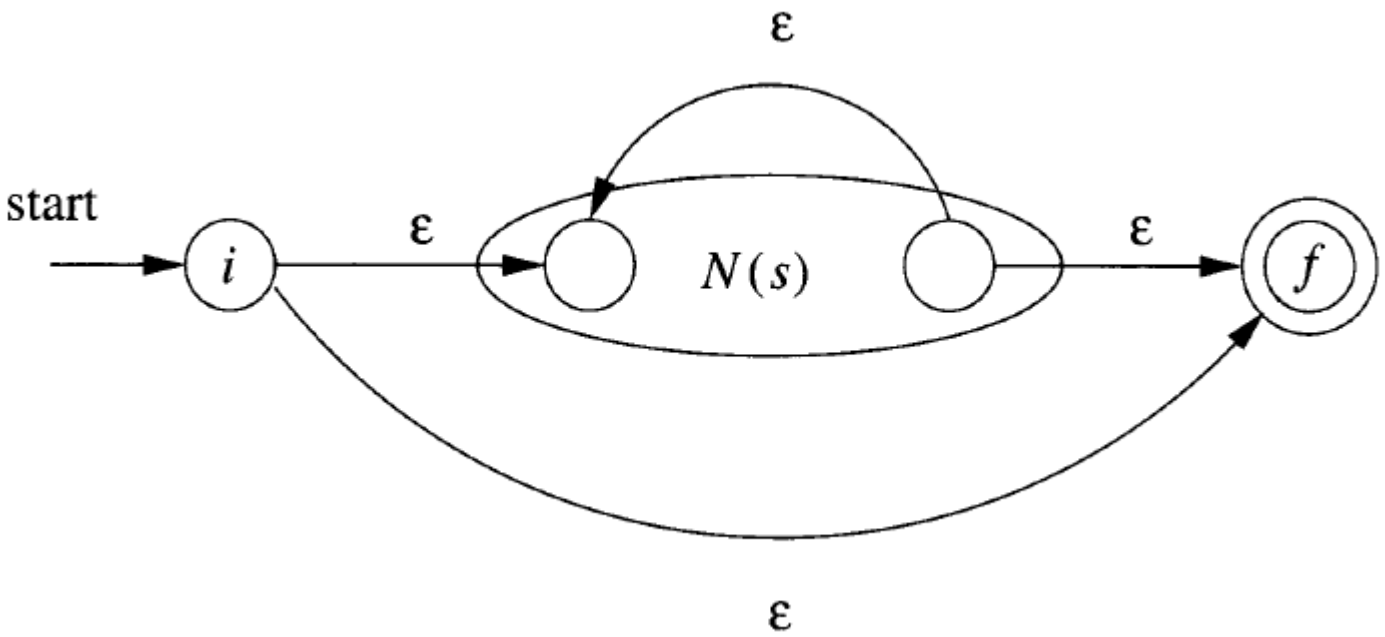


  - st



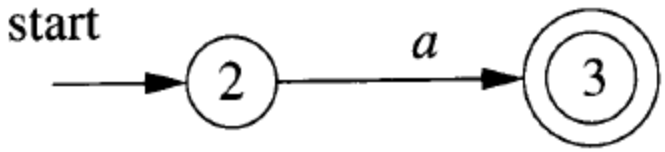
转换算法 (3)

- 归纳部分
  - s^*

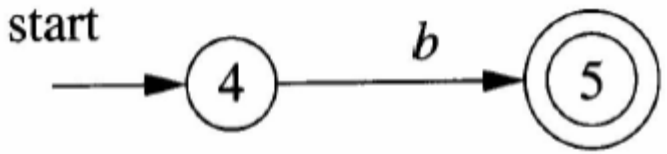


正则表达式到NFA的例子 (1)

- 正则表达式：(a|b)*abb
- 第一个a对应的NFA

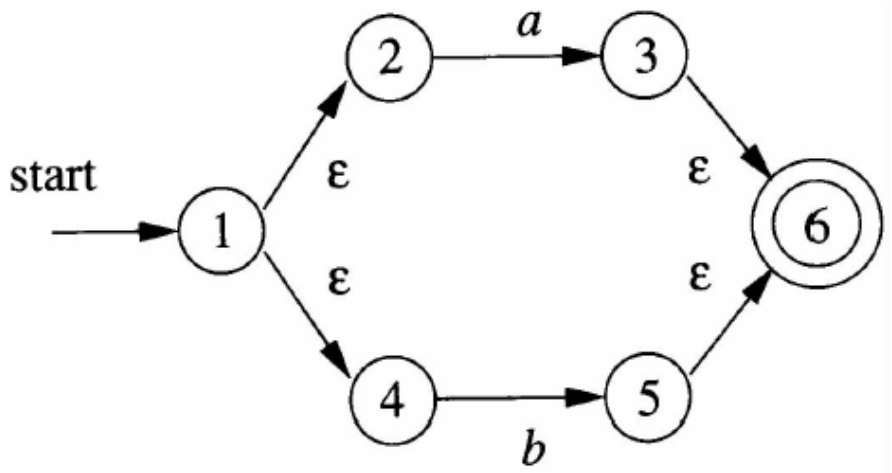


- 第一个b对应的NFA



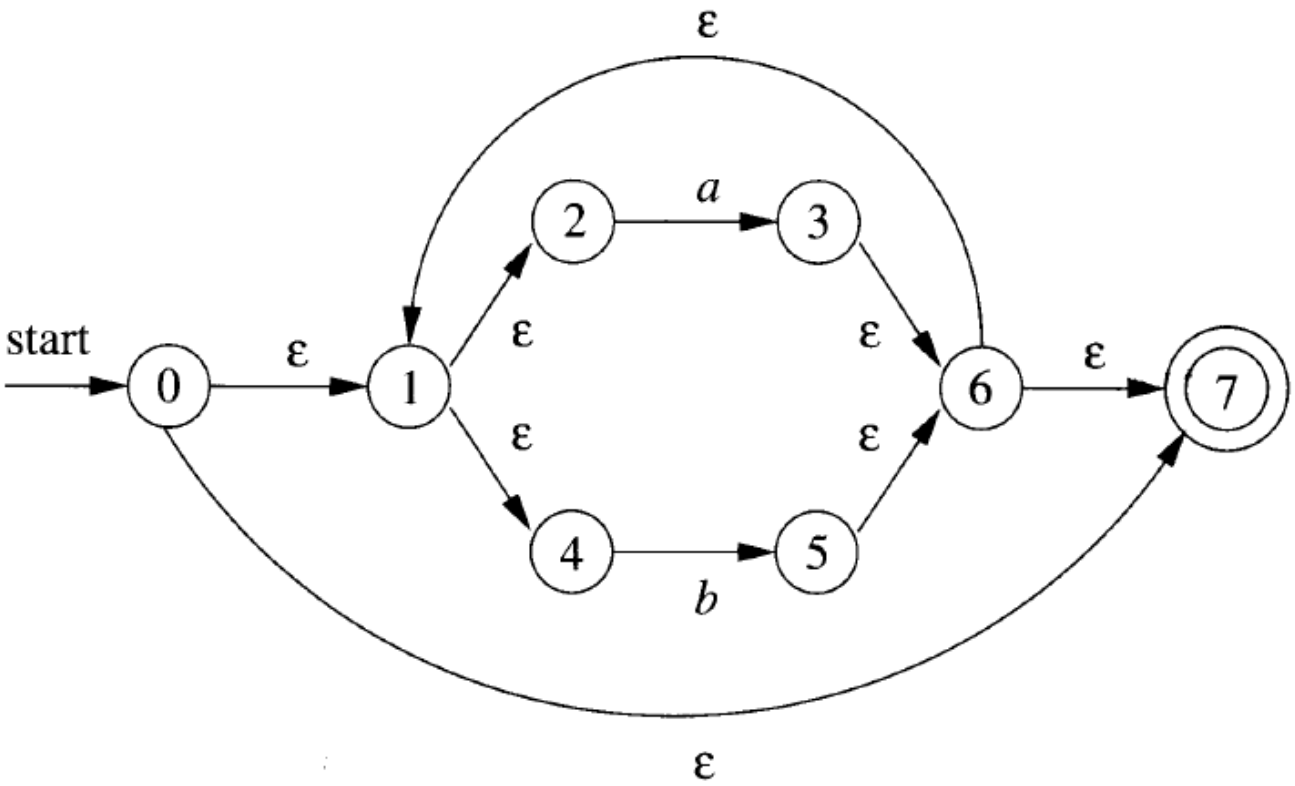
正则表达式到NFA的例子 (2)

- (a|b)的NFA



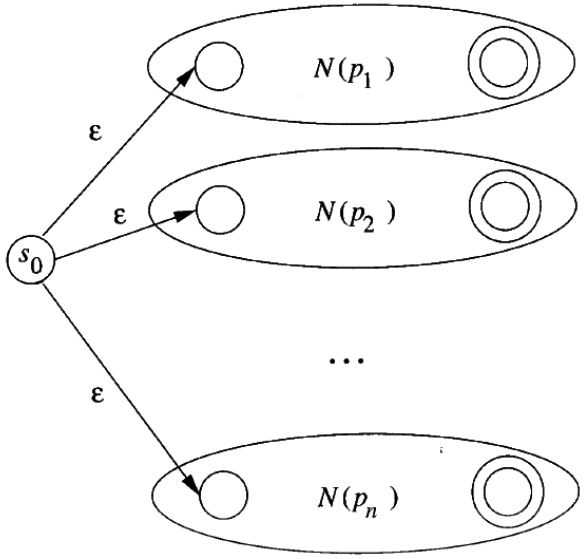
正则表达式到NFA的例子 (3)

- (a|b)*的NFA



NFA合并的方法

- 合并方法
  - 引入新的开始状态，并引入从该开始状态到各个原开始状态的ε转换
  - 得到的NFA所接受的语言是原来各个NFA语言的并集
  - 不同的接受状态代表不同的模式

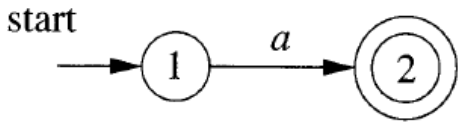


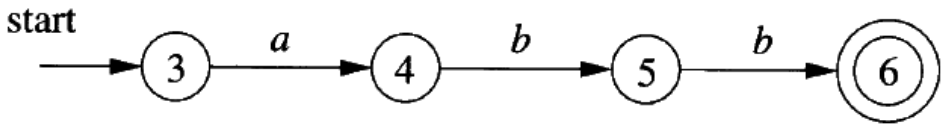
确定化NFA后的处理

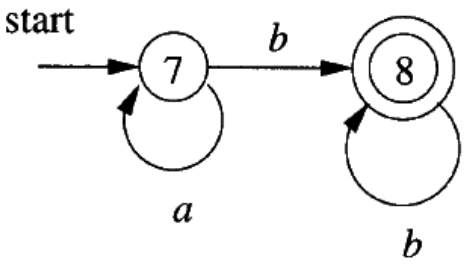
- 对得到的NFA进行确定化，得到DFA
- 一个DFA的接受状态对应于NFA的状态子集，其中至少包括一个NFA的接受状态
  - 如果其中包括多个对应于不同模式的NFA接受状态，则表示当前的输入前缀对应于多个模式，存在冲突
  - 找出第一个这样的模式，将该模式作为此DFA接受状态的输出

例子 (1)

- 假设有三个模式
  - a {A1}
  - abb {A2}
  - a^*b^+ {A3}
- 构造各模式的NFA

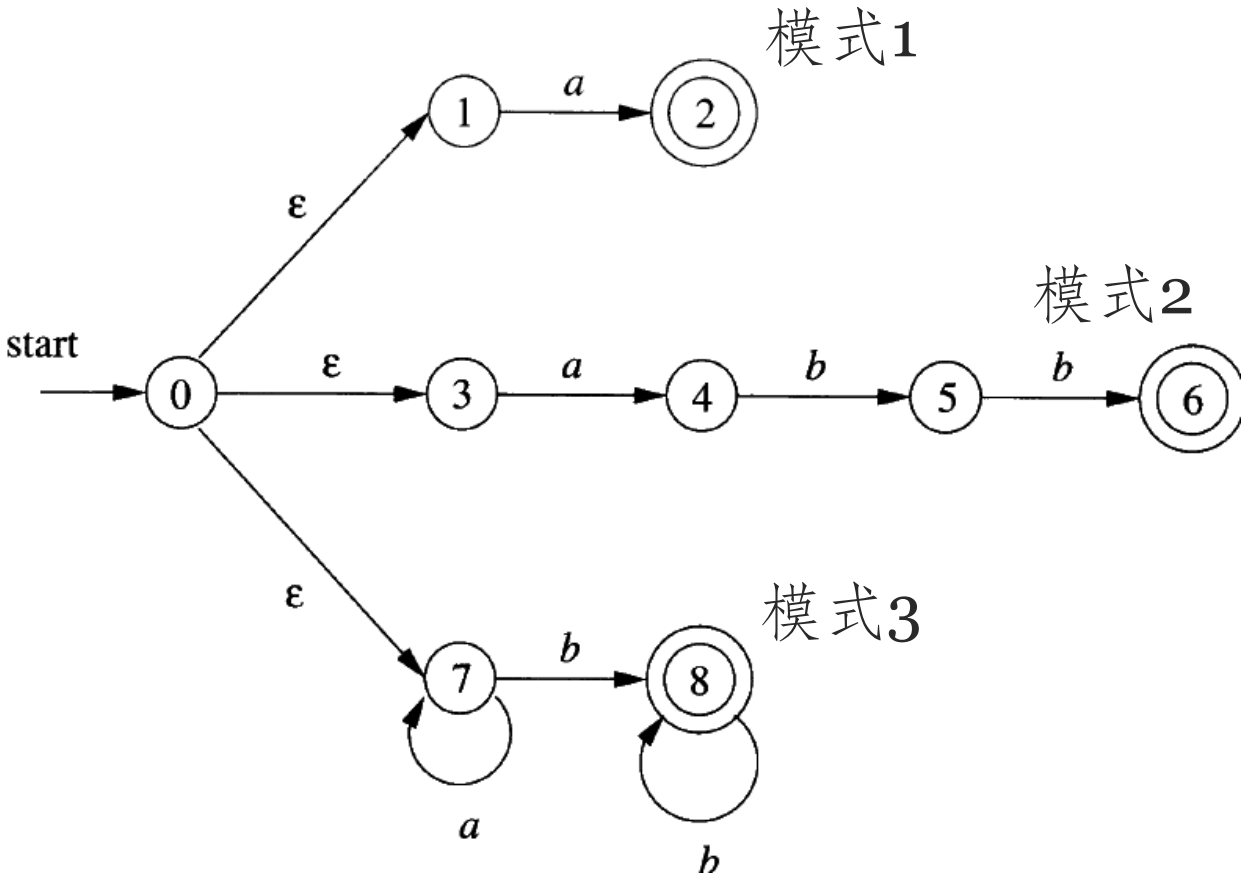






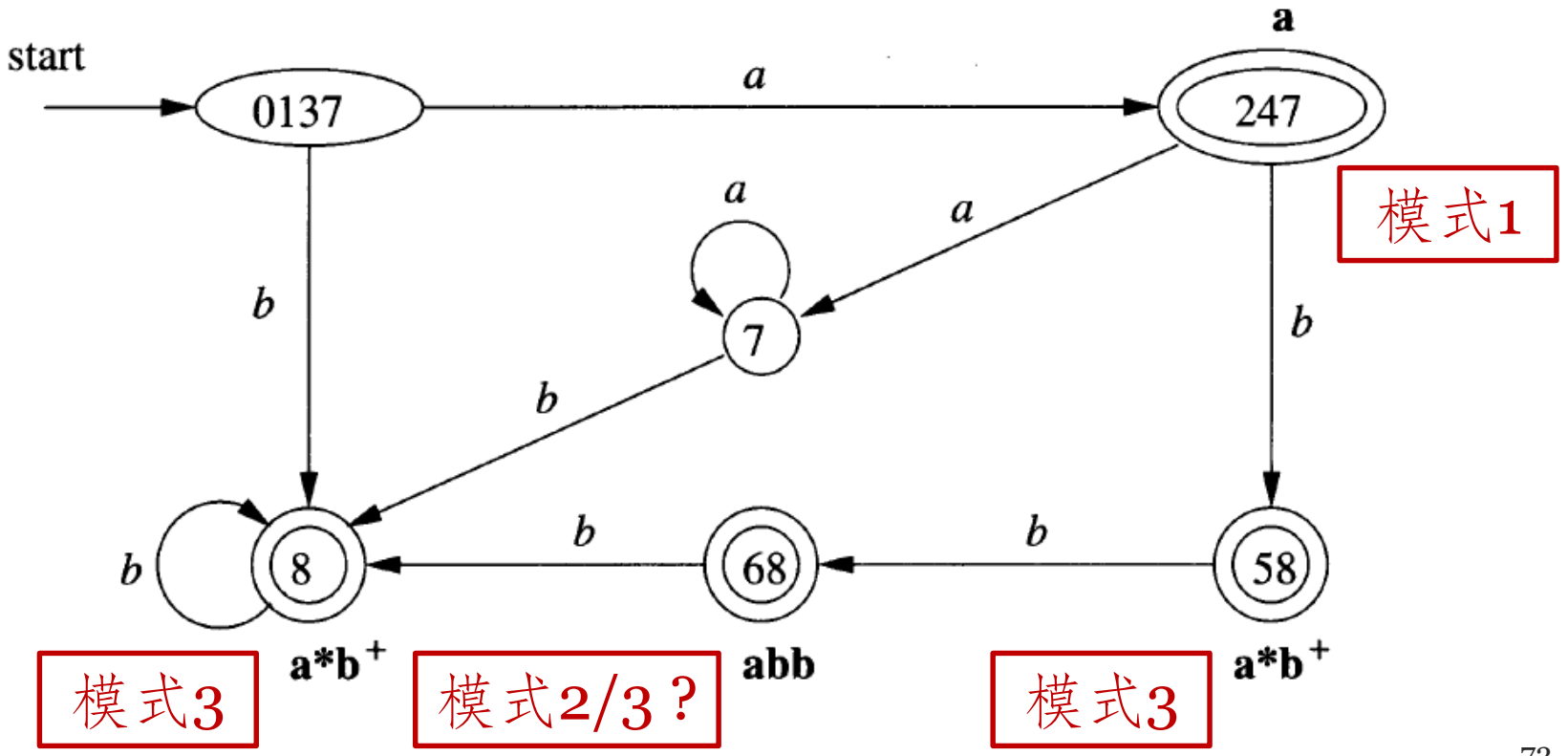
例子 (2)

- 合并NFA
  - 2：模式1
  - 6：模式2
  - 8：模式3



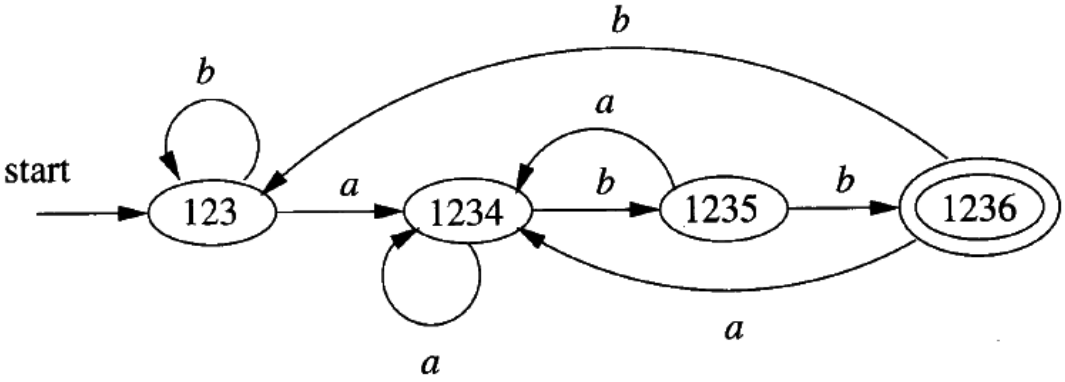
例子 (3)

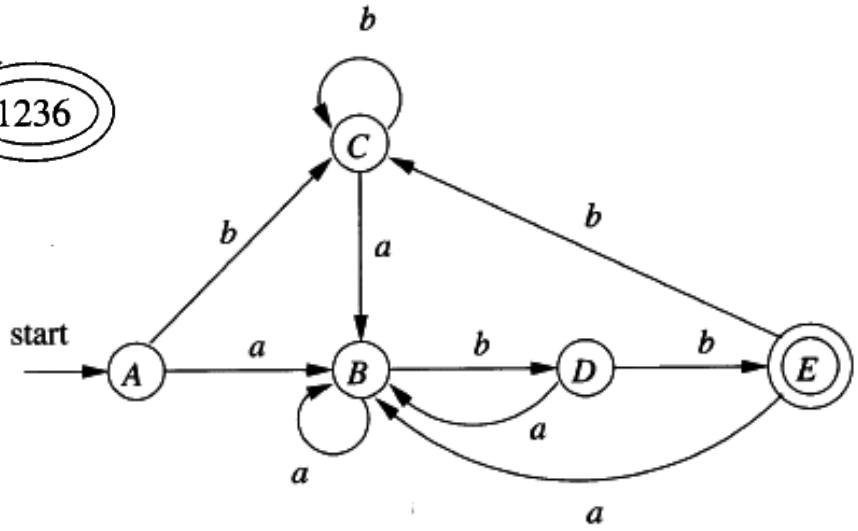
- 确定化得到如下DFA
  - DFA状态68对应的NFA状态子集为{6, 8}，其对应的模式是abb (即模式2)，而不是a^*b^+ (即模式3)



DFA状态数量的最小化

- 一个正则语言可对应于多个识别此语言的DFA
- 通过**DFA的最小化**可得到状态数量最少的DFA (不计同构，这样的DFA是唯一的)

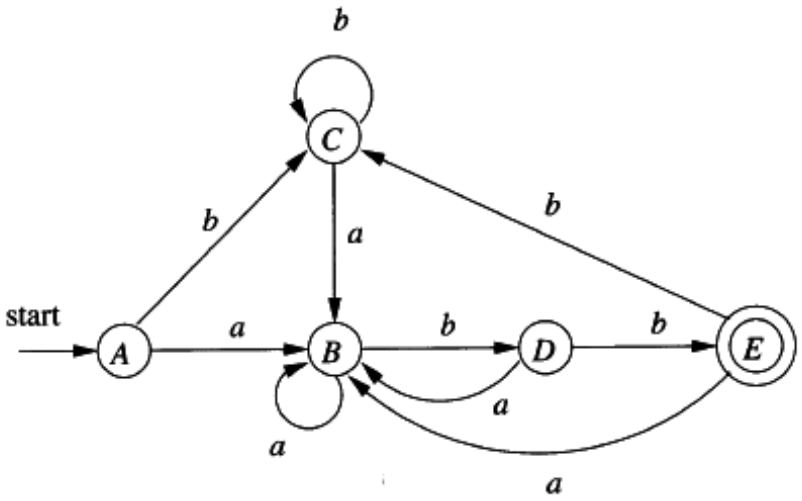




两个等价的DFA：都识别 $(a|b)^*abb$

状态的区分

- 状态的**可区分**
  - 如果存在串x，使得从状态s1和s2，一个到达接受状态而另一个到达非接受状态，那么x就区分了s1和s2
  - 如果存在某个串区分了s和t，我们说s和t是可区分的，否则它们是不可区分的
- 不可区分的两个状态就是**等价**的，可以合并



空串区分了E和其它状态；
bb区分了A和B

DFA最小化算法

- 把所有可区分的状态分开 (迭代过程)
  - 基本步骤：ε区分了接受状态和非接受状态
  - 归纳步骤：如果s和t是可区分的，且s'到s、t'到t有标号为a的边，那么s'和t'也是可区分的
- 最终没有区分开的状态就是等价的
  - 所有的**死状态**都是等价的
- 从划分得到的等价类中选取代表，并重建DFA

最小化算法(划分部分)

1. 设置初始划分：$\Pi = \{S - F, F\}$
2. 迭代，不断划分

for (Π中的每个元素G) {

细分G，使得G中的s、t仍然在同一组中 iff

对任意输入a，s、t都到达Π中的同一组；

Π_{new} = 将Π中的G替换为细分得到的小组

}

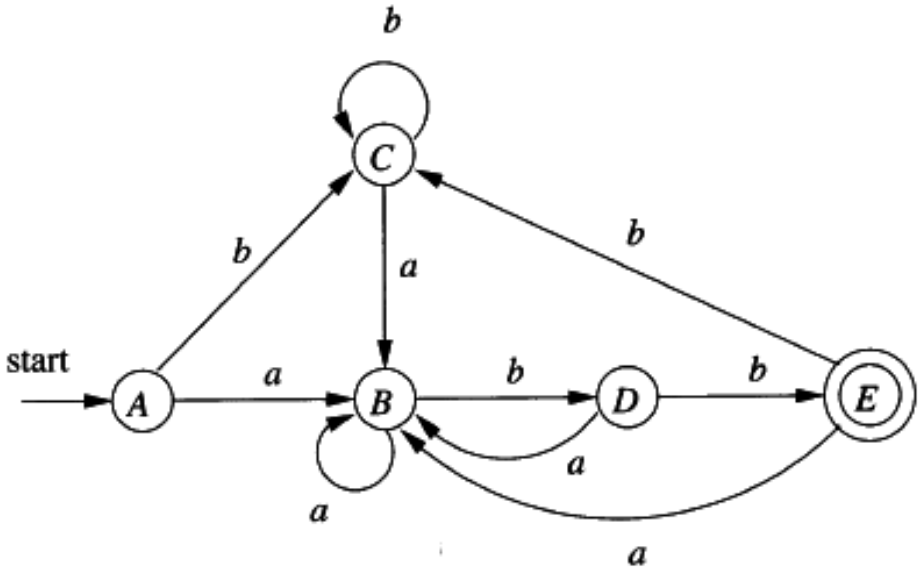
3. 如果$\Pi_{new} == \Pi$，令$\Pi_{final} = \Pi$，算法完成；否则$\Pi = \Pi_{new}$，转步骤2

最小化算法 (构造部分)

- 在Π_{final}的每个组中选择一个状态作代表，作为最小化DFA中的状态
  - **开始状态**就是包含原开始状态的组的代表
  - **接受状态**就是包含了原接受状态的组的代表 (这个组一定只包含接受状态)
  - 转换关系构造如下
    - 如果s是G的代表，而原DFA中s在a上的转换到达t，且t所在组的代表为r，那么最小化DFA中有从s到r的在a上的转换

DFA最小化的例子

- 初始划分：{A, B, C, D}，{E}
- 处理{A, B, C, D}：b把它细分为{A, B, C}和{D}
- 处理{A, B, C}：b把它细分为{A, C}和{B}
- 选取A, B, D和E为代表，构造得到最小DFA



状态	a	b
A	B	A
B	B	D
D	B	E
E	B	A

图 3-65　状态最少 DFA 的转换表

词法分析器状态的最小化

- 基本思想和DFA最小化算法相同
- 差别
  - 词法分析器中的接受状态对应于不同的模式
  - 对应不同模式的接受状态一定是不等价的
  - 初始划分为
    - 所有非接受状态集合 + 对应各模式的接受状态集合
- 其余划分和构造的方法均相同
- 接受状态对应的模式就是原来的模式

例子

- 初始划分：{0137, 7}, {247}, {68}, {8, 58}, {Φ}
  - 增加死状态Φ