编译原理

Compiler Construction Principles

朱 青

信息学院计算机系，

中国人民大学，

zqruc2012@aliyun.com

第2章:词法分析 (Lexical Analysis)

- ⌘2.1 词法分析程序的功能
- ⌘2.2 词法分析器的设计
- ⌘2.3 正规表达式 (Regular Expression)
- ⌘2.4 有限自动机
- ⌘2.5 词法分析器的自动生成

2.4 有限自动机

- ⌘2.4.1 确定有限自动机（DFA）
- ⌘2.4.2 非确定有限自动机（NFA)确定化
- ⌘2.4.3 具有ε-转移的NFA M确定化
- ⌘2.4.4 DFA的化简
- ⌘2.4.5 正规式与有限自动机的等价性
- ⌘2.4.6 正规文法与有限自动机

2.4.5 正规式与有限自动机的等价性

定义1：(子集法）若I为状态集S的子集。

$$I_a = \varepsilon\text{–}closure(J) \qquad a \in \Sigma$$

则称：J是从I出发，经过一条a弧所能到达的状态子集。

定理1:

Σ上的NFA M 所识别的字的全体构成Σ上的正规式V.

定理2:

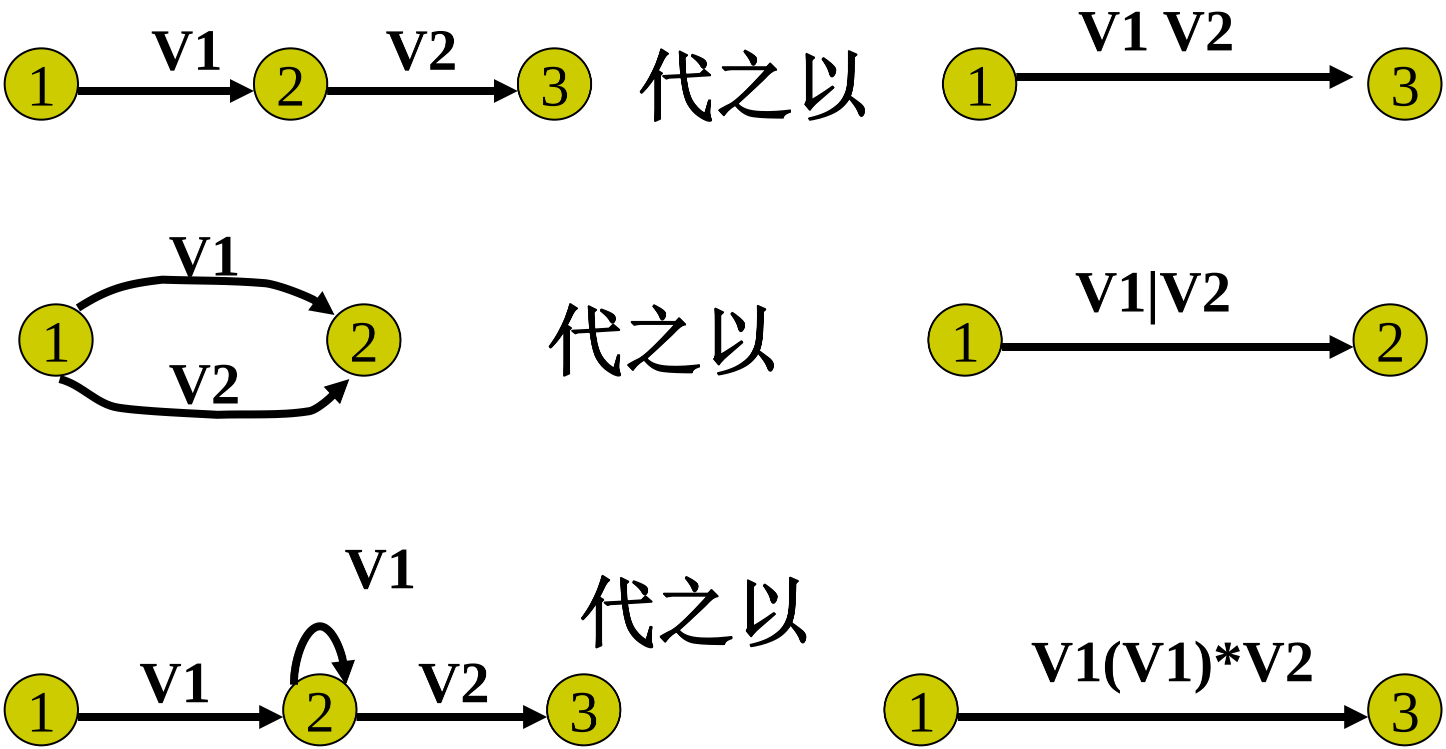
对Σ上的任何正规集,总存在一个 DFA M,使L(V)=L(M).

证明1:

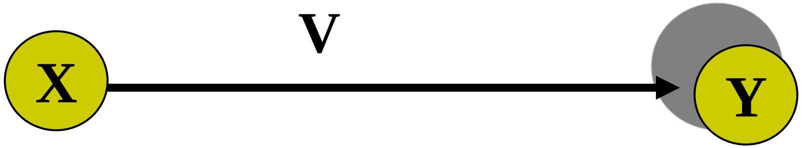
把转换图的意义拓宽,令每条弧上可以标记正规式.

在给定的NFA M上加二个新结,一个为初态X,从X用ε弧连接M的所有初态,另一为Y,从M的所有终态用ε弧连到Y,新的NFA M’与M等价.

对M’用以下等价规则,消去除X,Y以外的其它结点.

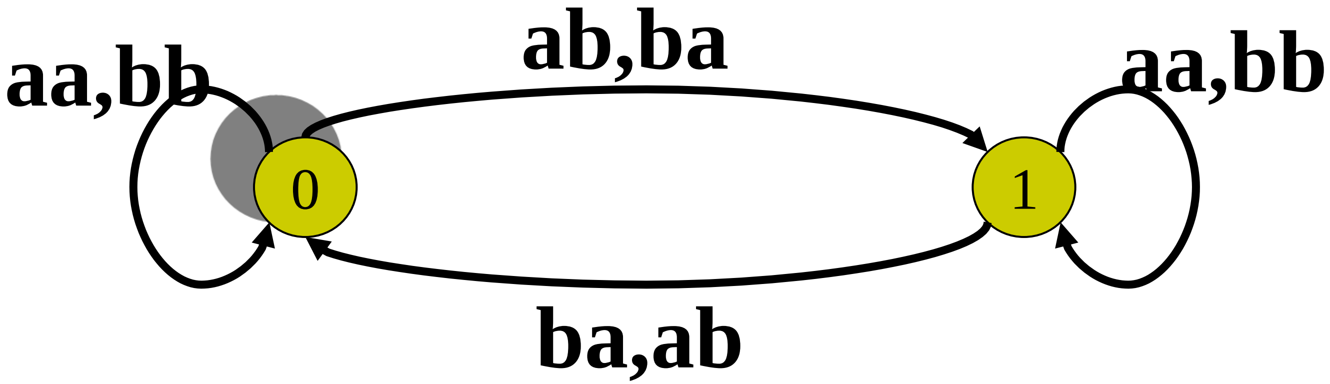


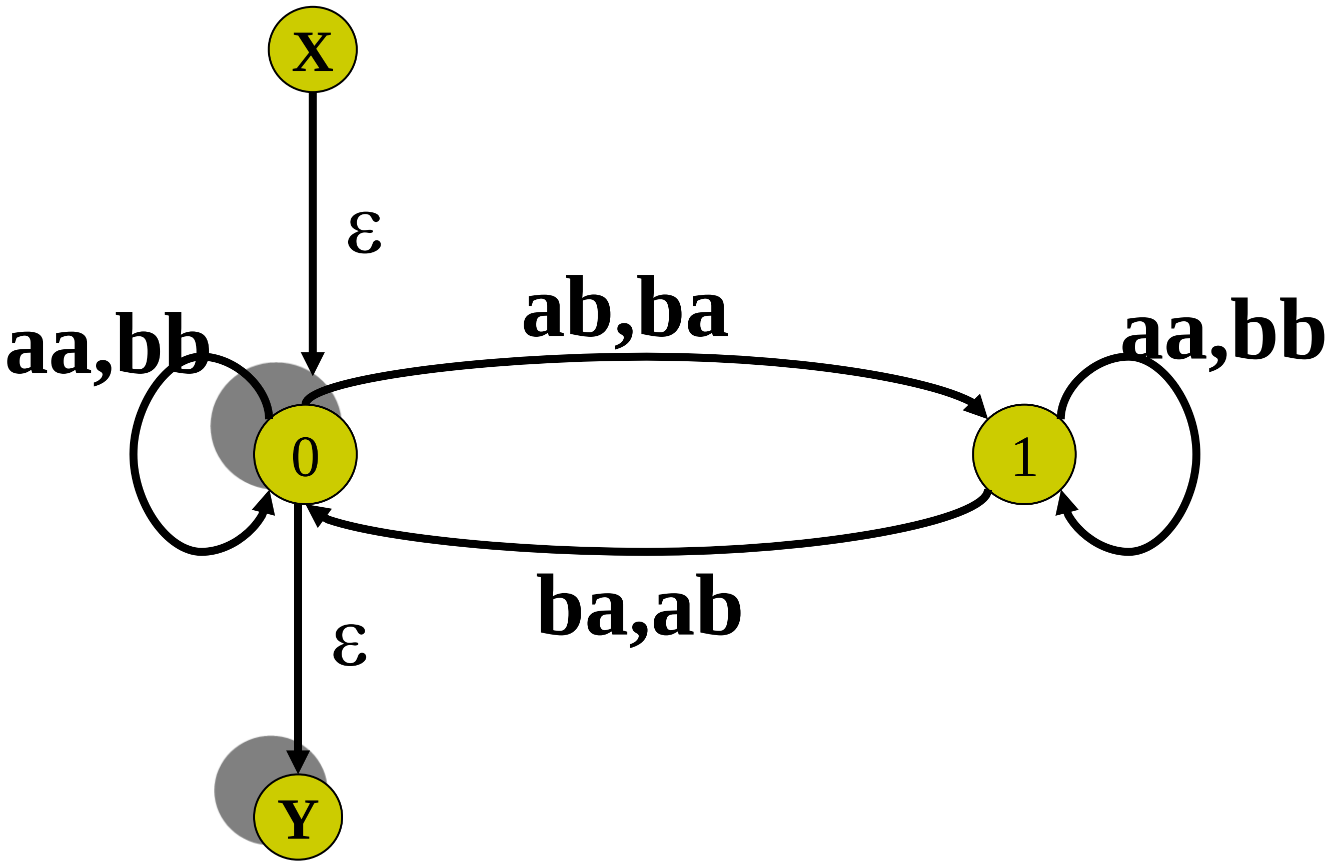
于是,最后得到只有初态X和终态Y的NFA,其标记为一正规式.即:

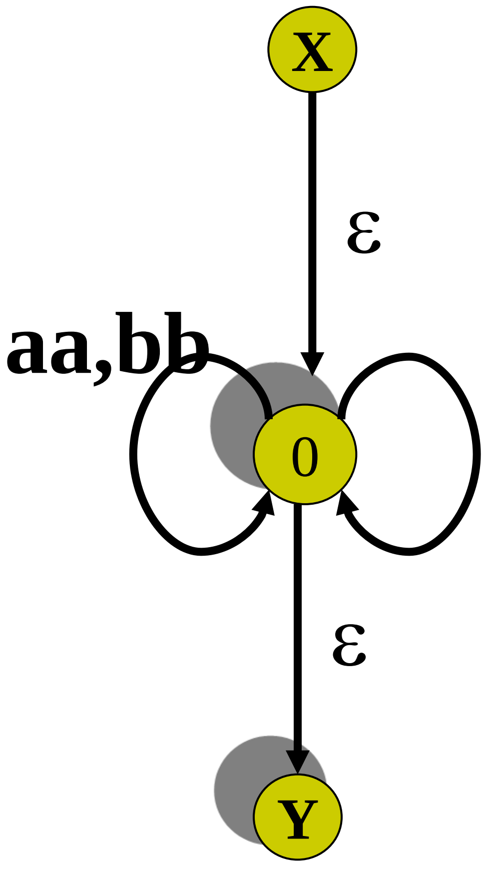


例如:把下面的NFA M ==>正规式.

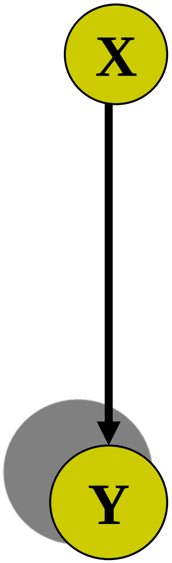
NFA M 是识别具有偶数a和偶数个b的非有限自动机:




X
ε
ab,ba
aa,bb
0
1
aa,bb
ba,ab
ε
Y



(ab|ba)(aa|bb)*(ba|ab)

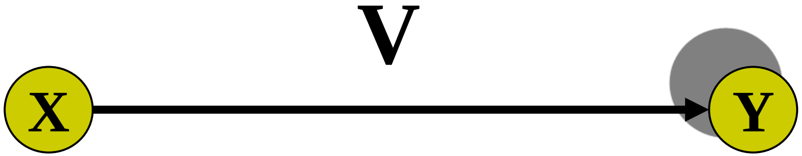


((aa|bb)|(ab|ba)(aa|bb)*(ba|ab))*

证明2：分两步：

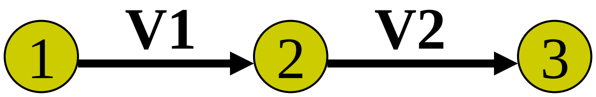
1）对给定的正规式构成一个NFA M。

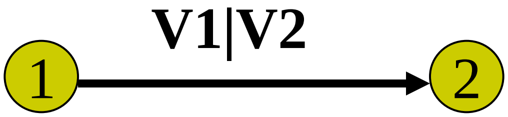
先写出：



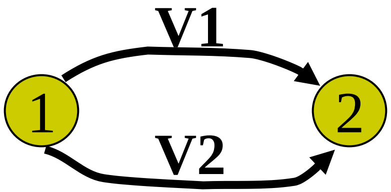
用以下规则对V进行分解并加进新结。

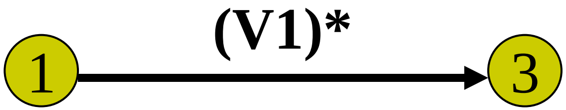
代之以



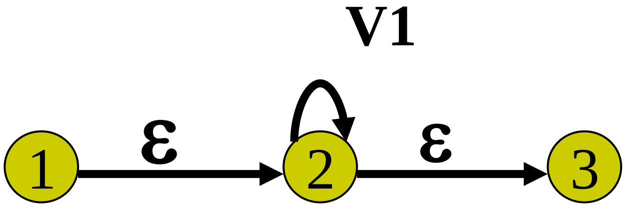


代之以





代之以



在分解过程中，要求：

（1）X，Y始终为唯一的初态和终态。

（2）所加新结其名字彼此不同。

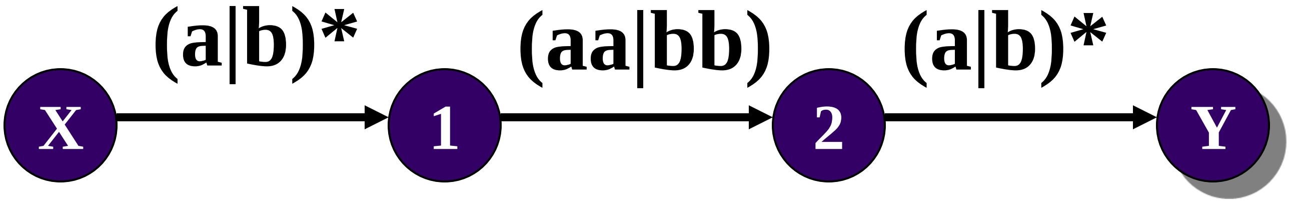
（3）弧上的标记必须是字符或空字。

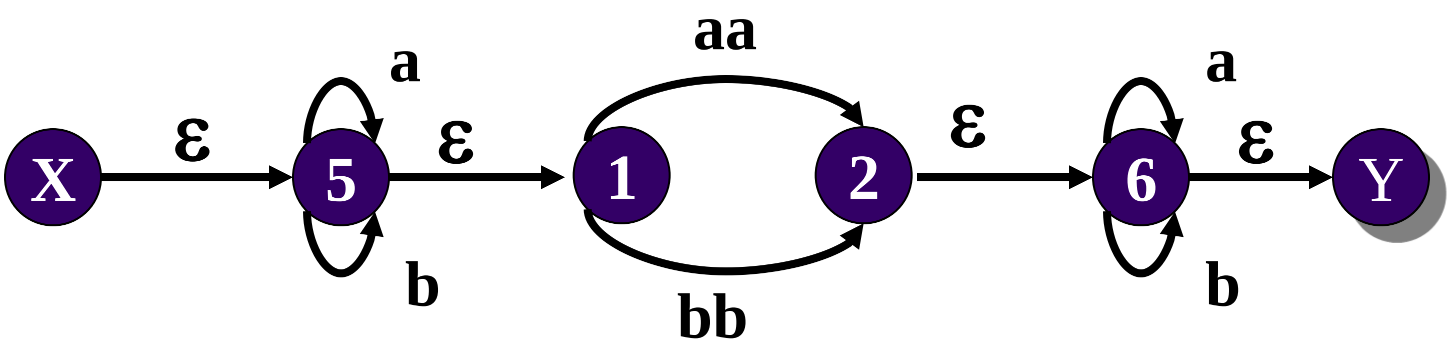
2）把NFA M 确定化：采用子集法进行确定化。
为简单记，设：Σ={a, b}.
因为一个状态矩阵可唯一的刻画一个DFA，故用子集法造该矩阵。

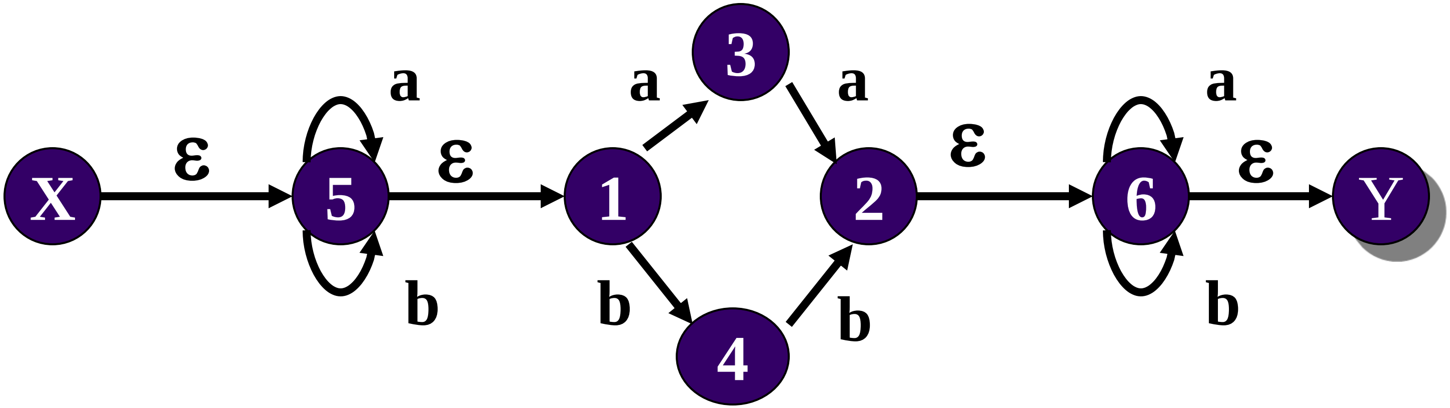
状态子集	Ia	Ib

（在前面章节中已讲）

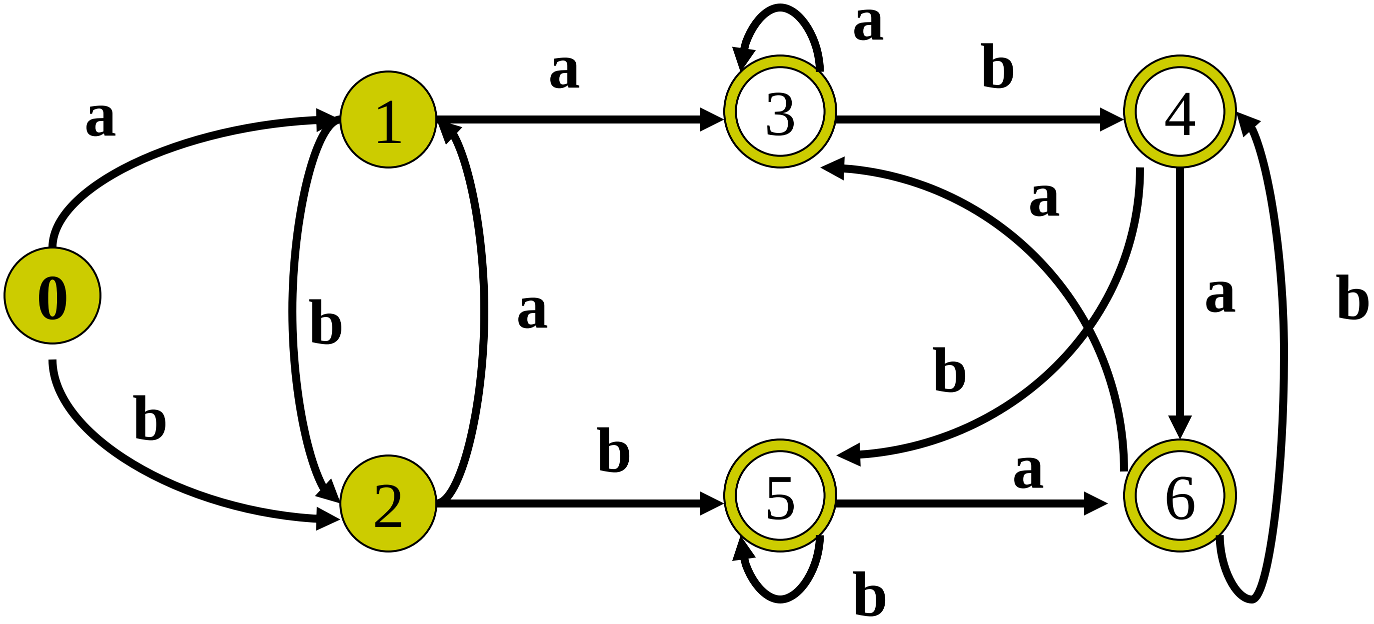
例：设 V=(a|b)*(aa|bb)(a|b)*




X
5
1
2
6
Y
ε
a
b
aa
bb


X
5
1
3
4
2
6
Y
ε
a
b

I	Ia	Ib
0{X,5,1}	{5,3,1}	{5,4,1}
1{5,3,1}	{5,3,1,2,6,Y}	{5,4,1}
2{5,4,1}	{5,3,1}	{5,3,1,2,6,Y}
3{5,3,1,2,6,Y}	{5,3,1,2,6,Y}	{5,4,1,6,Y}
4{5,4,1,6,Y}	{5,3,1,6,Y}	{5,3,1,2,6,Y}
5{5,4,1,2,6,Y}	{5,3,1,6,Y}	{5,4,1,2,6,Y}
6{5,3,1,6,Y}	{5,3,1,2,6,Y}	{5,4,1,6,Y}


a
a
a
b
0
1
3
4
b
a
a
a
b
b
b
2
b
5
a
6
b

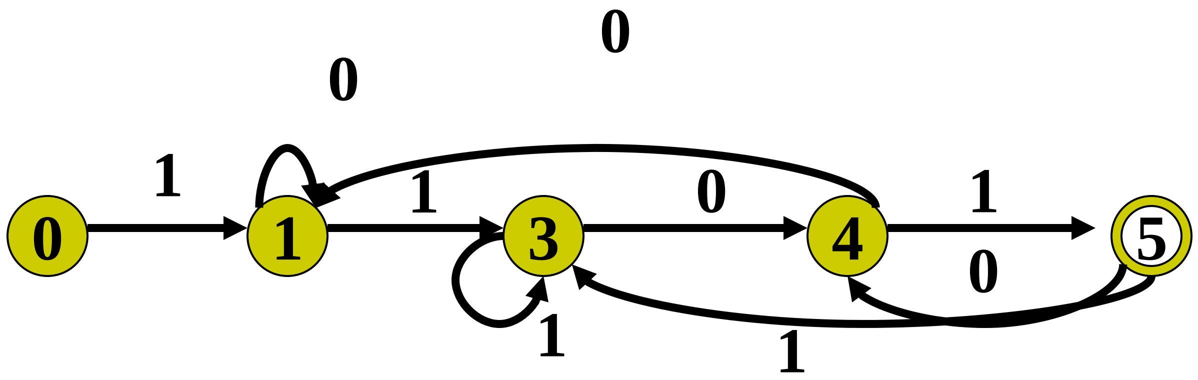
推论：

一个子集是正规的，当且仅当他可由一个DFA（或NFA）所识别。

例题：构造下列正规式相应的DFA

1（0|1）*101

答案:



2.4.6正规文法与有限自动机

正规文法是描述单词符号的另一种方法.

例如:

标识符 → 字母|<标识符> 字母

|<标识符>数字.

<整常数> → 数字|<整常数> 数字

<分界符> → + | - | * | / | , | ; | (|) | …

<分界符> → <冒号>=|<星号>*|< 斜竖>/ | …

<冒号> → :

<星号> → *

<斜竖> → /

定义式有两种类型： P → t

P → Qt

(1)正规文法

定义:

1）如果文法G=(V_T，V_N，S，P）

其中：(1)V_T是非空有限集,每个元素是一个终结符.

(2)V_N是非空有限集,每个元素是一个非终结符.

(3) S是一个非终结符,是开始符号.

(S在产生式的左部必须至少出现一次)

(4) P是产生式的集合:它的每一个产生式P的形式为:

$$A \longrightarrow aB \quad 或 \quad A \longrightarrow a$$

其中， A，B∈V_N，a∈V_TU{ε}，

则称 G 是右线性文法 。

2） 若文法G中的每一个产生式的形式为

$A \longrightarrow Ba$ 或 $A \longrightarrow a$

则称 G是左线性文法。

3）右线性文法和左线性文法都称为3型文法. 3型文法也称为正规文法. 它所产生的语言称为3型语言或正规语言.

(2)正规文法构造相应的状态转换图:

1) 对于右线性文法:

设:$G=\{V_N,V_T,P,S)$是一个右线性文法,并设$|V_N|=k$,

构造的状态转换图中共有$k+1$个结点.

结点的标记: -------- 用V_N的各个非终结符号分别标记其中的k 个结点, 且令G的开始符号S作为初态,余下的一个作为终态结点F,且F不属于V_N.

箭弧的规则:

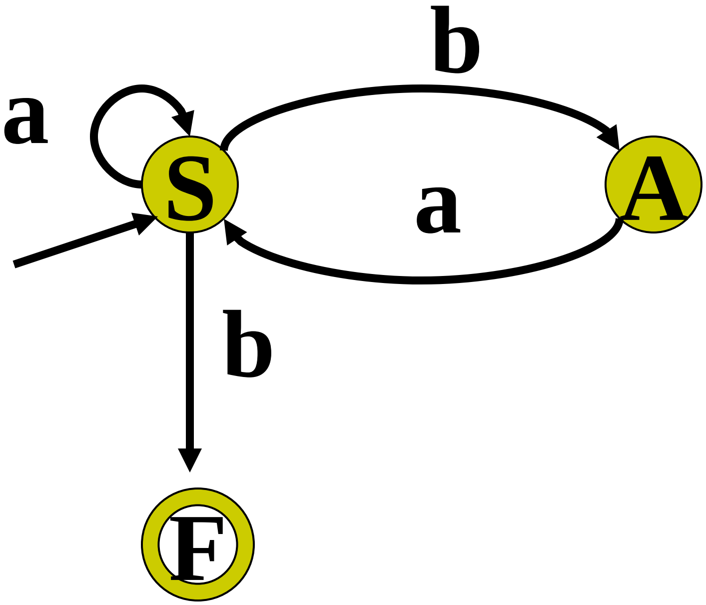
对于G中每一形如$A \rightarrow aB$ 的产生式,从结点A引一条箭弧到结点B,并用符号a标记这条箭弧.

对于G中每一形如$A \rightarrow a$的产生式,从结点A引一条箭弧到终态结点F,并用符号a标记这条箭弧.

例如：设给定右线性文法G：

$$S \longrightarrow aS|bA|b$$

$$A \longrightarrow aS$$



2) 对于左线性文法:

设:G={V_N,V_T,P,S)是一个左线性文法,并设|V_N|=k,

构造的状态转换图M中共有k+1个结点.

结点的标记: -------- 用V_N的各个非终结符号分别标记其中的k 个结点, 且 引入 开始符号R（R不属于V_N）作为初态, 用S作为终态结点.

箭弧的规则:

对于G中每一形如$A \rightarrow Ba$ 的产生式,从结点B引一条箭弧到结点A,并用符号a标记这条箭弧.

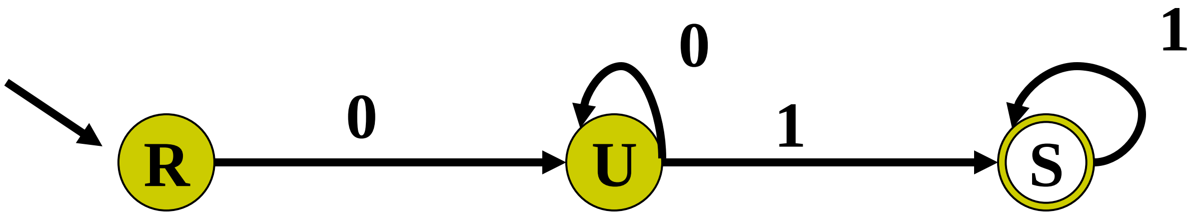
对于G中每一形如$A \rightarrow a$的产生式,从初态R引一条箭弧到结点A,并用符号a标记这条箭弧.

例如：对于左线性文法G=（{S，U}，{0，1}，P，S）其中

$$P=\{S \longrightarrow S1,\quad S \longrightarrow U1,$$
$$U \longrightarrow U0,\quad U \longrightarrow 0\ \}$$

构造状态转换图。

解：



定理：对任何一个右线性正规文法G，都存在一个左线性正规文法G’，使L（G’）=L（G），反之亦然。

（3）正规文法与有限自动机的等价性

定理1：对每一个 右线性正规文法G或左线性正规文法G，都存在有限自动机M，使L（M）=L（G）。

定理2：对每一个DFA M，都存在

一个右线性正规文法G和

一个左线性正规文法G’，

使

L（M）=L（G’）=L（G）。

由DFA M定义右线性文法:

- 设: DFA $M = (\Sigma, V_N, S, F, f)$,
  如初态符号S不属于F,令
  $$G = (\Sigma, V_N, S, P),$$
  其中: Σ为终结符.P是按下面规则定义:对任何a属于Σ且A,B属于V_N, 若有$f(A, a)=B$ 则
  a) 当B不属于F, 令$A \longrightarrow aB$
  b) 当B属于F, 令$A \longrightarrow a \mid aB$

- 如初态符号S属于F，因为$f(S,\varepsilon)=S$，所以，ε属于L（M），但ε不属于上面构造的G所产生的语言L(G).实际上

$$L(G)=L(M)-\{\varepsilon\}$$

因而对上面由M出发所构造的右线性正规文法G中添加一个非终结符号S_0不属于V_N和产生式

$$S0 \longrightarrow S \mid \varepsilon$$

并用S0代替S作开始符号.这样经过修正后的G仍是右线性正规文法且L(G)=L(M).

类似的从M出发可构造左线性文法.

例题：设DFA M=({0,1},{A,B,C,D},A,{B},f)

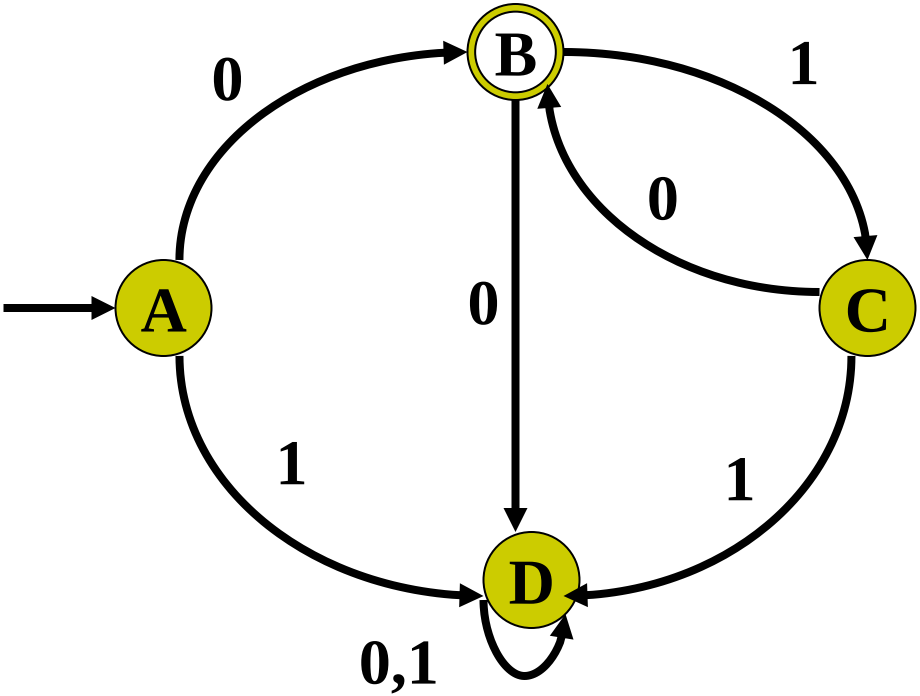
f(A,0)=B　　f(A,1)=D

f(B,0)=D　　f(B,1)=C

f(C,0)=B　　f(C,1)=D

f(D,0)=D　　f(D,1)=D

$L(M)=0(10)^*$.



构造右线性正规文法：

G=（{0，1}，{A，B，C，D}，A,P)

其中P为产生式的集合：

$A \rightarrow 0|0B|1D$　　　$B \rightarrow 0D|1C$

$C \rightarrow 0|0B|1D$　　　$D \rightarrow 0D|1D$

L(G)=L(G)=0(10)*

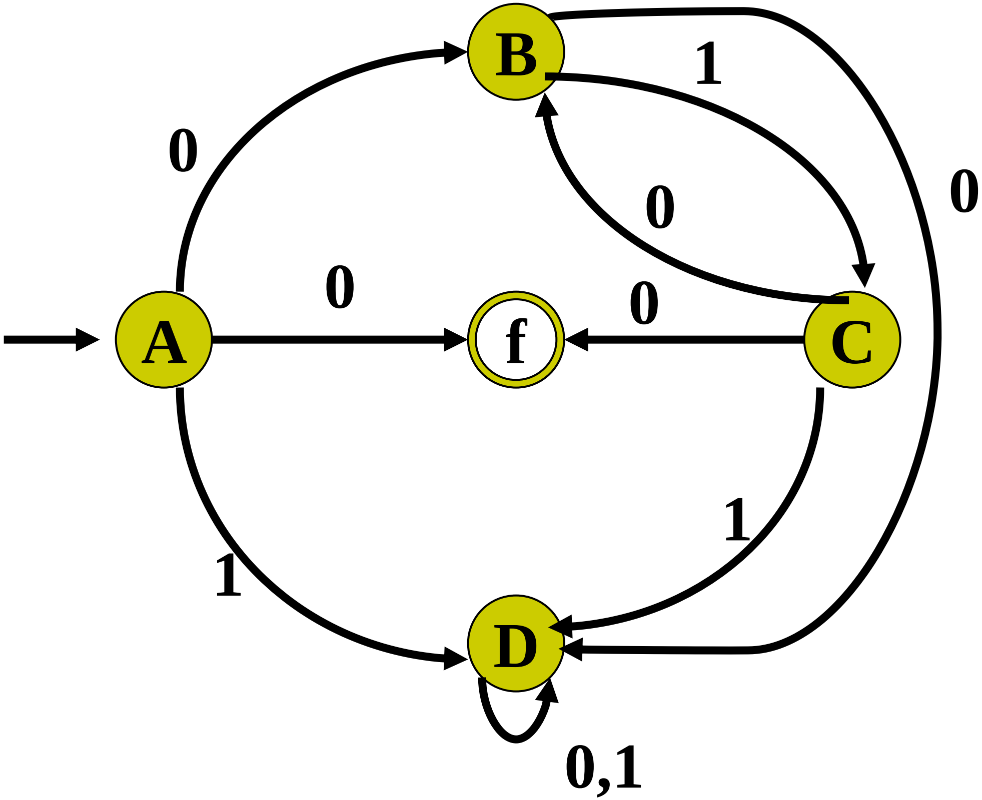
由右线性正规文法G出发构造的NFA M’为

M’=({0,1},{A,B,C,D,f},A,{f},F)

其中

F(A,0)={f,B}	F(A,1)={D}
F(B,0)={D}	F(B,1)={C}
F(C,0)={f,B}	F(C,1)={D}
F(D,0)={D}	F(D,1)={D}

状态转换图如下所示:



L（M’）=L（M）

据NFA M’的状态转换图，构造左线性正规文法

G’=（{0，1}，{B，C，D，S}，S，P）

其中S=f 为下面产生式的集合:

$S \rightarrow 0 \mid C0$ $\quad$ $C \rightarrow B1$

$B \rightarrow 0 \mid C0$

$D \rightarrow 1 \mid C1 \mid D0 \mid D1 \mid B0$

这里从M’构造左线性文法G’的产生式P的方法:

若对任何A,A1属于{B,C,D,S}且a属于{0,1}有F(A1,a)=A2,

则令$A2 \longrightarrow A1a$,

若有F(A1,a)=A(A是M’的初态),

则令$A1 \longrightarrow a$, 有

L(G’)=L(M’)=L(G)=L(M)=0(10)*

第2章:词法分析 (Lexical Analysis)

- ⌘2.1 词法分析程序的功能
- ⌘2.2 词法分析器的设计
- ⌘2.3 正规表达式 (Regular Expression)
- ⌘2.4 有限自动机
- ⌘2.5 词法分析器的自动生成

2.5 词法分析器的自动生成

用LEX语言来写词法分析器,

LEX编译程序的作用:

2.5.1 LEX语言的一般介绍:

LEX程序由两部分生成: (p58~59)

1) 正规式辅助定义式.

2)词法规则.

一.辅助定义式:

由一些LEX语句组成,形式为:

$$
\begin{array}{l}
D1 \longrightarrow R1 \\
D2 \longrightarrow R2 \\
\quad \ldots\ldots \\
Dm \longrightarrow Rm
\end{array}
$$

其中:Ri为正规式,它定义在

$\Sigma \cup \{D1,D2,\ldots,Di-1\}$.

Di 为其简名(用小写字符串记)

使用辅助定义式可以方便地定义单词:

如: FORTRN中的标识符.

letter → A|B|C|…|Z

digit → 0|1|2|…|9

iden → letter|digit

二.词法识别规则:

LEX语句组成:

P1 {A1}

P2 {A2}

....

Pn {An}

其中: Pi是词形,由定义在$\Sigma \cup \{D1,D2,\ldots Dm\}$上的正规式表示.

Ai是动作,当识别出词形Pi后应作的工作。

词法分析器的功能由Pi和Ai决定.

三.词法分析器如何工作:

1) 最长匹配原则:

L扫描输入串,寻找最长的子串匹配某一个Pi;并把该子串截下==>TOKEN
然后调用动作Ai,把表示Pi的二元式送给语法分析器.

2)优先匹配原则:

在服从最长匹配的前提下,处于前面的Pi,匹配优先权就越高.

3)出错处理:

在输入串中找不到与某一个Pi匹配的子串,要报告出错.

4) Ai返回单词的种别和内部值.在LEX程序中用RETURN(C,LEXVAL)

标识符: LEXVAL 是TOKEN中的内部值,

常数: LEXVAL是经DTB翻译后在TOKEN中的二进制值。

例题：用LEX识别单词符号： (p58~59)

AUXILIARY DEFINITIONS //正规定义式

Di Ri

……

RECOGNITION RULES //识别规则

Pi {Ai}

……

2.5.2 超前搜索

例如，Fortran 语句 DO 501 I=1,25

在正规式中引入运算符“/”,用来表示截断点.
如,识别基本字DO的规则要写成:
DO/(letter|digit)*= (letter|digit)*, {A}

识别时,要求词法分析器L向前扫描到“,”,从“/”处截断,后一半还给输入串,前一半作为L的输出.

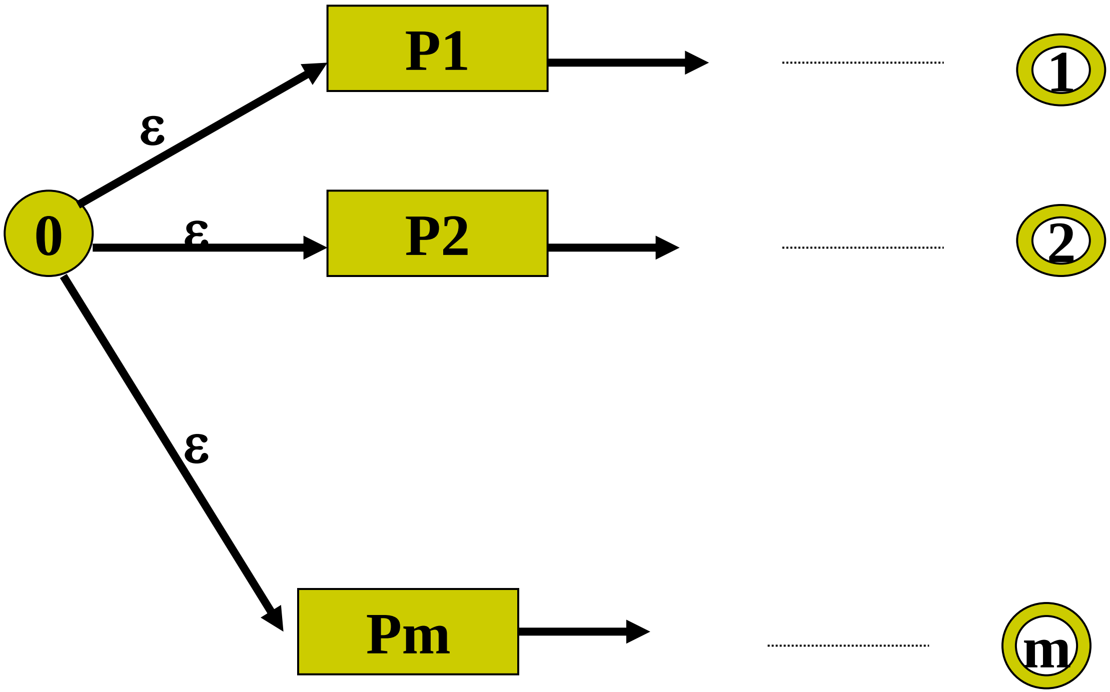
2.5.3 LEX的实现

LEX源程序经LEX编译程序翻译成词法分析器L(状态转换表和控制程序组成).

LEX编译过程:

1.给每一个Pi造相应的NFA Mi,

2.输入一个初态,经ε弧,把所有的NFA Mi连接成一个NFA M.


P1
P2
Pm
0
ε
ε
ε
1
2
m

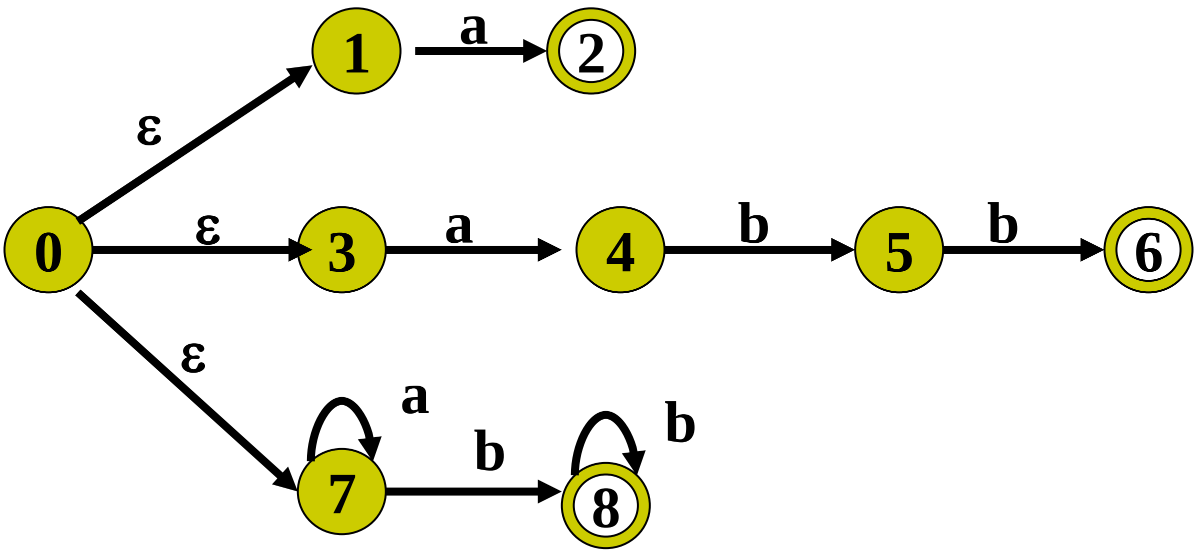
3.把NFA M 确定化.

4.最小化.

控制程序按状态转换表处理输入串,并能实现最长匹配和最优匹配出错处理.

例如:要识别的词形为:

a	{ }
abb	{ }
a*bb*	{ }


0
1
2
3
4
5
6
7
8
ε
ε
ε
a
a
a
b
b
b
b
b

确定化后得:

	状态	a	b	到达终态时所识别的词形
初态	0137	247	8	
终态	247	7	58	a
终态	8	--	8	a*bb*
	7	7	8	
终态	58	--	68	a*bb*
终态	68	--	8	abb (因为6在8的前面)

实现问题：

- 缓冲区预处理，超前搜索，
- 关键字的处理，符号表的实现
- 查找效率，算法的优化实现
- 词法错误处理

小结

- 用正规式编写词法，设置单词种别和属性
- 从单词的描述出发，逐步实现词法分析程序
- 词法分析器的自动生成LEX

正规表达式

↓

正规文法

↓

状态图(DFA,NFA)

↓

识别过程的实现算法

↓

程序实现和测试

词法分析技术的其他应用：

- 查询语言，信息检索系统
  - 识别由正规式描述的字符串
- 命令语言
  - 识别命令格式
- 报文的词处理
  - 识别报文格式的词处理
- 应用范围：
  - 数据格式可以用三型文法描述。

习题

- 1）考虑 C 语言十六进制常整数，
  - a)用正规式描述其词法
  - b)构造对应的 DFA
- 2）构造下列正规式的状态图，并以五元组的形式构造对应的 DFA
  - $a(ab|ab^*a)^*b$

习题

- 3）给出下述文法所对应的正规式
  - S -> 0 A | 1 B
  - A -> 1 S | 1
  - B -> 0 S | 0

附： Finite Automata

- A *recognizer* for a language is a program that takes a string x, and answers “yes” if x is a sentence of that language, and “no” otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic(DFA)* or *non-deterministic (NFA)*

- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.

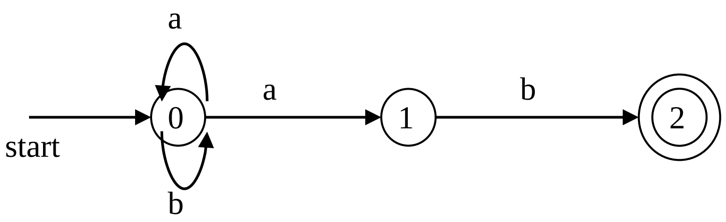
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.
  - Algorithm1: Regular Expression ➔ NFA ➔ DFA (two steps: first to NFA, then to DFA)
  - Algorithm2: Regular Expression ➔ DFA (directly convert a regular expression into a DFA)

Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - S - a set of states
  - Σ - a set of input symbols (alphabet)
  - move – a transition function move to map state-symbol pairs to sets of states.
  - s_0 - a start (initial) state
  - F – a set of accepting states (final states)

- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

NFA (Example)



Transition graph of the NFA

0 is the start state s_0

{2} is the set of final states F

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

$S = \{0,1,2\}$

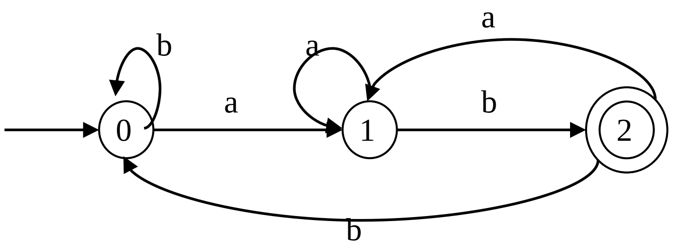
Transition Function:

	a	b
0	{0,1}	{0}
1	_	{2}
2	_	_

The language recognized by this NFA is $(a|b)^*\ a\ b$

Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
- no state has ε- transition
- for each symbol a and state s, there is at most one labeled edge a leaving s.
  i.e. transition function is from pair of state-symbol to state (not set of states)



The language recognized by this DFA is also $(a|b)^*\ a\ b$

Implementing a DFA

- Let us assume that the end of a string is marked with a special symbol (say eos). The algorithm for recognition will be as follows: (an efficient implementation)

```
s ← s0                  { start from the initial state }
c ← nextchar            { get the next character from the input string }
while (c != eos) do     { do until the end of the string }
   begin
      s ← move(s,c)     { transition function }
      c ← nextchar
   end
if (s in F) then        { if s is an accepting state }
   return "yes"
else
   return "no"
```

Implementing a NFA

S 🡐 ε-closure({s_0}) //set all of states can be accessible from s_0 by ε-transitions
c 🡐 nextchar
while (c != eos) {
  begin
    s 🡐 ε-closure(move(S,c))   // set of all states can be accessible from a state in S
    c 🡐 nextchar   // by a transition on c
  end
if (S∩F != Φ) then   { if S contains an accepting state }
  return "yes"
else
  return "no"

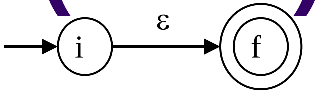
- This algorithm is not efficient.

Converting A Regular Expression into A NFA (Thomson's Construction)

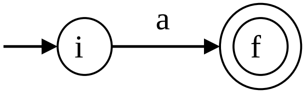
- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
  It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
  To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA,

Thomson's Construction (cont.)

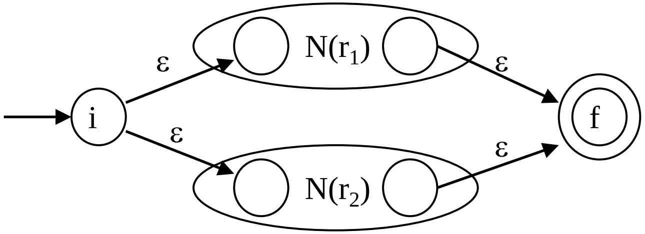
- To recognize an empty string ε



- To recognize a symbol a in the alphabet Σ



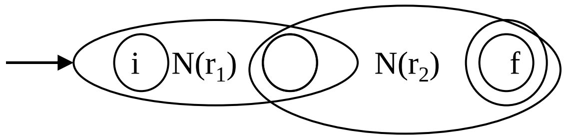
- If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2
  - For regular expression $r_1 | r_2$



NFA for $r_1 | r_2$

Thomson's Construction (cont.)

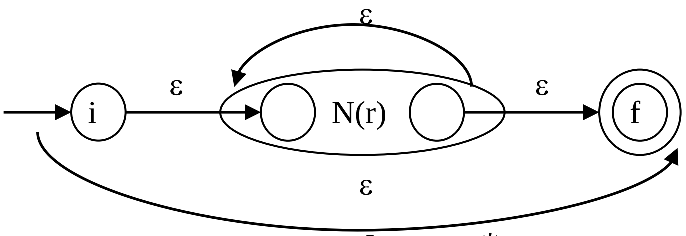
- For regular expression $r_1\ r_2$



Final state of $N(r_2)$ become final state of $N(r_1 r_2)$

NFA for $r_1\ r_2$

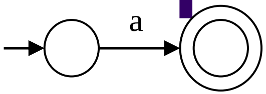
- For regular expression r^*



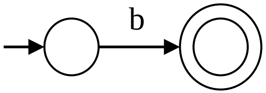
NFA for r^*

Thomson’s Construction (Example - (a|b) * a)

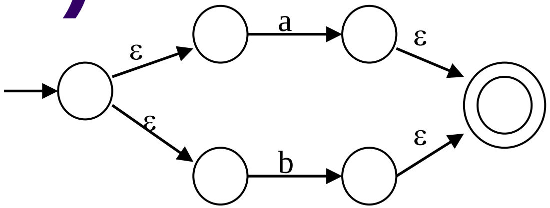
a:



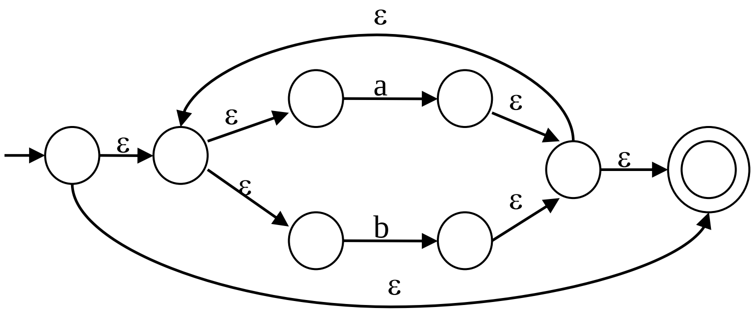
b:



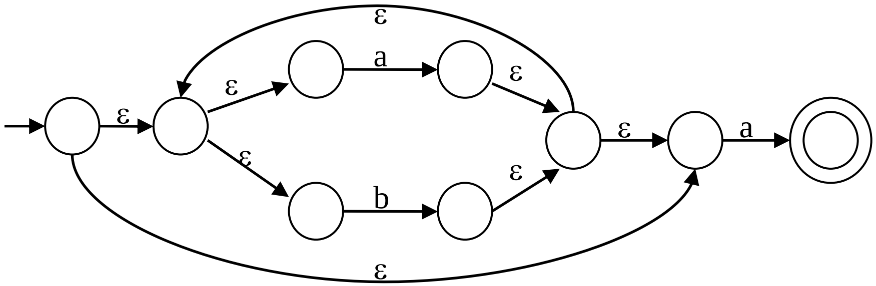
(a | b)



(a|b) *



(a|b) * a



Converting a NFA into a DFA (subset construction)

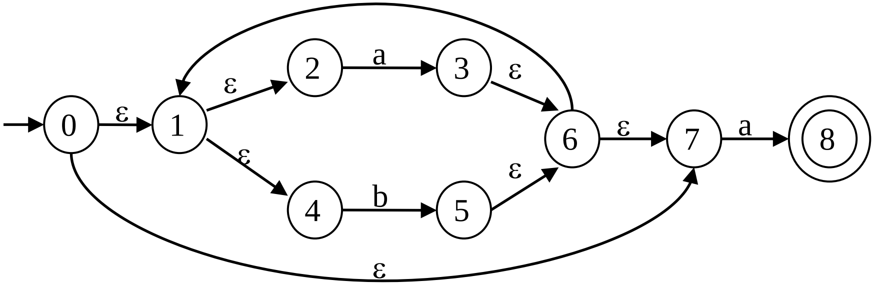
put ε-closure($\{s_0\}$) as an unmarked state into the set of DFA (DS)
while (there is one unmarked S_1 in DS) do
    begin
        mark S_1
        for each input symbol a do
            begin
                $S_2 \leftarrow \varepsilon$-closure(move($S_1$,a))
                if (S_2 is not in DS) then
                    add S_2 into DS as an unmarked state
                transfunc[S_1,a] $\leftarrow S_2$
            end
    end

ε-closure($\{s_0\}$) is the set of all states can be accessible from s_0 by ε-transition.

set of states to which there is a transition on a from a state s in S_1

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure($\{s_0\}$)

Converting a NFA into a DFA (Example)



$S_0 = \varepsilon\text{-closure}(\{0\}) = \{0,1,2,4,7\}$ S_0 into DS as an unmarked state

$\Downarrow$ mark S_0

$\varepsilon\text{-closure}(move(S_0,a)) = \varepsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\} = S_1$ S_1 into DS

$\varepsilon\text{-closure}(move(S_0,b)) = \varepsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\} = S_2$ S_2 into DS

$transfunc[S_0,a] \Leftarrow S_1$ $transfunc[S_0,b] \Leftarrow S_2$

$\Downarrow$ mark S_1

$\varepsilon\text{-closure}(move(S_1,a)) = \varepsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\} = S_1$

$\varepsilon\text{-closure}(move(S_1,b)) = \varepsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\} = S_2$

$transfunc[S_1,a] \Leftarrow S_1$ $transfunc[S_1,b] \Leftarrow S_2$

$\Downarrow$ mark S_2

$\varepsilon\text{-closure}(move(S_2,a)) = \varepsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\} = S_1$

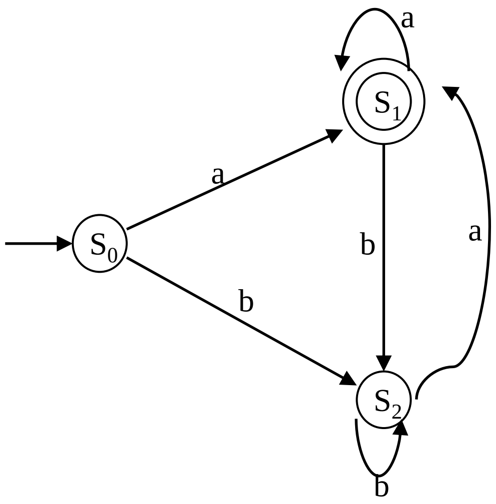
$\varepsilon\text{-closure}(move(S_2,b)) = \varepsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\} = S_2$

$transfunc[S_2,a] \Leftarrow S_1$ $transfunc[S_2,b] \Leftarrow S_2$

Converting a NFA into a DFA (Example – cont.)

S_0 is the start state of DFA since 0 is a member of $S_0=\{0,1,2,4,7\}$

S_1 is an accepting state of DFA since 8 is a member of $S_1 = \{1,2,3,4,6,7,8\}$



Converting Regular Expressions Directly to DFAs

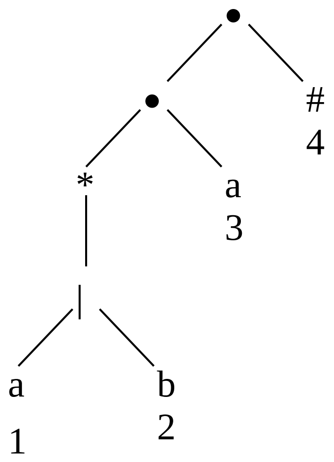
- We may convert a regular expression into a DFA (without creating a NFA first).
- First we augment the given regular expression by concatenating it with a special symbol #.

  r $\rightarrow$ (r)# augmented regular expression

- Then, we create a syntax tree for this augmented regular expression.
- In this syntax tree, all alphabet symbols (plus # and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.
- Then each alphabet symbol (plus #) will be numbered (position numbers).

Regular Expression ➔ DFA (cont.)

$(a|b)^*\ a \rightarrow (a|b)^*\ a\ \#$ augmented regular expression



Syntax tree of $(a|b)^*\ a\ \#$

- each symbol is numbered (positions)
- each symbol is at a leave
- inner nodes are operators