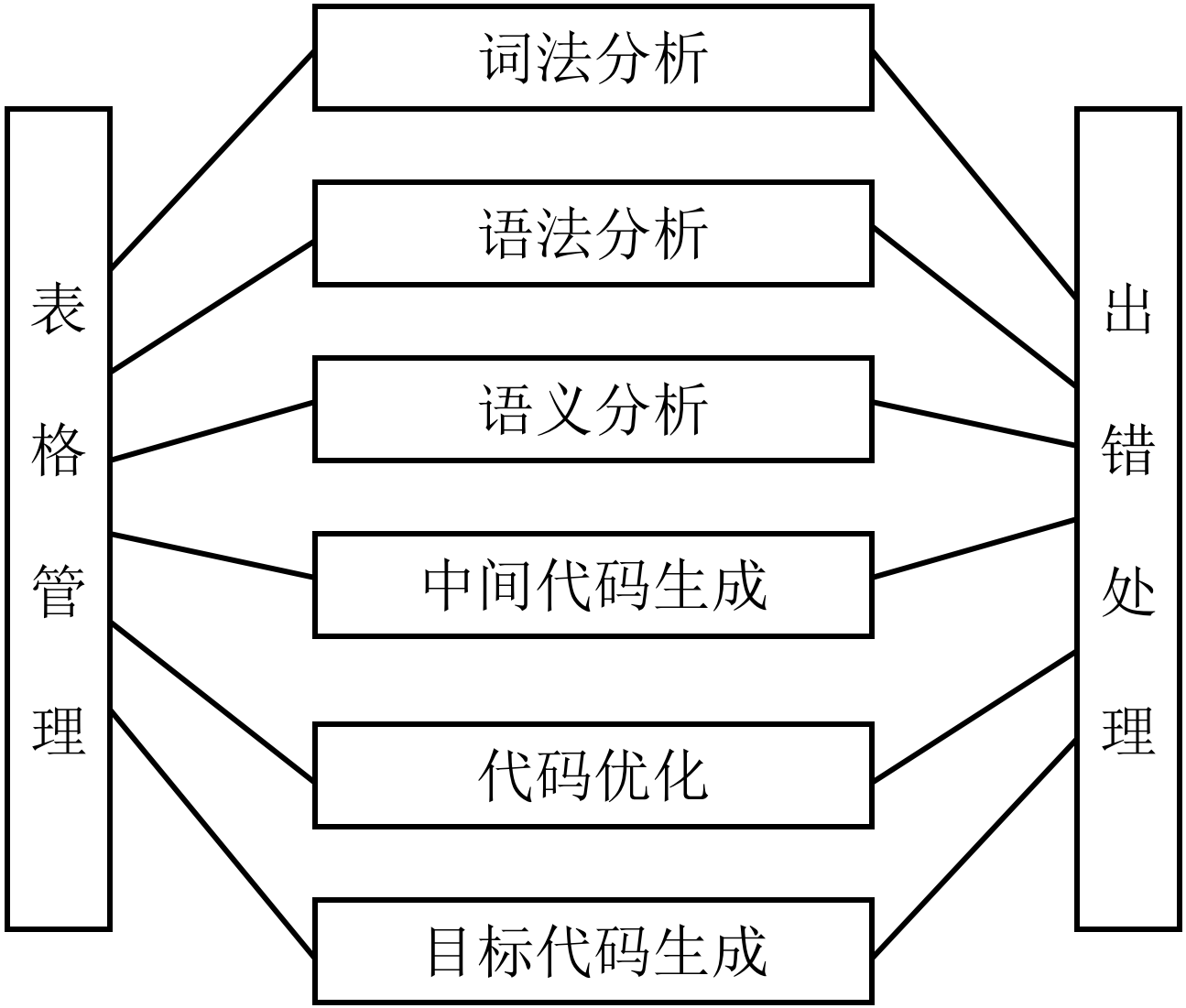
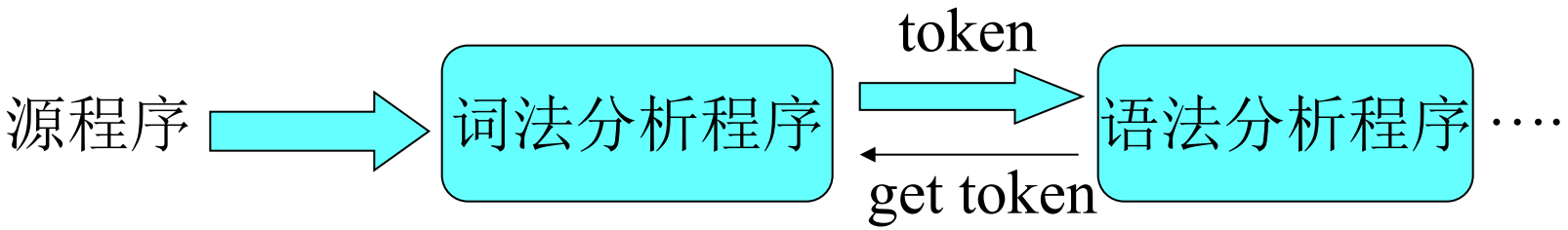
# 第3章 词法分析

词法分析


词法分析
语法分析
语义分析
中间代码生成
代码优化
目标代码生成
表格管理
出错处理

- 词法分析是编译过程中的一个阶段，在语法分析前进行
- 也可以和语法分析结合在一起作为一遍，由语法分析程序调用词法分析程序来获得当前单词供语法分析使用

# 词法分析程序和语法分析程序的关系



# 词法分析的主要功能

- 功能
  - 分析输入的源程序，输出单词符号
  - 把构成源程序的字符串转换成单词符号的序列，并构造符号表，记录其属性

- 单词符号（token）形式
  - 按照最小语义单位设计
  - 通常表示为二元组：

    （单词类别，属性值）（类号，内码）

# 词法分析程序的其他任务

- 过滤掉空格、跳过注释、换行符
- 追踪换行标志
- 复制出错源程序
- 宏展开 ……

# 单词符号的表示

- 常用单词类别
  - 各关键字(保留字、基本字)，各种运算符，各种分界符——各用一个类别码标识
  - 标识符——用一个类别码标示
  - 常量(整数，实数，字符串等)——用一个类别码标示
- 属性(值)——单词的值
  - 常数的值，标识符的名字等
  - 保留字、运算符、分界符的属性值可以省略

# 词法分析中的表格

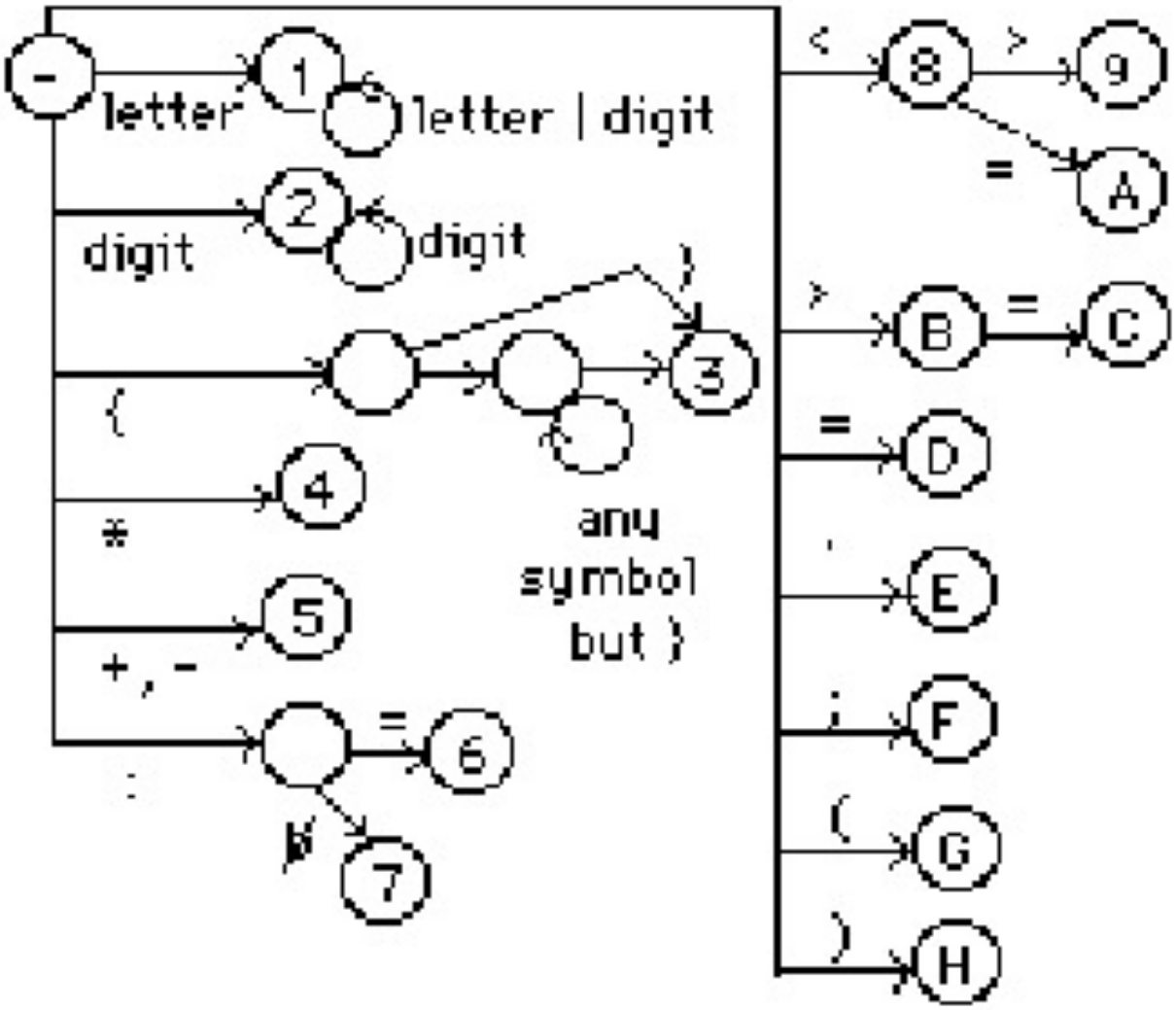
- 保留字表
- 常量表
- 标识符表*
- 自定义函数名表
- 标准函数名表
- 标号表

# 源程序示例

```
main( ) {
  printf(“hello”);
}
```

```
double p, q;
double r, s, t, z;
  // convert to canonical form
  r = b/a;
  s = c/a;
  t = d/a;
  p = s-(r*r)/3.0;
  q = (2*r*r*r)/27.0-(r*s)/3.0+t;
  double D, phi, R;
  R = bgSign(q)*sqrt(fabs(p)/3.0);
  D = pow(p/3.0,3)+pow(q/2,2);
```

```
while (j<=ir)
    {
      if (j<ir && ra[j-1]<ra[j+1-1])
        j++;
      if (rra<ra[j-1])
      {
        ira[i-1] = ira[j-1];
        ra[i-1] = ra[j-1];
        i = j;
        j <<= 1;
      }
      else
        j = ir+1;
    }
```


letter
letter | digit
digit
digit
{
*
+ , -
:
=
anu
symbol
but }
<
>
=
>
=
=
,
;
(
)
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F
G
H

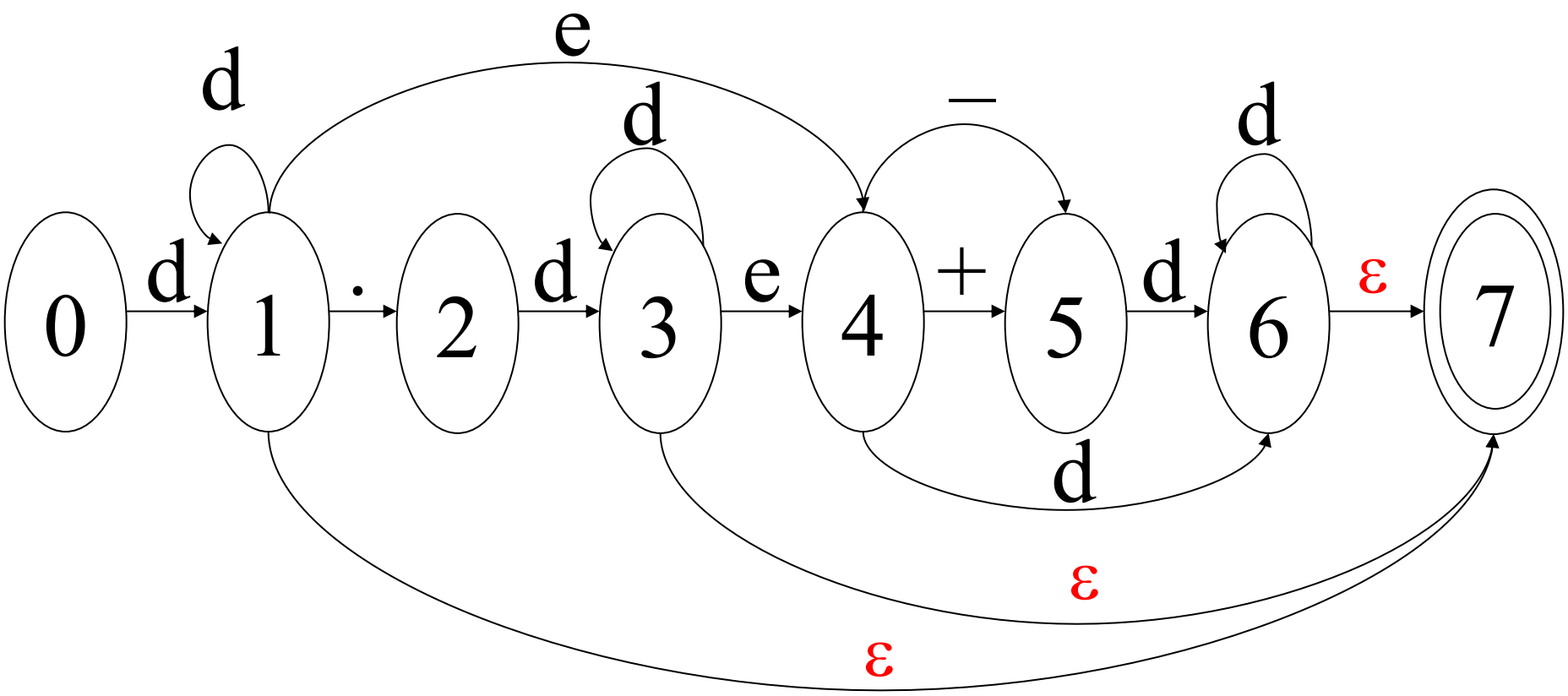
# 词法分析程序的构造

- 词法规则（正规文法）
- 正规表达式
- 自动机（NFA->DFA->最简DFA）
- 程序实现自动机状态转换

# 无正负号（实）数

- dd*(•dd*|ε)(e(+|-|ε)dd*|ε)

dd*(•dd*|ε)(e(+|-|ε)dd*|ε)



# C语言风格的注释

- 有限自动机？

# Review:利用状态转换图DFA识别单词

1. 从初态出发
2. 读入一字符
3. 按当前字符转入下一状态
4. 重复 2,3 直到无法继续转移

```
state=1;
ch=getchar;
while not accept(state) and not error(state) do
{
   newstate=T[state][ch];
   if not error(newstate) then ch=getchar;
   else break;
   state=newstate;
}
if accept(state) then OK;
```

程序实现：

```
state=1;
while state=1,2,3,4 do
{       case state of
        1: if(inputchar==‘/’) {getchar; state=2;}
           else {goto other state or error}
        2: if(inputchar==‘*’) {getchar; state=3;}
           else {goto other state or error}
        3: if(inputchar==‘*’) {getchar; state=4;}
           else {getchar}/*stay at state 3*/
        4: if(inputchar==‘/’) {getchar; state=5;}
           else if (inputchar==‘*’) {getchar;}
                else {getchar; state=3;}
}
if (state==5) accept; else error;
```

# 状态转换矩阵

| | / | * | other | 终止状态 |
|---|---|---|---|---|
| 1 | 2 | | | No |
| 2 | | 3 | | No |
| 3 | 3 | 4 | 3 | No |
| 4 | 5 | 4 | 3 | No |
| 5 | | | | Yes |

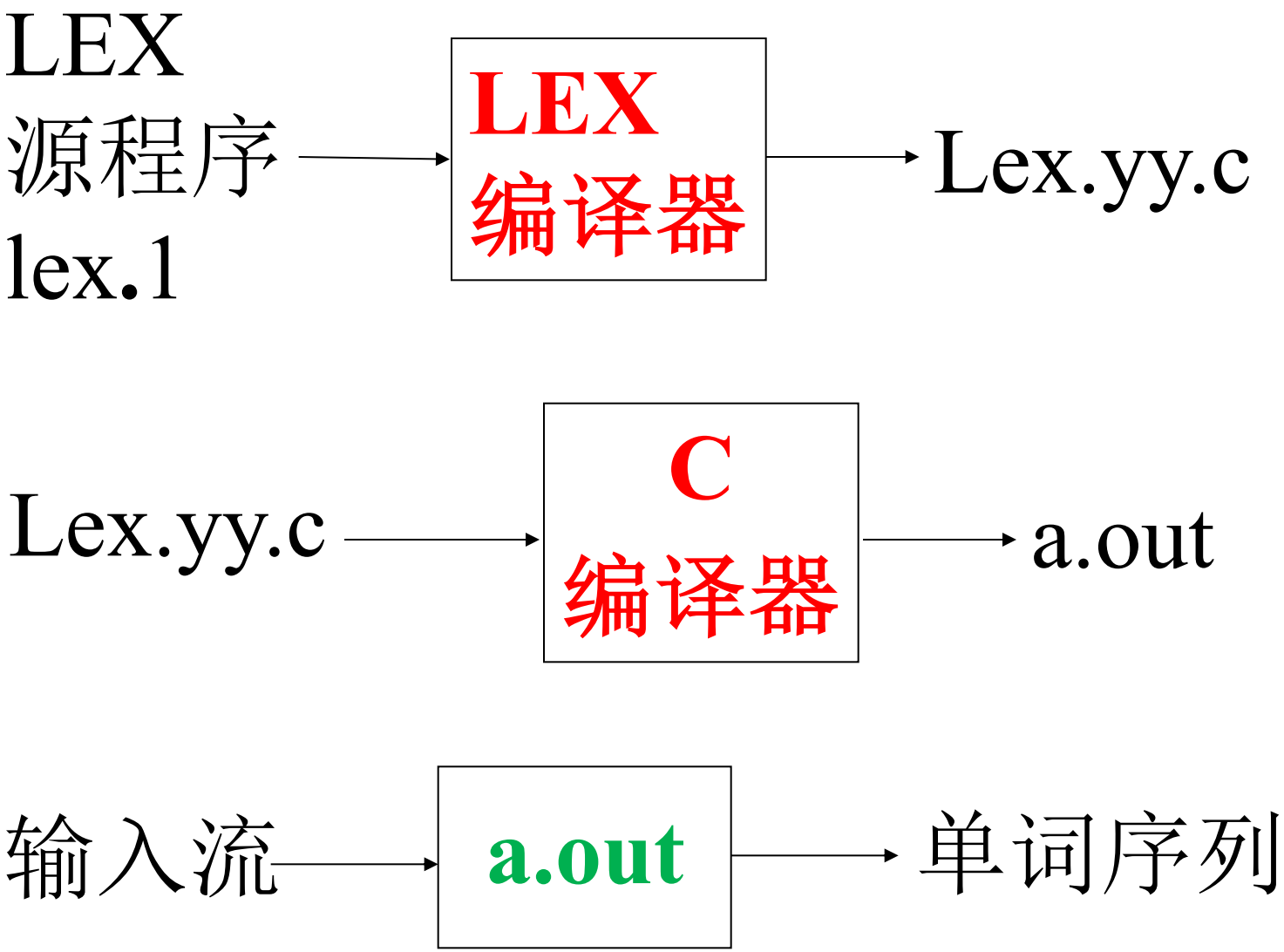
# 词法分析程序的自动构造工具LEX

一、原理

单词结构用正则式描述

正则式⇒ NFA ⇒DFA ⇒min DFA

二、用LEX建立词法分析程序的过程


LEX
源程序
lex.l
LEX
编译器
Lex.yy.c
Lex.yy.c
C
编译器
a.out
输入流
a.out
单词序列

## 三、lex源程序

**lex源程序由三部分组成**

- **说明部分**（辅助定义）
- %%
- **规则部分**（识别规则）
- %%
- **辅助过程**（用户子程序）

$D_1 \longrightarrow R_1$

$D_2 \longrightarrow R_2$

：

：

$D_n \longrightarrow R_n$

**辅助定义**

其中：

R1, R2, ……, Rn 为正则表达式

D1, D2, ……, Dn 为正则表达式名字

**举例：**

**辅助定义**

标识符：

letter→A|B| ¨ ¨ ¨ |Z

digit →0|1| ¨ ¨ ¨ |9

iden → letter(letter|digit)*

带符号整数：

integer→digit (digit)*

sign → +| - |ε

signinteger → sign integer

**识别规则**：

- $p_1$ { 动作$_1$}
- $p_2$ { 动作$_2$}
- … …
- $p_n$ { 动作$_n$}

- 每个$p_i$是正则式名字，每个动作$i$是正则式$p_i$识别某类单词时词法分析器应执行动作的程序段
- 用C书写
- 基本动作：返回单词的类别编码和单词值

例：LEX 源程序

AUXILIARY DEFINITIONS /*辅助定义*/

letter → A|B|…|Z

digit → 0|1|…|9

%%

RECOGNITION RULES /*识别规则*/

1.BEGIN {RETURN(1,—) }

2.END {RETURN(2,—) }

3.FOR {RETURN(3,—) }

**RETURN**是LEX过程，该过程将单词传给语法分析程序 RETURN(C, LEXVAL)

- **C为单词类别编码**
- **LEXVAL：**
  - 标识符：**TOKEN（字符数组）**
  - 整常数：**DTB**
    - **数值转换函数**：将TOKEN中的数字串转换成二进制
  - 其他单词：无定义

| | |
|---|---|
| 4.DO | {RETURN(4,—) } |
| 5.IF | {RETURN(5,—) } |
| 6.THEN | {RETURN(6,—) } |
| 7.ELSE | {RETURN(7,—) } |
| 8.letter(letter \|digit)* | {RETURN(8,TOKEN) } |
| 9.digit(digit)* | {RETURN(9,DTB } |
| 10. : | {RETURN(10,—) } |
| 11. + | {RETURN(11,—) } |
| 12. * | {RETURN(12,—) } |

13. , {RETURN(13,—) }
14. ( {RETURN(14,—) }
15. ) {RETURN(15,—) }
16. := {RETURN(16,—) }
17. = {RETURN(17,—) }

- 辅助过程是需要的，这些过程用C书写，可以分别编译并置于词法分析器中

```
%{
/*  Lex program to convert uppercase to
    lowercase except inside comments
*/
#include <stdio.h>
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
#endif
%}
%%
[A-Z]   {putchar(tolower(yytext[0]));
        /* yytext[0] is the single
           uppercase char found */
        }
"/*"    { char c ;
          int done = FALSE;
          ECHO;
          do
          { while ((c=input())!='*')
              putchar(c);
            putchar(c);
            while((c=input())=='*')
              putchar(c);
            putchar(c);
            if (c == '/') done = TRUE;
          } while (!done);
        }
%%
void main(void )
{ yylex();}
```

辅助过程

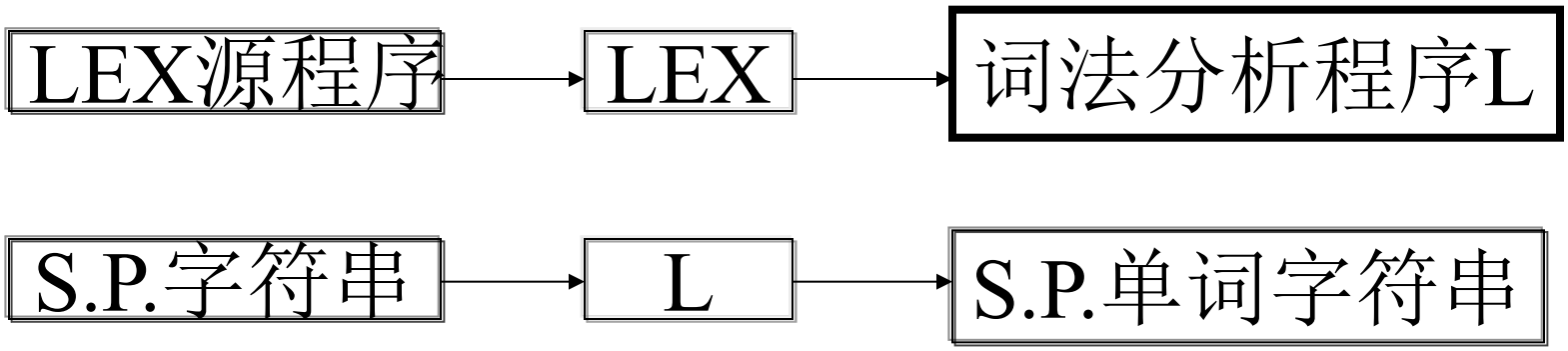
表2-3　一些Lex内部名字

| Lex 内部名字 | 含义/使用 |
| --- | --- |
| `lex.yy.c`或`lexyy.c` | Lex 输出文件名 |
| `yylex` | Lex 扫描例程 |
| `yytext` | 当前行为匹配的串 |
| `yyin` | Lex 输入文件（缺省：`stdin`） |
| `yyout` | Lex 输出文件（缺省：`stdout`） |
| `input` | Lex 缓冲的输入例程 |
| `ECHO` | Lex 缺省行为（将`yytext`打印到`yyout`） |

资料来自于：

K. C. Louden, 编译原理及实践, 冯博琴译, 机械工业出版社

# LEX的实现过程

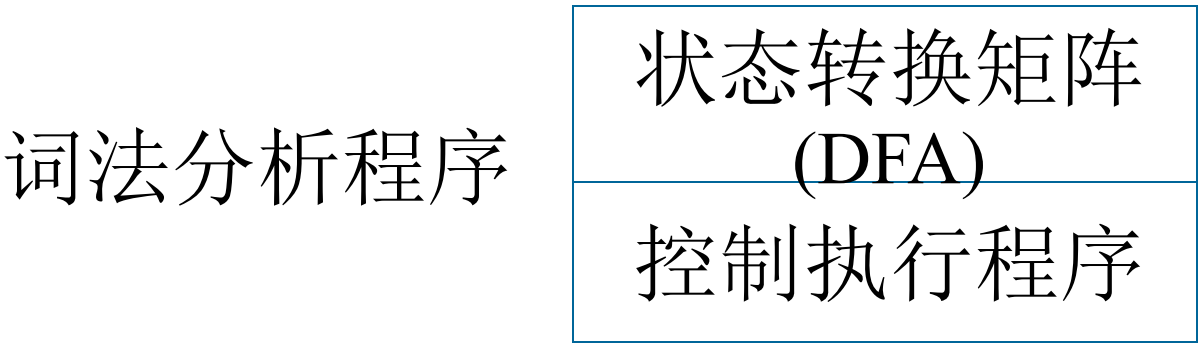


# LEX的实现

LEX的功能：根据LEX源程序构造一个词法分析程序

词法分析器实质上是一个**有穷自动机**

LEX生成的词法分析程序由两部分组成:




∴LEX的功能是根据LEX源程序生成状态转换矩阵和控制程序

# LEX的处理过程

. 扫描每条识别规则$P_i$构造一个相应的非确定有穷自动机$M_i$

.. 将各条规则的有穷自动机$M_i$合并成一个新的NFA M

…确定化与最简化　NFA⇒DFA⇒min DFA

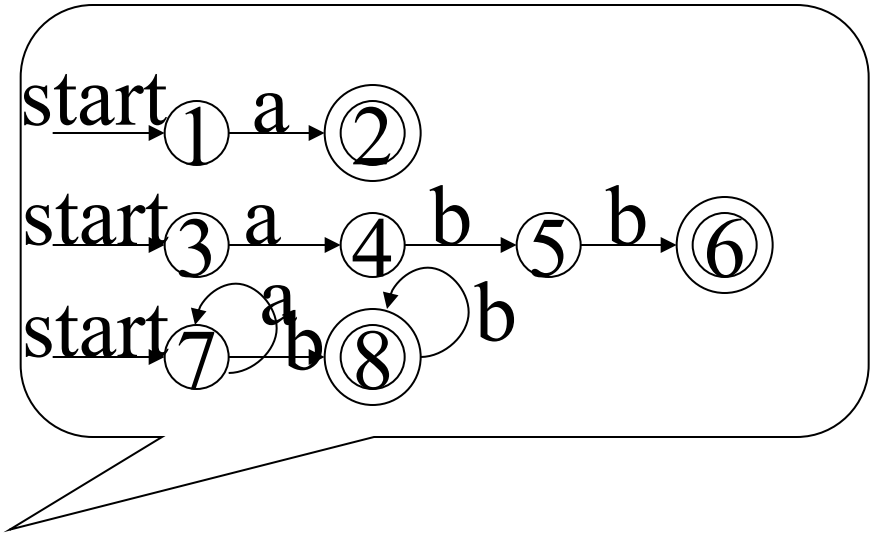
生成DFA的状态转换矩阵和控制执行程序

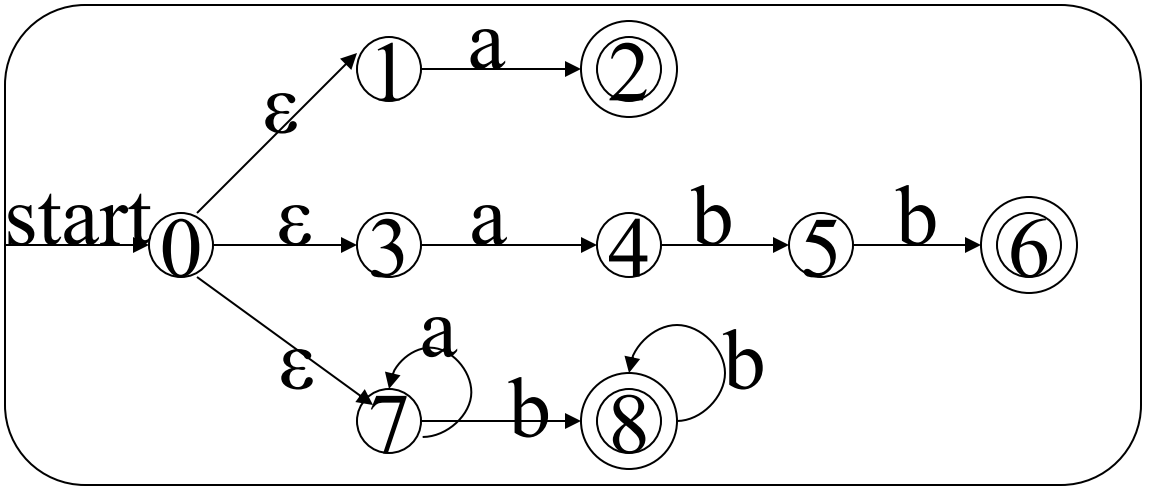
**例**：LEX源程序，

a { }

abb { }

a* bb* { }



| 状态 | | a | b | 所识别的单词 |
|---|---|---|---|---|
| 初态 | {0,1,3,7} | {2,4,7} | {8} | |
| 终态 | {2,4,7} | {7} | {5,8} | a |
| 终态 | {8} | φ | {8} | abb |
| | {7} | {7} | {8} | $a^*bb^*$ |
| 终态 | {5,8} | φ | {6,8} | |
| 终态 | {6,8} | φ | {8} | |

- LEX是一个**通用的工具**，用它可以生成各种语言的词法分析程序，只需要根据不同的语言书写不同的LEX源文件
- LEX不但能自动生成词法分析器，而且还可以产生多种模式识别器及文本编辑程序等

# Questions

- 如何编程实现正规式到NFA的转换？
- 如何编程实现NFA2DFA？
- 如何编程实现DFA最小化？