编译原理

Gamma 移动Ai团队

很多推导内容的细节理解会比较耗费时间

计划时间：90分钟内，所以.....

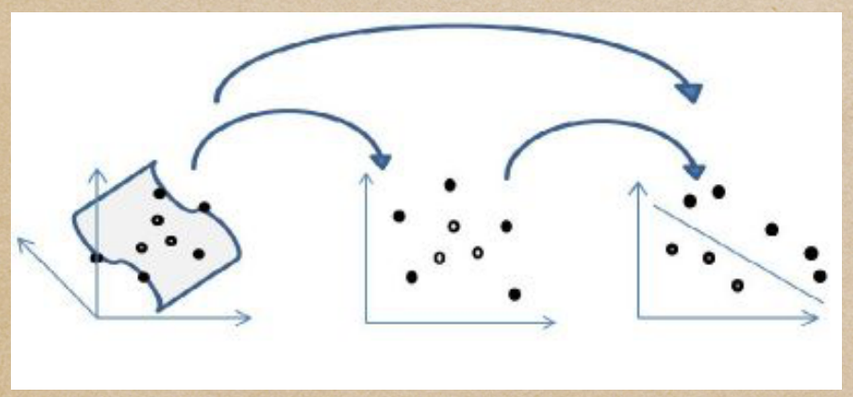
过程中请先跟着我的讲解思路能够理顺每个小部分的逻辑。推导细节的理解可以通过课后的练习和重新推导该PPT的Demo的方式去进行。

不然时间实在来不及呀.....

一、概述

1.1 翻译器、编译器、解释器

翻译器



编译器

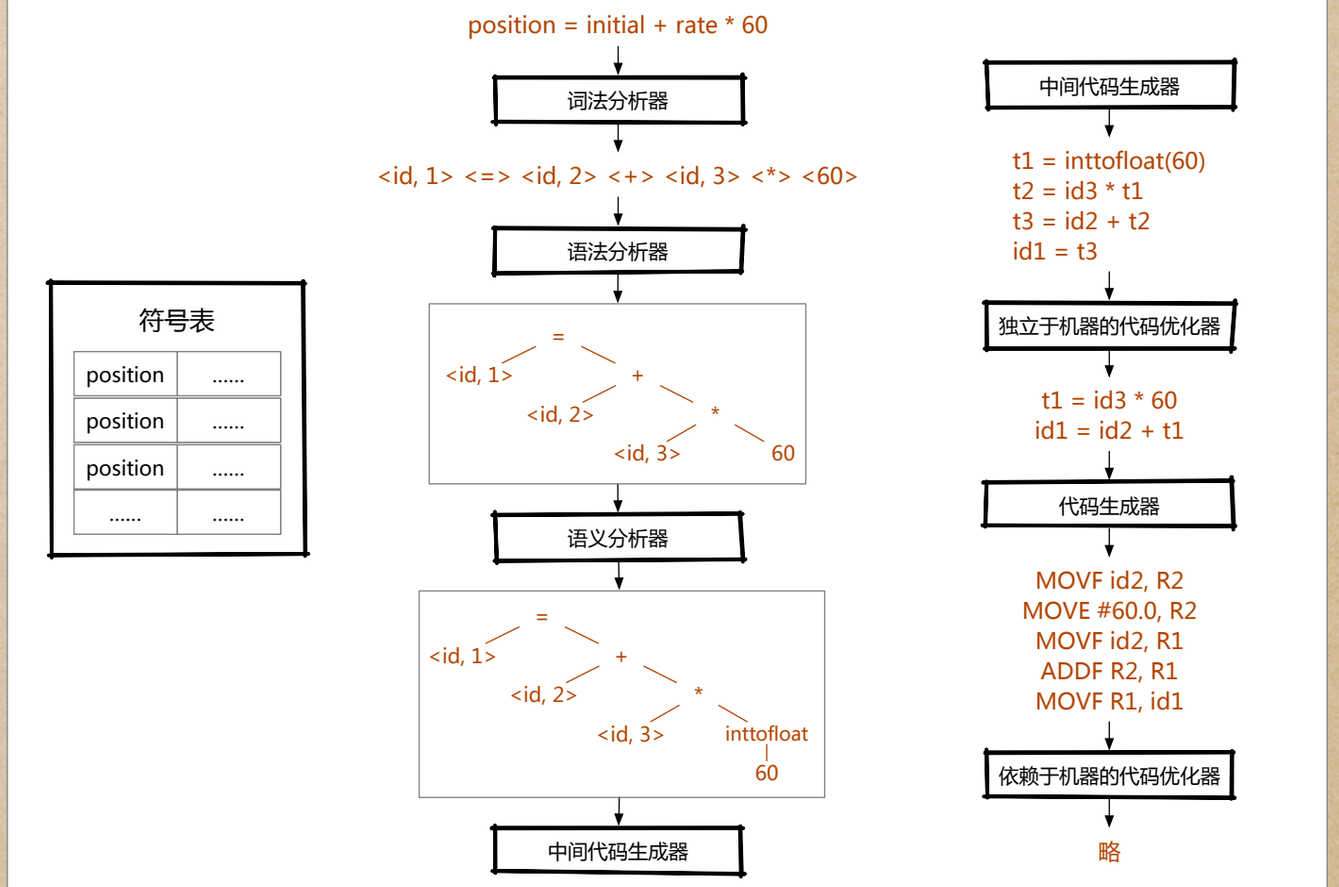
解释器

1.2 编译7阶段

- 词法分析
- 语法分析
- 语义分析
- 中间代码生成
- 独立于机器优化
- 代码生成
- 依赖于机器优化

1.3 编译过程举例

position = initial + tate * 60



二、词法分析

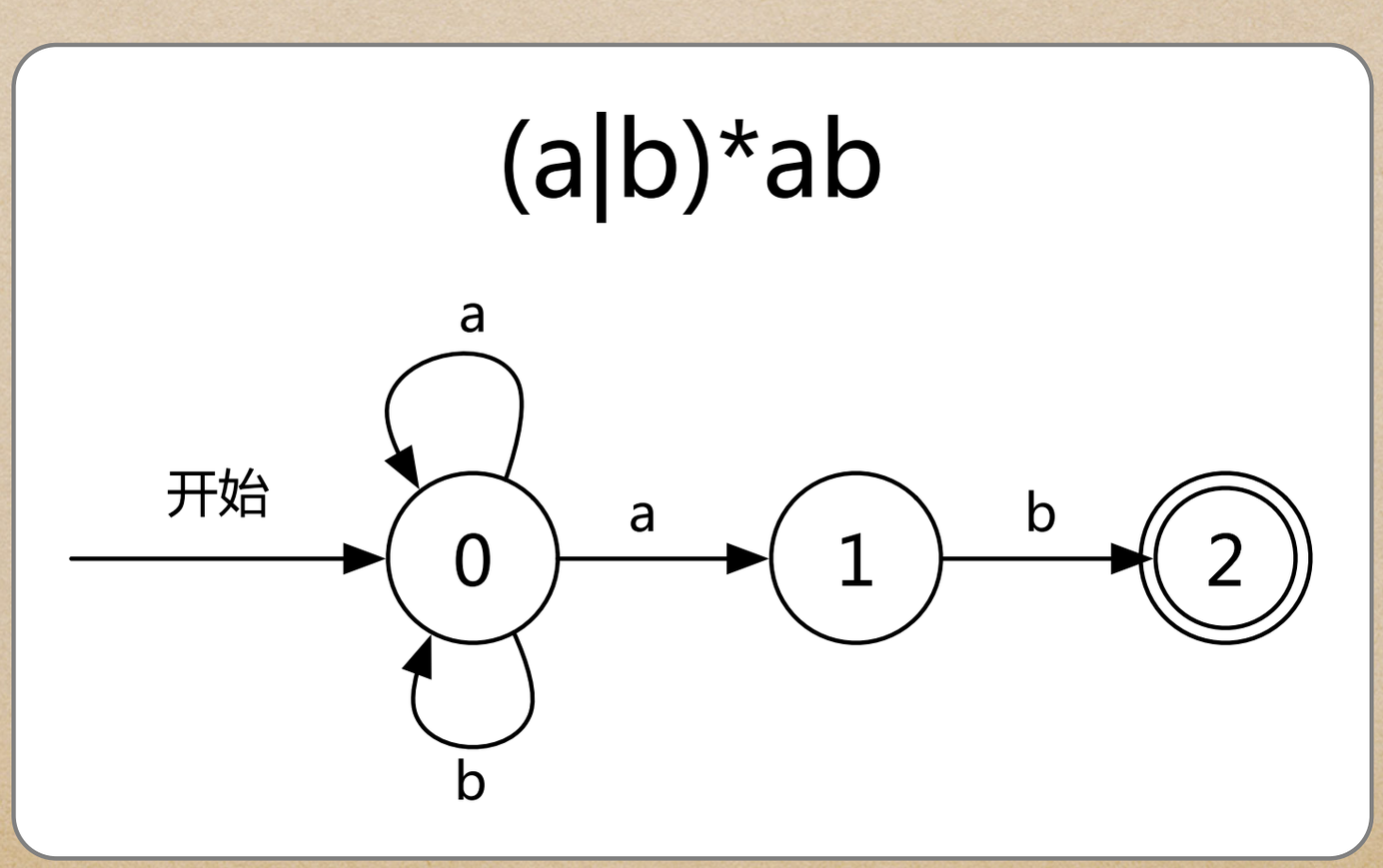
2.1 正规表达式

- 应用参考：百度百科
- https://baike.baidu.com/item/正则表达式/1700215?fr=aladdin

元字符	描述
\	将下一个字符标记符、或一个向后引用、或一个八进制转义符。例如，“\\n”匹配\n。“\n”匹配换行符。序列“\\”匹配“\”而“\(”则匹配“(”。即相当于多种编程语言中都有的“转义字符”的概念。
^	匹配输入字行首。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。
$	匹配输入行尾。如果设置了RegExp对象的Multiline属性，$也匹配“\n”或“\r”之前的位置。
*	匹配前面的子表达式任意次。例如，zo*能匹配“z”，也能匹配“zo”以及“zoo”。*等价于{0,}。
+	匹配前面的子表达式一次或多次(大于等于1次）。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。+等价于{1,}。
?	匹配前面的子表达式零次或一次。例如，“do(es)?”可以匹配“do”或“does”。?等价于{0,1}。
{*n*}	*n*是一个非负整数。匹配确定的*n*次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个o。
{*n*,}	*n*是一个非负整数。至少匹配*n*次。例如，“o{2,}”不能匹配“Bob”中的“o”，但能匹配“foooood”中的所有o。“o{1,}”等价于“o+”。“o{0,}”则等价于“o*”。
{*n*,*m*}	*m*和*n*均为非负整数，其中*n*<=*m*。最少匹配*n*次且最多匹配*m*次。例如，“o{1,3}”将匹配“fooooood”中的前三个o为一组，后三个o为一组。“o{0,1}”等价于“o?”。请注意在逗号和两个数之间不能有空格。

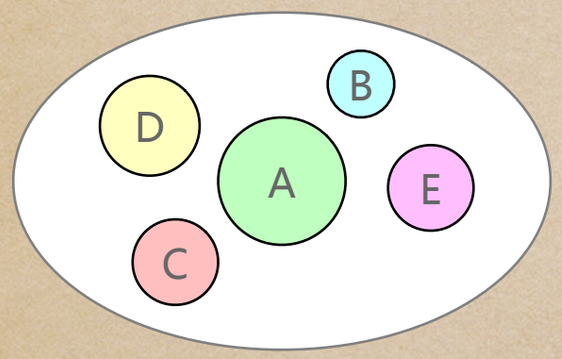
2.2 自动机

- 将正规表达式以状态转换图的方式表示，能完成该状态转换功能的机器，我们称为自动机。
- 当然，我们常常直接将状态转换图称为自动机



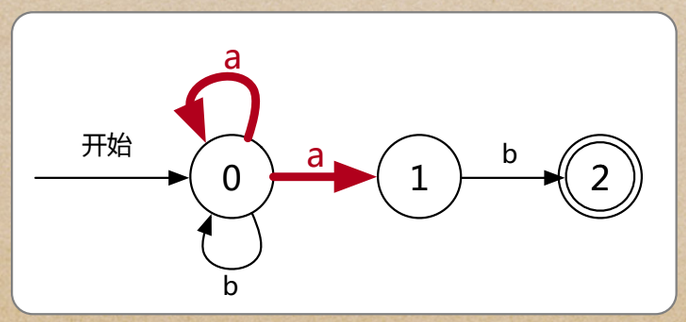
2.3 NFA & DFA cmp

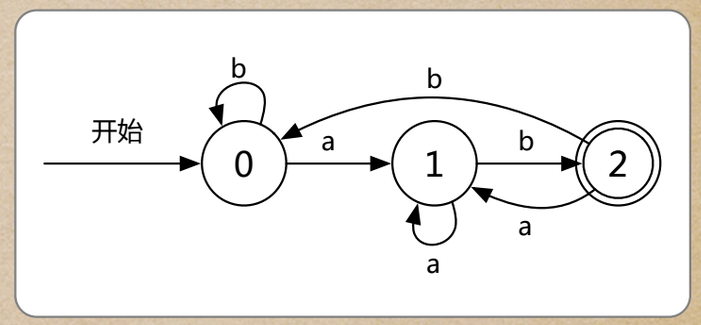
- NFA : Non-deterministic Finite Automate
- DFA : Deterministic Finite Automata



Finite

NFA

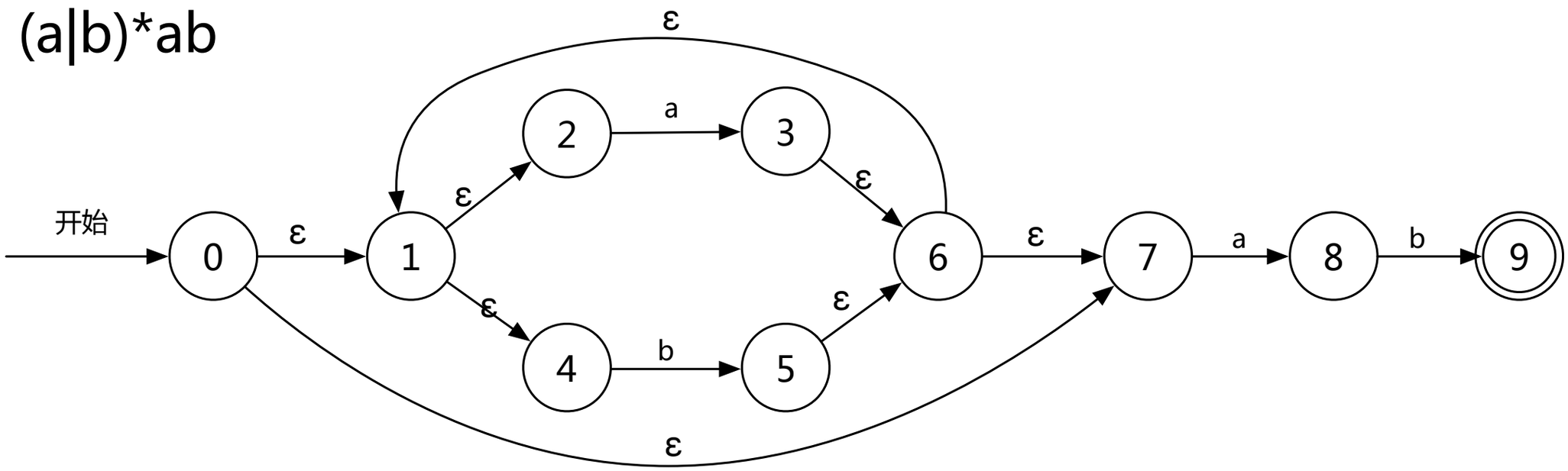




DFA

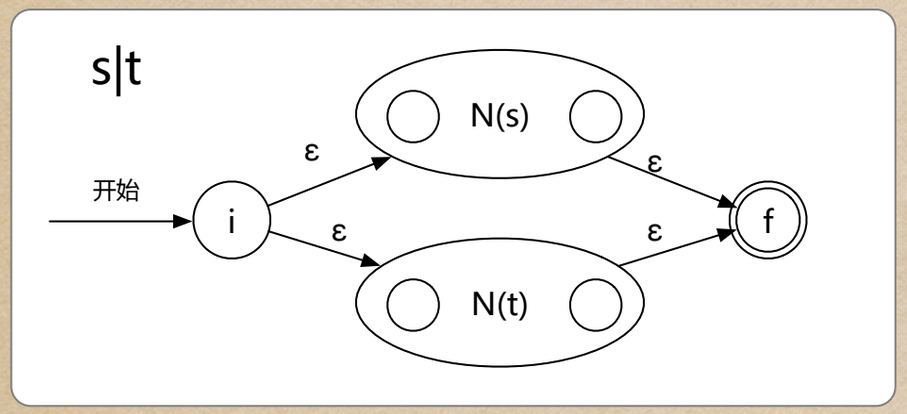
2.4 NFA有规则构建（1）

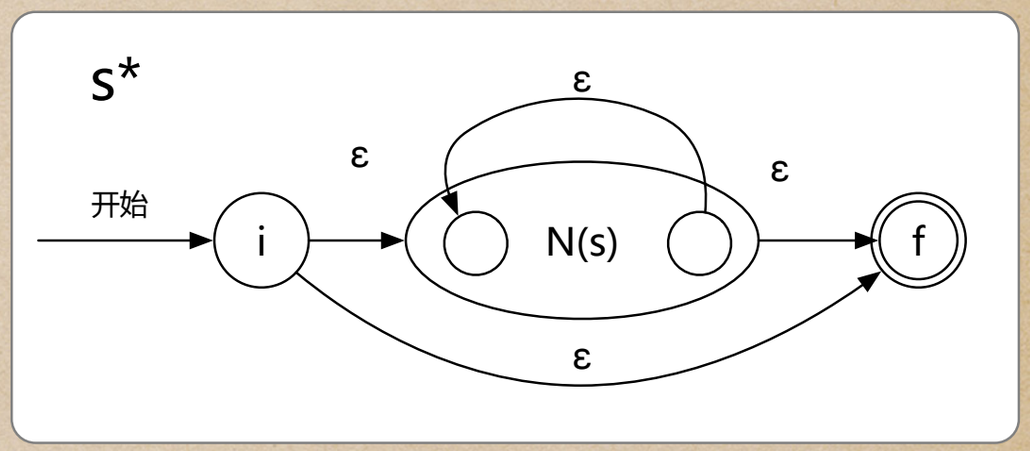
(a|b)*ab

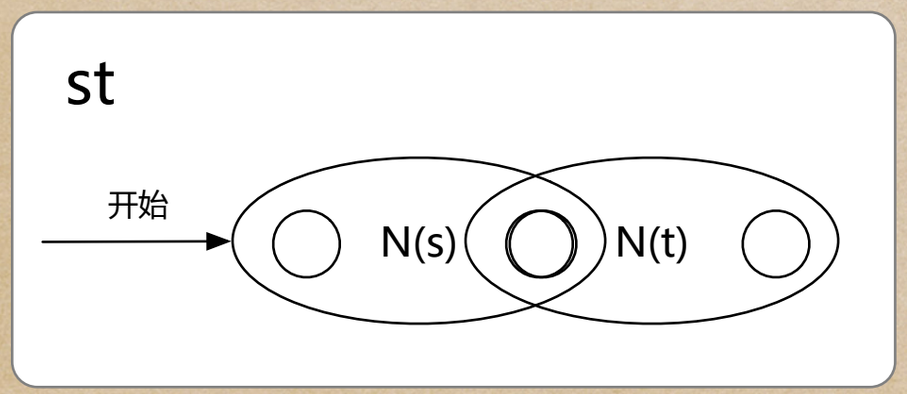


2.4 NFA有规则构建（2）

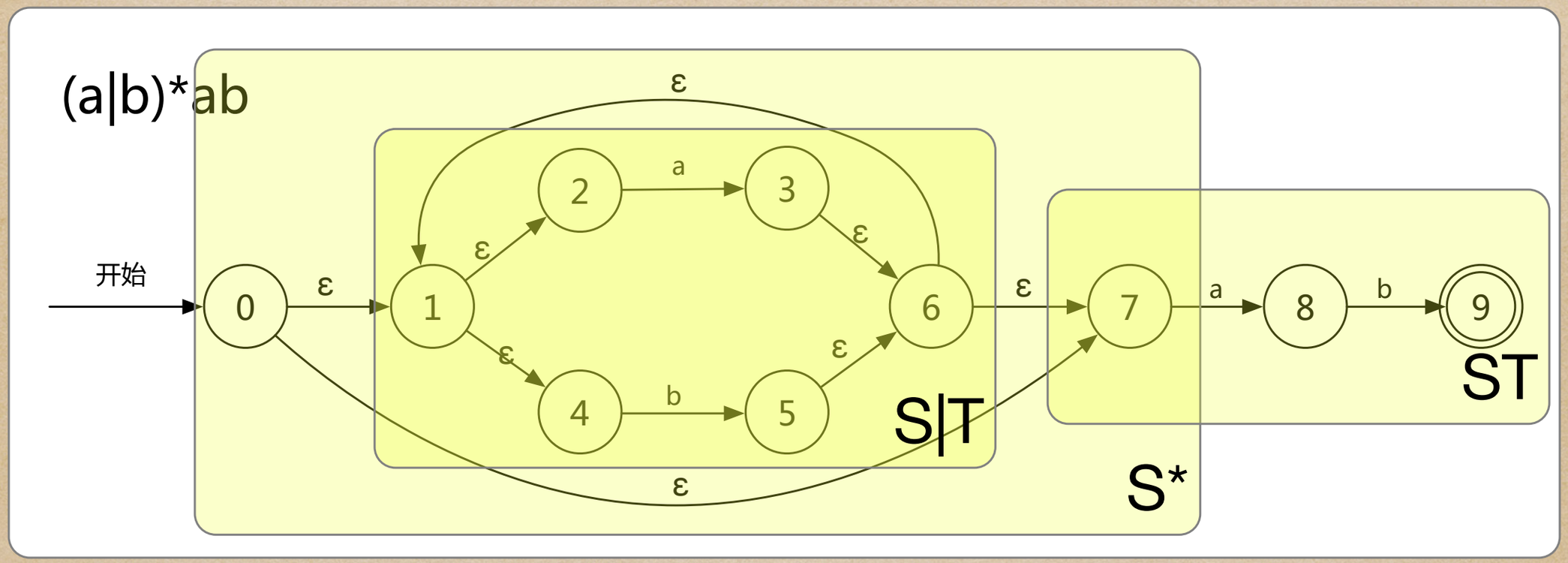
- 识别3种模式：s|t、st、s*

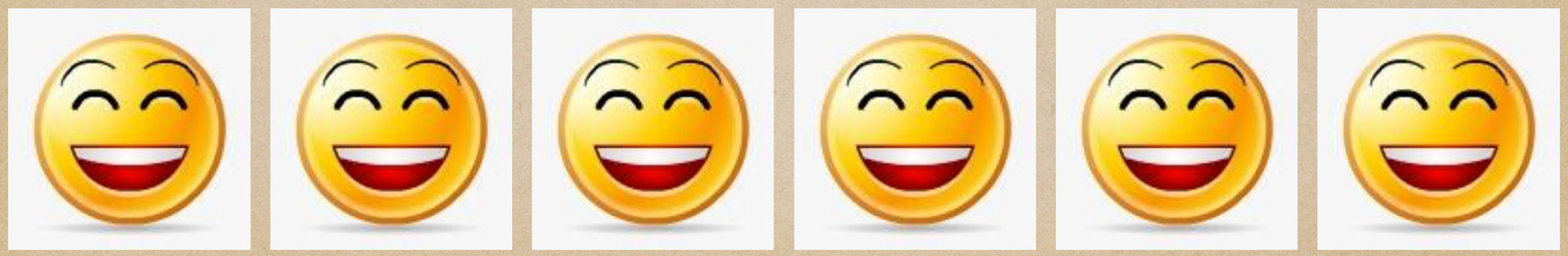






2.4 NFA有规则构建（3）

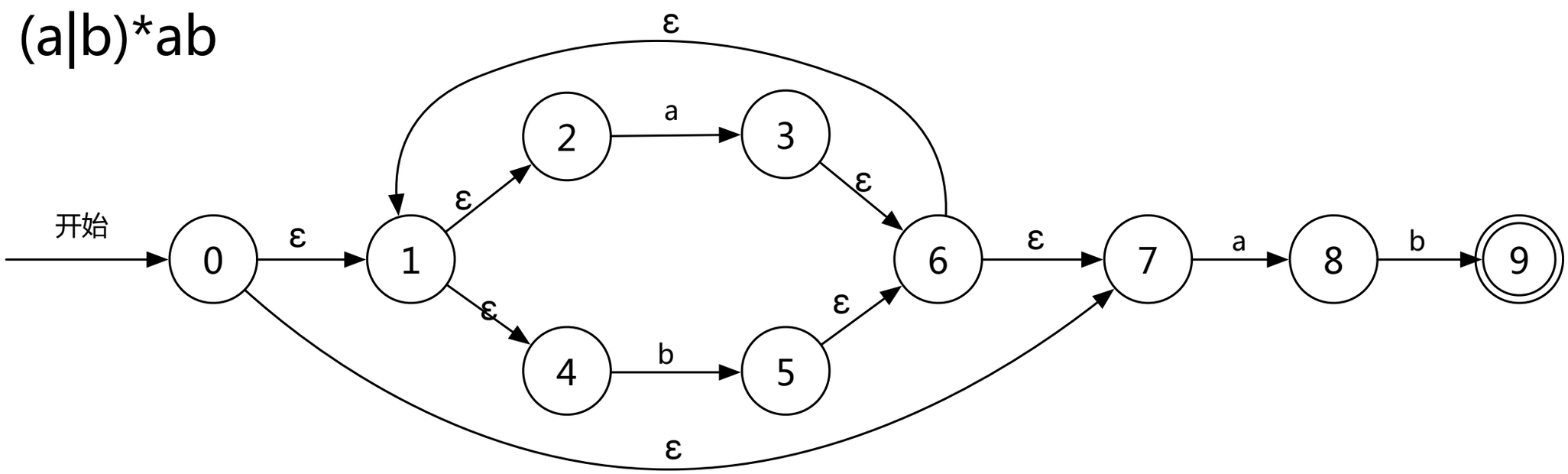




2.5 NFA -> DFA (1)

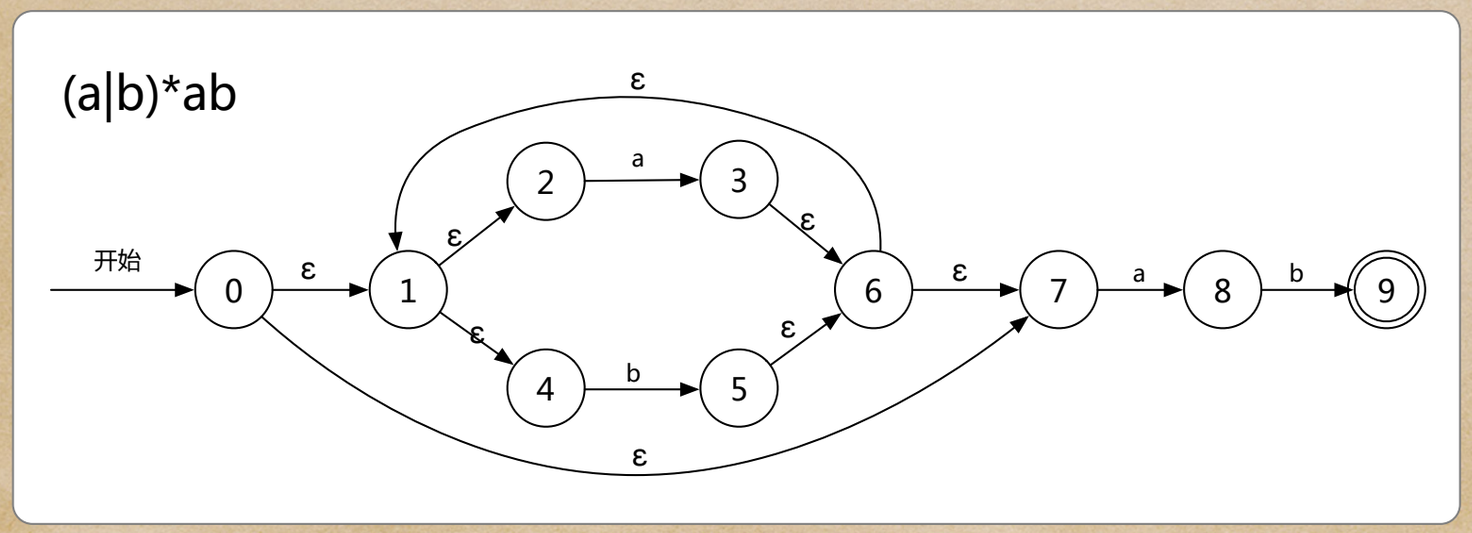
- 还记得 NFA & DFA 的差别么？

(a|b)*ab



2.5 NFA -> DFA (2)

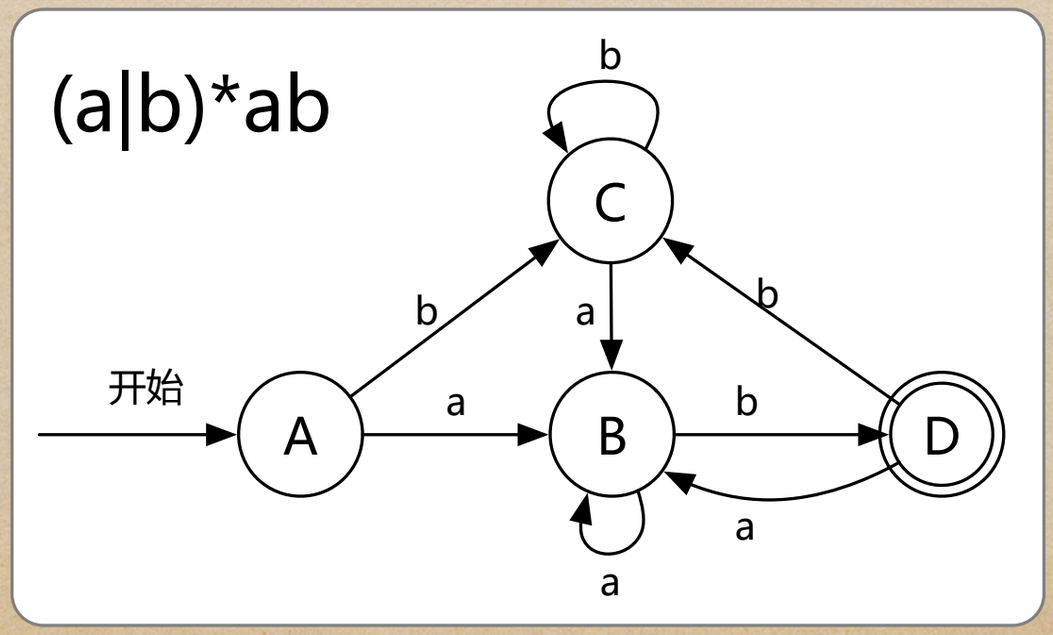
- A = {0,1,2,4,7}
- ε-closure(move(A,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = B
- ε-closure(move(A,b)) = ε-closure({5}) = {1,2,4,5,6,7} = C
- ε-closure(move(B,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = B
- ε-closure(move(B,b)) = ε-closure({5,9}) = {1,2,4,5,6,7,9} = D
- ε-closure(move(C,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = B
- ε-closure(move(C,b)) = ε-closure({5}) = {1,2,4,5,6,7} = C
- ε-closure(move(D,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = B
- ε-closure(move(D,b)) = ε-closure({5}) = {1,2,4,5,6,7} = C



2.5 NFA -> DFA (3)

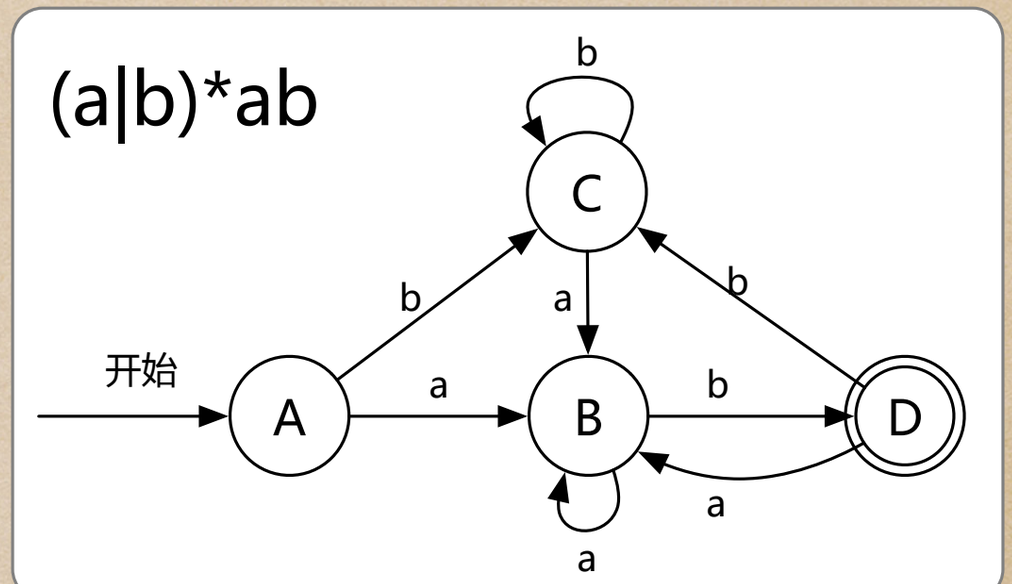
- A = {0,1,2,4,7}
- B = {1,2,3,4,6,7,8}
- C = {1,2,4,5,6,7}
- D = {1,2,4,5,6,7,9}

状态/输入符号	a	b
A	B	C
B	B	D
C	B	C
D	B	C



2.6 DFA化简（1）

- 1) {A,B,C} & {D}
- 2-1) move(A, a) = B = move(B, a) = move(C, a)
- 2-2) move(A, b) = C = move(C, b)
- 2-3) move(C, b) = D
- 2-4) {A,B,C} -> {A,C} & {B}
- 3) 结果：{A,C} & {B} & {D} 即AC状态合并

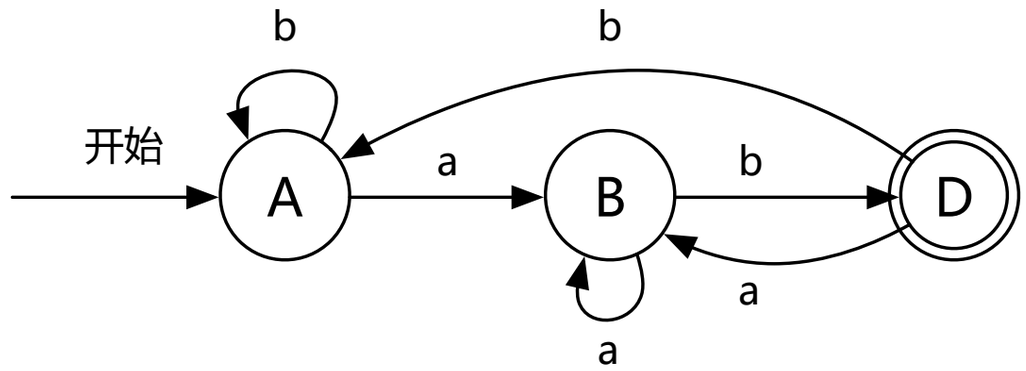


状态/输入符号	a	b
A	B	C
B	B	D
C	B	C
D	B	C

2.6 DFA化简（2）

- 检查一下是不是DFA？
- 想一想为何AC可以合并？

(a|b)*ab



状态/输入符号	a	b
A	B	C
B	B	D
D	B	C

2.7 价值

- 为什么要学习正规表达式？
- 为什么要将正规表达式转换成状态转换图？
- 为什么要将NFA转换成DFA？

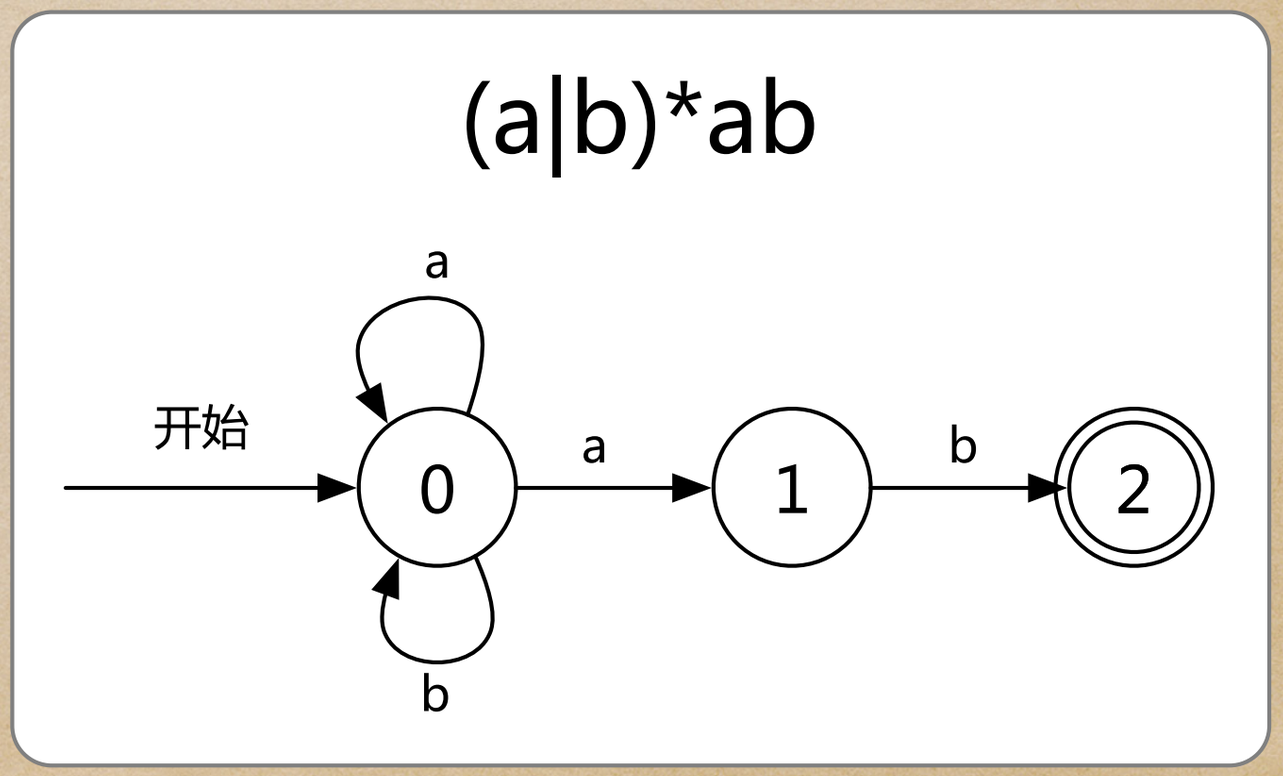
2.8 练习

如图，该状态转换图也是 (a|b)*ab 的一种表示形式，请直接基于这张状态转换图：

1. 将其转换成DFA

2. 比较你转换的DFA和

我们Demo中转换的DFA



三、实践 LEX

3.1 Lex文件构成

- 文件名以.l为后缀（如test.l）
- 开头代码段（%{ ... }%）
- 正规式定义段
- 标记识别段（%% ... %%）
- 末尾代码段

3.2 Lex Demo

```
/* 开头代码段 */
%{
#include <stdio.h>
%}

/* 正规式定义段 */
word                    [a-zA-Z]+
number                  [0-9]+

/* 词法规则段 */
%%
{number}                printf("NUMBER");
{word}                  {
                            printf("WO");
                            printf("RD");
                        }
%%

/* 末尾代码段 */
int main(int argc, char *argv[]) {
    FILE *fp;
    fp = fopen(argv[1], "r");
    yyin = fp;
    while (yylex () != 0) ;
    fclose(fp);
    return 0;
}
```

3.3 Lex编译

1）.l -> .c

flex -o test.c test.l

2）.c -> 可执行文件

gcc -o test test.c -lfl

3.4 执行

./test

```
yangyifans-MacBook-Pro:test1 Number & Word yangyifan$ ./test


123
NUMBER
hi
WORD
hello
WORD
453
NUMBER
```

3.5 体验

- 运行Demo中的 test, 结合对应的 test.l 查看其效果

3.6 练习

- 修改Demo中的 test.l, 使之以test.txt文件 为输入的输出符合期望输出

输入文件test.txt

```
/* test.txt */
int a = 5;
int b = 3;
int c = 0;
do {
    a = a - 1;
    c = (a <> b);
} while (a > b);
```

期望输出结果

```
/* 期望输出 */
符号类型      具体值
变量          int
变量          a
操作符         =
数字          5
界符          ;
变量          int
变量          b
操作符         =
数字          3
界符          ;
变量          int
变量          c
操作符         =
数字          0
界符          ;
关键字         do
界符          {
变量          a
操作符         =
变量          a
操作符         -
数字          1
界符          ;
变量          c
操作符         =
界符          (
变量          a
操作符         <>
变量          b
界符          )
界符          ;
界符          }
关键字         while
界符          (
变量          a
操作符         >
变量          b
界符          )
界符          ;
```