北京大学计算机学院

2024 年春季学期　　《编译原理》

第二讲　词法分析

Lexical Analysis

主要内容

- 词法分析的作用
- 词法分析的规约
- 词法分析的手动实现
- 词法分析的自动生成

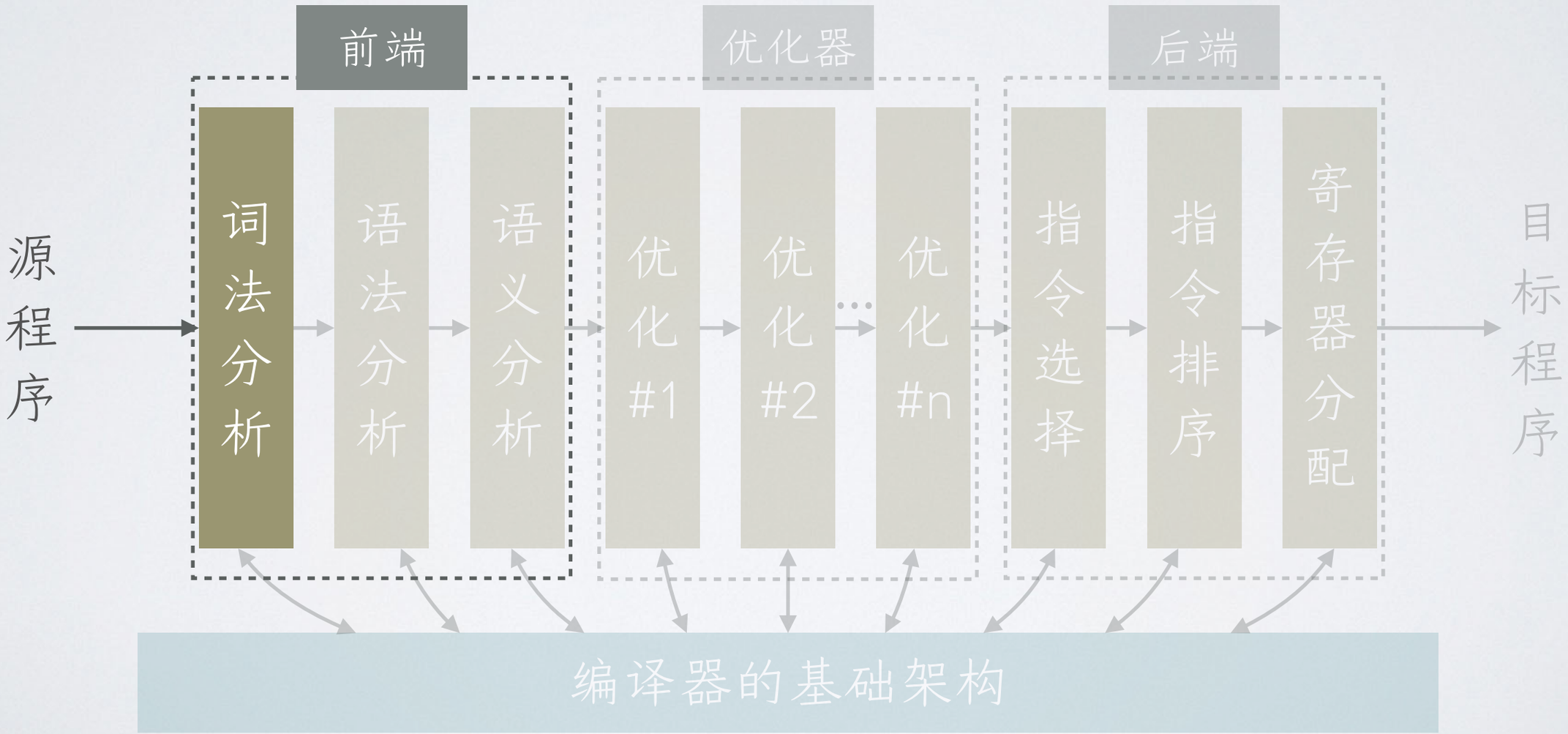
- 对应章节：第 3 章

主要内容

- **词法分析的作用**
- 词法分析的规约
- 词法分析的手动实现
- 词法分析的自动生成

词法分析的作用

- 读取源程序的字符流，输出**词法单元(token)**序列
- 把「字母」组合成「单词」



词法分析示例

```
a = a * 2 * b * c * d;
```

<ID, “a”> <EQ, “=”> <ID, “a”> <MUL, “*”> <NUMBER, “2”> <MUL, “*”>
<ID, “b”> <MUL, “*”> <ID, “c”> <MUL, “*”> <ID, “d”> <SEMI, “;”>

词法单元类别

词素(lexeme)

词法单元

可以附带**属性值**
例：**NUMBER** 单元的数值

- 注：词素是源程序中的字符序列，一般不用记录在 token 中

```
while (i != j) {
  if (i > j) i = i - 1; else j = j - 1;
}
```

<WHILE> <LPAREN> <ID,"i"> <NEQ> <ID,"j"> <RPAREN> <LCURLY>
<IF> <LPAREN> <ID,"i"> <GT> <ID,"j"> <RPAREN> <ID,"i"> <EQ> <ID,"i"> <SUB> <NUMBER,1> <SEMI>
<ELSE> <ID,"j"> <EQ> <ID,"j"> <SUB> <NUMBER,1> <SEMI>
<RCURLY>

词法分析的工作

- 主要工作：识别**词法单元(token)**
  - 输出词法单元的**类别**和(可选的)**属性值**
- 其它工作
  - 去除**注释**和**空白符号**
  - 进行一些预处理，如识别和展开**宏(macro)**
  - 对**数值常数**进行数值字符串到机器表示(如 int、float)的转换
  - 把编译器生成的**错误信息**关联到源程序(行、列)
  - 与后续的语法分析、语义分析进行交互
  - ……

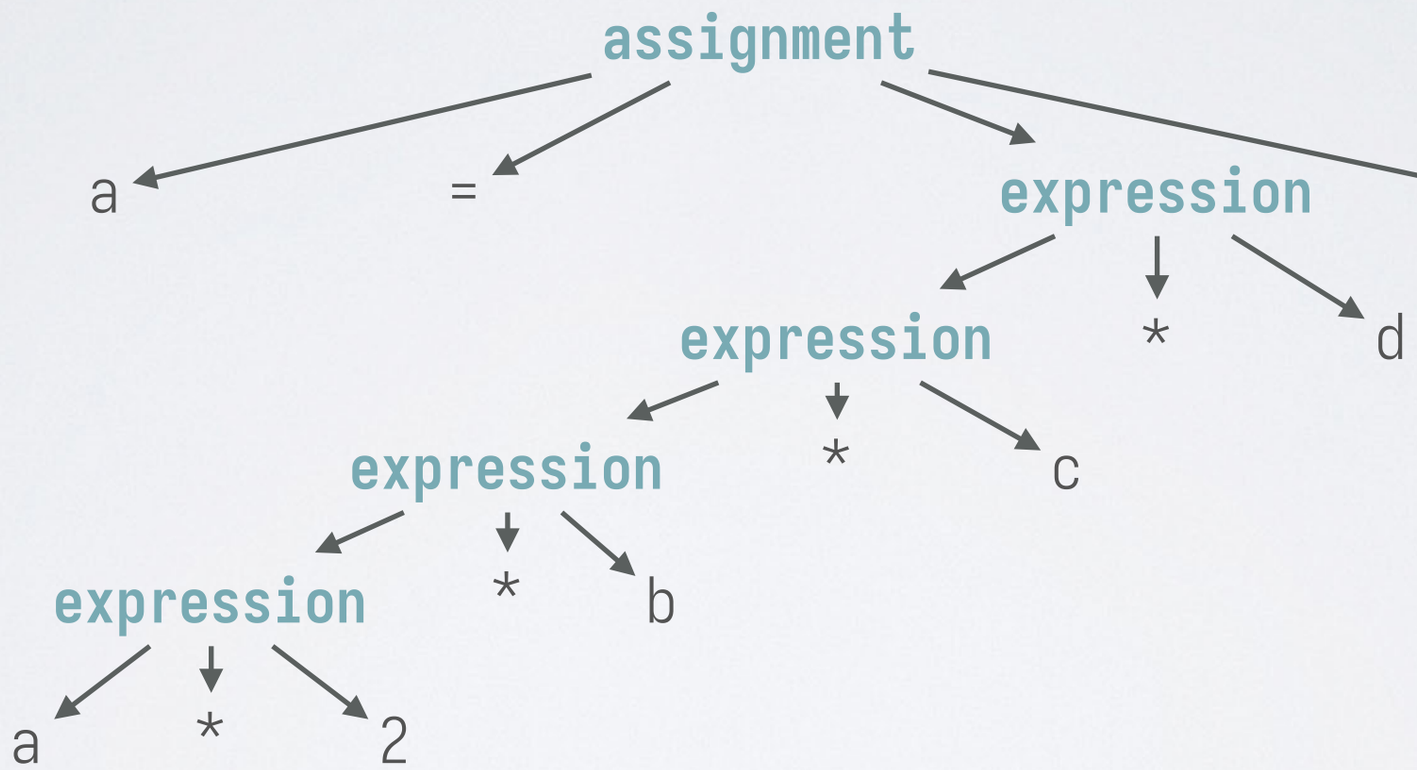
常用的 token 类别和属性

- **关键字、保留字**(keyword)
  - while, if, for, void, typedef, ……
  - 也可以每个关键字单做一类
- **标识符**(identifier)
  - 用来表示各种名字，如变量名、函数名等
- **字面常数**(literal)
  - 256, 3.1415926, 1e20, true, "Hello World\n"
- **运算符**(operator)
  - 如 +、-、*、/ 等
- **分界符**(delimiter)
  - 如分号、逗号、冒号等

为什么需要词法分析？

- 为什么不直接从源程序的字符流进行语法分析呢？

```
a = a * 2 * b * c * d;
```

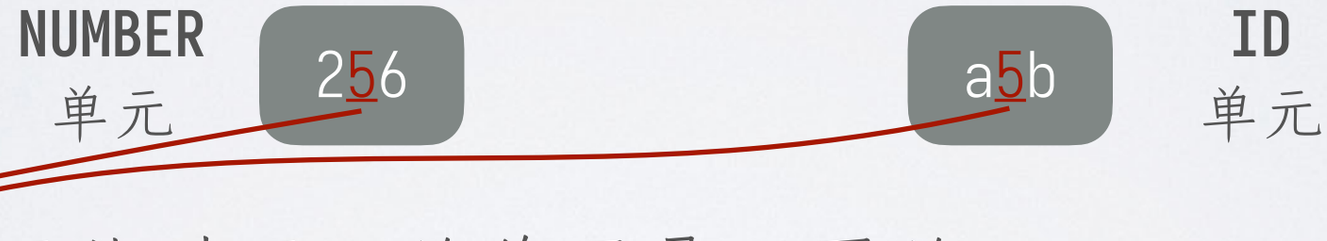


- 使用 token 序列表示程序有什么好处？

自然语言模型中的 token

```
Language Learning Models (LLMs) have revolutionized the field of natural
 language processing, enabling machines to understand and generate human-
like text. At the core of LLMs lies the concept of tokens, which serve as
 the fundamental building blocks for processing and representing text
 data. In this blog post, we'll demystify tokens in LLMs, unraveling
 their significance and exploring how they contribute to the power and
 flexibility of these remarkable models.
```

- 「表达语义的最小单元」



- 字符 5 在这两处对语义的作用是不同的
- 自然语言模型可以考虑比单词更细粒度的 token(为什么?)
- **注:** 在讲语法分析时我们会看到编译中的词法、语法分析可以合并

图片来源: https://www.linkedin.com/pulse/demystifying-tokens-llms-understanding-building-blocks-lukas-selin/.

主要内容

- 词法分析的作用
- **词法分析的规约**
- 词法分析的手动实现
- 词法分析的自动生成

什么是规约？

- **规约(specification)**
  - 描述问题、定义、算法等的形式化记号
- **词法分析的规约：**每一种词法单元类别的定义
- **正则表达式(regular expression)**
  - 用一个正则表达式来表示一类词法单元

字母表（alphabet）

- **字母表**是符号的非空有限集合，一般用 Σ 表示
- 可以类比自然语言的字母表
- 每个编程语言都有自己的字母表
  - 机器语言：由符号 0 和 1 组成，$\Sigma = \{0,1\}$
  - ASCII 字符集
  - Pascal 语言：$\Sigma = \{A\text{-}Z,a\text{-}z,0\text{-}9,+,-,*,/,<,=,>,:,,,;,..,^,(,),\{,\},[,]\}$

符号串（string）

- 给定字母表 Σ，**符号串**是由 Σ 中符号组成的有限序列

- **归纳定义法**

  (1) ϵ 是 Σ 上的一个符号串(空串)

  (2) 若 β 是 Σ 上的符号串，而 a 是 Σ 中的符号，则 $a\beta$ 是 Σ 上的符号串

- 归纳法定义了满足上述条件的**最小集合**
  - 若 α 是 Σ 上的符号串，当且仅当它由(1)或(2)导出

- 例：如何通过归纳法定义自然数？

  (1) 0 是自然数

  (2) 若 n 是自然数，则 $n+1$ 也是自然数

记号约定

- 用英文字母表比较靠前的小写字母表示**符号**
  - 如：$a, b, c, d, \cdots\cdots$
- 用小写希腊字母或字母表比较靠后的小写字母表示**符号串**
  - 如：$\alpha, \beta, \gamma, \theta, \pi, \rho, \cdots\cdots$
    - 注：ϵ 比较特殊，通常用来表示空串
  - 如：$\cdots\cdots, x, y, z$
- 用大写字母表示**符号串集合**
  - 如：$A, B, C, D, \cdots\cdots, X, Y, Z$

符号串的运算

- **连接**(concatenation)：设 x 和 y 是符号串，它们的连接 xy 是把 y 的符号写在 x 的符号之后得到的符号串
  - 例：$x = \text{ba}, y = \text{nana}, xy = \text{banana}$
  - 归纳定义：

    (1) 若 $x = \epsilon$，则 $xy = y$

    (2) 若 $x = az$，其中 a 是符号，z 是符号串，则 $xy = a\underline{zy}$

- 如何证明对任意符号串 x，有 $x\epsilon = x$?
  - **归纳证明：**

    (1) 若 $x = \epsilon$，则左边和右边都等于 ϵ

    (2) 若 $x = az$，根据归纳有 $z\epsilon = z$，则 $x\epsilon = \underline{az}\epsilon = a\underline{z\epsilon} = az = x$

符号串的运算

- **方幂**(exponentiation)：设 x 是符号串，n 是自然数，x^n 定义为
  (1) 若 $n=0$，则 $x^n=\epsilon$
  (2) 若 $n=k+1$，则 $x^n=x^kx$，即 x^k 与 x 的连接

- 如何证明对任意自然数 n，有 $\epsilon^n=\epsilon$?
  - 归纳证明：
    (1) 若 $n=0$，则 $\epsilon^n=\epsilon$
    (2) 若 $n=k+1$，根据归纳有 $\epsilon^k=\epsilon$，则 $\epsilon^n=\epsilon^k\epsilon=\epsilon\epsilon=\epsilon$

- **在归纳定义的数学对象上通过归纳法证明其性质**

正则表达式

- 给定字母表 Σ，**正则表达式**按照下面的方式进行归纳定义：
  - (1) ϵ 是一个正则表达式，它能匹配空串
  - (2) 若 $a \in \Sigma$，则 a 是一个正则表达式，它能匹配单个 a 形成的串
  - (3) 若 r 和 s 是正则表达式，那么
    - (a) **选择**：$r|s$ 是一个正则表达式，它能匹配的串可以被 r 或 s 匹配
    - (b) **连接**：rs 是一个正则表达式，它能匹配的串可以分成前后两个部分，分别能被 r 和 s 匹配
    - (c) **闭包**：r^* 是一个正则表达式，它能匹配的串可以分成 n 个部分 $(n \geq 0)$，其中每个部分都能被 r 匹配

- 注：正则表达式中可以加括号明确计算顺序，一般认为三种运算中选择的优先级最低，闭包的优先级最高

正则表达式示例

- 考虑 $\Sigma = \{a,b\}$
  - a|b　　能匹配 a, b
  - (a|b)(a|b)　　能匹配 aa, ab, ba, bb
  - a^*　　能匹配 ϵ, a, aa, aaa, aaaa, ……
  - (a|b)* 或 $(a^*b^*)^*$　　能匹配 ϵ, a, b, aa, ab, ba, bb, aaa, ……
  - a^*b　　能匹配 b, ab, aab, aaab, ……

- **C 语言的标识符:**
  (A|B|...|Z|a|b|...|z|_)(A|B|...|Z|a|b|...|z|_|0|1|...|9)*

- **有符号整数:**
  (+|-|ϵ)(0|1|...|9)(0|1|...|9)*

语言（language）

- **如何严格定义正则表达式的含义？**
- 一个**语言(language)**是某个字母表上的符号串集合
  - 该集合可以有(可数)无限个元素
- 语言的例子
  - 空集 $\varnothing$
  - 只包含空串的集合 $\{\epsilon\}$
  - 所有符合规范的 C 语言标识符的集合
  - 所有语法正确的 C 语言程序的集合
  - 所有语法正确的英语句子的集合

正则语言

- 正则表达式 r 匹配的符号串构成其对应的语言 $L(r)$
  - (1) $L(\epsilon) = \{\epsilon\}$
  - (2) 若 a 是字母表中的符号，则 $L(a) = \{a\}$
  - (3) 若 r 和 s 是正则表达式，那么
    - (a) 选择：$L(r \mid s) = L(r) \cup L(s)$
    - (b) 连接：$L(rs) = \{xy \mid x \in L(r), y \in L(s)\}$
    - (c) 闭包：$L(r^*) = \{x_1 x_2 \cdots x_n \mid n \geq 0, x_1, x_2, \cdots, x_n \in L(r)\}$
- 能被一个正则表达式表达的语言被称为**正则语言**
- 若 $L(r) = L(s)$，则认为正则表达式 r 和 s 等价

正则表达式的性质

- 设 r_1、r_2、r_3 是三个正则表达式，则有
  - 单位正则表达式：$\epsilon r_1 = r_1 \epsilon = r_1$
  - 交换律：$r_1 | r_2 = r_2 | r_1$
  - 结合律：
    - $r_1 | (r_2 | r_3) = (r_1 | r_2) | r_3$
    - $r_1(r_2 r_3) = (r_1 r_2) r_3$
  - 分配率：
    - $r_1(r_2 | r_3) = r_1 r_2 | r_1 r_3$
    - $(r_1 | r_2) r_3 = r_1 r_3 | r_2 r_3$
  - $r_1^* = (r_1 | \epsilon)^*$　　$(r_1^*)^* = r_1^*$　　$(r_1 | r_2)^* = (r_1^* r_2^*)^*$

正则表达式的扩展

- 为了方便表达，可以对正则表达式进行如下扩展：
  - **1 次或多次出现：** r^+ 能匹配的串可以分成 n 个部分$(n > 0)$，其中每个部分都能被 r 匹配
    - $r^* = r^+ | \epsilon$
    - $r^+ = rr^* = r^*r$
  - **0 次或 1 次出现：** $r?$ 能匹配空串或 r 能匹配的串
    - $r? = r | \epsilon$
  - **字符类：** [abc] 表示 a|b|c，[a-z] 表示 a|b|c|...|z
- **C 语言的标识符：** [A-Za-z_]([A-Za-z_0-9])*

正则表达式作为词法分析的规约

- 使用一系列的正则表达式，标明它们每个对应的 token 类别

token 类别	正则表达式
IF	`if`
WHILE	`while`
ID	`[A-Za-z_]([A-Za-z_0-9])*`
NUMBER	`[+-]?([0-9])+`
LPAREN	`(`
NEQ	`!=`

- 反复从源程序字符流中用正则表达式进行匹配，输出 token
- 一般而言，在实现时采取如下策略：
  - 选择**匹配长度最大**的正则表达式
  - 如果有多个匹配长度相同，则选择**排在前面**的正则表达式

正则表达式作为词法分析的规约

token 类别	正则表达式
WHILE	while
ID	[A-Za-z_]([A-Za-z_0-9])*
LPAREN	(
NOT	!
NEQ	!=

<LPAREN, "(">

```
while (i != j) {
  if (i > j)
    i = i - 1;
  else
    j = j - 1;
}
```

- **WHILE** 和 **ID** 的正则表达式都能匹配 while
  - 选择排在前面的，输出词法单元 <**WHILE**>
- **NOT** 和 **NEQ** 的正则表达式分别能匹配 ! 和 !=
  - 选择匹配长度更大的，输出词法段元 <**NEQ**>

主要内容

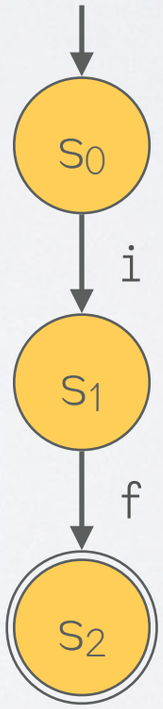
- 词法分析的作用
- 词法分析的规约
- **词法分析的手动实现**
- 词法分析的自动生成

如何识别词法单元？

- 前面我们使用正则表达式来描述词法单元的规约
- 要实现词法分析，需要按照正则表达式的表述处理字符流

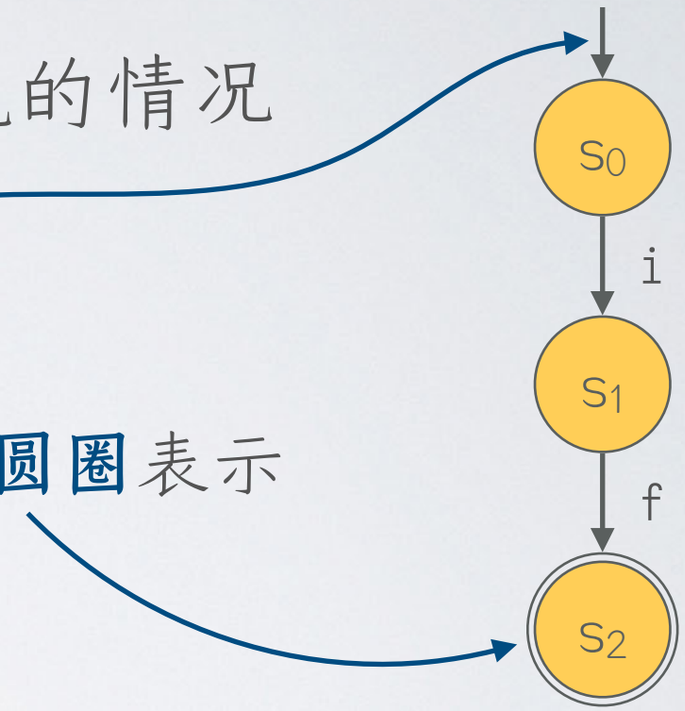
token 类别	正则表达式
IF	`if`

```
c = next_char();
if (c == 'i') {
  c = next_char();
  if (c == 'f') {
    // 识别成功，输出词法单元 IF
  } else {
    // 识别失败
  }
} else {
  // 识别失败
}
```



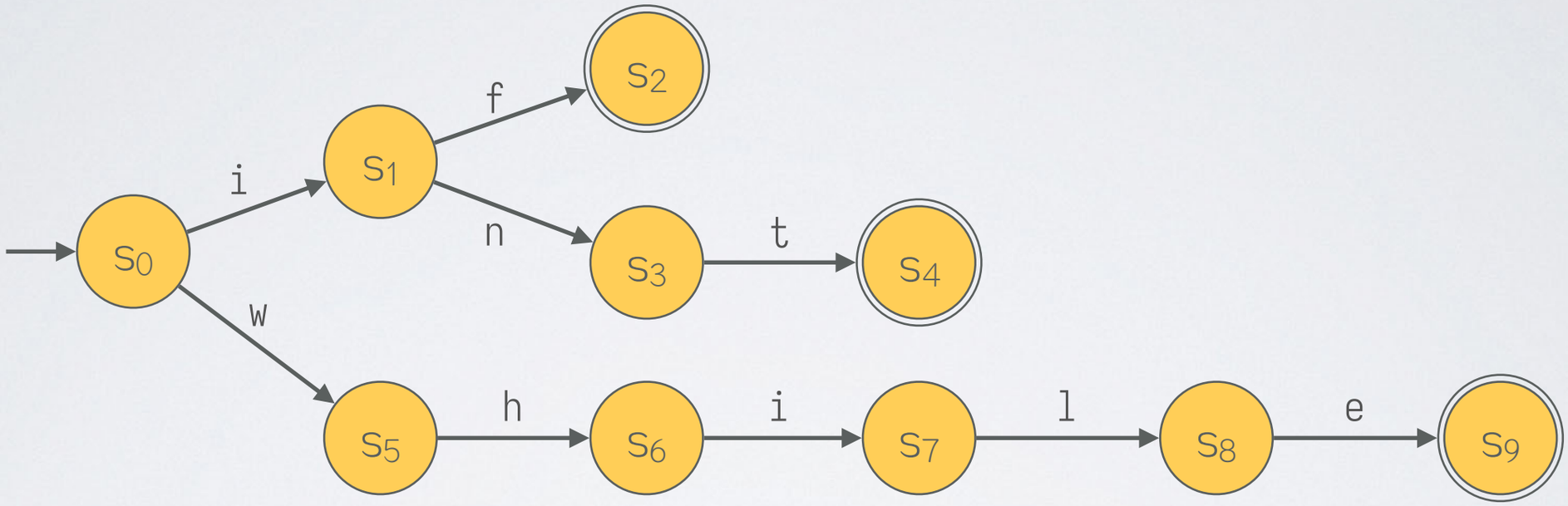
状态转移图

- **状态转移图**(transition diagram)
- **状态(state)**：表示在识别过程中可能出现的情况
  - 状态可看作已识别部分的总结
  - 有一个特殊的**初始状态**，用一条**初始边**表示
  - 某些状态为**接受状态**，表明识别成功，用**双层圆圈**表示



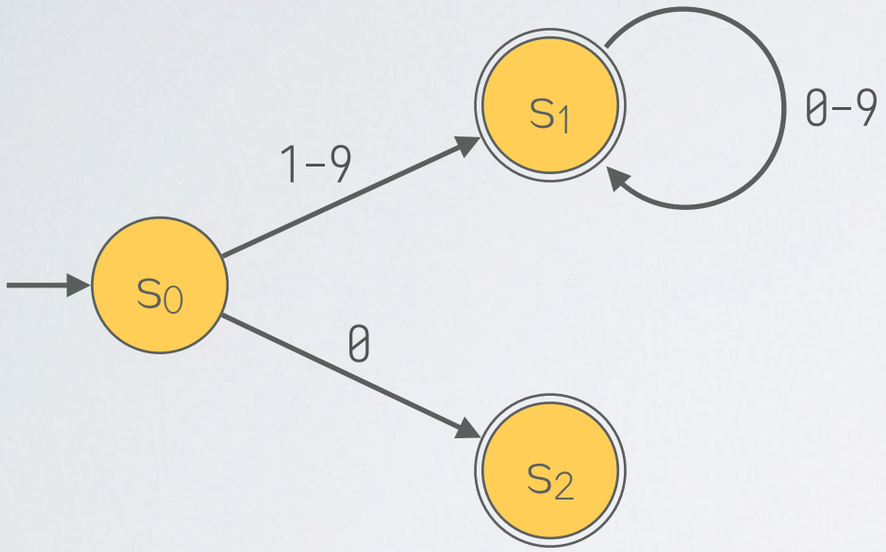
- **边(edge)**：表示识别了若干符号后，状态发生转移
  - 边的标号表示转移状态时识别的符号
  - 没有画出的边(比如从 s_1 出发标号为 a)默认表示转移到错误状态

状态转移图



识别 if、int、while 的状态转移图

带环的状态转移图



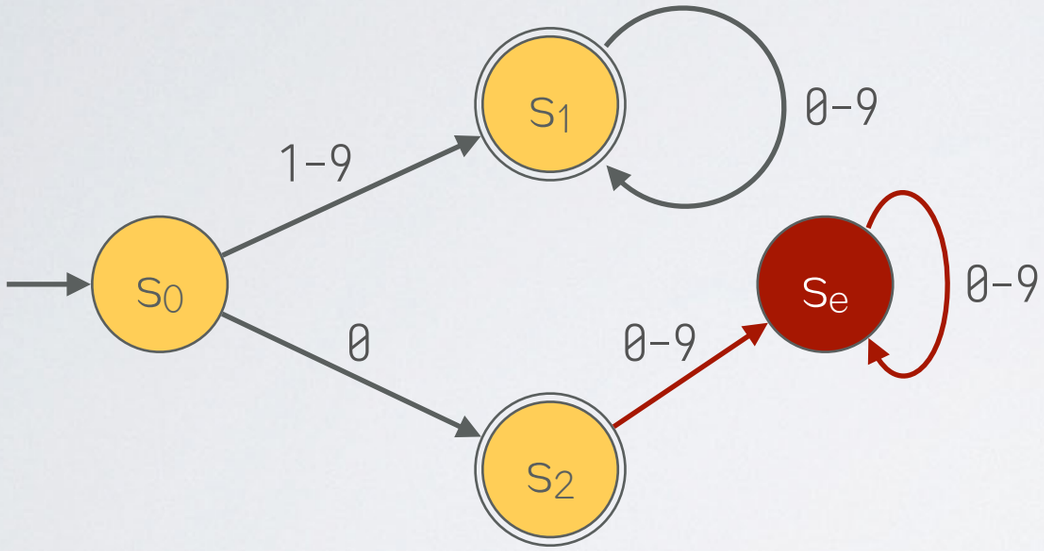
识别无符号整数的转移图
(前面没有多余的零)

```
c = next_char();
s = STATE_0;
while (c != EOF && s != STATE_ERROR) {
  switch (s) {
    case STATE_0:
      if (c == '0') s = STATE_2;
      else if (c == '1' || ... || c == '9') s = STATE_1;
      else s = STATE_ERROR;
      break;
    case STATE_1:
      if (c == '0' || ... || c == '9') s = STATE_1;
      else s = STATE_ERROR;
      break;
    case STATE_2:
      s = STATE_ERROR;
      break;
  }
  c = next_char();
}
if (s == STATE_1 || s == STATE_2) {
  // 识别成功
} else {
  // 识别失败
}
```

确定性有限自动机

- **Deterministic Finite Automaton, DFA**
- 对状态转移图的形式化定义
- 一个 DFA D 是一个五元组 $D = (S, \Sigma, \delta, s_0, F)$，其中
  - S 是一个有限的**状态集合**
  - Σ 是该 DFA 使用的**字母表**
  - δ 是一个从 $S \times \Sigma$ 到 S 的映射，称为**转移函数**
    - $\delta(s, a) = s'$ 表示当前状态为 s，输入符号为 a 时，自动机状态转移到 s'，此时称 s' 是 s 的一个**后继**
  - $s_0 \in S$ 为**初始状态**
  - $F \subseteq S$ 为**接受状态**的集合

DFA 示例



识别无符号整数的转移图
(前面没有多余的零)

$$D = (S, \Sigma, \delta, s_0, F)$$

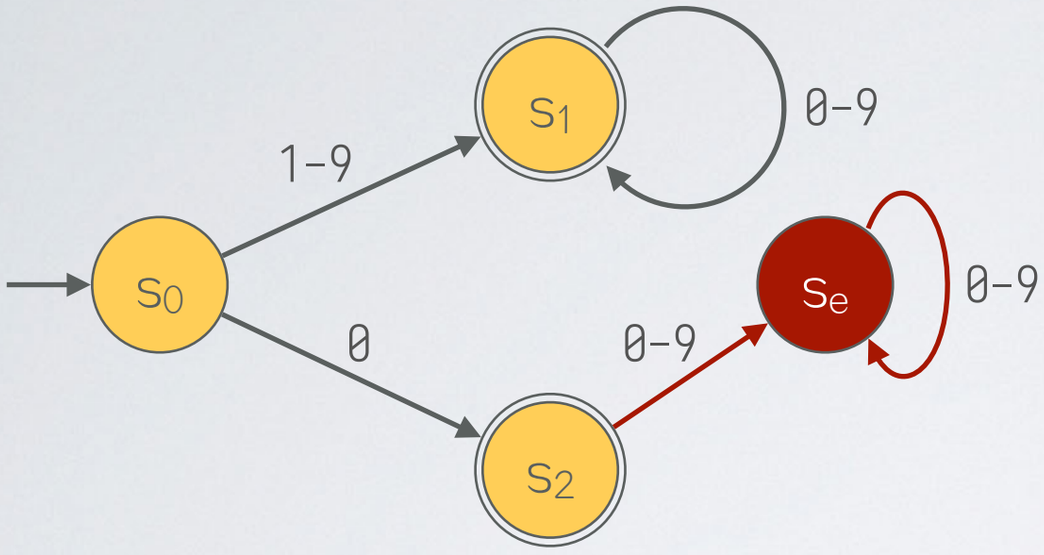
$$S = \{s_0, s_1, s_2, \boxed{s_e}\}$$

$$\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$$

$$\delta = \left\{ \begin{array}{ccc} s_0 \xrightarrow{1\text{-}9} s_1 & , & s_0 \xrightarrow{0} s_2 \\ s_1 \xrightarrow{0\text{-}9} s_1 & , & s_2 \xrightarrow{0\text{-}9} s_e \\ s_e \xrightarrow{0\text{-}9} s_e & & \end{array} \right\}$$

$$F = \{s_1, s_2\}$$

实现基于 DFA 的识别



$$S = \{s_0, s_1, s_2, s_e\}$$

$$\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$$

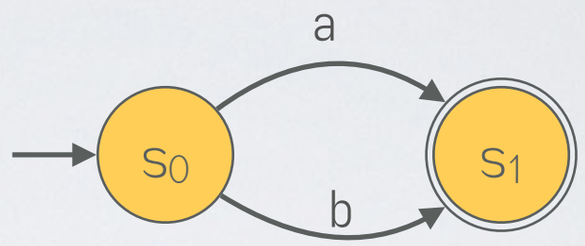
$$\delta = \left\{ \begin{array}{ccc} s_0 \xrightarrow{1\text{-}9} s_1 & , & s_0 \xrightarrow{0} s_2 \\ s_1 \xrightarrow{0\text{-}9} s_1 & , & s_2 \xrightarrow{0\text{-}9} s_e \end{array} \right\}$$

$$F = \{s_1, s_2\}$$

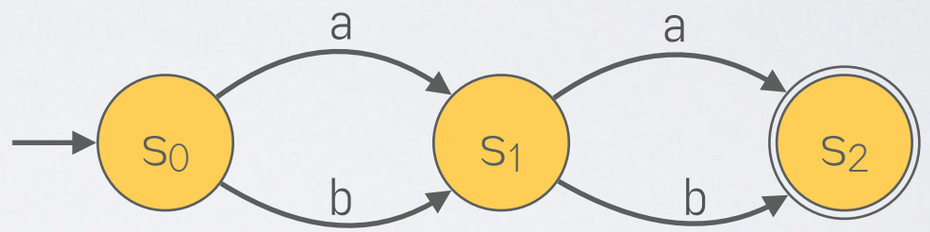
```
c = next_char();
s = s_0;
while (c != EOF && s != s_e) {
  s = δ(s, c);
  c = next_char();
}
if (s ∈ F) {
  // 识别成功
} else {
  // 识别失败
}
```

正则表达式转换为 DFA

- 考虑 $\Sigma = \{a,b\}$
  - a|b



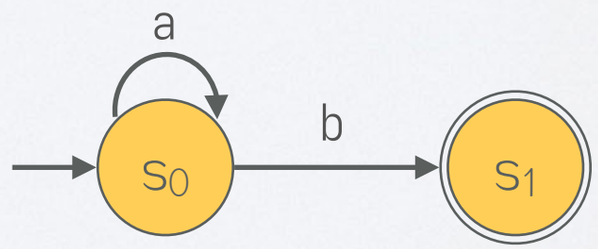
  - (a|b)(a|b)



  - a*

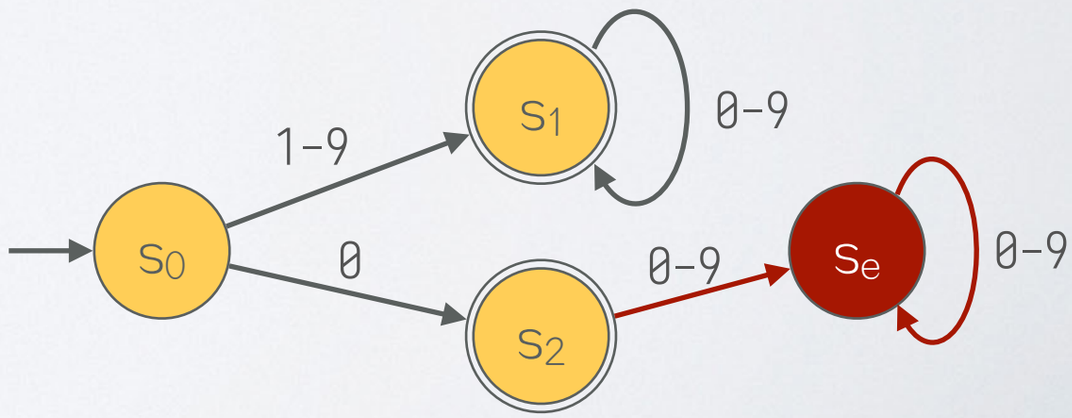
  - (a|b)*

  - a*b



DFA 识别的语言

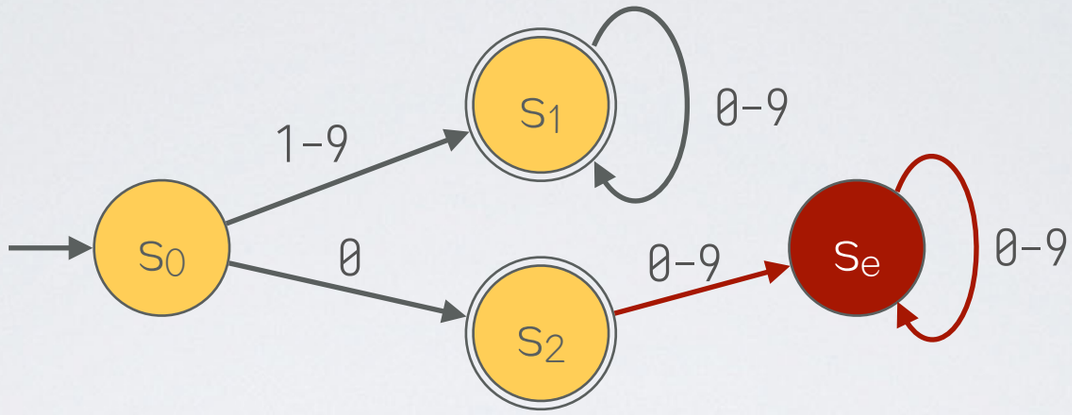
- 怎么证明构造出的 DFA 与正则表达式**等价**?
  - 需要严格定义能被 DFA 识别的**语言**
- 考虑 $D = (S, \Sigma, \delta, s_0, F)$, $\delta : S \times \Sigma \to S$ 为状态转移函数
  - 对任意符号串 w, 用归纳法定义 $\delta(s, w)$, 从状态 s 出发识别串 w 后到达的状态
  - 若 $w = \epsilon$, 则 $\delta(s, w) = s$
  - 若 $w = az$, 其中 a 是一个符号, 则 $\delta(s, w) = \delta(\delta(s, a), z)$
- 例:
  - $\delta(s_0, 0) = s_2$, $\delta(s_0, 01) = s_e$
  - $\delta(s_1, 0099) = s_1$, $\delta(s_0, 2024) = s_1$



DFA 识别的语言

- 考虑 $D = (S, \Sigma, \delta, s_0, F)$, $\delta : S \times \Sigma \to S$ 为状态转移函数
- 对任意串 w, $\delta(s, w)$ 表示状态 s 出发识别串 w 后到达的状态
- DFA D 定义的**语言**定义为: $L(D) = \{w \mid \delta(s_0, w) \in F\}$
  - 即从初始状态出发后，根据串 w 进行状态转移可以到达接受状态
- 怎么证明 DFA D 与正则表达式 r **等价**?
  - 需要说明 $L(D) = L(r)$

DFA 识别的语言

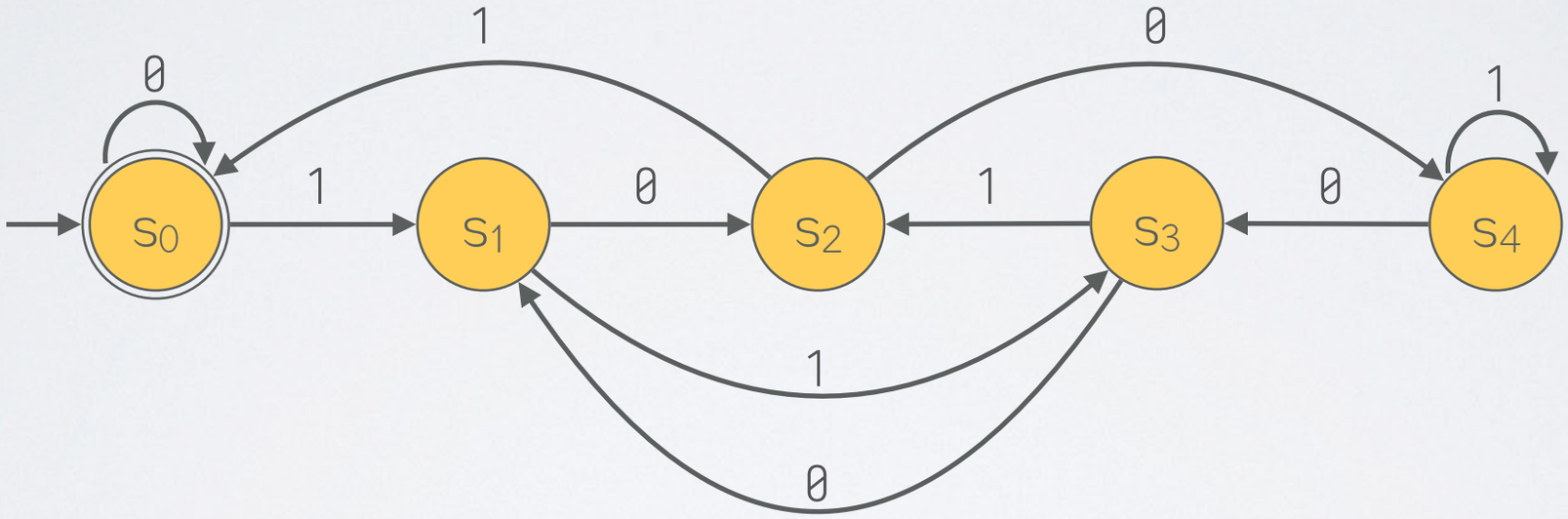


- 证明：这个 DFA (D) 与正则表达式 `0|[1-9]([0-9])*` (r) 等价
  - $L(r) \subseteq L(D)$
    - 根据定义，有

      $L(r) = \{0\} \cup \{w_0 w_1 \cdots w_n \mid w_0 \in [1\text{-}9], n \geq 0, w_1, \cdots, w_n \in [0\text{-}9]\}$
    - 对 $w \in L(r)$ 分类讨论，说明 $\delta(s_0, w) \in \{s_1, s_2\}$
  - $L(D) \subseteq L(r)$
    - 需要说明如果 $\delta(s_0, w) \in \{s_1, s_2\}$，那么 $w \in L(r)$
    - 先说明从 s_1 出发对应 `([0-9])*` 以及从 s_2 出发对应 ϵ

构造 DFA

- 有时，构造 DFA 可能比写正则表达式更容易
- **例：**识别 $\Sigma = \{0,1\}$ 上能被 5 整除的二进制数
  - 构造状态 s_i 表示目前识别的二进制数模 5 余 i



- **问：**怎么构造识别所有被 32 除余 1 的二进制数的 DFA?
- **问：**怎么构造匹配以上两个语言的正则表达式?

实现词法分析

- 构造一个识别所有需要的 token 类别的 DFA D
  - 词法分析过程通常优先考虑**排在前面**的识别规则
  - 每个接受状态对应一个 token 类别
- 反复从源程序字符流中读取符号，直到将会转移到错误状态
  - 词法分析过程通常寻求**最长匹配**
  - 记录最近一次到达过的接受状态
- 处理并输出 token，重设 D 的状态为初始状态，继续读取符号
  - 根据接受状态判断 token 的类别
- **词法识别的规约是正则表达式，怎么自动转换为 DFA?**

主要内容

- 词法分析的作用
- 词法分析的规约
- 词法分析的手动实现
- **词法分析的自动生成**

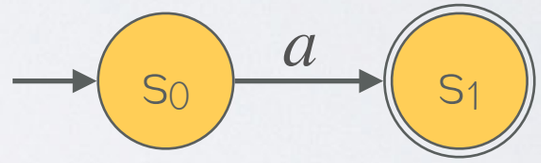
从规约到实现的转换

- **词法分析的规约：**每类 token 的正则表达式
- **词法分析的实现：**基于有限自动机处理字符流、生成 token
- **词法分析的自动生成：**自动化正则表达式到 DFA 的转换

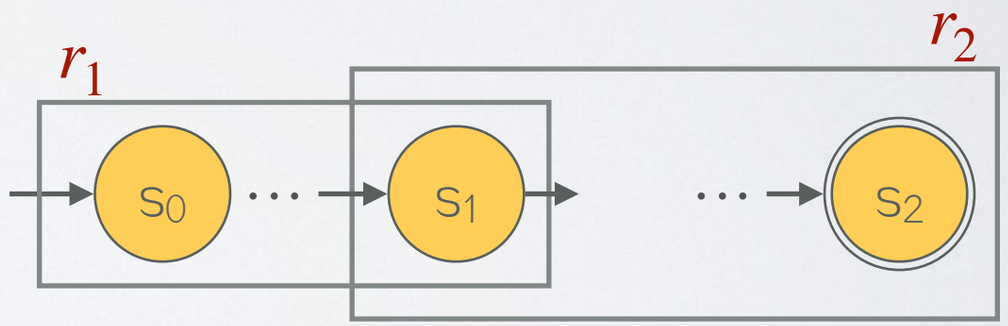
正则表达式转换为 DFA，但是自动

- 给定字母表 Σ 和正则表达式 r，输出 DFA D 满足 $L(D) = L(r)$
- 按照 r 的形式进行分类讨论
  - $r = \epsilon$

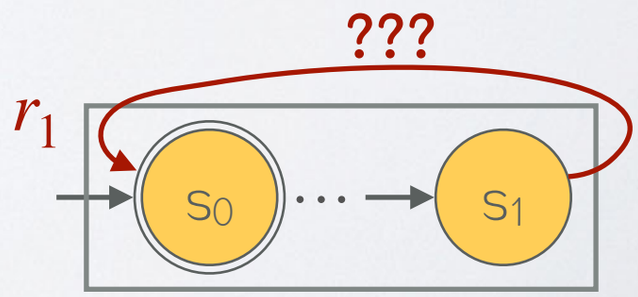
  - $r = a\ (a \in \Sigma)$



  - $r = r_1 r_2$
    - r_1 对应的 DFA 有多个接受状态怎么办？

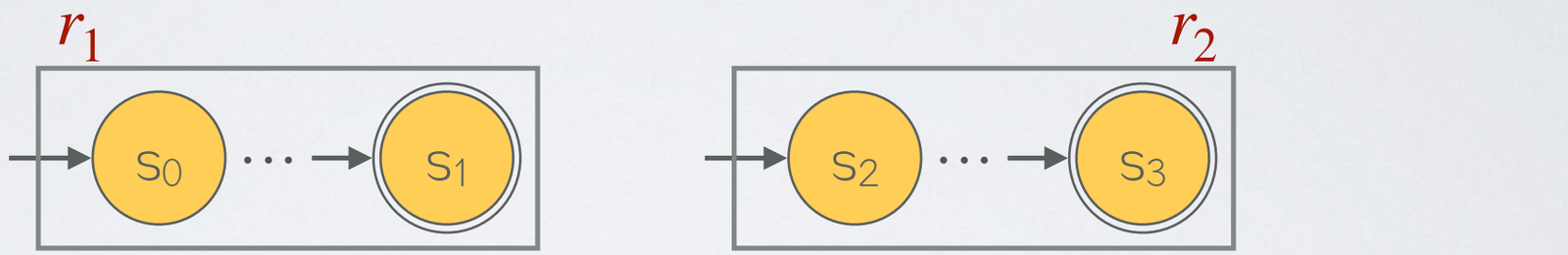


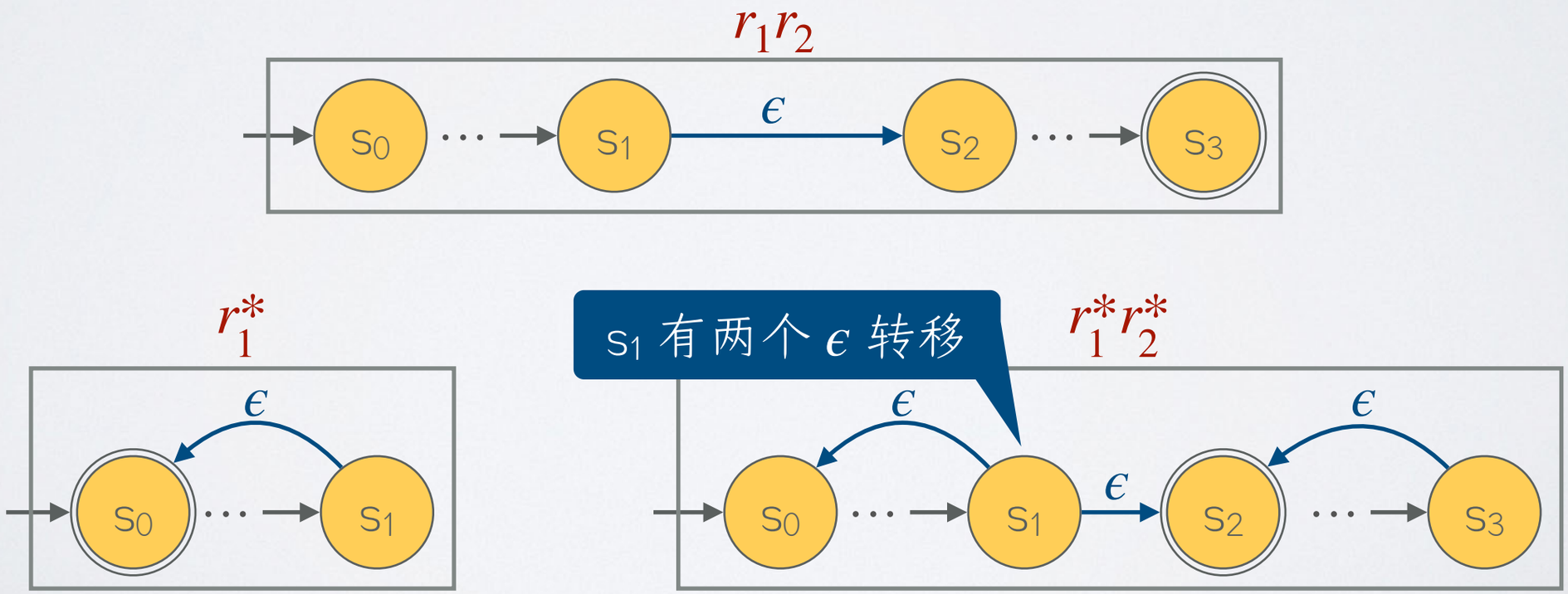
  - $r = r_1^*$ 怎么处理呢？



ϵ 转移

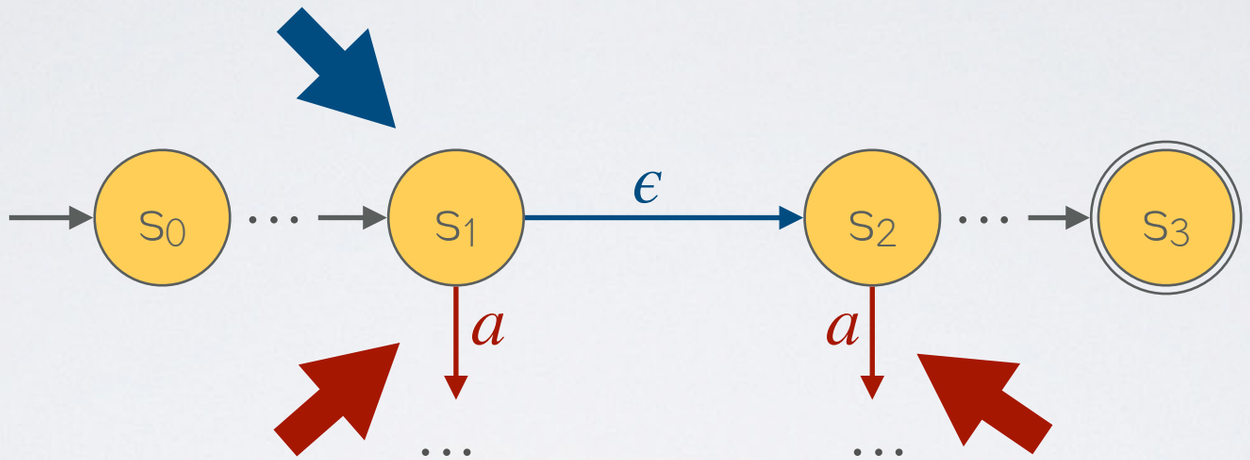
- 在根据正则表达式构造 DFA 时，我们常希望能进行 **ϵ 转移**
  - 即自动机不需要读取符号就进行状态转移





ϵ 转移对状态转移的影响

- 在引入 ϵ 转移后，状态转移不再具有**确定性**

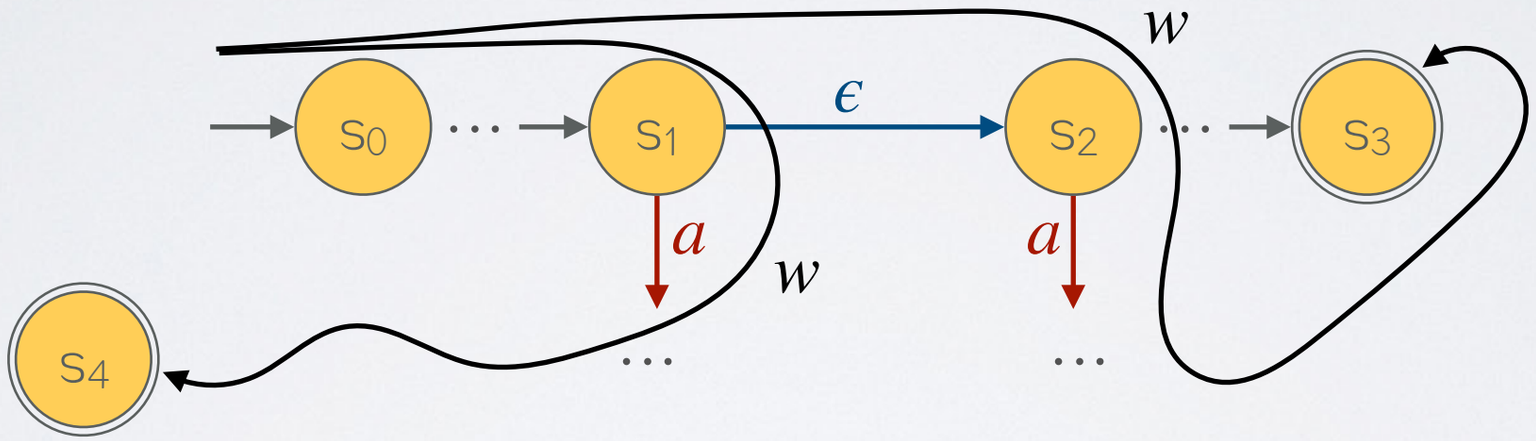


- 假设识别了若干符号后到了状态 s_1，下一个符号是 a，**应该怎么进行转移？**
  - 可以从 s_1 直接转移
  - 也可以先进行 ϵ 转移到 s_2，再从 s_2 转移

非确定性有限自动机

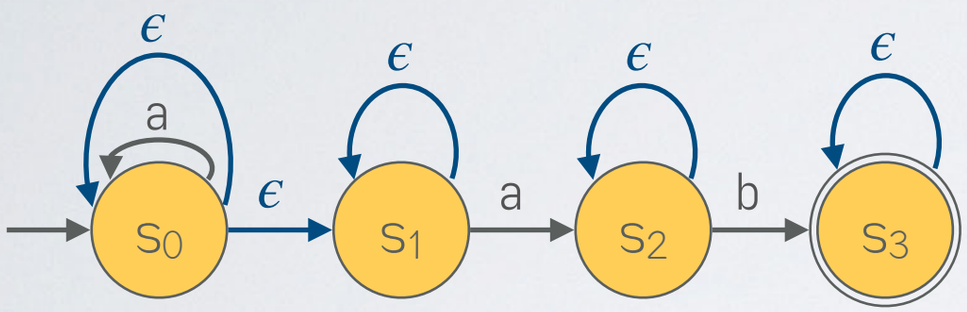
- Nondeterministic Finite Automaton, NFA

- 状态转移具有不确定性的自动机
  - ❖ 只要存在一条转移路径能到达接受状态，就认为识别成功



- 定义：$N = (S, \Sigma, \delta, s_0, F)$ 与 DFA 基本相同，不同之处为
  - ❖ $\delta : S \times (\Sigma \cup \{\epsilon\}) \to 2^S$
    - 2^S 表示 S 所有子集的集合
    - NFA 允许状态对同一符号进行不同的转移

NFA 示例



- 通常不需要错误状态和错误转移
- 转移到自身的 ϵ 转移通常不用画

$$N = (S, \Sigma, \delta, s_0, F)$$

$$S = \{s_0, s_1, s_2, s_3\}$$

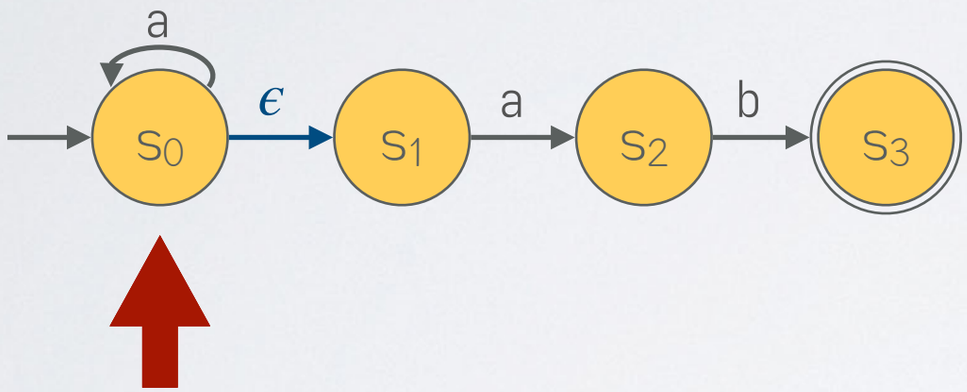
$$\Sigma = \{\text{a,b}\}$$

用空集表示转移到错误状态

$$\delta = \left\{ \begin{array}{llll} s_0 \xrightarrow{a} \{s_0\}, & s_0 \xrightarrow{b} \boxed{\varnothing}, & s_0 \xrightarrow{\epsilon} \{s_0, s_1\} \\ s_1 \xrightarrow{a} \{s_2\}, & s_1 \xrightarrow{b} \varnothing, & s_1 \xrightarrow{\epsilon} \{s_1\} \\ s_2 \xrightarrow{a} \varnothing, & s_2 \xrightarrow{b} \{s_3\}, & s_2 \xrightarrow{\epsilon} \{s_2\} \\ s_3 \xrightarrow{a} \varnothing, & s_3 \xrightarrow{b} \varnothing, & s_3 \xrightarrow{\epsilon} \{s_3\} \end{array} \right\}$$

$$F = \{s_3\}$$

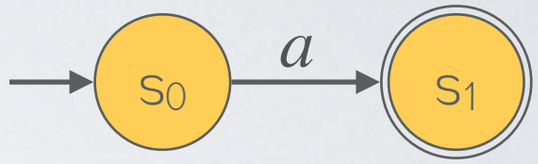
基于 NFA 的识别



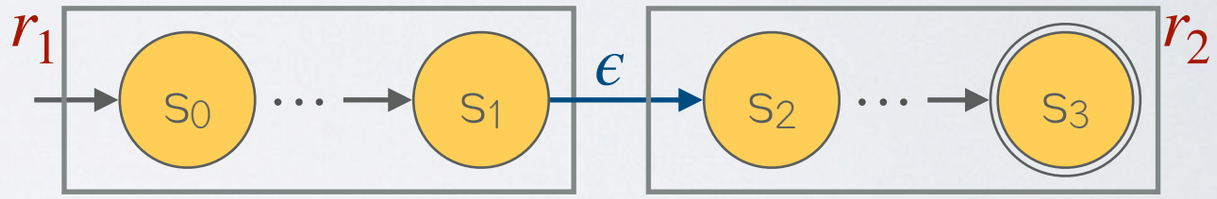
$w =$ a a b

正则表达式转换为 NFA

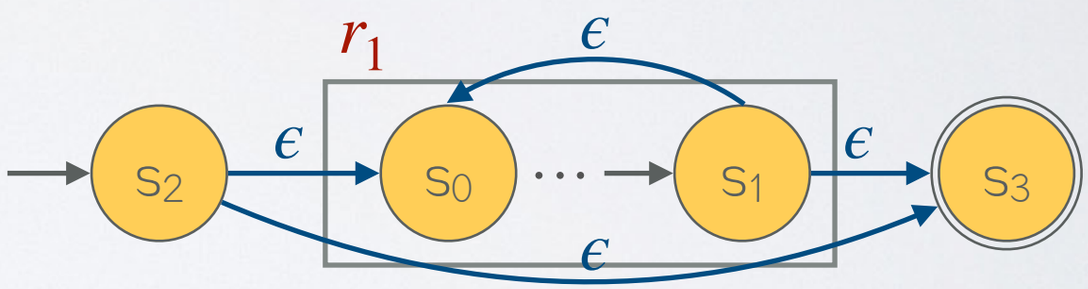
- 给定字母表 Σ 和正则表达式 r, 输出 NFA N 满足 $L(N) = L(r)$
- 按照 r 的形式进行分类讨论
  - $r = \epsilon$　　$r = a\ (a \in \Sigma)$



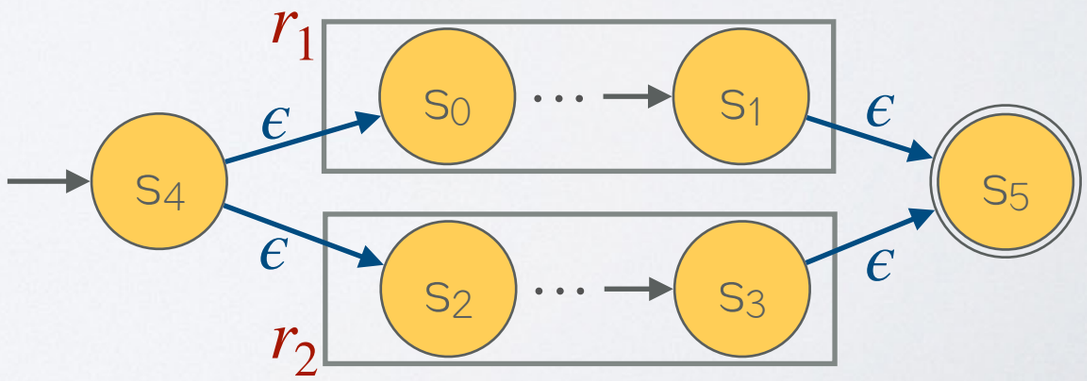
  - $r = r_1 r_2$



  - $r = r_1^*$

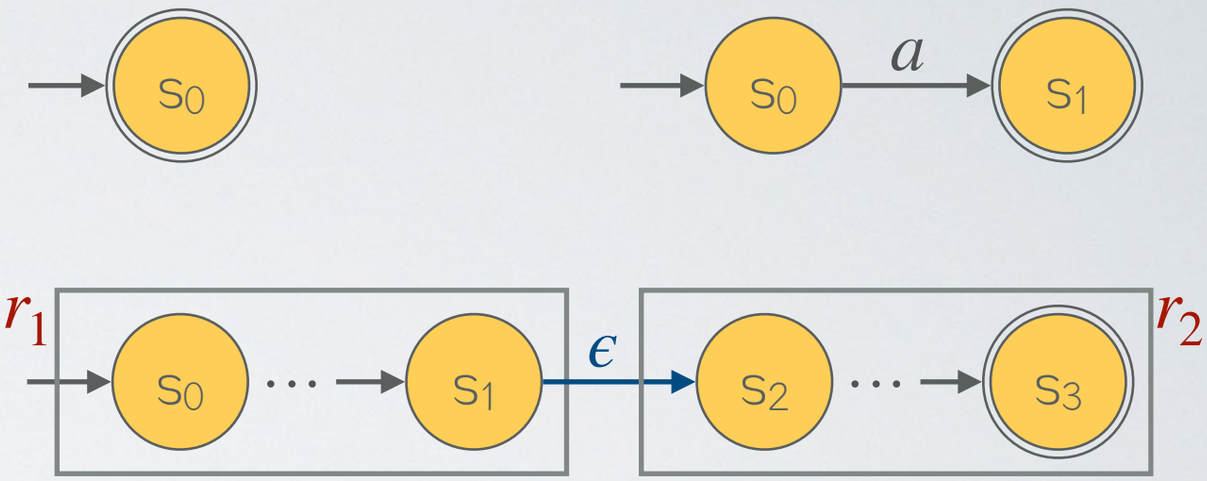


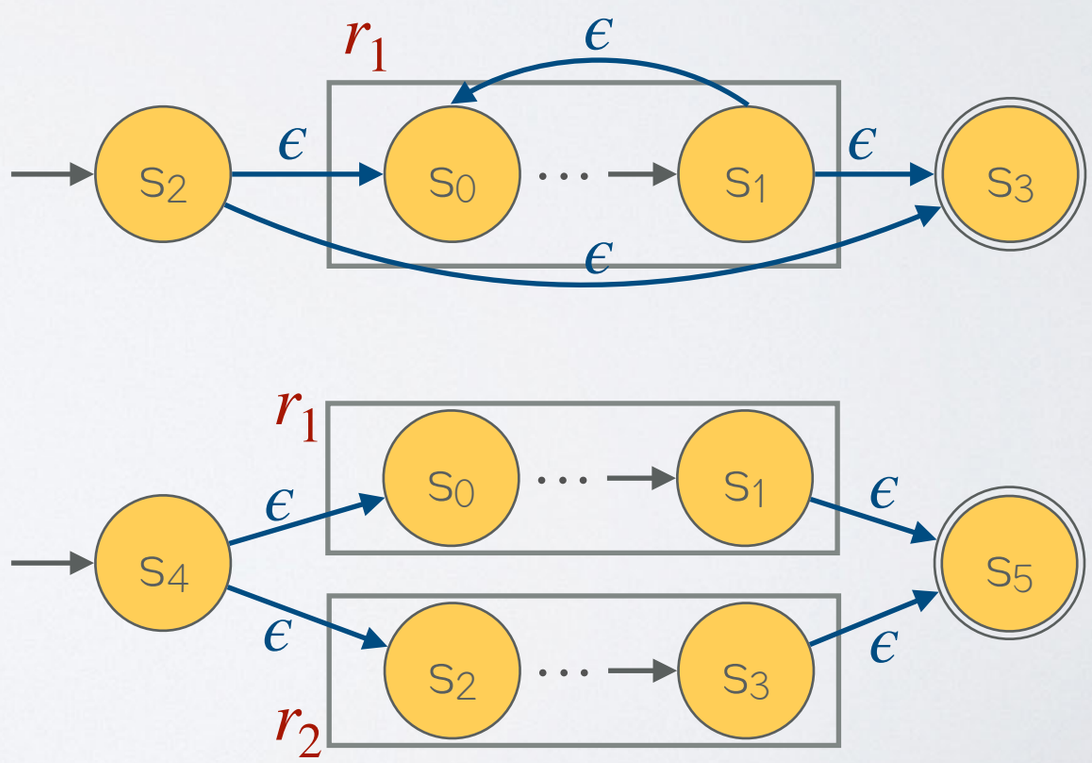
  - $r = r_1 \mid r_2$



正则表达式转换为 NFA

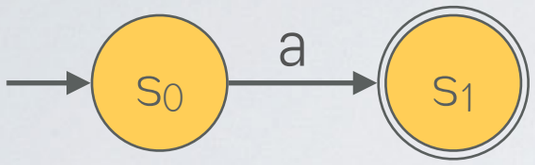
- Thompson 构造法
- **这种构造法有什么优点？**
- **只有一个初始状态**
  - 只有初始转移进入该状态
- **只有一个接受状态**
  - 没有转移从该状态发出
- **每个状态至多有**
  - 两条进入、两条发出的 ϵ 转移
  - 一条进入、一条发出的 Σ 转移



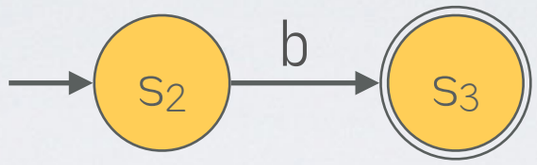


转换示例

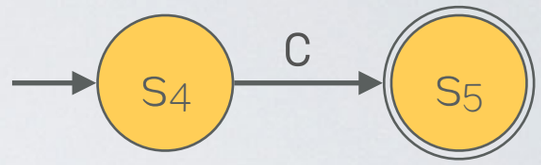
a



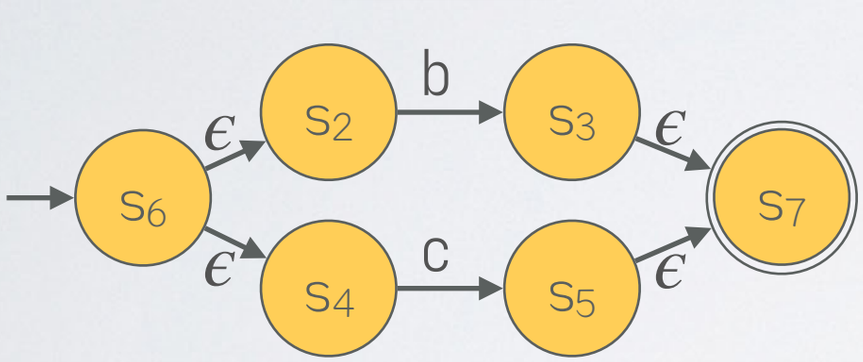
b



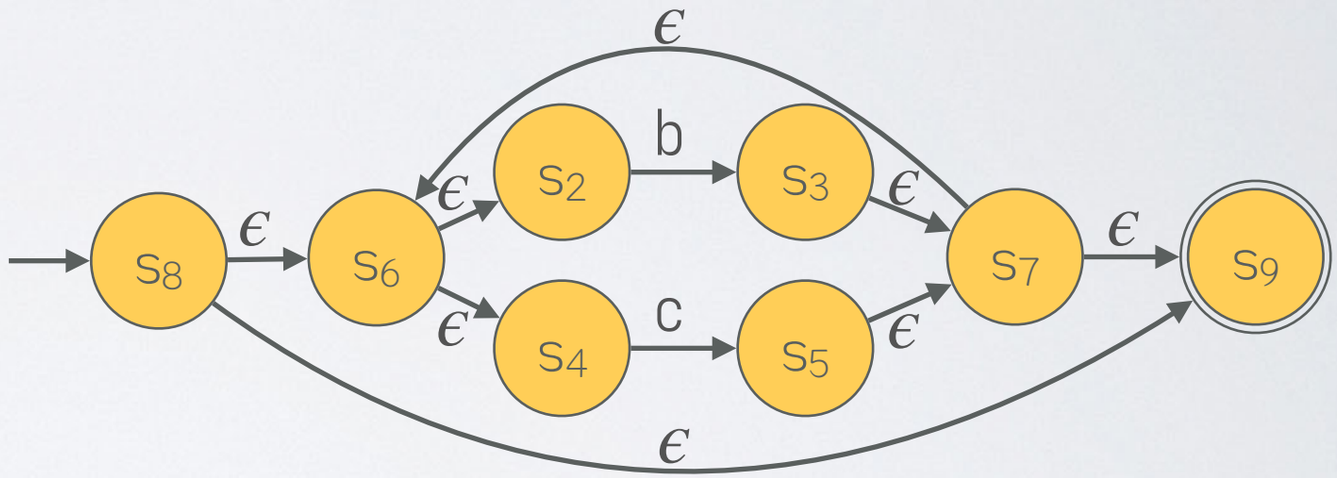
c



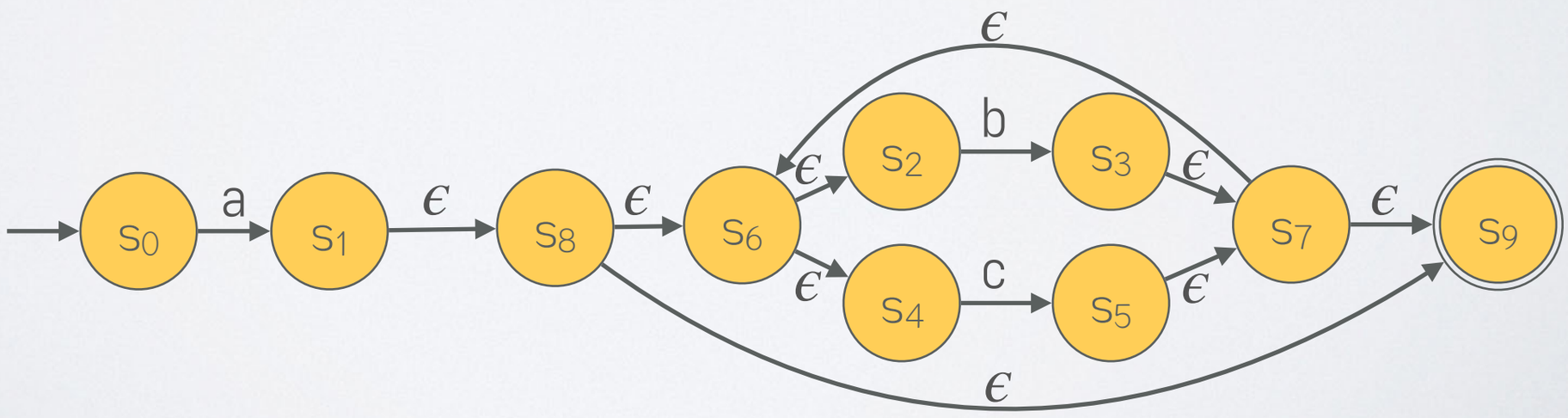
b | c



(b | c)*



a(b | c)*



NFA 转换为 DFA

- NFA 容易构造
- DFA 容易模拟
- NFA 识别语言的能力至少应该与 DFA 相当

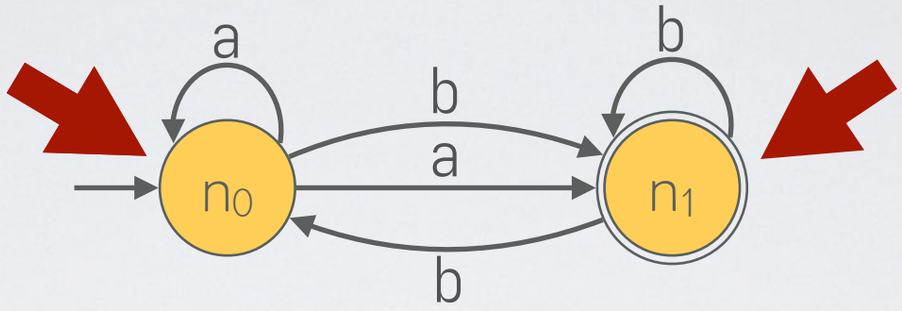
- **定理：对任意一个 NFA N，都存在一个 DFA D，使得 $L(N) = L(D)$**

- **为什么？**
  - **「可能处于的状态」在 NFA 识别过程中仍是有限的**

$$D = (S_D, \Sigma, \delta_D, d_0, F_D)$$

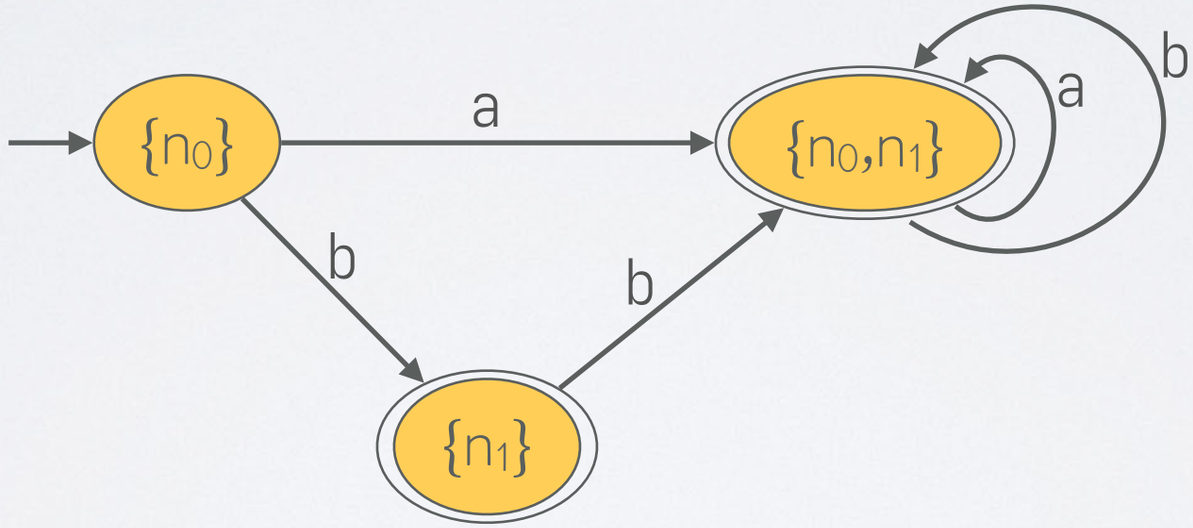
```
c = next_char();
s = d_0;
while (c != EOF && s != d_e) {
  s = δ_D(s, c);
  c = next_char();
}
if (s ∈ F_D) {
  // 识别成功
} else {
  // 识别失败
}
```

子集构造法



NFA

- 要构造确定性自动机，需要分析 NFA 可能**处于哪些状态**

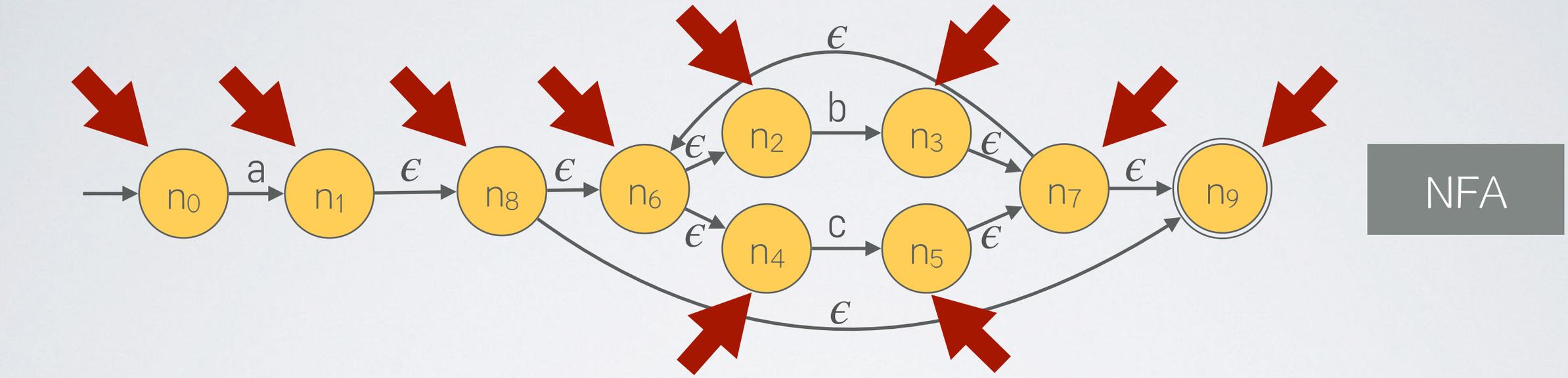


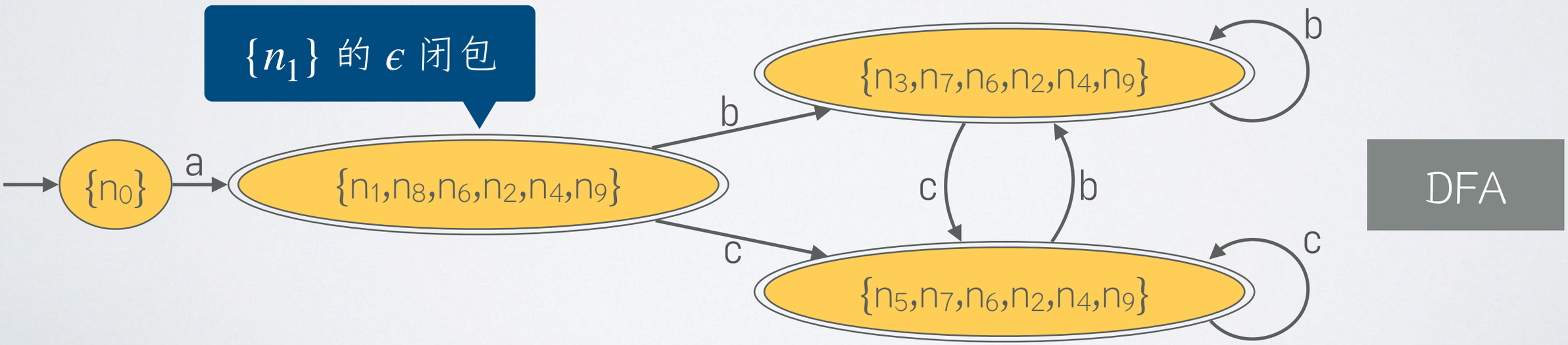
DFA

- 上面的 $\{n_0, n_1\}$ 表示存在符号串 w，在 NFA 上从 n_0 出发识别 w 后既可能处于 n_0，也可能处于 n_1

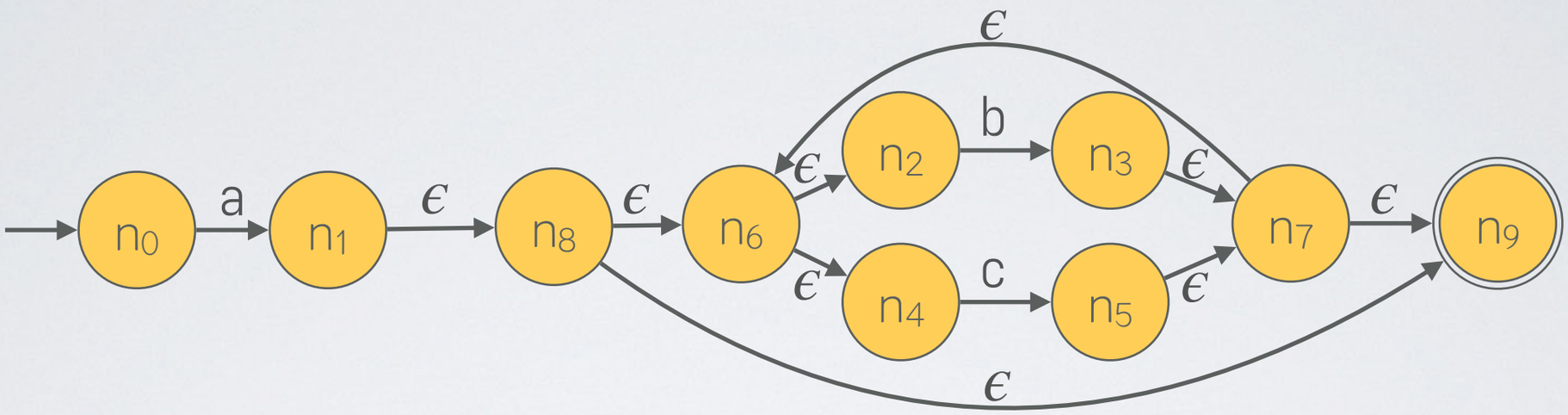
ϵ 闭包

- NFA 中还存在 **ϵ 转移**，在构造子集时需要额外处理





实现 ϵ 闭包的计算

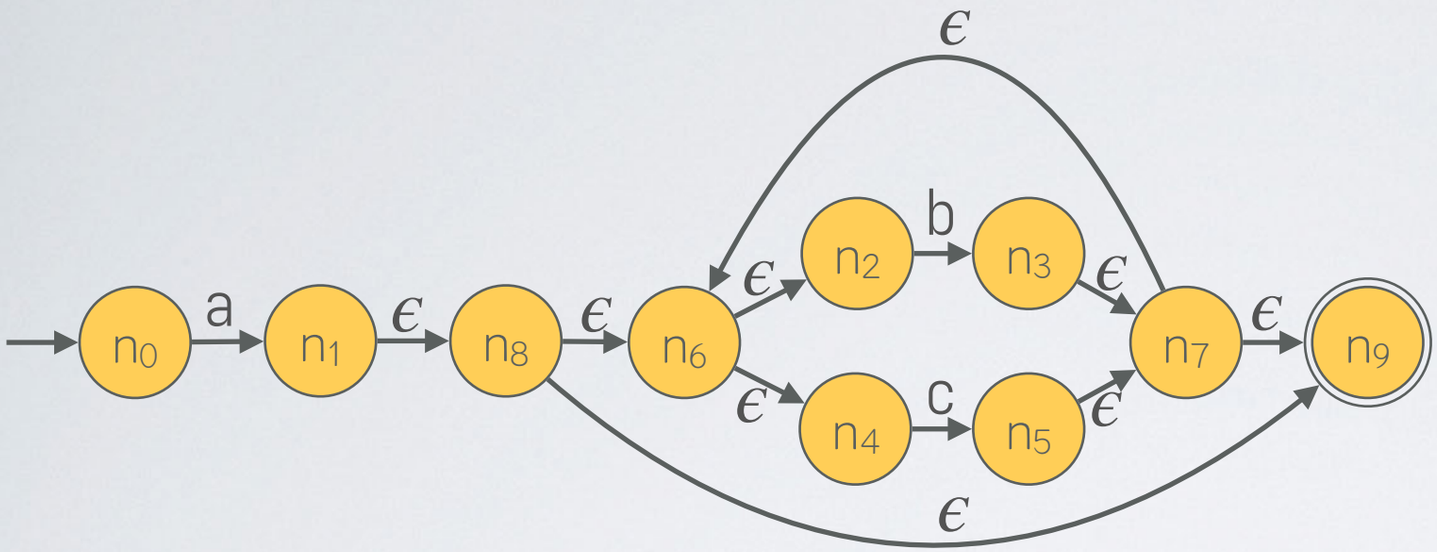


$$N = (S_N, \Sigma, \delta_N, n_0, F_N)$$

- 例：计算 $\{n_1\}$ 的 ϵ 闭包
  - 第一轮：$T = \{n_1, n_8\}$
  - 第二轮：$T = \{n_1, n_8, n_6, n_9\}$
  - 第三轮：$T = \{n_1, n_8, n_6, n_9, n_2, n_4\}$
  - 第四轮：$T = \{n_1, n_8, n_6, n_9, n_2, n_4\}$

```
ϵ_closure(S) {
  T = S;
  do {
    T′ = T;
    T = T′ ∪ ⋃_{s∈T′} δ_N(s, ϵ);
  } while (T != T′);
  return T;
}
```

实现子集构造法



- 第一轮：$q = \{n_0\}$

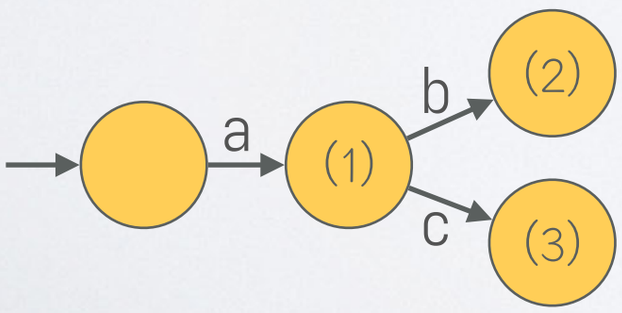
  (1) $\delta_D(q, \mathrm{a}) = \{n_1, n_8, n_6, n_2, n_4, n_9\}$

- 第二轮：$q = \{n_1, n_8, n_6, n_2, n_4, n_9\}$

  (2) $\delta_D(q, \mathrm{b}) = \{n_3, n_7, n_6, n_2, n_4, n_9\}$

  (3) $\delta_D(q, \mathrm{c}) = \{n_5, n_7, n_6, n_2, n_4, n_9\}$

- ……



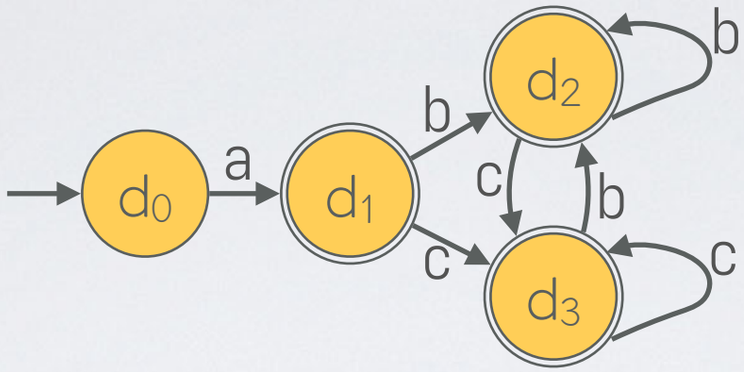
$$N = (S_N, \Sigma, \delta_N, n_0, F_N)$$

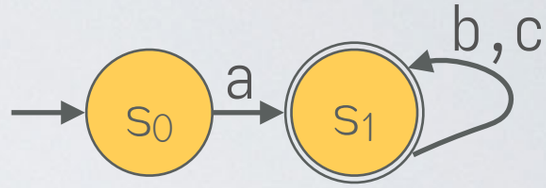
$$D = (S_D, \Sigma, \delta_D, d_0, F_D)$$

```
d0 = ϵ_closure({n0});
SD = {d0};
work_list = {d0};
while (work_list != ∅) {
  q = remove_from(work_list);
  foreach (c ∈ Σ) {
    t = ϵ_closure(⋃s∈q δN(s, c));
    δD(q, c) = t;
    if (t ∉ SD) {
      SD = SD ∪ {t};
      work_list = work_list ∪ {t};
    }
  }
}
FD = {q | q ∈ SD, q ∩ FN ≠ ∅};
```

DFA 的最小化

a(b | c)*





- 这是识别正则表达式所匹配语言的**状态数最少**的 DFA 吗？

- 最小化 DFA 的思路：**找出等价的状态并合并它们**

- 给定 DFA $D = (S, \Sigma, \delta, s_0, F)$，两个状态 p 和 q 等价意味着，对任意符号串 w，有 $\delta(p, w) \in F$ **当且仅当** $\delta(q, w) \in F$

- 例：上面中间的 DFA 中 d_1、d_2、d_3 三个状态等价

等价状态的判别条件

- 两个状态 p 和 q 等价应该满足如下条件：
  - **一致性条件**
    - p 和 q 必须同时为接受状态，或者同时为非接受状态
  - **蔓延性条件**
    - 对任意符号 $a \in \Sigma$, $\delta(p,a) = r$, $\delta(q,a) = s$, r 和 s 必须等价

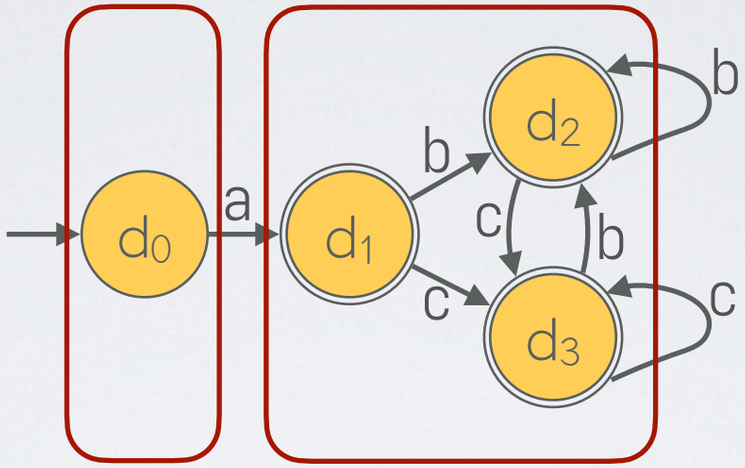
- **Hopcroft 算法：**

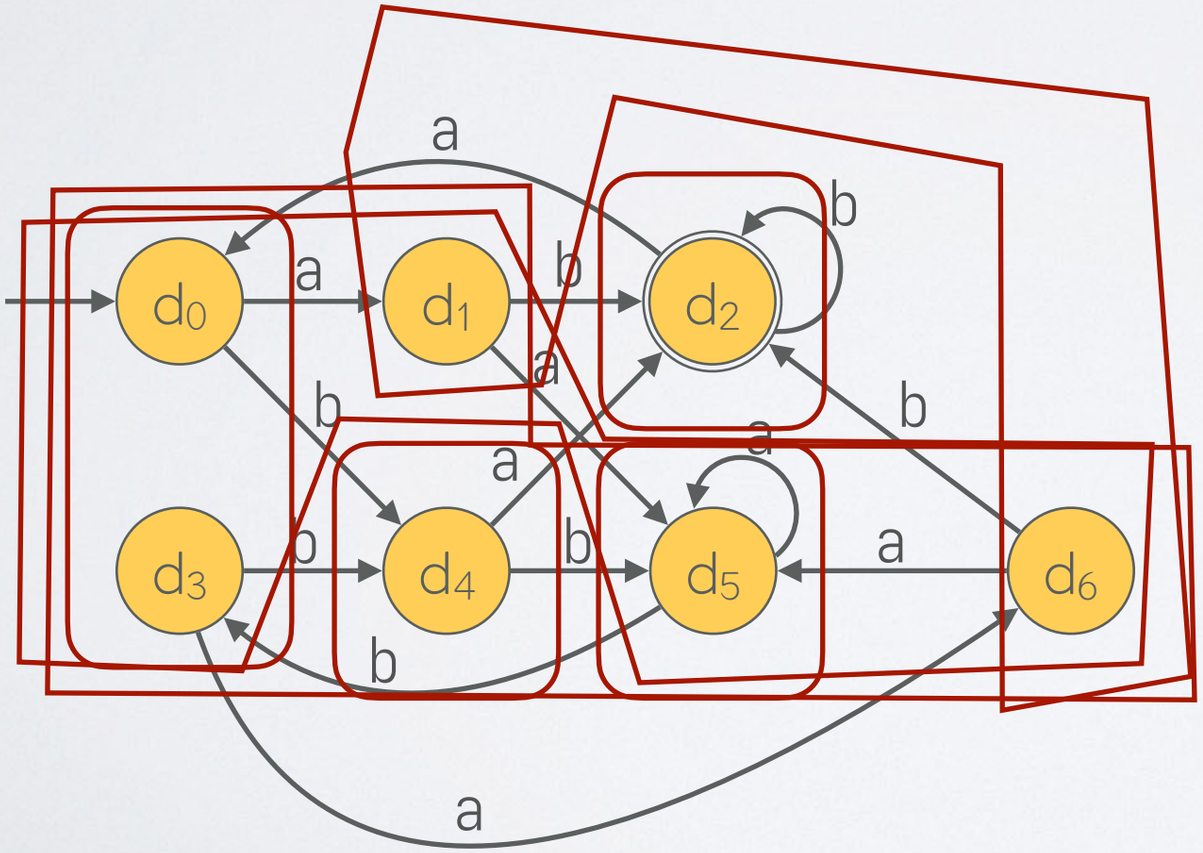
  (1) 把所有状态划分为两个组：接受状态组、非接受状态组

  (2) 任选符号 a，判断每个组中的状态对于 a 的转移，如果落入不同的组中，就把该组中的状态按照转移之后的组进行划分，使划分之后的每个组对于 a 的转移都会落入同一个组

  (3) 重复第(2)步，直到无法再划分

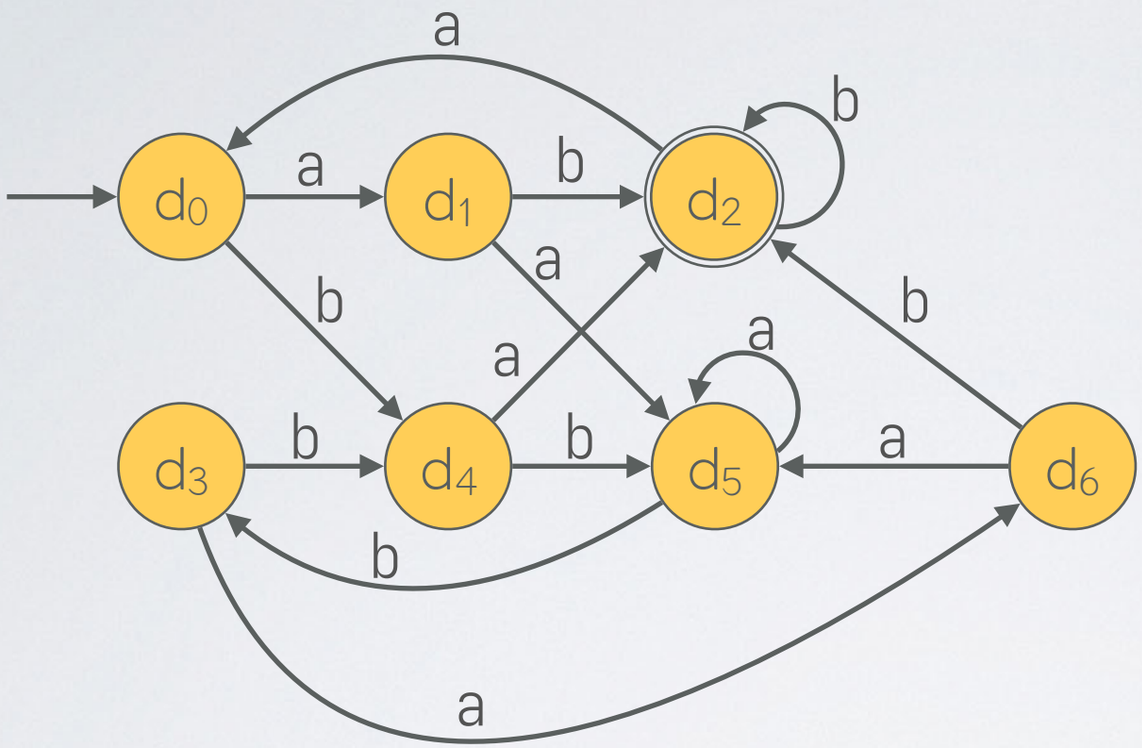
等价类划分示例





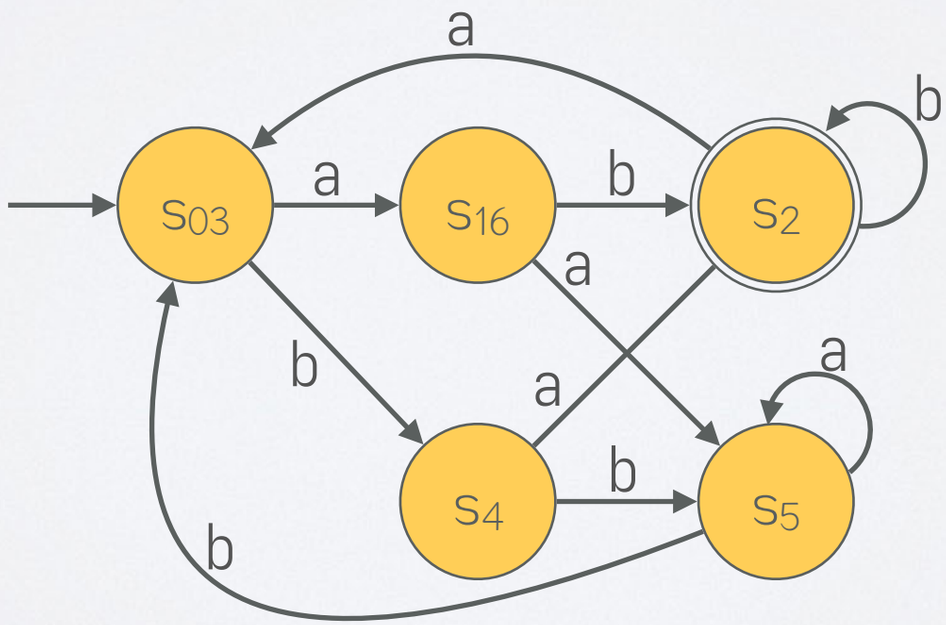
- 使用符号 a 来划分
  - d_4 会转移到 d_2
- 使用符号 b 来划分
  - d_0、d_3 会转移到 d_4
  - d_5 会转移到 d_3
  - d_1、d_6 会转移到 d_2

最小化 DFA 示例



等价类：

- $\{d_0, d_3\}$
- $\{d_5\}$
- $\{d_1, d_6\}$
- $\{d_4\}$
- $\{d_2\}$

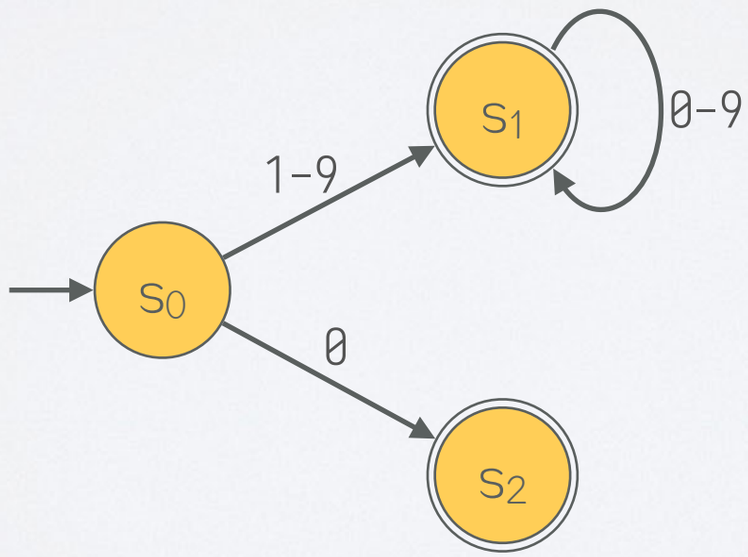


实现词法分析

- 构造一个识别所有需要 token 类别的 DFA D
  - 词法分析过程通常有限考虑**排在前面**的识别规则
  - 每个接受状态对应一个 token 类别
- 反复从源程序字符流中读取符号，直到将会转移到错误状态
  - 词法分析过程通常寻求**最长匹配**
  - 记录最近一次到达过的接受状态
- 处理并输出 token，重设 D 的状态为初始状态，继续读取符号
  - 根据接受状态判断 token 的类别

表驱动的词法分析


```
0|[1-9]([0-9])*
```



	0	[1-9]	其它
S_0	S_2	S_1	S_e
S_1	S_1	S_1	S_e
S_2	S_e	S_e	S_e

主要内容

- 词法分析的作用
- 词法分析的规约
- 词法分析的手动实现
- 词法分析的自动生成
- One More Thing

正则表达式转换为 DFA，但是求导

- 语言 L 对符号 a 的「导数」是另一个语言 $D_a(L)$
  - $D_a(L)$ 中的串加上前缀 a 后是 L 中的串
- 例子：
  - $D_{\mathsf{b}}\{\mathsf{foo}, \mathsf{bar}, \mathsf{baz}\} = \{\mathsf{ar}, \mathsf{az}\}$
  - $D_{\mathsf{f}}\{\mathsf{foo}, \mathsf{bar}, \mathsf{baz}\} = \{\mathsf{oo}\}$
  - $D_{\mathsf{a}}\{\mathsf{foo}, \mathsf{bar}, \mathsf{baz}\} = \varnothing$
- **定义：$D_a(L) = \{w \mid aw \in L\}$**
  - Brzozowski 提出“导数”[Brz64]
  - Antimirov 提出“偏导数”[Ant96]

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. J. ACM, 11(4): 481–494, 1964.

[Ant96] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. Theor. Comput. Sci., 155(2): 291–319, 1996.

对正则表达式求导

- 定理：正则语言的「导数」仍是正则语言
- 归纳证明：正则表达式「求导」后仍是正则表达式
  - $D_a(\epsilon) = \varnothing \qquad D_a(a) = \epsilon \qquad D_a(b) = \varnothing \ (a \neq b)$
  - $D_a(r|s) = D_a(r)|D_a(s)$
  - $D_a(rs) = D_a(r)s\,|\,emp(r)D_a(s)$
    - $emp(r)$ 判断 r 是否匹配空串，若是返回 ϵ，否则返回 $\varnothing$
  - $D_a(r^*) = D_a(r)r^*$

- 例子：$r = 01^*|1$

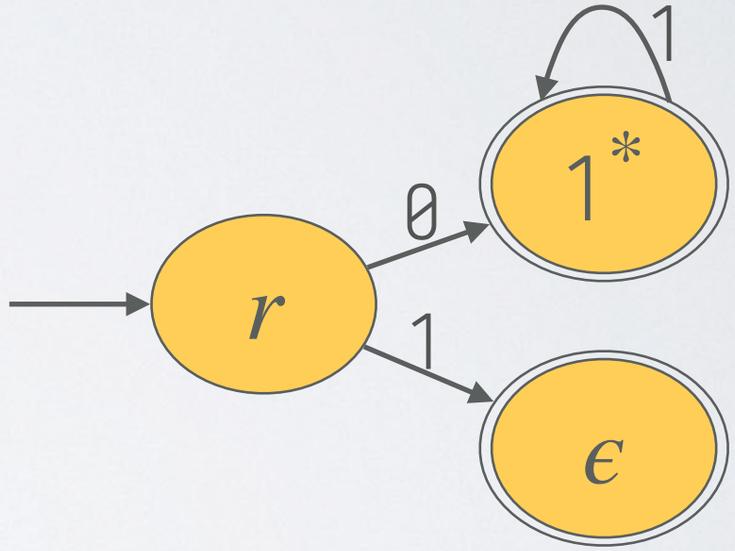
$$
\begin{aligned}
D_1(r) &= D_1(01^*|1) \\
&= D_1(01^*)|D_1(1) \\
&= D_1(0)1^*|emp(0)D_1(1^*)|\epsilon \\
&= \varnothing 1^*|\varnothing D_1(1^*)|\epsilon \\
&= \epsilon
\end{aligned}
$$

判断正则表达式是否匹配空串

- $emp(r)$ 判断 r 是否匹配空串，可以归纳定义：
  - $emp(\epsilon) = \epsilon \qquad emp(a) = \emptyset$
  - $emp(r|s) = emp(r)|emp(s)$
  - $emp(rs) = emp(r)emp(s)$
  - $emp(r^*) = \epsilon$
- 例子：
  - $emp(01^*|1) = emp(01^*)|emp(1) = emp(0)emp(1^*)|\emptyset = \emptyset\epsilon = \emptyset$
- **判断正则表达式 r 是否匹配串 w：依次将 r 对 w 中的符号「求导」，检查最终得到的语言是否包含空串**

正则表达式转换为 DFA，但是求导

- 以正则表达式为状态，状态转换通过「求导」进行
- 例子：$r = 01^* | 1$
  - $D_0(r) = 1^*$
  - $D_1(r) = \epsilon$
  - $D_0(1^*) = D_0(1)1^* = \varnothing$
  - $D_1(1^*) = D_1(1)1^* = 1^*$
  - $D_0(\epsilon) = \varnothing, D_1(\epsilon) = \varnothing$
  - 共有 r、1^*、ϵ、$\varnothing$ 四种状态
  - 使用 emp 判断哪些状态是终止状态



本讲小结

- 词法分析的规约给出每类词法单元的定义
  - 通常使用正则表达式来描述
- 词法分析可以通过有限自动机进行实现
  - 确定性有限自动机(DFA)
  - 基于 DFA 的词法单元识别
- 词法分析的自动生成
  - 非确定性有限自动机(NFA)
  - 正则表达式转换为 NFA，NFA 转换为 DFA，DFA 的最小化
  - 表驱动的词法分析

思考问题

- 为什么编译过程需要词法分析?
- 词法分析需要知道编程语言的哪些信息?
- 词法分析为什么引入词法单元类别(而非直接记录词素)?
- 词法分析可以不忽略空白字符或注释吗?
- 为什么词法分析不直接基于 NFA 来实现?
- 每个关键字/保留字(keyword)可以单独对应一个词法单元类别,这样做有什么优点和缺点?