



第二讲 词法分析（基础版）

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

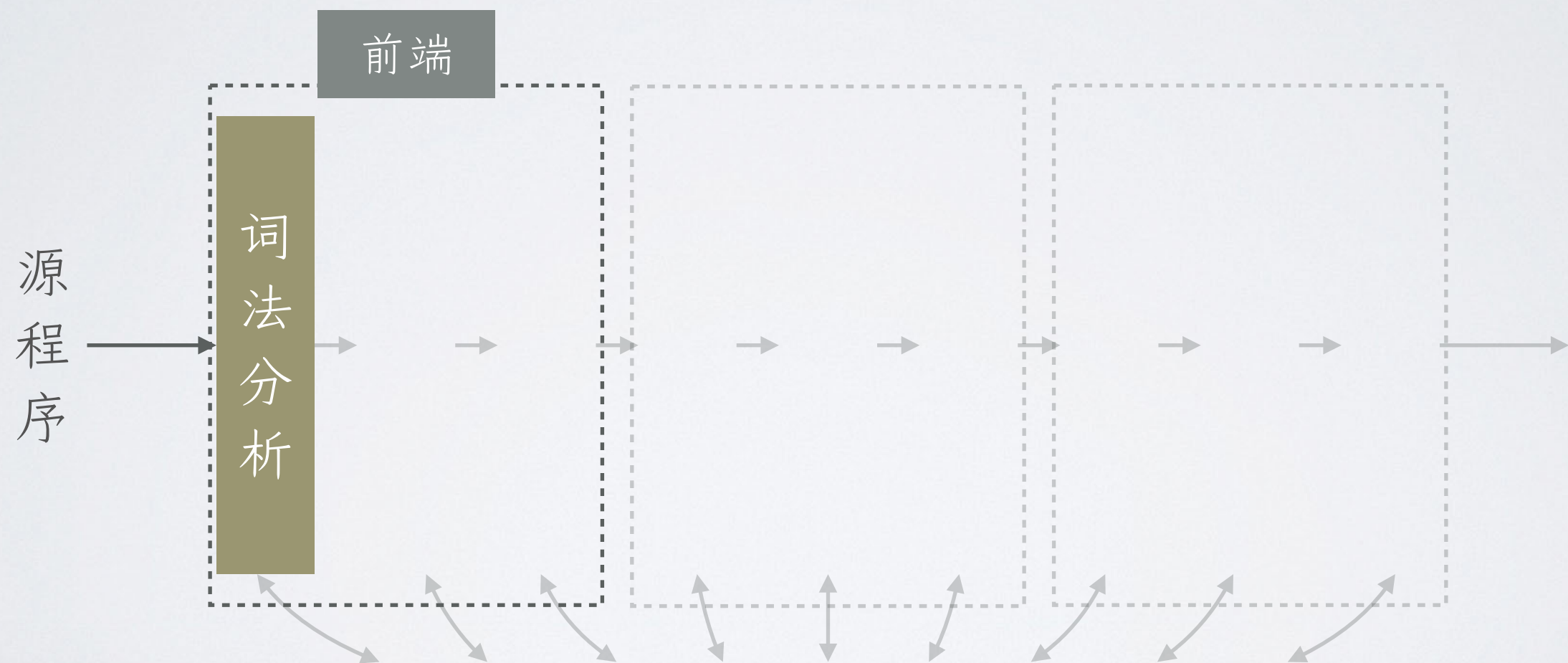
- ◎ 对应龙书章节：第 3 章

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
 - ❖

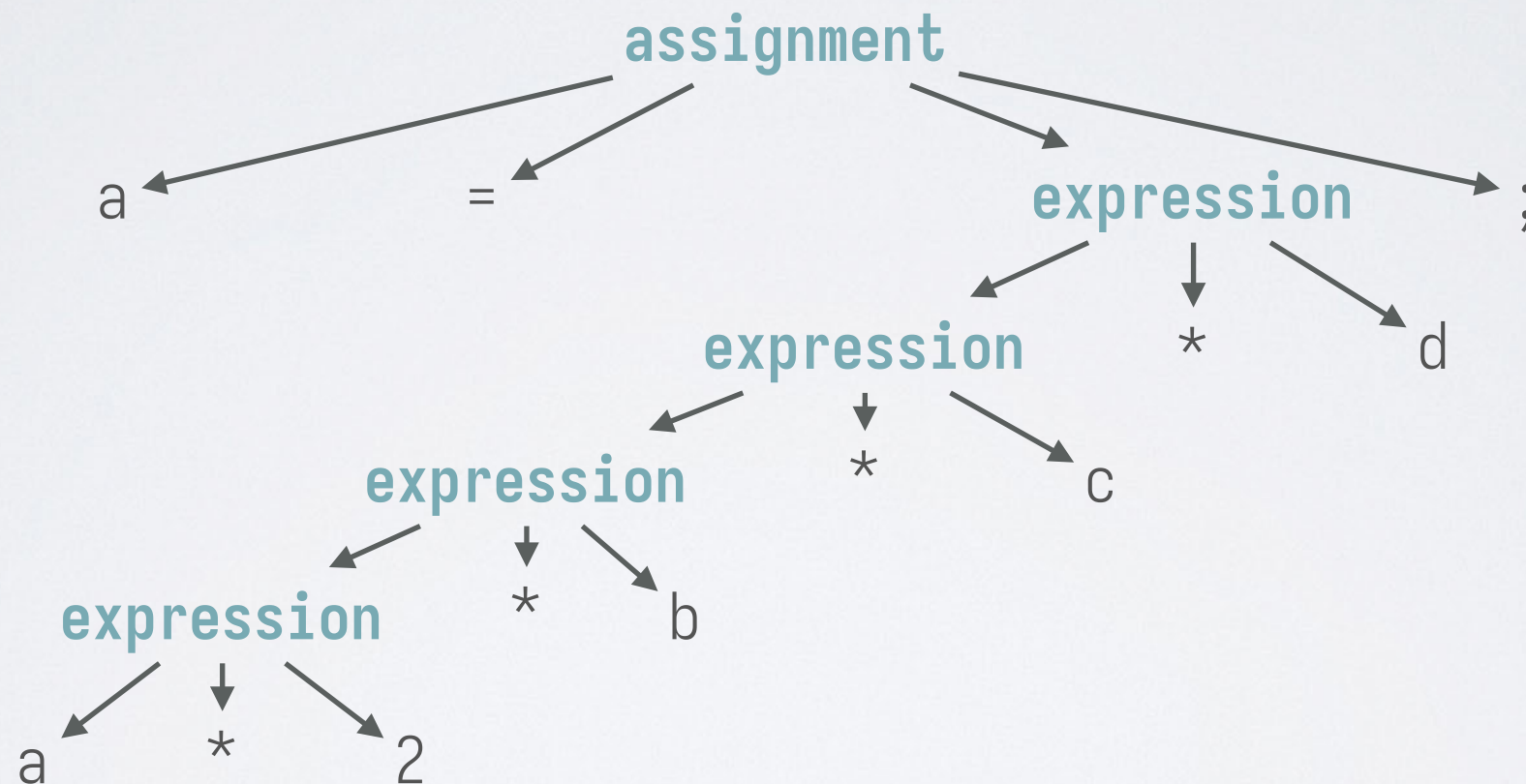
常用的 token 类别和属性

- **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?

- 为什么不能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」

NUMBER

单元

256

ID

单元

a5b

◎ 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

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: $\Sigma = \{A-Z, a-z, 0-9, +, -, *, /, <, =, >, :, ,, ;, ., ^, (,), \{, \}, [,]\}$

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

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: $x = ba, y = nana, xy = banana$
- ◎ **方幂** (exponentiation): 设 x 是符号串, n 是自然数, x^n 定义为 n 份 x 前后连接起来
 - ❖ 例: $x = ba, x^0 = \epsilon, x^3 = bababa$

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

- ❖ $a \mid b$ 能匹配 a, b
- ❖ $(a \mid b)(a \mid b)$ 能匹配 aa, ab, ba, bb
- ❖ a^* 能匹配 $\epsilon, a, aa, aaa, aaaa, \dots$
- ❖ $(a \mid b)^*$ 或 $(a^*b^*)^*$ 能匹配 $\epsilon, a, b, aa, ab, ba, bb, aaa, \dots$
- ❖ a^*b 能匹配 $b, ab, aab, aaab, \dots$

◎ C 语言的标识符:

$(A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \mid _)(A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \mid _ \mid 0 \mid 1 \mid \dots \mid 9)^*$

◎ 有符号整数:

$(+ \mid - \mid \epsilon)(0 \mid 1 \mid \dots \mid 9)(0 \mid 1 \mid \dots \mid 9)^*$

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 $\emptyset$
 - ❖ 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: $L(r|s) = L(r) \cup L(s)$

(b) 连接: $L(rs) = \{xy \mid x \in L(r), y \in L(s)\}$

(c) 闭包: $L(r^*) = \{x_1x_2\cdots x_n \mid n \geq 0, x_1, x_2, \cdots, x_n \in L(r)\}$

◎ 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 $a | b | c | \dots | z$
- ◎ 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	<code>if</code>
WHILE	<code>while</code>
ID	<code>[A-Za-z_]([A-Za-z_0-9])*</code>
NUMBER	<code>[+-]?([0-9])⁺</code>
LPAREN	<code>(</code>
NEQ	<code>!=</code>

- 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	<code>while</code>
ID	<code>[A-Za-z_][A-Za-z_0-9]*</code>
LPAREN	<code>(</code>
NOT	<code>!</code>
NEQ	<code>!=</code>

<**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

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	<code>i f</code>

```

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
}

```

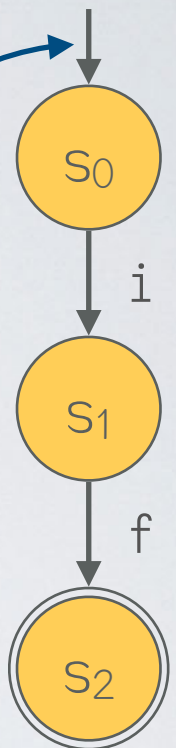


状态转移图

● 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

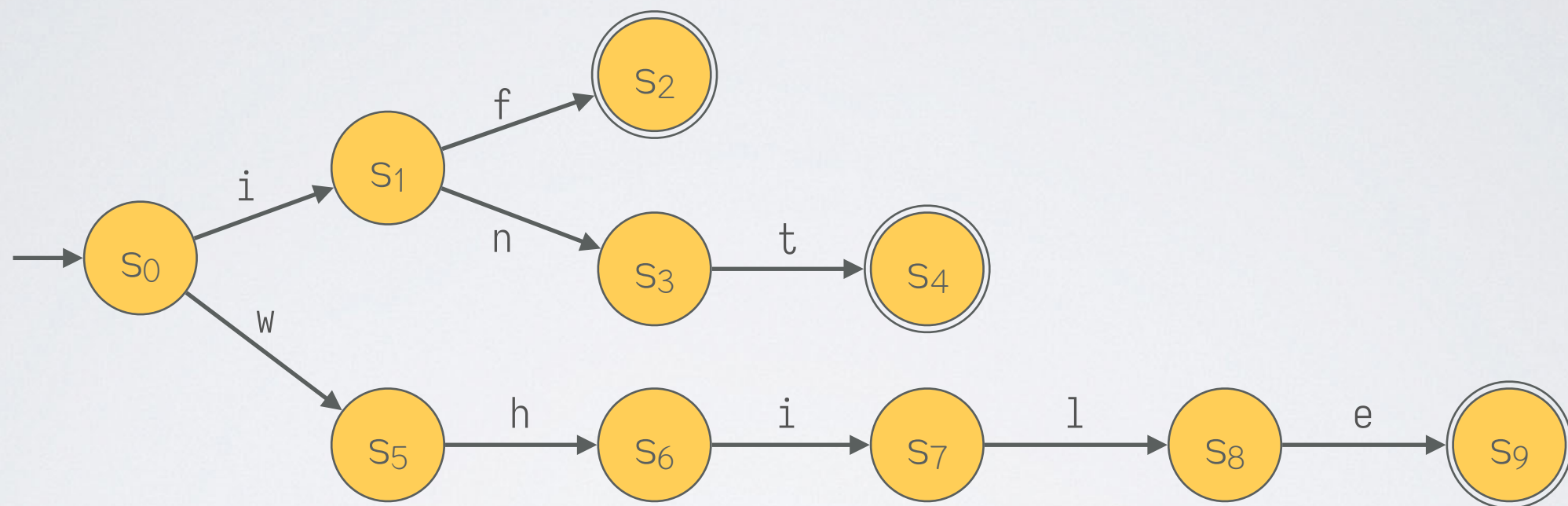
- ❖ 状态可看作已识别部分的总结
- ❖ 有一个特殊的**初始状态**，用一条**初始边**表示
- ❖ 某些状态为**接受状态**，表明识别成功，用**双层圆圈**表示



● 边 (edge): 表示识别了若干符号后，状态发生转移

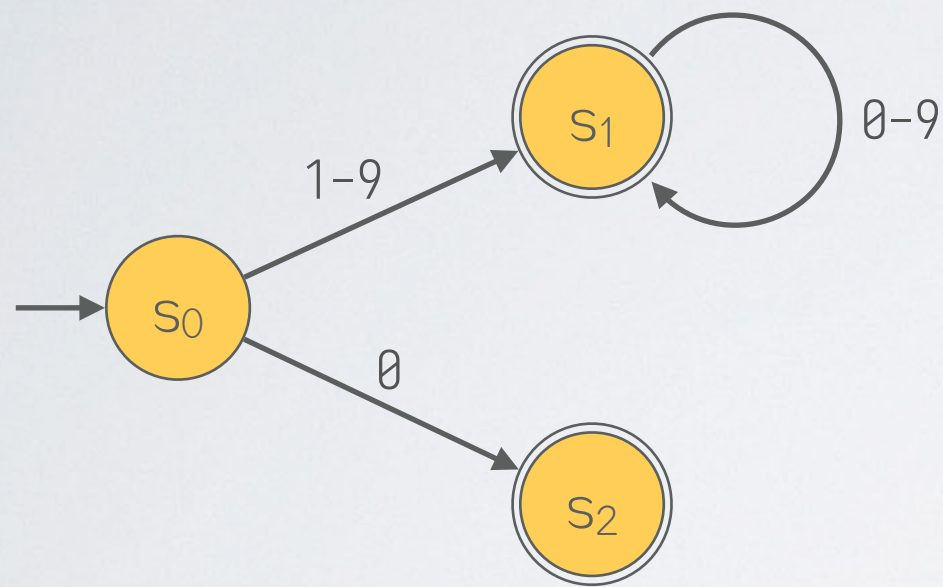
- ❖ 边的标号表示转移状态时识别的符号
- ❖ 没有画出的边 (比如从 s_1 出发标号为 a) 默认表示转移到错误状态

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**

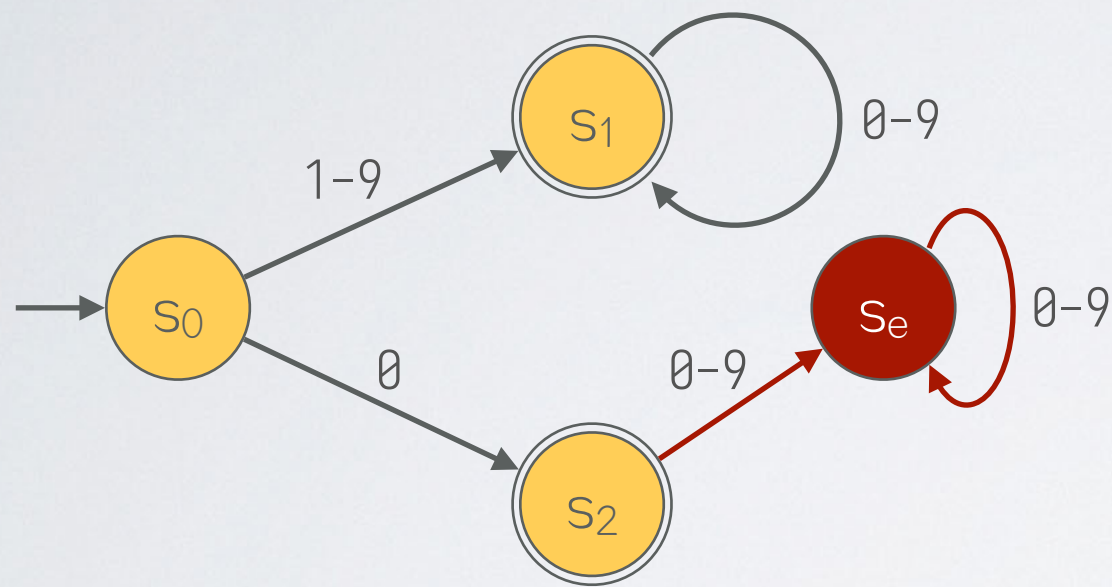
- ❖ δ 是一个从 $S \times \Sigma$ 到 S 的映射, 称为**转移函数**

- ❖ $\delta(s, a) = s'$ 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)

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

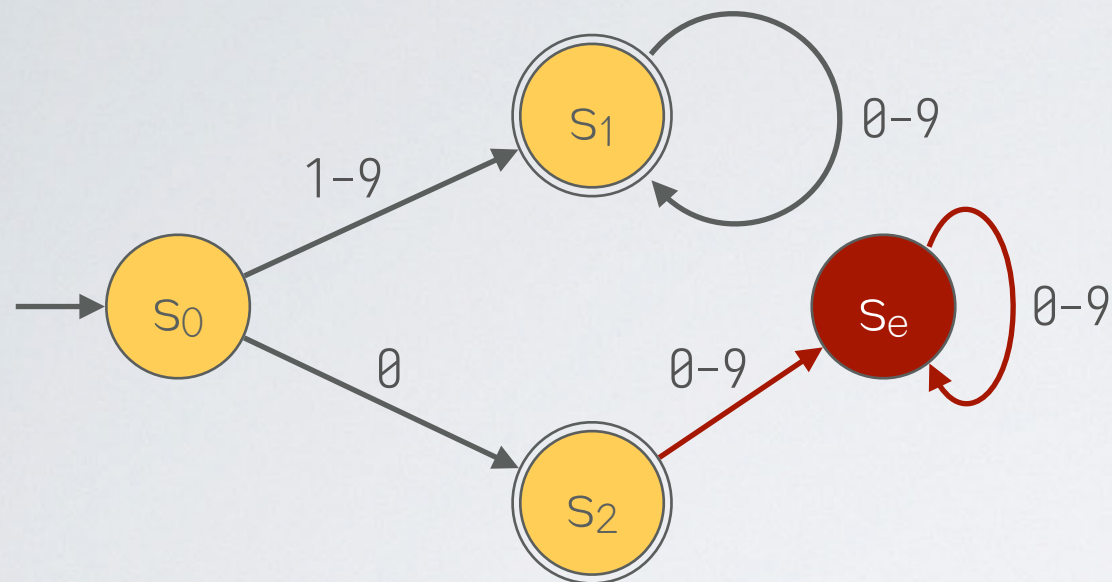
$$S = \{s_0, s_1, s_2, s_e\}$$

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

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

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

实现基于 DFA 的识别



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

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

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

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

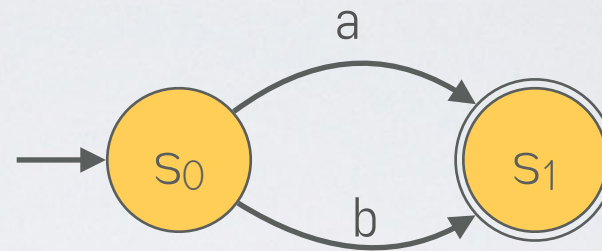
```

c = next_char();
s = s0;
while (c != EOF && s != se) {
    s = δ(s, c);
    c = next_char();
}
if (s ∈ F) {
    // 识别成功
} else {
    // 识别失败
}
  
```

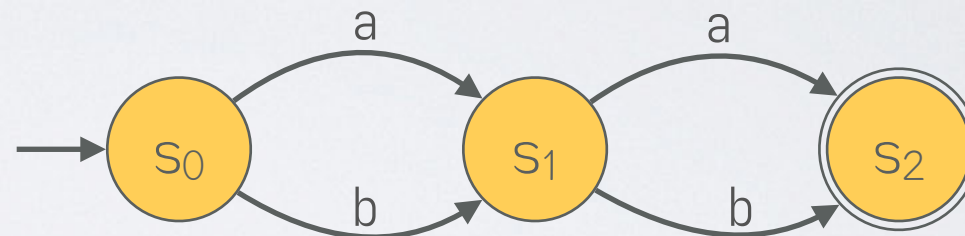
正则表达式转换为 DFA

● 考虑 $\Sigma = \{a, b\}$

❖ $a \mid b$



❖ $(a \mid b)(a \mid b)$



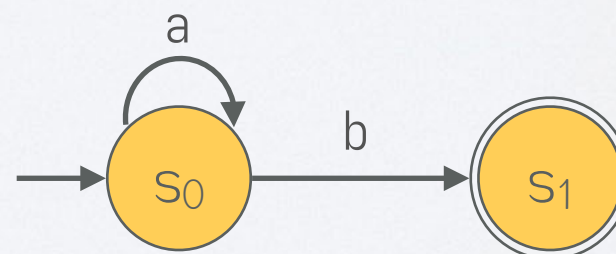
❖ a^*



❖ $(a \mid b)^*$

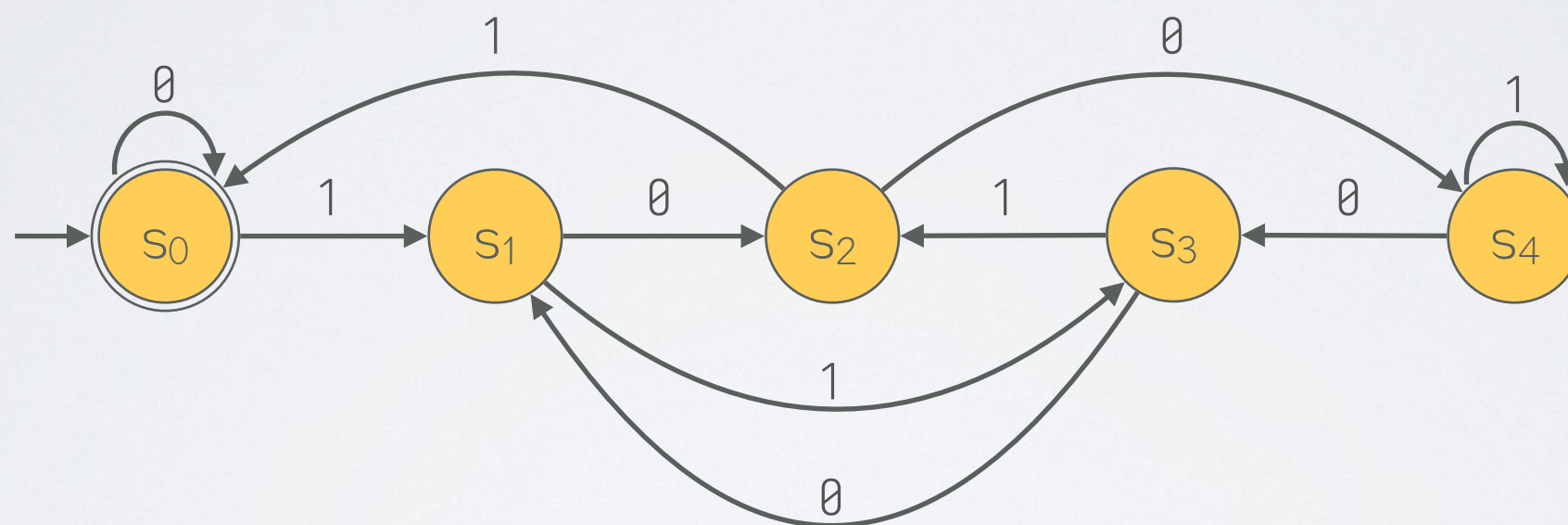


❖ a^*b



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, 怎么转换为 DFA?**

正则表达式转换为 DFA

◎ 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:

❖ $D_b\{\text{foo}, \text{bar}, \text{baz}\} = \{\text{ar}, \text{az}\}$

❖ $D_f\{\text{foo}, \text{bar}, \text{baz}\} = \{\text{oo}\}$

❖ $D_a\{\text{foo}, \text{bar}, \text{baz}\} = \emptyset$

◎ 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

$$\diamond D_a(\epsilon) = \emptyset \quad D_a(a) = \epsilon \quad D_a(b) = \emptyset \ (a \neq b)$$

$$\diamond D_a(r | s) = D_a(r) | D_a(s)$$

$$\diamond D_a(rs) = D_a(r)s | \text{emp}(r)D_a(s)$$

 ◇ $\text{emp}(r)$ 判断 r 是否匹配空串, 若是返回 ϵ , 否则返回 $\emptyset$

$$\diamond D_a(r^*) = D_a(r)r^*$$

◎ 例子: $r = \emptyset 1^* | 1$

$$\begin{aligned} D_1(r) &= D_1(\emptyset 1^* | 1) \\ &= D_1(\emptyset 1^*) | D_1(1) \\ &= D_1(\emptyset) 1^* | \text{emp}(\emptyset) D_1(1^*) | \epsilon \\ &= \emptyset 1^* | \emptyset D_1(1^*) | \epsilon \\ &= \epsilon \end{aligned}$$

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:

❖ $emp(\epsilon) = \epsilon$ $emp(a) = \emptyset$

❖ $emp(r | s) = emp(r) | emp(s)$

❖ $emp(rs) = emp(r)emp(s)$

❖ $emp(r^*) = \epsilon$

◎ 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

正则表达式通过求导转换为 DFA

◎ 以正则表达式为状态，状态转换通过「求导」进行

◎ 例子: $r = 01^* | 1$

❖ $D_0(r) = 1^*$

❖ $D_1(r) = \epsilon$

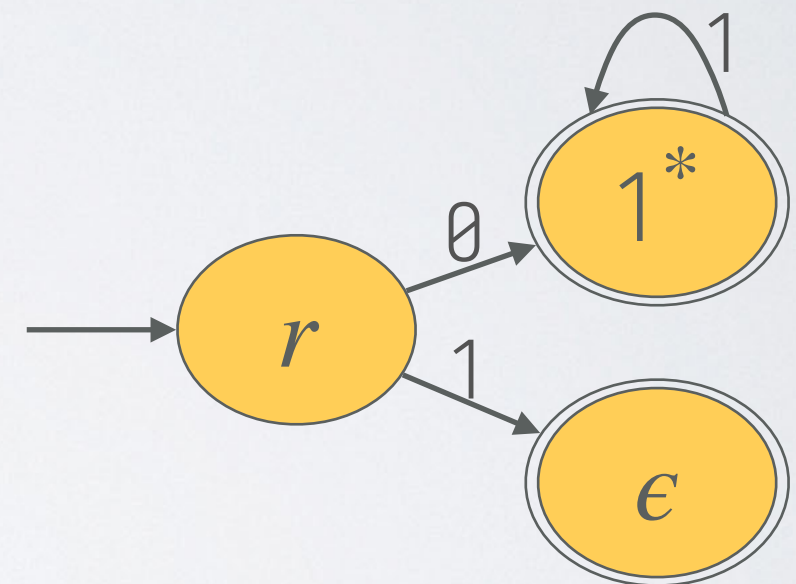
❖ $D_0(1^*) = D_0(1)1^* = \emptyset$

❖ $D_1(1^*) = D_1(1)1^* = 1^*$

❖ $D_0(\epsilon) = \emptyset, D_1(\epsilon) = \emptyset$

❖ 共有 r 、 1^* 、 ϵ 、 $\emptyset$ 四种状态

❖ 使用 *emp* 判断哪些状态是终止状态



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

- ◎ 实践演示

玩具语言

- ◎ C 语言, 但只有主函数, 只有 int 类型, 只有直线代码

```
int main() {  
    int x = 10;  
    int y = x + 1 * x;  
    int z = x * (y + 1);  
    return z % 66;  
}
```

```
int main() {  
    int y;  
    y = 1;  
    int main;  
    main = y + 1;  
    return main;  
}
```

```
int main() {  
    int a = 1;  
    {  
        a = a + 2;  
        int a = 3;  
        a = a + 4;  
    }  
    a = a + 5;  
    return a;  
}
```

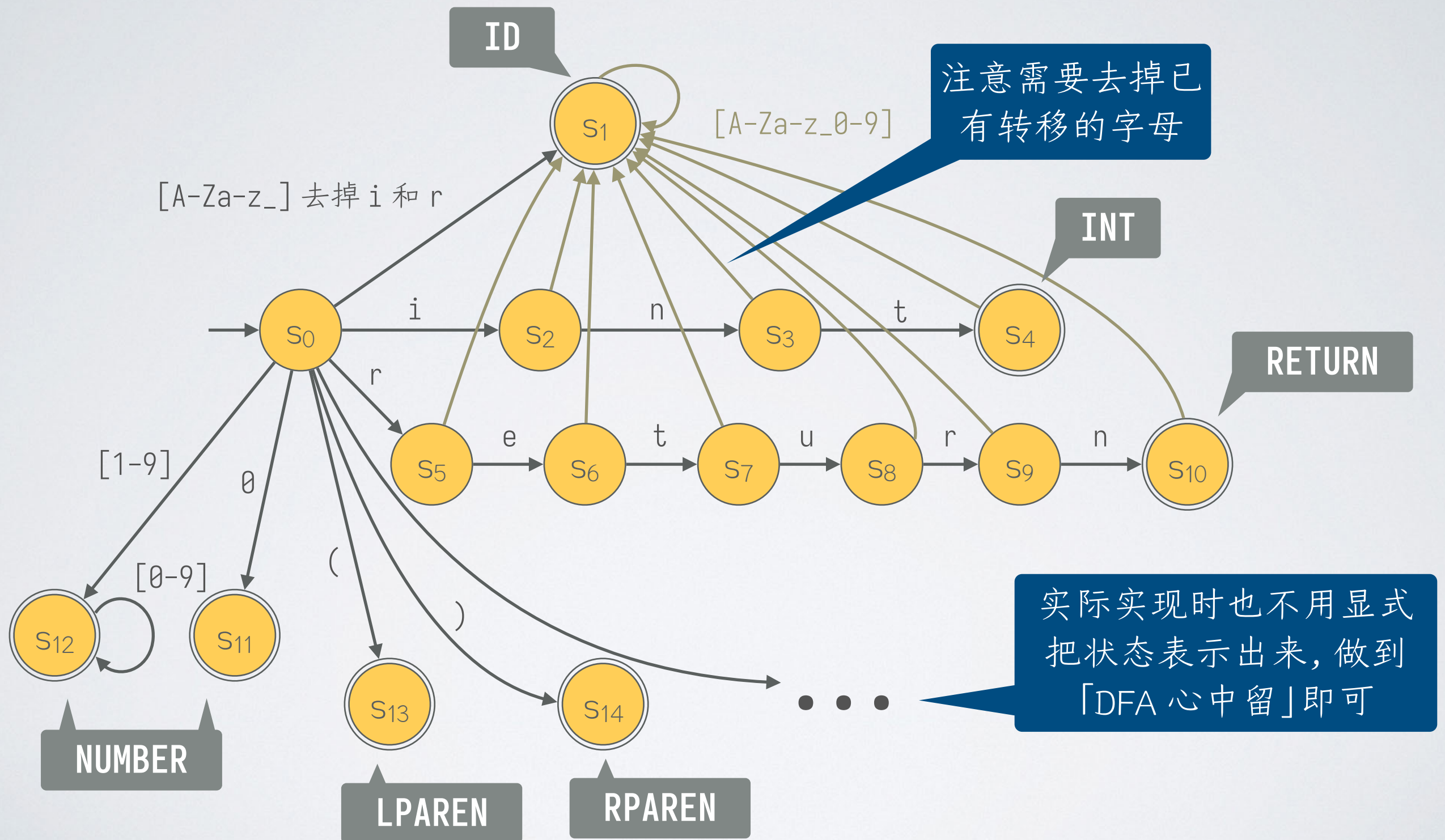
```
int main() {  
    int a = 1;  
    int b = a + 2;  
    {  
        b = a + b;  
        {  
            int a = 3;  
            b = b + a;  
            { int b = 1; }  
            { return b; }  
        }  
        int b = 1;  
    }  
}
```

玩具语言的词法规约

token 类别	正则表达式
INT	<code>int</code>
RETURN	<code>return</code>
ID	<code>[A-Za-z_]([A-Za-z_0-9])*</code>
NUMBER	<code>0 [1-9]([0-9])*</code>
LPAREN/ RPAREN	<code>()</code>
LCURLY/ RCURLY	<code>{ }</code>
EQ	<code>=</code>
SEMI	<code>;</code>
PLUS	<code>+</code>
MINUS	<code>-</code>
STAR	<code>*</code>
SLASH	<code>/</code>
PERCENT	<code>%</code>

- 把所有的正则表达式通过 **选择** 操作 | 合并为一个
- 为该正则表达式构造 DFA, 在接受状态上标记对应的 token 类别
- 在构造 DFA 时, 尽可能合并 **等价** 的转移, 比如 **ID** 中可以接受的字符

按照词法规约构造 DFA



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
 - ❖ 正则表达式转换为 DFA
- ◎ 后续可能讲的专题:
 - ❖ 词法分析的自动生成
 - ❖ 正则表达式的若干扩展

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)？
- ◎ 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？
- ◎ 词法分析可以不忽略空白字符或注释吗？
- ◎ 有没有什么语言的词法分析的规约是不能用正则表达式表达的？它们的词法分析应该如何进行？