

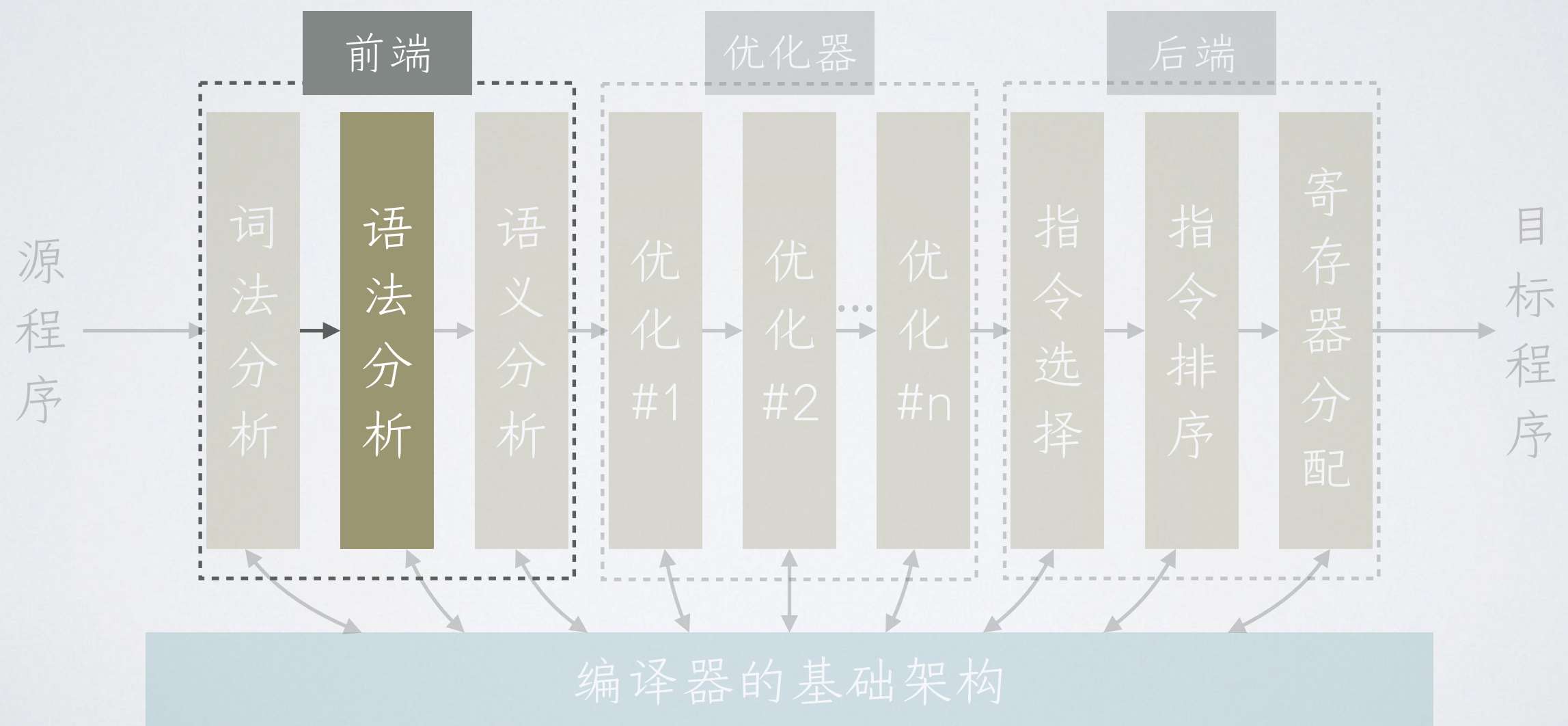


第十一讲 语法分析（进阶版）

Advanced Topics in Syntax Analysis

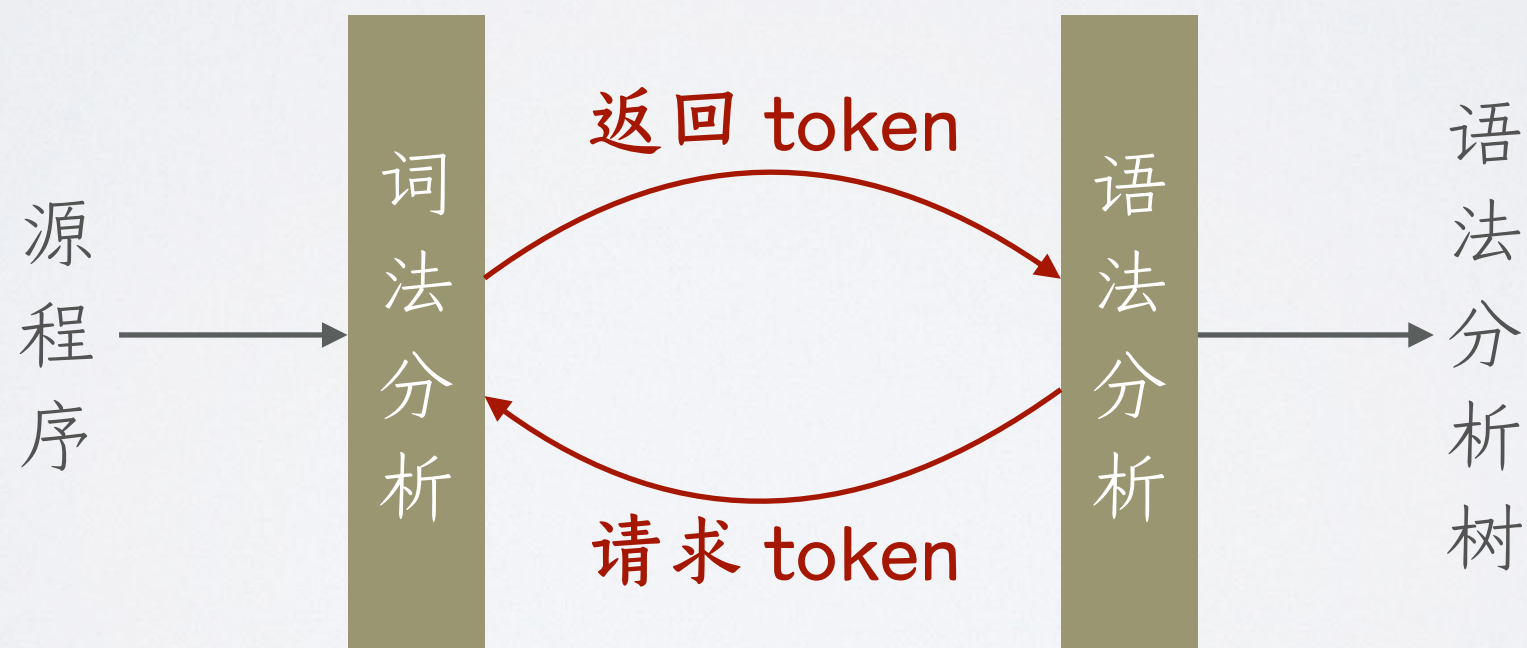
回顾：语法分析的作用

- 处理词法单元(token)序列, 构造语法分析树
- 把「单词」组合成「句子」

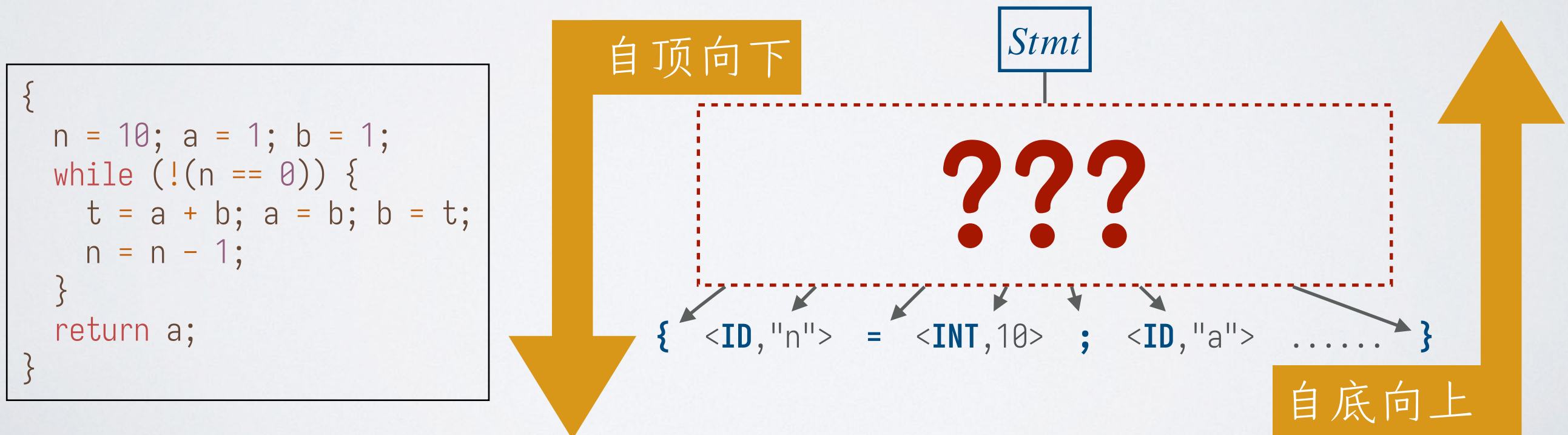


回顾：语法分析的工作

- 识别**语法变量**，进行**语法检查**，构造**语法分析树**
- 输入为词法分析得出的 token 序列
- 如果语法有错，尽可能详细地报告语法错误



回顾：语法分析的实现



回顾：不能用 LL 文法表达的语言

- **LL 文法的短板：**通过有限个「向前看」符号选择推导规则

- 例：右侧 CFG 表示了语言 $\{a^i b^j \mid i \geq j\}$

- 该文法不是 LL(1) 的

- ❖ [1] 和 [2] 两条规则右侧的 FIRST 集合有交集

[1]	$S \rightarrow a S$
[2]	$S \rightarrow P$
[3]	$P \rightarrow a P b$
[4]	$P \rightarrow \epsilon$

- 不存在一个 LL 文法可以表示该语言

- ❖ 例如允许 k 个「向前看」, 那么 $a^k b^k$ 和 $a^{k+1} b^k$ 都是以 k 个 a 开头, 但是前者应该选择规则 [2] 展开, 而后者应该选择规则 [1] 展开

- 直到看到 token 流中的 b 之后才能决定应该如何选择规则

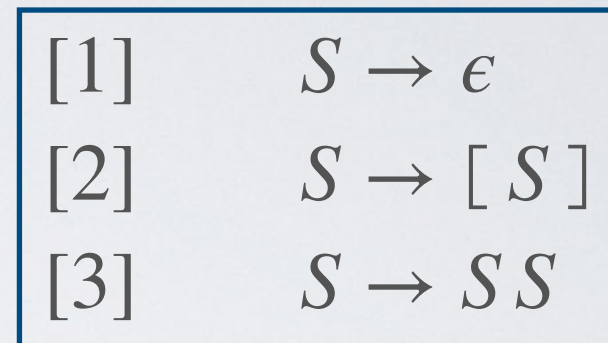


主要内容

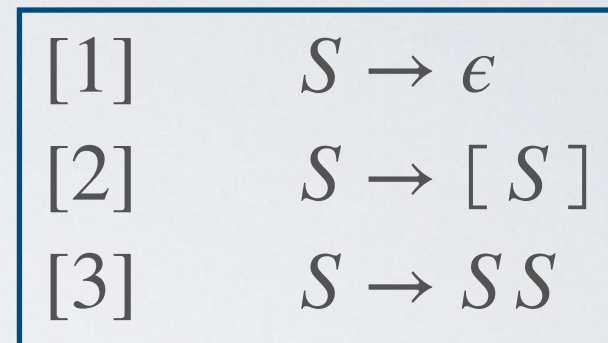
- ◎ 自底向上的语法分析
- ◎ 语法分析的自动生成
- ◎ 自顶向下的语法分析(进阶版)

主要内容

- ◎ 自底向上的语法分析
- ◎ 语法分析的自动生成
- ◎ 自顶向下的语法分析(进阶版)



- 用 $w : \beta$ 表示存在归约关系 $\beta \Leftarrow^* w$, 其中 w 表示 token 流, β 表示分析目标
- 从**左往右**读取输入的 token, 同时加到 w 和 β 后面
- 如果 **β 最右边**出现了可以被归约的「模式」, 则归约为对应的非终结符号


$$\begin{array}{lcl}
S & \Leftarrow & [S] \\
& \Leftarrow & [SS] \\
& \Leftarrow & [S[S]] \\
& \Leftarrow & [S[]] \\
& \Leftarrow & [[S][]] \\
& \Leftarrow & [[][]]
\end{array}$$

基于移进-归约实现自底向上分析



类比前面的自顶向下分析, 使用无二义性文法

$S' \rightarrow S \text{ EOF}$

[1] $S \rightarrow \epsilon$

[2] $S \rightarrow [S] S$



◎ 移进 (shift)

❖ 把一个 token 压入分析栈

◎ 归约 (reduce)

❖ 把分析栈顶部的若干符号归约为非终结符号

预测分析

● 问题：如何确定应该移进还是归约？

● 一种简单的策略：

- ❖ 当分析栈顶部有能够归约的「模式」时，则归约
- ❖ 否则，则移进
- ❖ 但是，像 [1] 这种会导致**冲突**

● 根据以下信息进行预测分析

- ❖ 分析栈的顶部状态
- ❖ 待处理的向前看符号

$$S' \rightarrow S \text{ EOF}$$

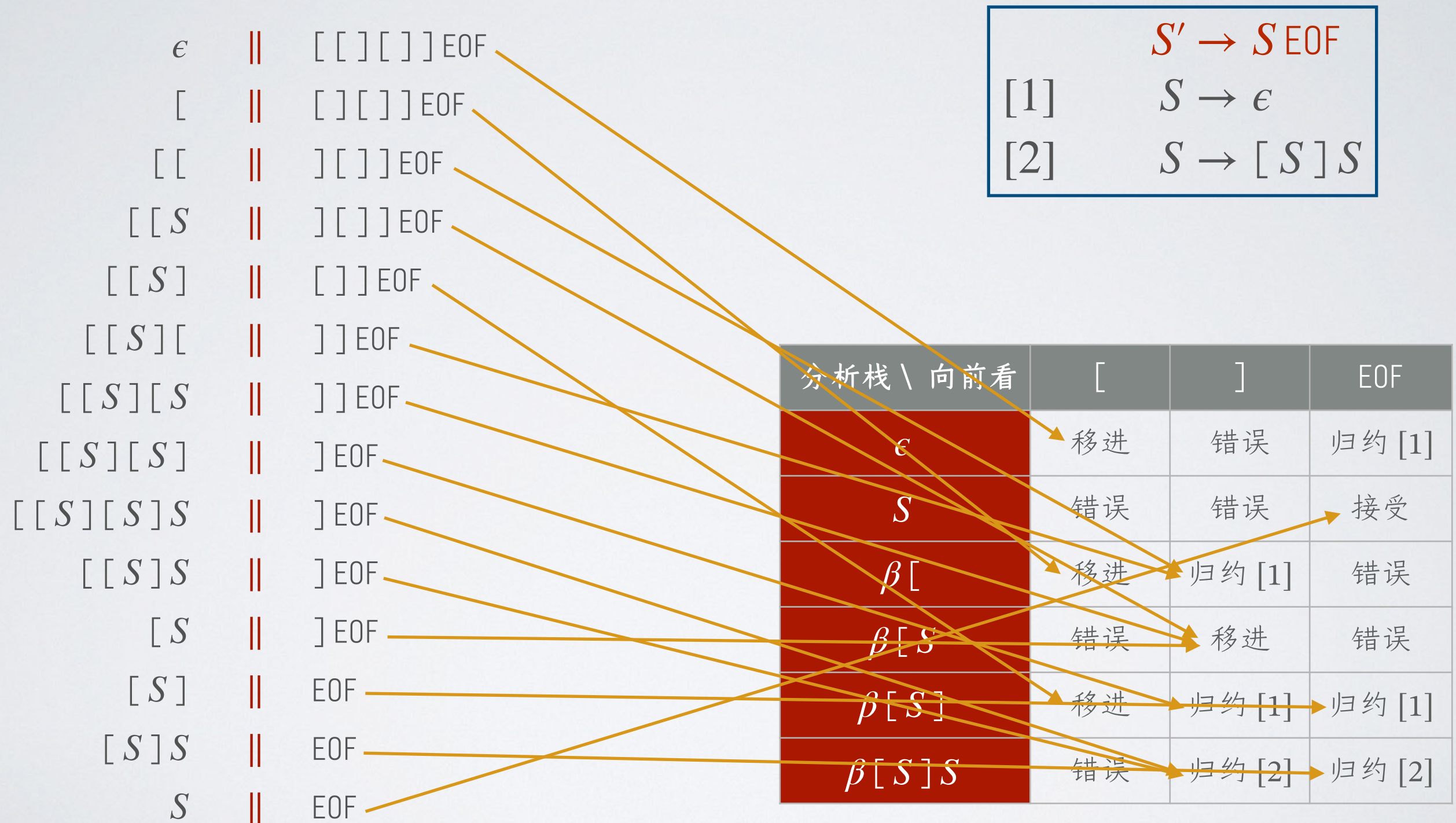
$$[1] \quad S \rightarrow \epsilon$$

$$[2] \quad S \rightarrow [S]S$$

$$\beta \parallel w$$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

预测分析



实现预测分析

```

top = 0;
token = next_token();
while (true) {
    if (top == 0) {
        switch (token) {
            case LSQUARE:
                top++; stack[top] = LSQUARE;
                token = next_token(); break;
            case RSQUARE:
                return false;
            case EOF:
                top++; stack[top] = S; break;
        }
    } else if (top == 1 && stack[top] == S) {
        switch (token) {
            case LSQUARE: return false;
            case RSQUARE: return false;
            case EOF: return true;
        }
    } else if .....
}

```

$$S' \rightarrow S \text{ EOF}$$

[1] $S \rightarrow \epsilon$

[2] $S \rightarrow [S]S$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

实现预测分析

```

.....
} else if (top >= 1 && stack[top] == LSQUARE) {
    switch (token) {
        case LSQUARE:
            top++; stack[top] = LSQUARE;
            token = next_token(); break;
        case RSQUARE:
            top++; stack[top] = S; break;
        case EOF:
            return false;
    }
} else if (top >= 2 && stack[top] == S &&
           stack[top - 1] == LSQUARE) {
    switch (token) {
        case LSQUARE: return false;
        case RSQUARE:
            top++; stack[top] = RSQUARE;
            token = next_token(); break;
        case EOF: return false;
    }
} else if .....

```

$$S' \rightarrow S \text{ EOF}$$

[1] $S \rightarrow \epsilon$

[2] $S \rightarrow [S]S$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

实现预测分析

$$S' \rightarrow S \text{ EOF}$$

[1] $S \rightarrow \epsilon$

[2] $S \rightarrow [S]S$

```

.....
} else if (top >= 4 && stack[top] == S &&
           stack[top - 1] == RSQUARE &&
           stack[top - 2] == S &&
           stack[top - 3] == LSQUARE) {
    switch (token) {
        case LSQUARE:
            return false;
        case RSQUARE:
            top = top - 3; stack[top] = S;
            break;
        case EOF:
            top = top - 3; stack[top] = S;
            break;
    }
} else .....

```

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

构造移进-归约分析表

首先，找出可能的分析栈「模式」

- ❖ 空栈 ϵ 和只包含初始符号的栈 S
- ❖ 对每个产生规则 $A \rightarrow \alpha \gamma$ 和非空的 α ，把 $\beta \alpha$ 作为一种「模式」，即将来可能在栈顶发现产生规则的右端 $\alpha \gamma$ 并归约到 A

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow [S]$
[2]	$S \rightarrow a$

然后，确定每一种情况的动作

- ❖ ϵ : $\text{FIRST}(S)$ 中的符号可以移进
- ❖ S : EOF 符号可以接受
- ❖ $\beta \alpha$ (对应规则 $A \rightarrow \alpha \gamma$):
 - ❖ $\text{FIRST}(\gamma)$ 中的符号可以移进
 - ❖ 若 $\gamma = \epsilon$ ，则可以归约

分析栈 \ 向前看	[	]	a	EOF
ϵ	移进		移进	
S				接受
$\beta [$	移进		移进	
$\beta [S$		移进		
$\beta [S]$	归约 [1]	归约 [1]	归约 [1]	归约 [1]
βa	归约 [2]	归约 [2]	归约 [2]	归约 [2]

问题：移进-归约冲突

- ◎ $S \rightarrow \epsilon$ 这种规则可以在任意位置归约
- ◎ 简单解法：用 FOLLOW 集合判断是否归约
 - ❖ 分析栈「模式」为 $\beta\alpha$ (对应规则 $A \rightarrow \alpha$) 时, 如果向前看符号在 $\text{FOLLOW}(A)$ 中, 且归约后得到的分析栈具有某种已知的「模式」, 则可以归约

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow [S]S$

$$\text{FOLLOW}(S) = \{\text{EOF},]\}$$

分析栈 \ 向前看	[	]	EOF
ϵ	移进/归约	归约 [1]	归约 [1]
S			接受
$\beta[$	移进	归约 [1]	归约 [1]
$\beta[S$		移进	
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$		归约 [2]	归约 [2]

问题：移进-归约冲突

- 用 FOLLOW 集合比较粗糙，可能无法排除冲突

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow a b d$
[2]	$S \rightarrow a A c$
[3]	$S \rightarrow b A d$
[4]	$A \rightarrow b$

$$\text{FOLLOW}(A) = \{c, d\}$$

分析栈 \ 向前看	a	b	c	d	EOF
ϵ	移进	移进			
S					接受
$\beta a b$			归约 [4]	移进/归约	

- 如果还是只使用一个向前看符号，**是否还有信息可挖掘？**
 - ❖ 如果在分析栈为 $\beta a b$ 且向前看为 d 时，归约 [4]，分析栈变成 $\beta a A$ ，此时只有 [2] 符合且只允许向前看为 c
 - ❖ 需要对分析栈的「模式」进行更细致的分类

分析栈模式的分类

◎ 前面我们是这么得出可能的分析栈的「模式」的：

- ❖ 空栈 ϵ 和只包含初始符号的栈 S
- ❖ 对每个产生规则 $A \rightarrow \alpha \gamma$ 和非空的 α , 把 $\beta \alpha$ 作为一种模式, 即将来可能在栈顶发现产生规则的右端 $\alpha \gamma$ 并归约到 A

◎ 每种「模式」对应了一个「部分分析状态」：

圆点的左边表示
分析栈顶内容

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow [S]$
[2]	$S \rightarrow a$

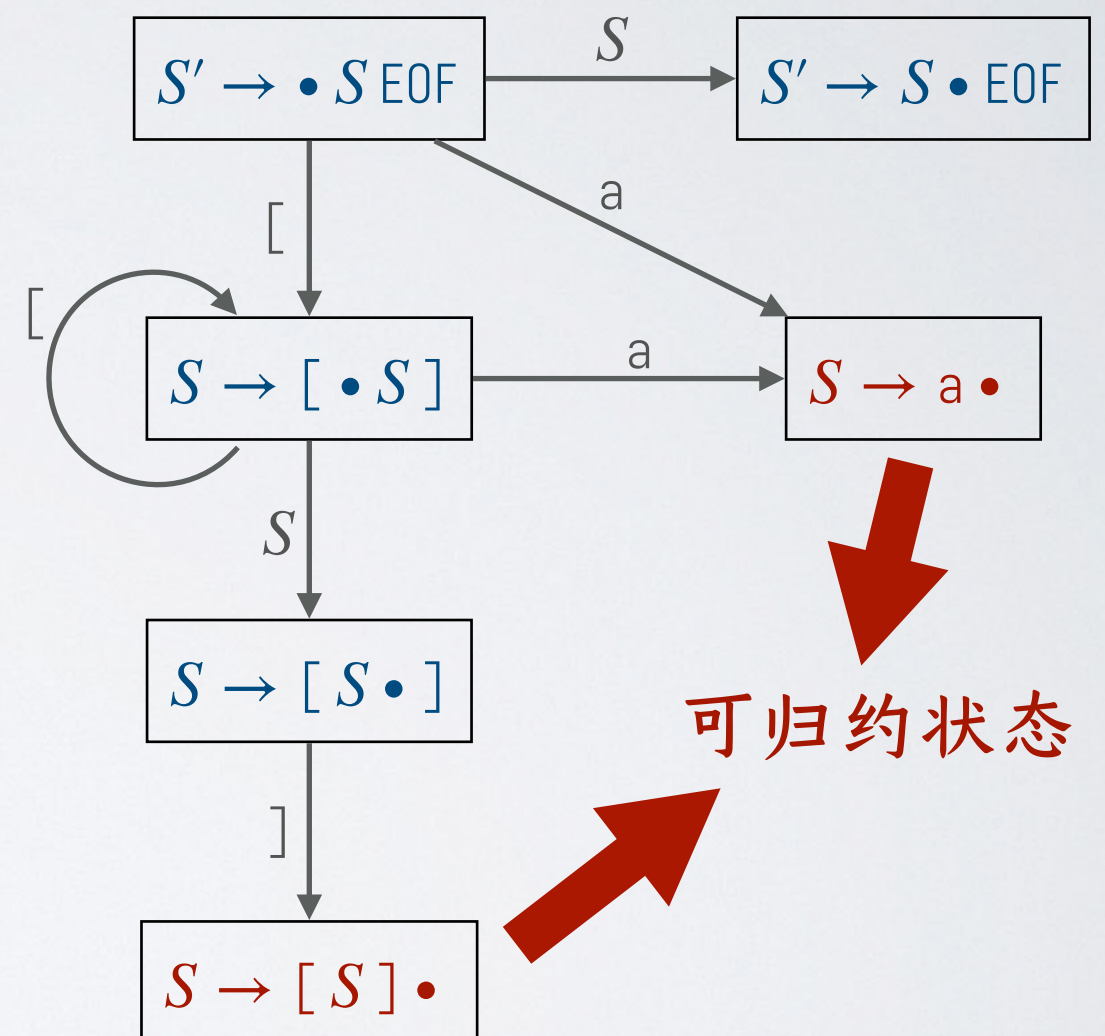
ϵ	$S' \rightarrow \bullet S \text{ EOF}$
S	$S' \rightarrow S \bullet \text{ EOF}$
$\beta [$	$S \rightarrow [\bullet S]$
$\beta [S$	$S \rightarrow [S \bullet]$
$\beta [S]$	$S \rightarrow [S] \bullet$
βa	$S \rightarrow a \bullet$

部分分析状态

- 移进/归约可以表示为这些「部分分析状态」间的转移

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow [S]$
[2]	$S \rightarrow a$

分析栈 \ 向前看	[	]	a	EOF
ϵ	移进		移进	
S				接受
$\beta[$	移进		移进	
$\beta[S$		移进		
$\beta[S]$	归约 [1]	归约 [1]	归约 [1]	归约 [1]
βa	归约 [2]	归约 [2]	归约 [2]	归约 [2]



部分分析状态间的转移

● 为了方便构造转移, 增加形如

$A \rightarrow \bullet \alpha$ 的「部分分析状态」

● 对于状态 $A \rightarrow \alpha \bullet \gamma$:

❖ 如果 $\gamma = c \gamma'$, 其中 c 是终结符号, 则可以通过 c 转移到 $A \rightarrow \alpha c \bullet \gamma'$

❖ 表示栈 $\beta \alpha$ 移进 c 变为栈 $\beta \alpha c$

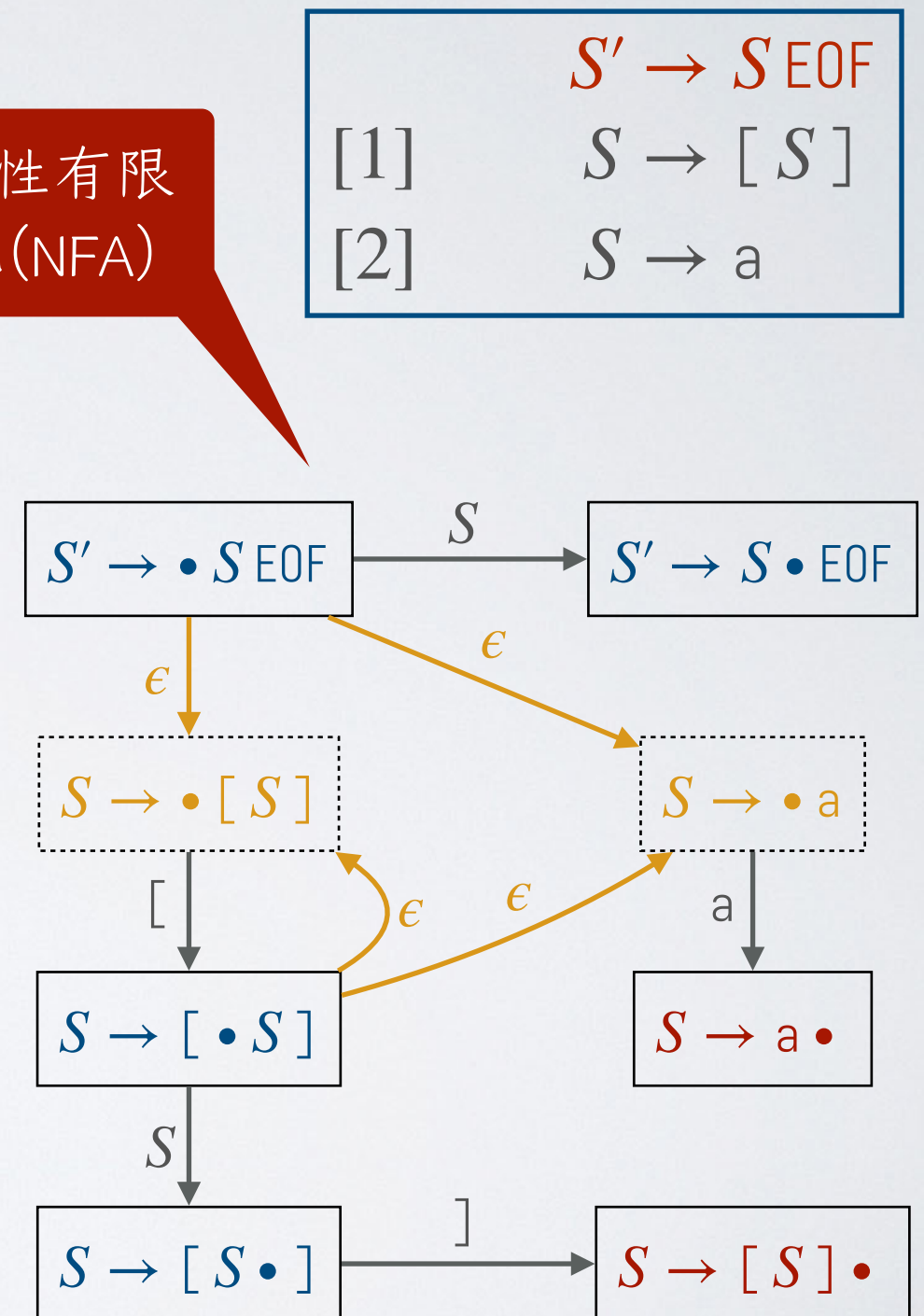
❖ 如果 $\gamma = X \gamma'$, 其中 X 是非终结符号, 则有两种转移:

❖ 通过 X 转移到 $A \rightarrow \alpha X \bullet \gamma'$

❖ 表示栈 $\beta \alpha w$ 归约成为栈 $\beta \alpha X$

❖ 对任意规则 $X \rightarrow \delta$, 可以 ϵ 转移到 $X \rightarrow \bullet \delta$

非确定性有限自动机(NFA)

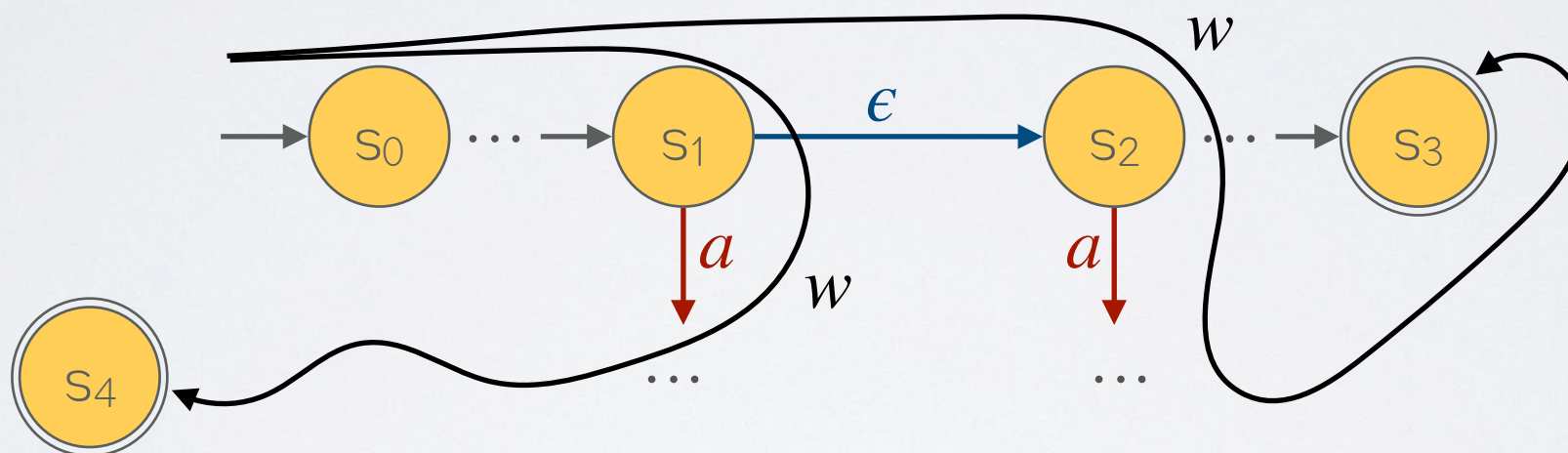


非确定性有限自动机

● 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



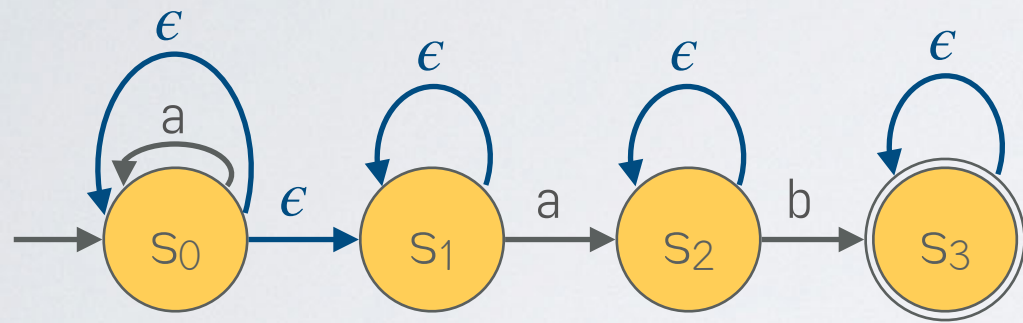
● 定义: $N = (S, \Sigma, \delta, s_0, F)$ 与 DFA 基本相同, 不同之处为

❖ $\delta : S \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^S$

● 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

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

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

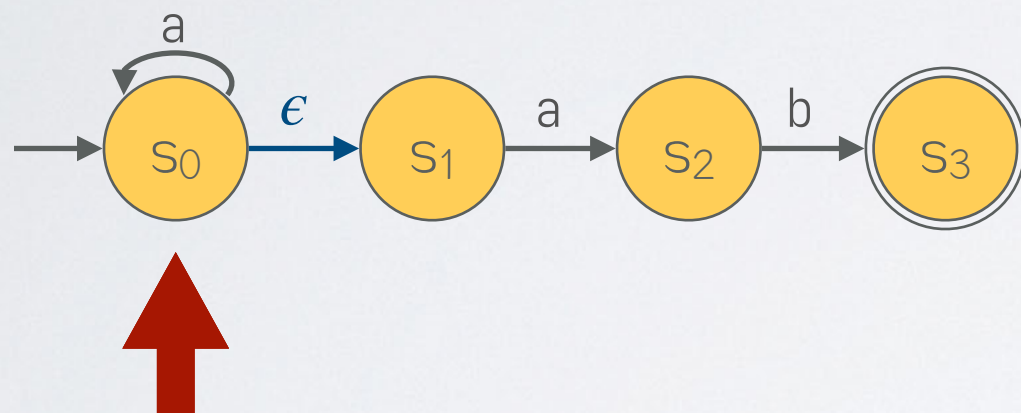
$$\Sigma = \{a, b\}$$

用空集表示转移到错误状态

$$\delta = \left\{ \begin{array}{l} s_0 \xrightarrow{a} \{s_0\}, s_0 \xrightarrow{b} \emptyset, s_0 \xrightarrow{\epsilon} \{s_0, s_1\} \\ s_1 \xrightarrow{a} \{s_2\}, s_1 \xrightarrow{b} \emptyset, s_1 \xrightarrow{\epsilon} \{s_1\} \\ s_2 \xrightarrow{a} \emptyset, s_2 \xrightarrow{b} \{s_3\}, s_2 \xrightarrow{\epsilon} \{s_2\} \\ s_3 \xrightarrow{a} \emptyset, s_3 \xrightarrow{b} \emptyset, s_3 \xrightarrow{\epsilon} \{s_3\} \end{array} \right\}$$

$$F = \{s_3\}$$

基于 NFA 的识别



$w = \boxed{a} a b$



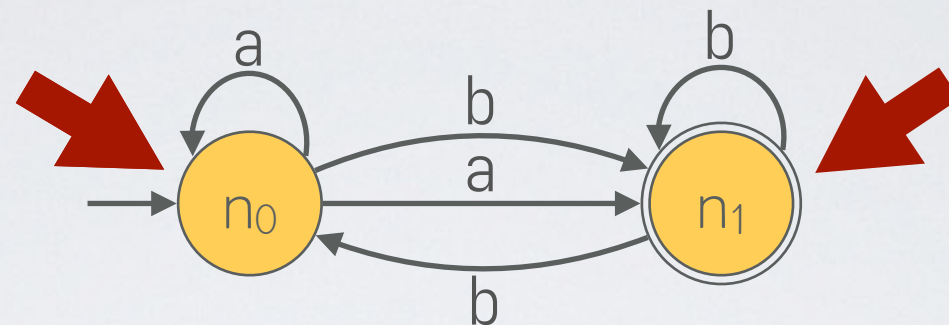
NFA 转换为 DFA

- ◎ 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

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

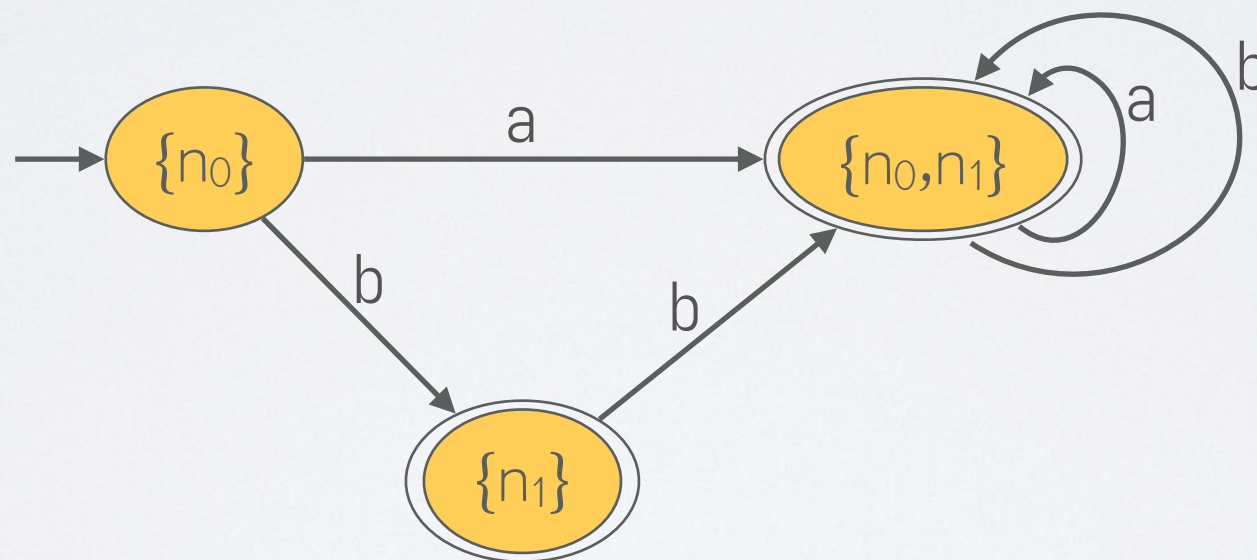
```
c = next_char();
s = d0;
while (c != EOF && s != de) {
    s = δD(s, c);
    c = next_char();
}
if (s ∈ FD) {
    // 识别成功
} else {
    // 识别失败
}
```

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

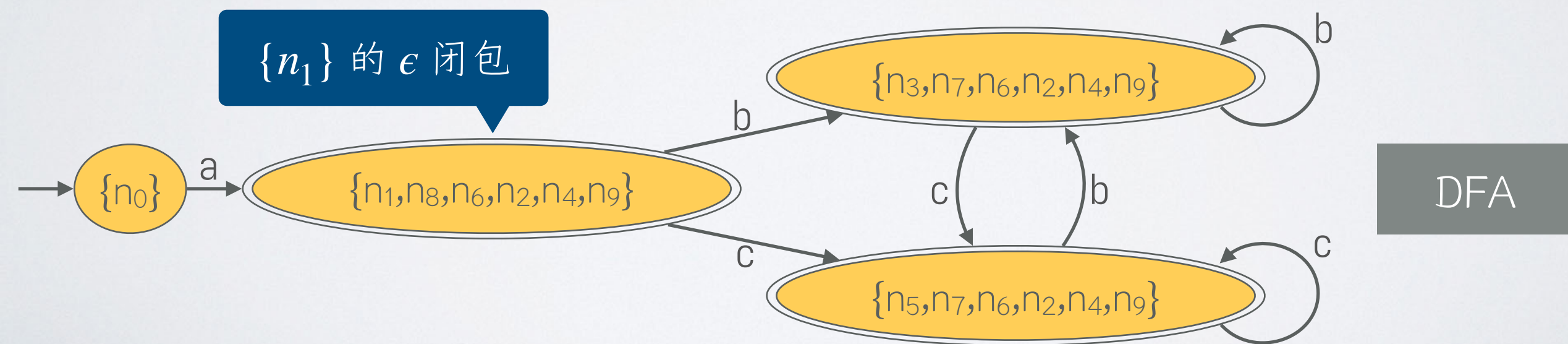
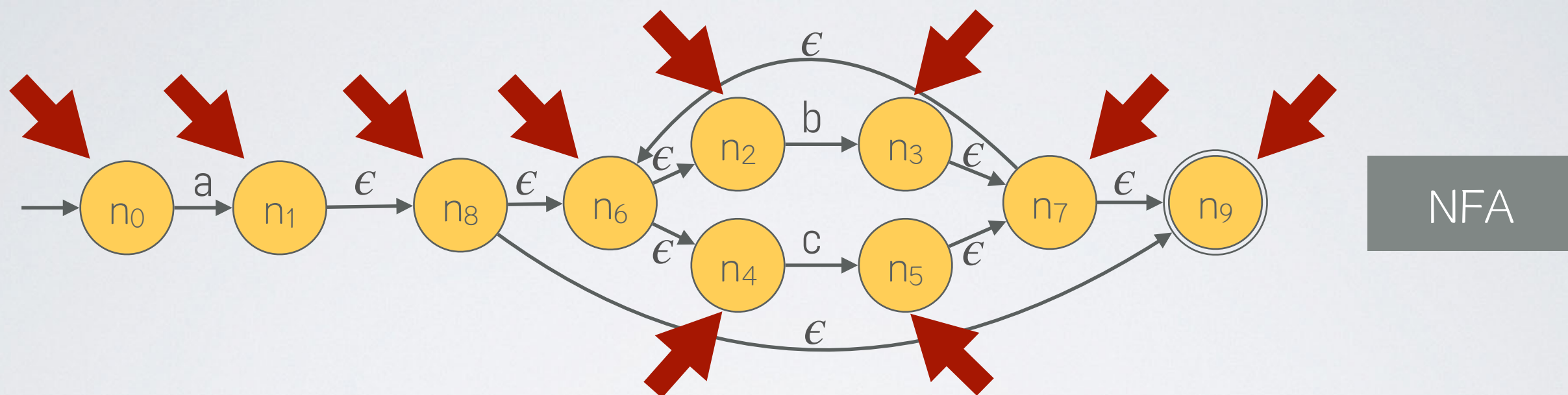


DFA

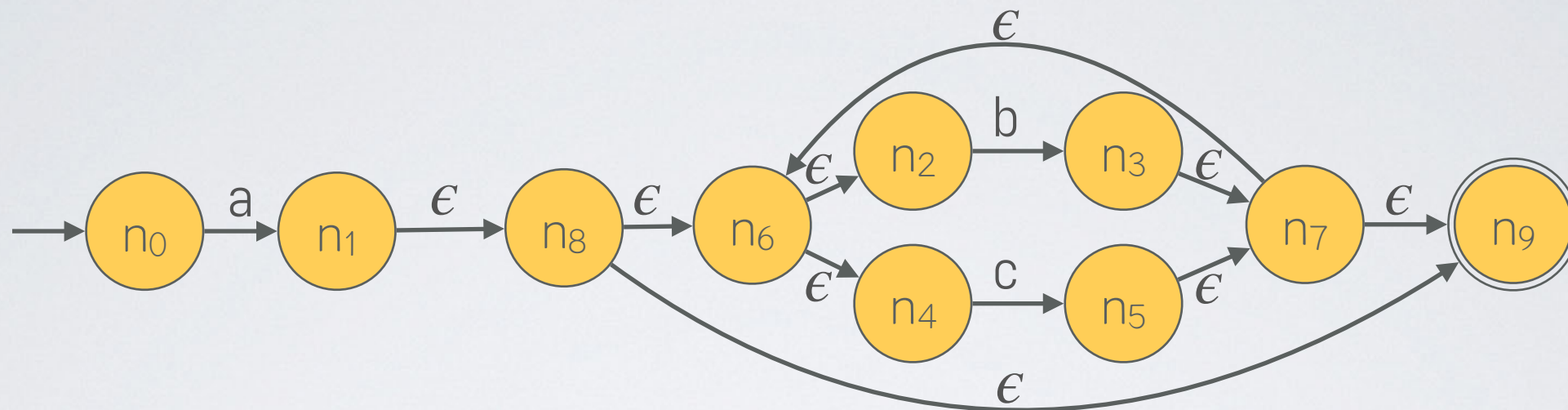
- 上面的 $\{n_0, n_1\}$ 表示存在符号串 w ，在 NFA 上从 n_0 出发识别 w 后既可能处于 n_0 ，也可能处于 n_1

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



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

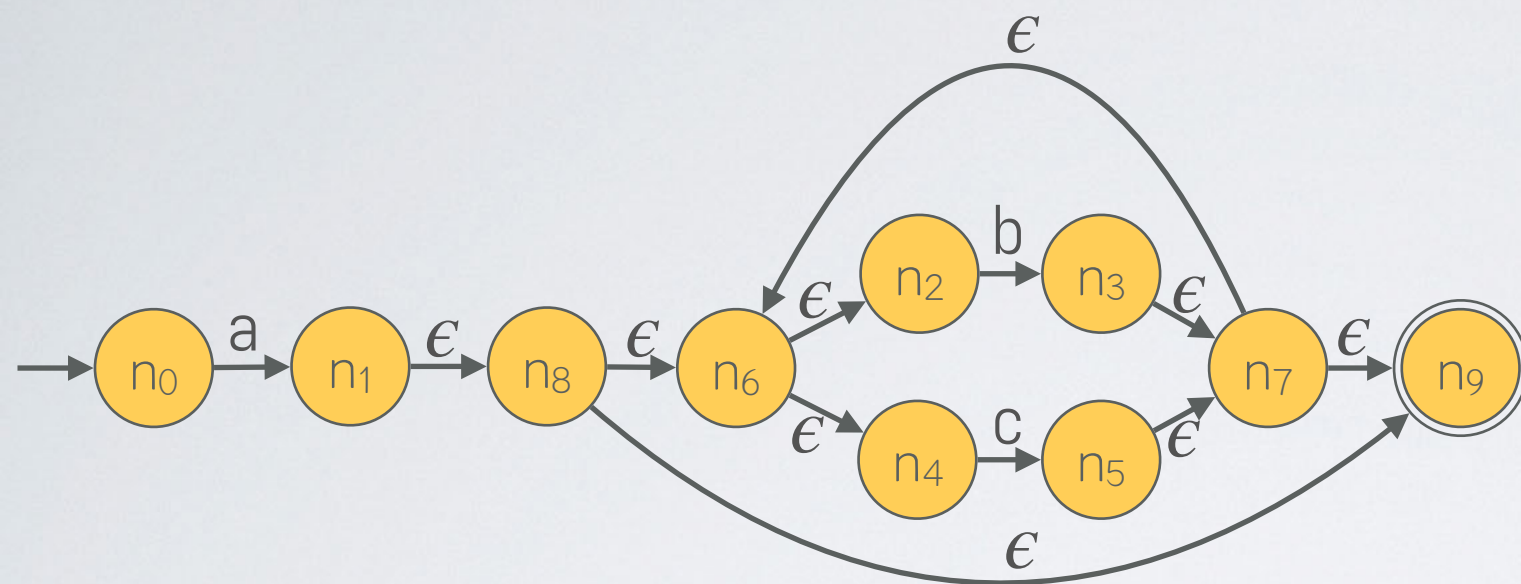
◎ 例：计算 $\{n_1\}$ 的 ϵ 闭包

- ❖ 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: $T = \{n_1, n_8, n_6, n_9, n_2, n_4\}$

```

 $\epsilon$ _closure( $S$ ) {
     $T = S$ ;
    do {
         $T' = T$ ;
         $T = T' \cup \bigcup_{s \in T'} \delta_N(s, \epsilon)$ ;
    } while ( $T \neq T'$ );
    return  $T$ ;
}
    
```

实现子集构造法



● 第一轮: $q = \{n_0\}$

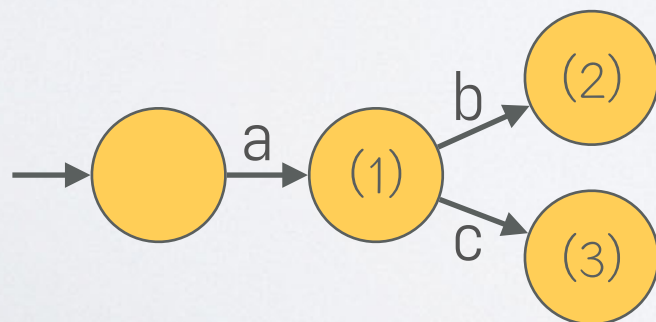
$$(1) \quad \delta_D(q, a) = \{n_1, n_8, n_6, n_2, n_4, n_9\}$$

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

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

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

●



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

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

```

 $d_0 = \epsilon\_closure(\{n_0\});$ 
 $S_D = \{d_0\};$ 
work_list =  $\{d_0\};$ 
while (work_list  $\neq \emptyset$ ) {
     $q = \text{remove\_from}(\text{work\_list});$ 
    foreach ( $c \in \Sigma$ ) {
         $t = \epsilon\_closure(\bigcup_{s \in q} \delta_N(s, c));$ 
         $\delta_D(q, c) = t;$ 
        if ( $t \notin S_D$ ) {
             $S_D = S_D \cup \{t\};$ 
            work_list = work_list  $\cup \{t\};$ 
        }
    }
}
 $F_D = \{q \mid q \in S_D, q \cap F_N \neq \emptyset\};$ 

```

部分分析状态的自动机

非确定性有限自动机(NFA)

$S' \rightarrow S \text{ EOF}$

[1]

$S \rightarrow [S]$

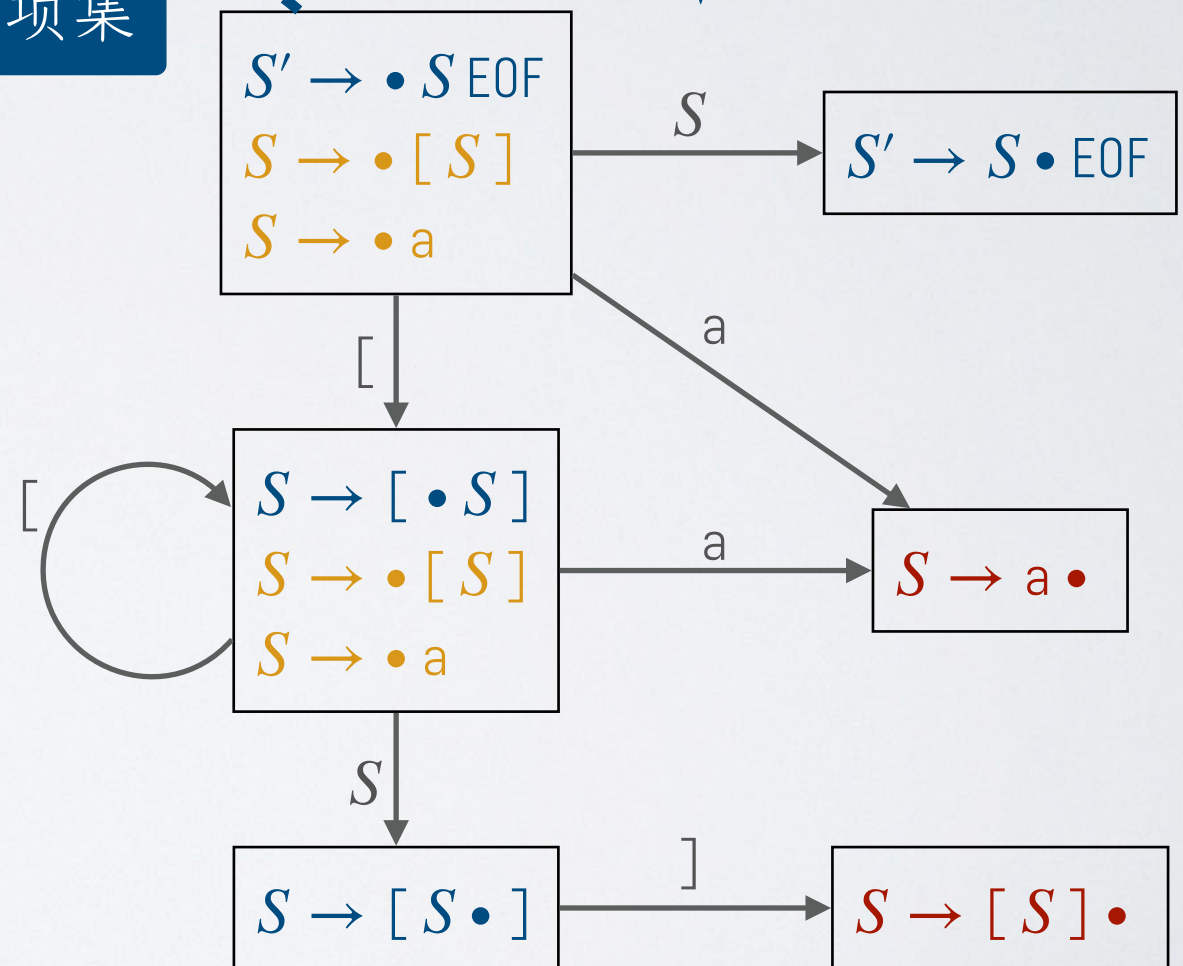
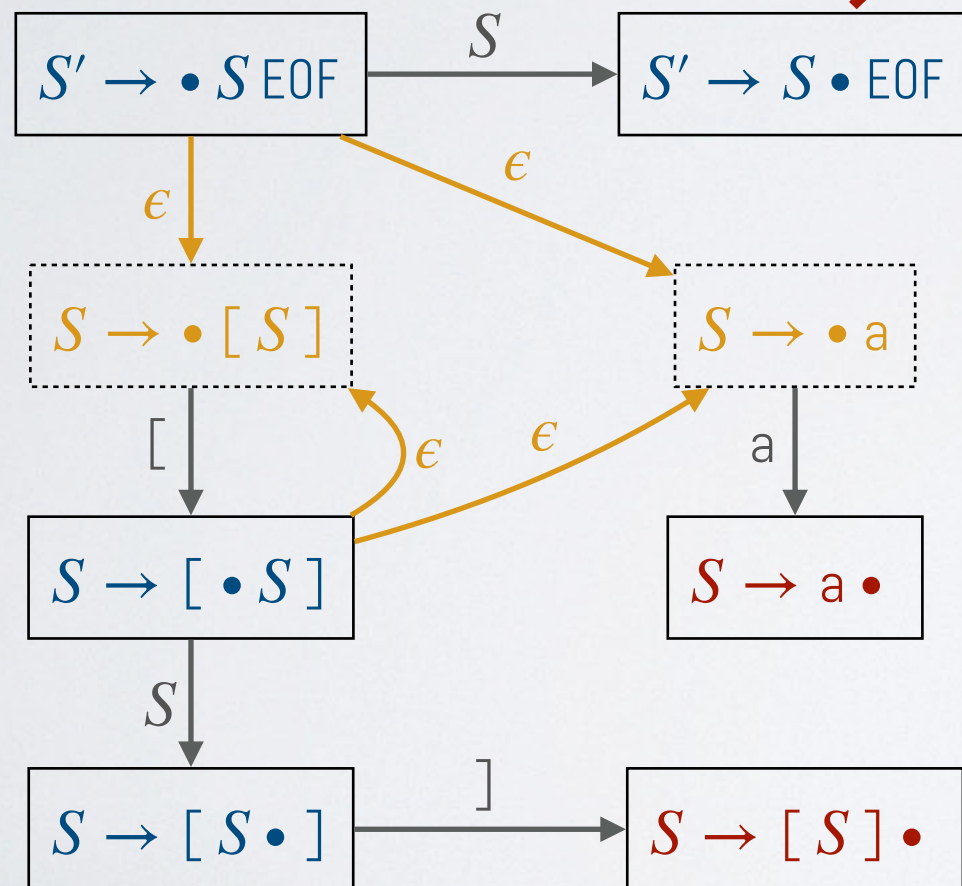
[2]

$S \rightarrow a$

确定性有限自动机(DFA)

项

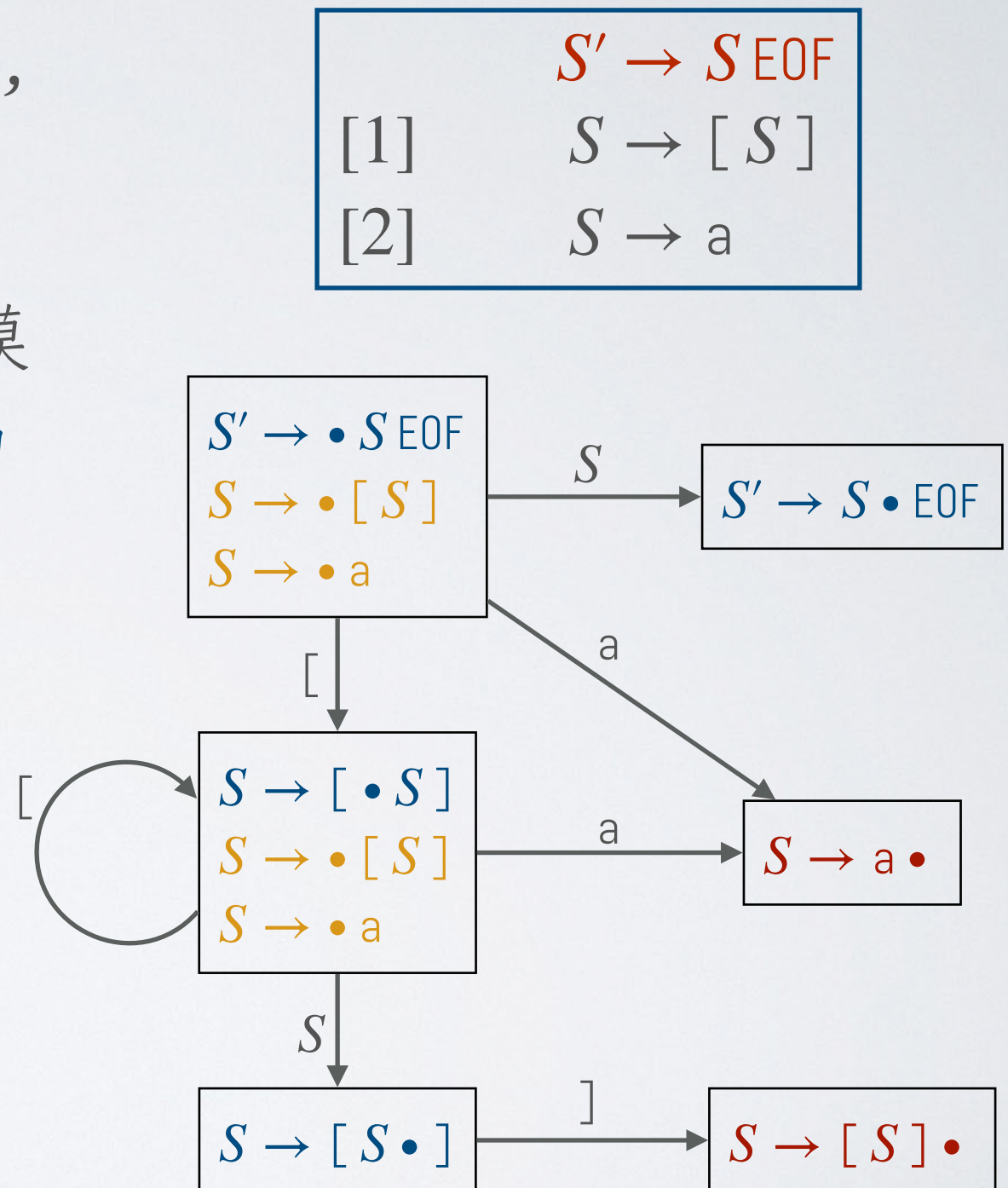
项集



根据自动机构造移进-归约分析表

- 如果状态有终结符号 c 的转移, 则可以对 c 进行**移进**
- 如果状态是可归约状态(存在模式 $A \rightarrow \alpha \bullet$), 则可以进行**归约**

分析栈 \ 向前看	[	]	a	EOF
ϵ	移进		移进	
S				接受
$\beta[$	移进		移进	
$\beta[S$		移进		
$\beta[S]$	归约 [1]	归约 [1]	归约 [1]	归约 [1]
βa	归约 [2]	归约 [2]	归约 [2]	归约 [2]



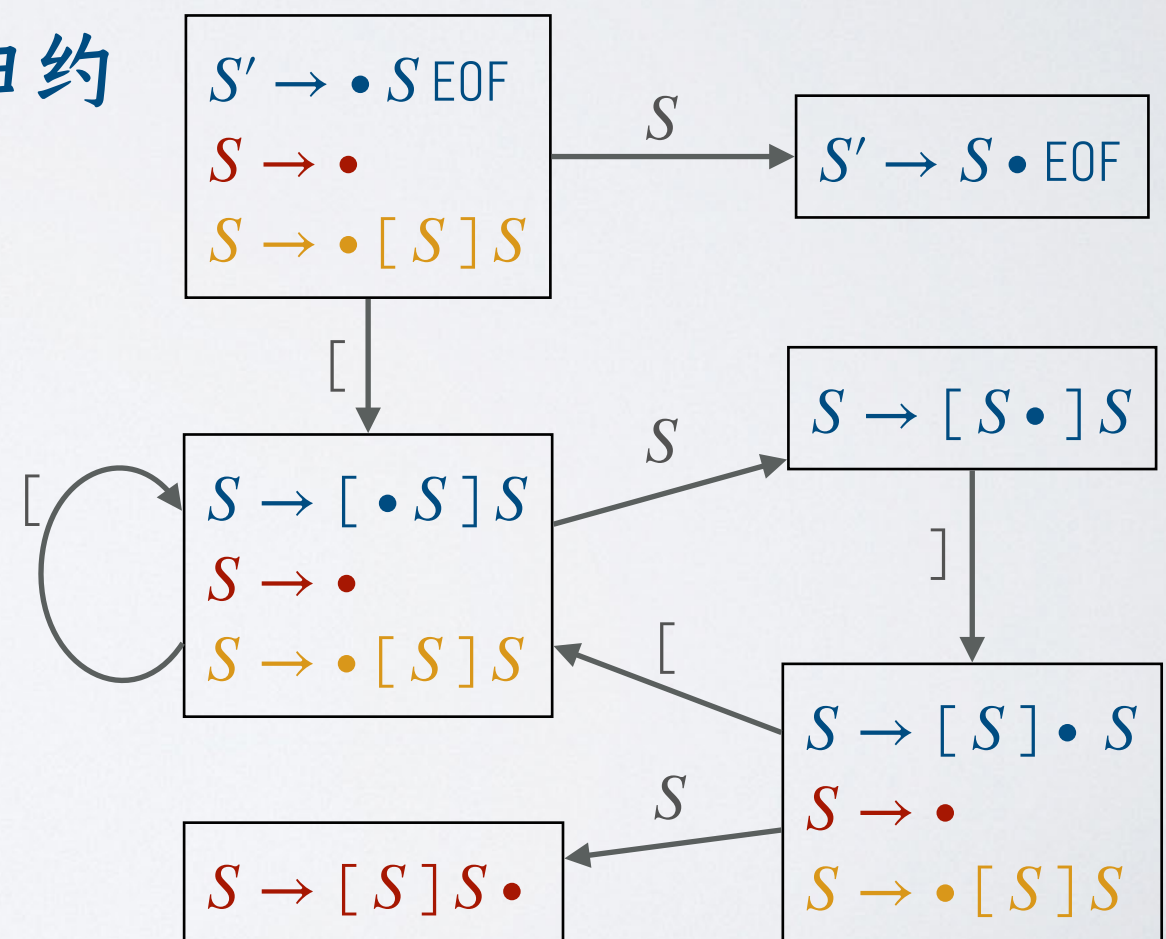
根据自动机构造移进-归约分析表

- 如果状态有终结符号 c 的转移, 则可以对 c 进行**移进**
- 如果状态是可归约状态(存在模式 $A \rightarrow \alpha \bullet$), 且向前看符号在 $\text{FOLLOW}(A)$ 中, 则可以进行**归约**

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow [S] S$

$\text{FOLLOW}(S) = \{\text{EOF},]\}$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	归约 [1]	归约 [1]
S			接受
$\beta[$	移进	归约 [1]	归约 [1]
$\beta[S$		移进	
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$		归约 [2]	归约 [2]

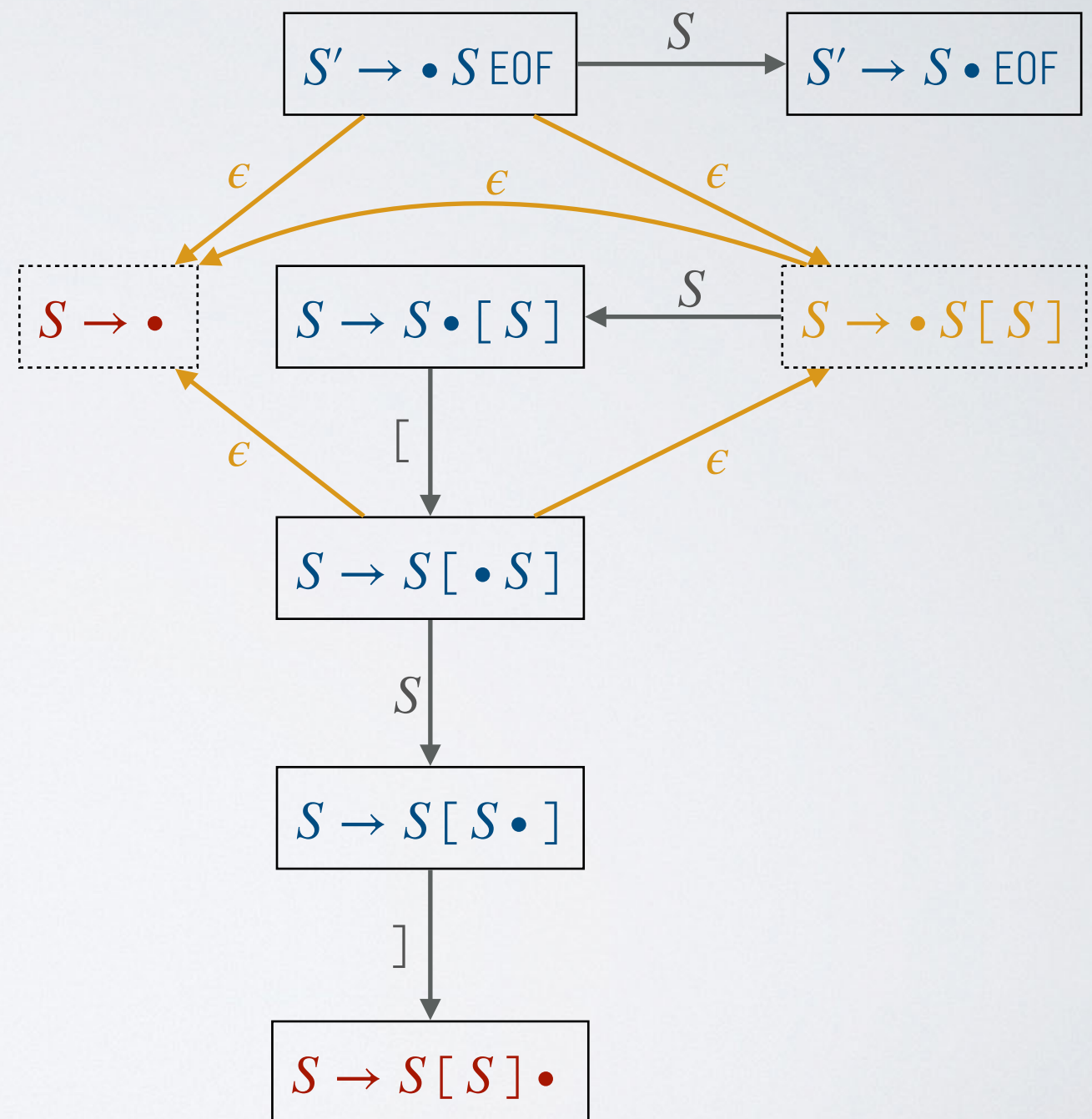


构造分析表示例

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow S[S]$

$\text{FOLLOW}(S) = \{\text{EOF}, [,]\}$

分析栈 \ 向前看	[	]	EOF
ϵ			
S			
βS			
$\beta S[$			
$\beta S[S$			
$\beta S[S]$			

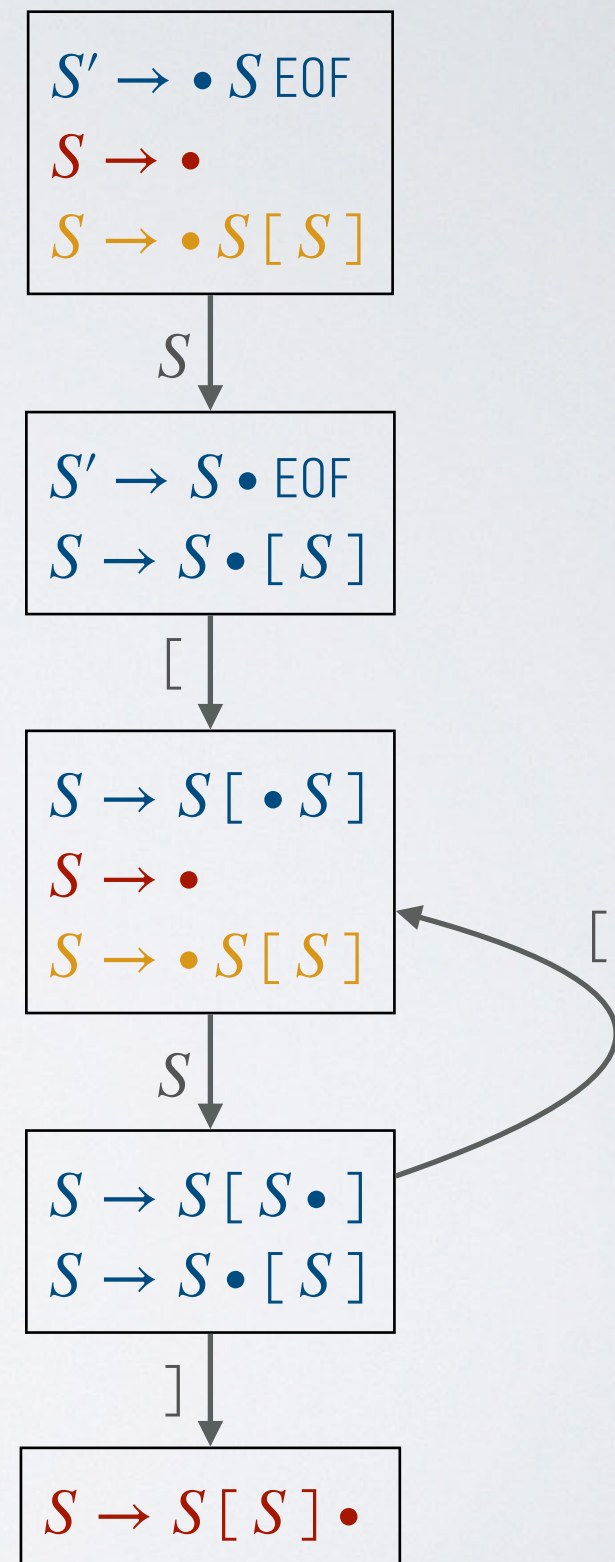


构造分析表示例

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow S[S]$

$\text{FOLLOW}(S) = \{\text{EOF}, [,]\}$

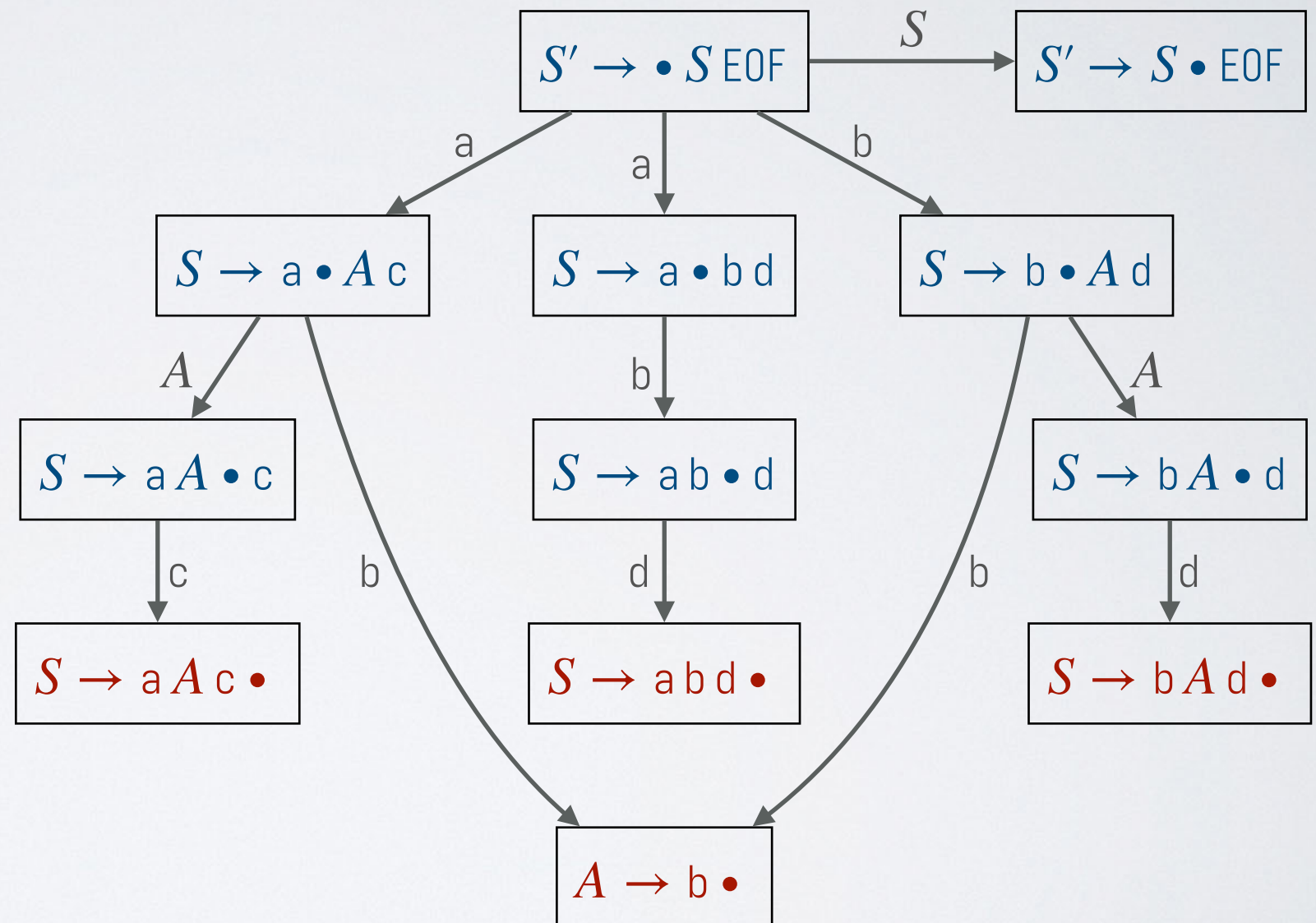
分析栈 \ 向前看	[	]	EOF
ϵ	归约 [1]	归约 [1]	归约 [1]
S	移进		接受
$\beta S[$	归约 [1]	归约 [1]	归约 [1]
$\beta S[S$	移进	移进	
$\beta S[S]$	归约 [2]	归约 [2]	归约 [2]



使用 FOLLOW 集合的局限性

- 为方便展示, 对 ϵ 转移进行了化简

	$S' \rightarrow S EOF$
[1]	$S \rightarrow a b d$
[2]	$S \rightarrow a A c$
[3]	$S \rightarrow b A d$
[4]	$A \rightarrow b$

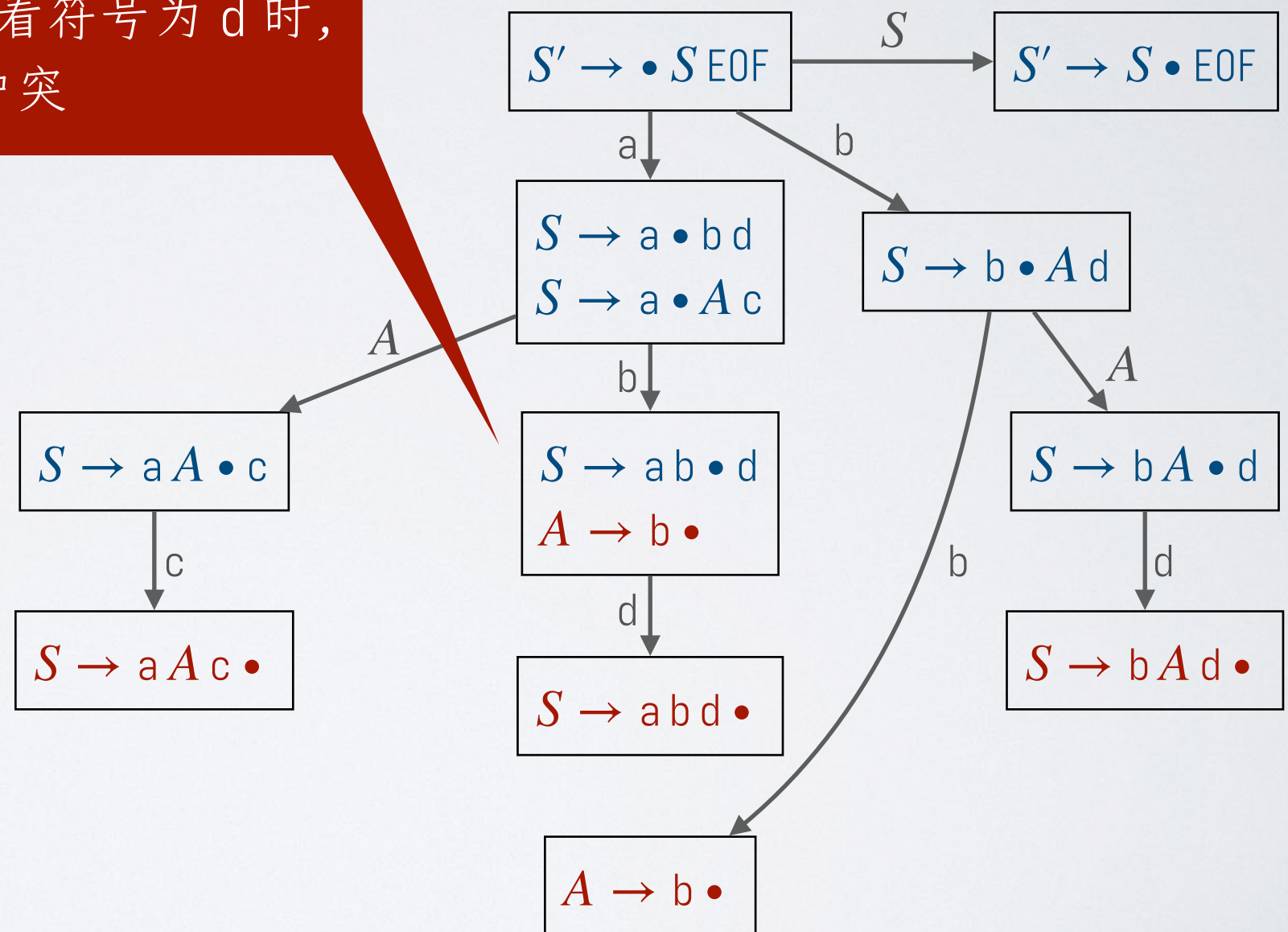


使用 FOLLOW 集合的局限性

当分析栈「模式」为 ab 、向前看符号为 d 时，
存在移进-归约冲突

	$S' \rightarrow S EOF$
[1]	$S \rightarrow abd$
[2]	$S \rightarrow aAc$
[3]	$S \rightarrow bAd$
[4]	$A \rightarrow b$

$FOLLOW(A) = \{c, d\}$

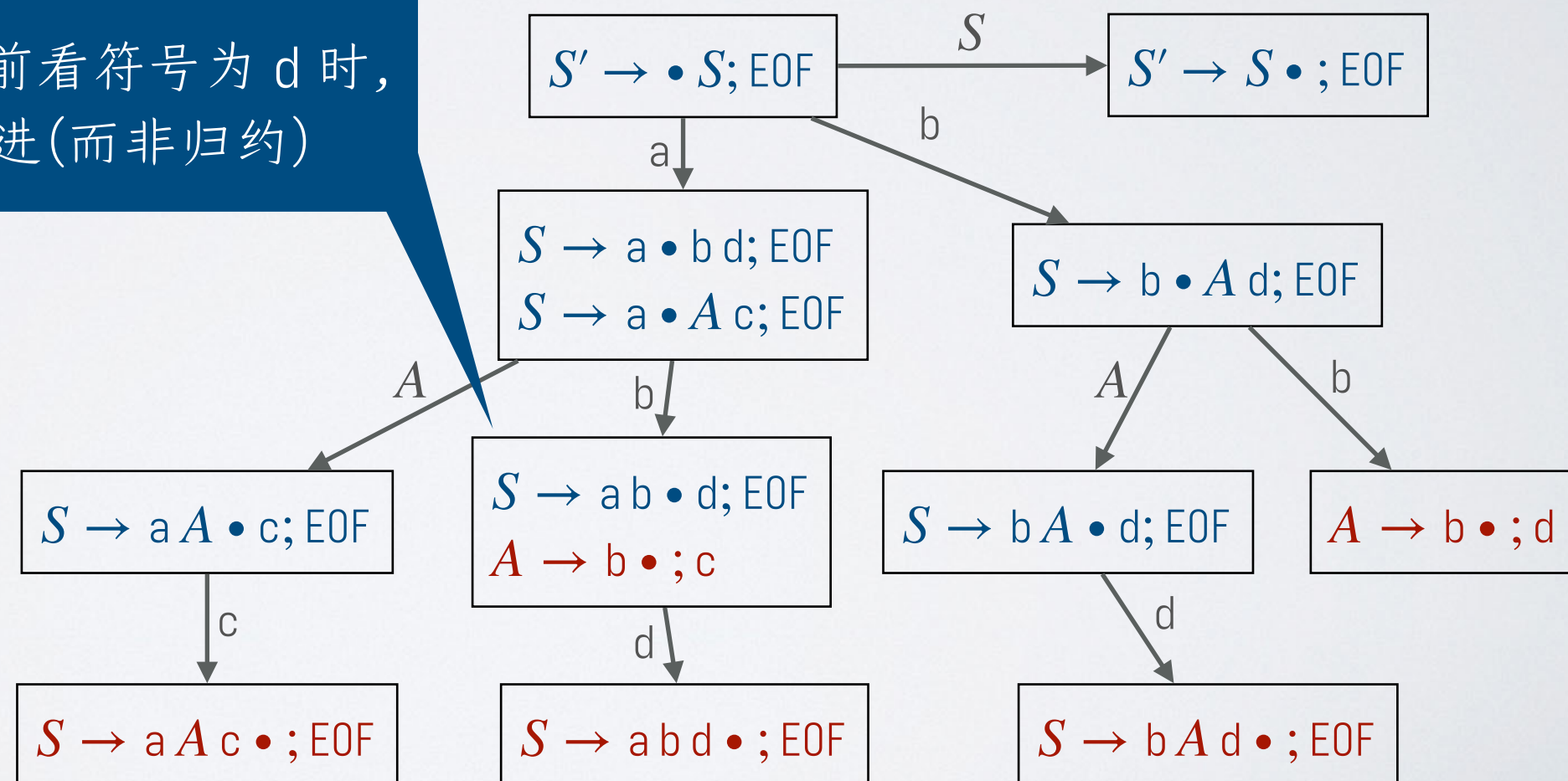


部分分析状态 + 「再向前看」

- 使用 FOLLOW 集合来判断是否归约的方式比较粗糙
- 扩充每个部分分析状态 $A \rightarrow \alpha \bullet \gamma$ 为 $\langle A \rightarrow \alpha \bullet \gamma; c \rangle$
 - 从该部分分析状态完成 γ 的分析后,「再向前看」为 c 时可以进行归约

当分析栈「模式」为 ab 、向前看符号为 d 时, 不存在冲突, 动作为移进(而非归约)

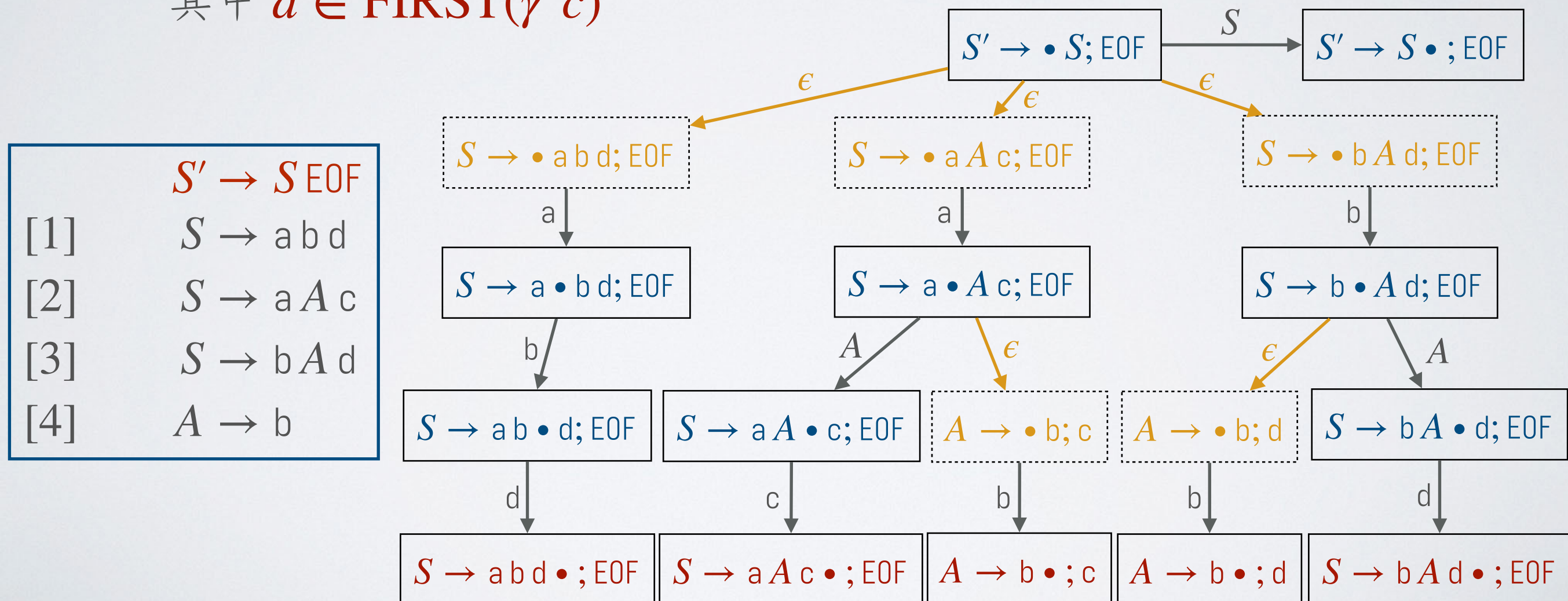
	$S' \rightarrow S EOF$
[1]	$S \rightarrow a b d$
[2]	$S \rightarrow a A c$
[3]	$S \rightarrow b A d$
[4]	$A \rightarrow b$



部分分析状态 + 「再向前看」

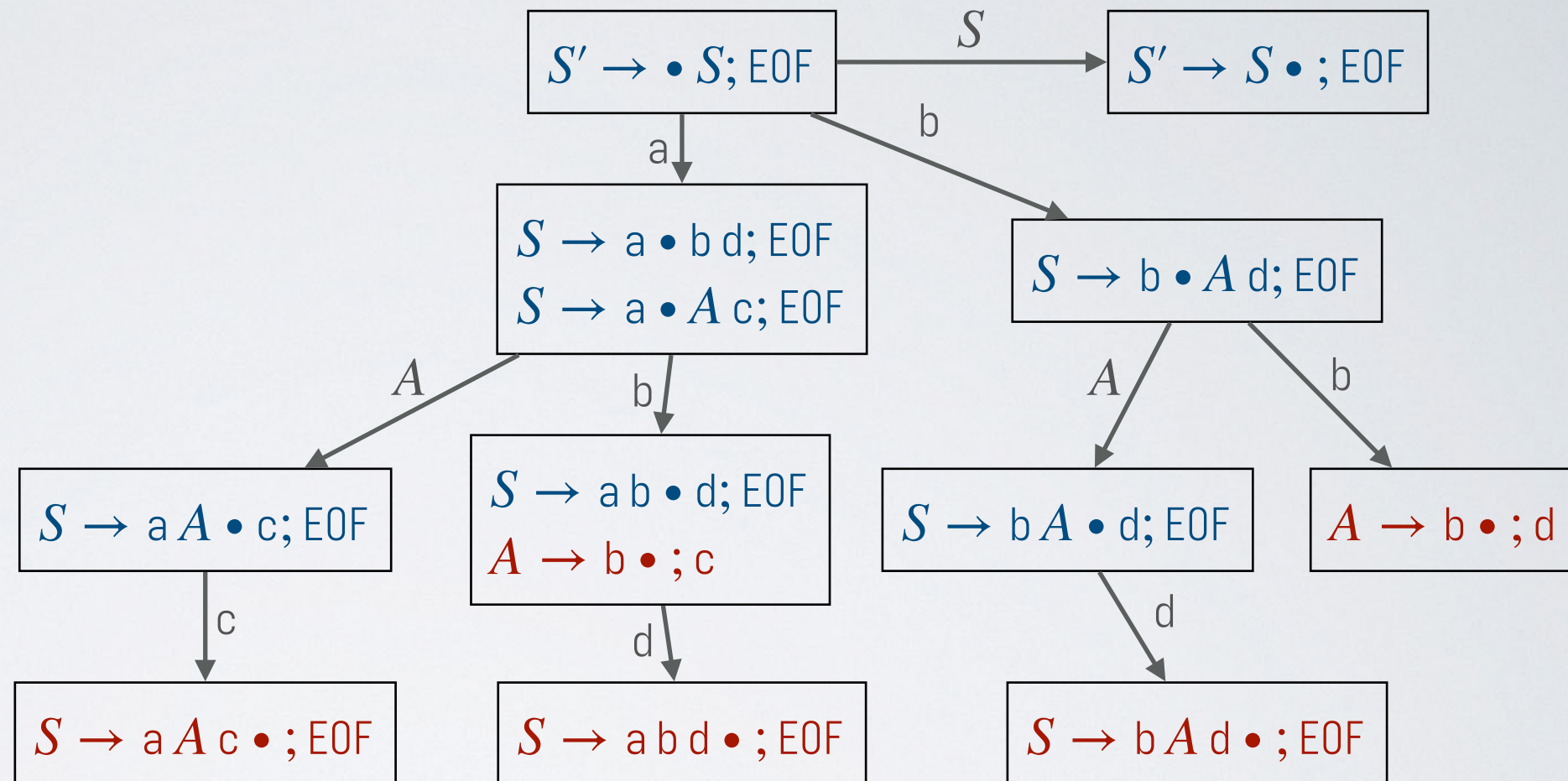
构造时, 主要考虑 ϵ 转移对「再向前看」符号的影响

- ❖ 回顾: 对于状态 $A \rightarrow \alpha \bullet X \gamma'$ 和规则 $X \rightarrow \delta$, ϵ 转移到 $X \rightarrow \bullet \delta$
- ❖ 对于状态 $\langle A \rightarrow \alpha \bullet X \gamma'; c \rangle$ 和规则 $X \rightarrow \delta$, ϵ 转移到 $\langle X \rightarrow \bullet \delta; d \rangle$, 其中 $d \in \text{FIRST}(\gamma' c)$



部分分析状态 + 「再向前看」

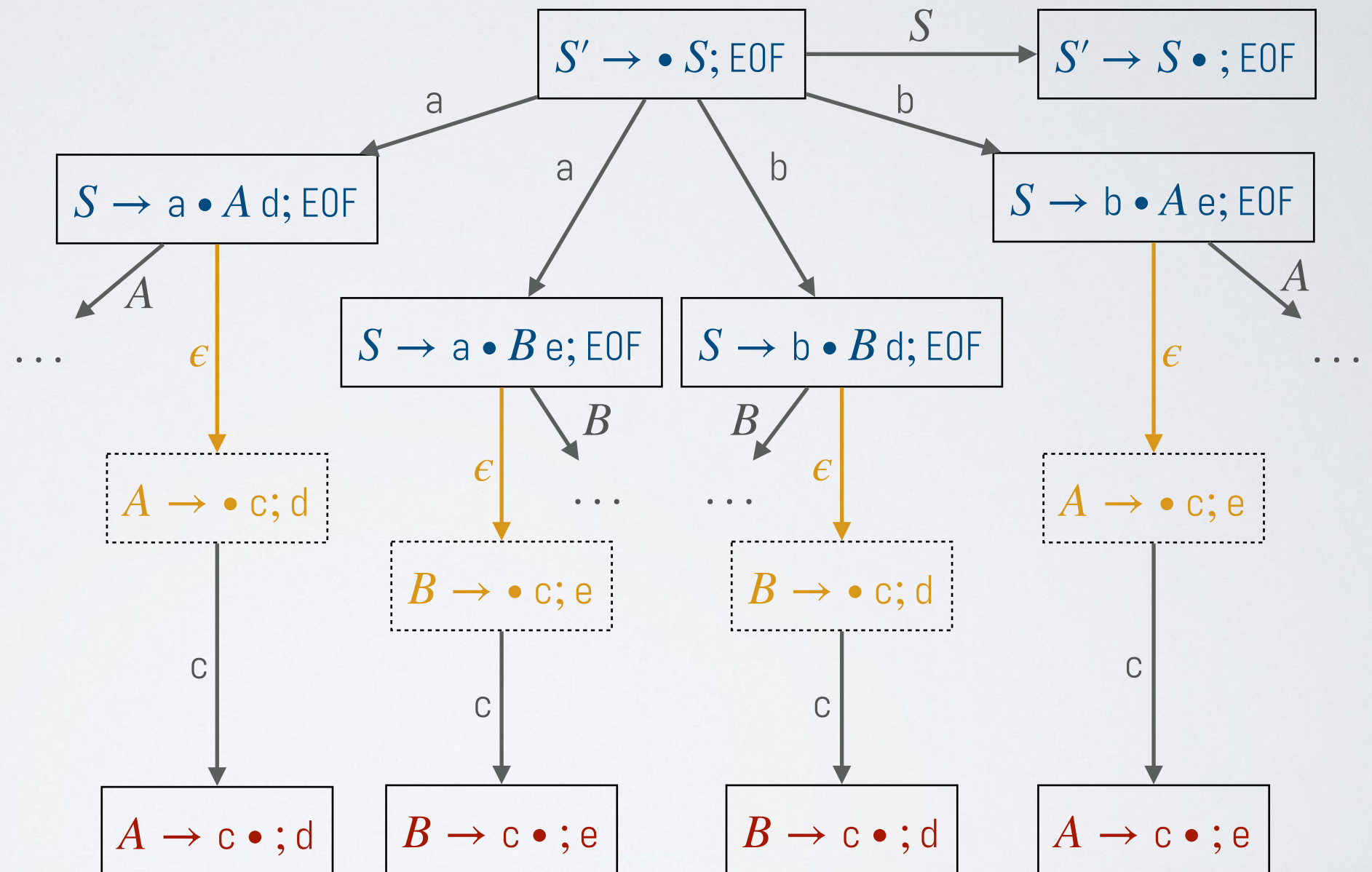
	$S' \rightarrow S EOF$
[1]	$S \rightarrow a b d$
[2]	$S \rightarrow a A c$
[3]	$S \rightarrow b A d$
[4]	$A \rightarrow b$



向前看 \ 分析栈	ϵ	S	a	aA	aAc	ab	abd	b	bA	bAd	bb
a	移进										
b	移进		移进					移进			
c				移进		归约 [4]					
d						移进			移进		归约 [4]
EOF		接受			归约 [2]		归约 [1]			归约 [3]	

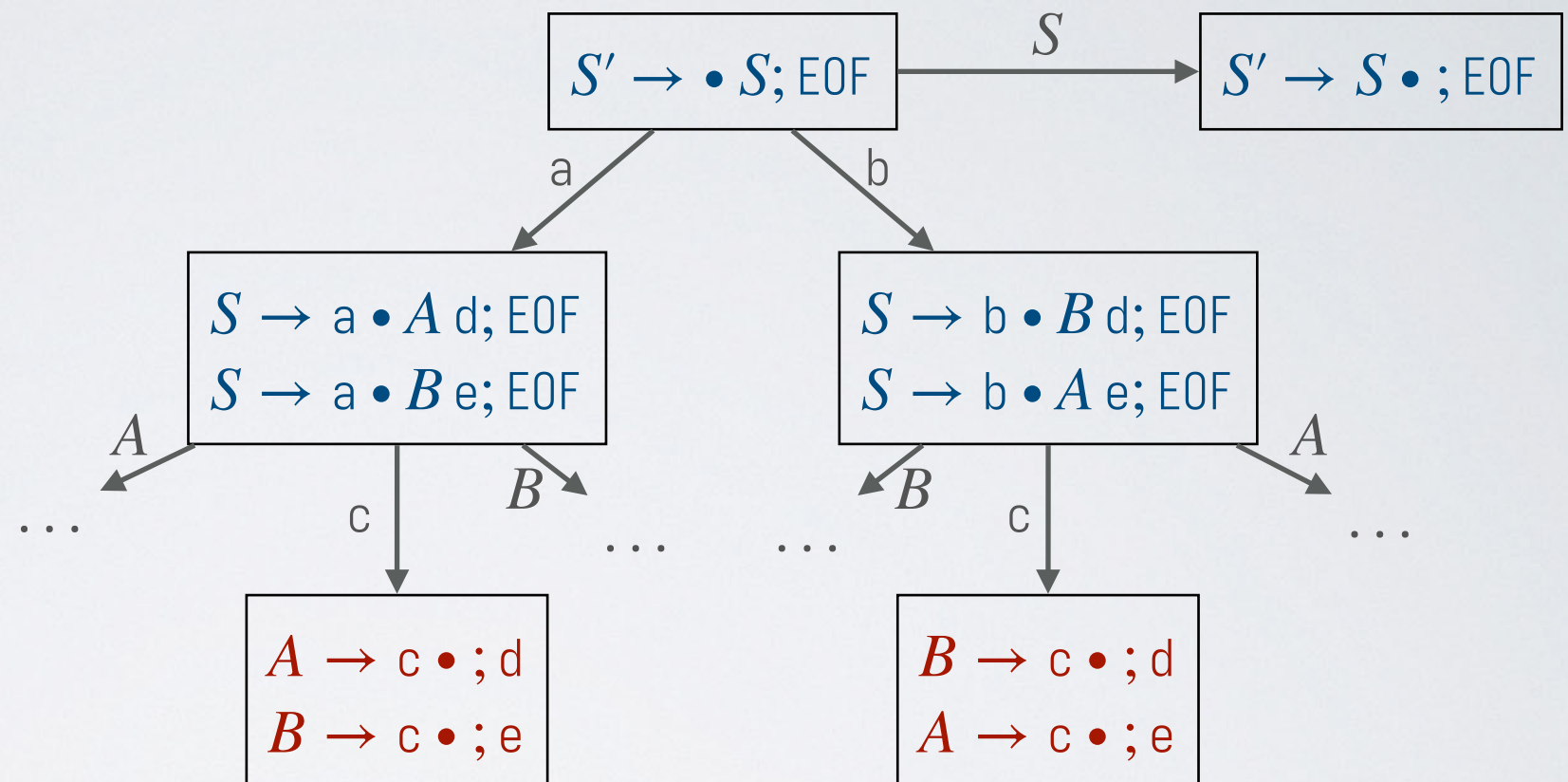
构造分析表示例

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow a A d$
[2]	$S \rightarrow b B d$
[3]	$S \rightarrow a B e$
[4]	$S \rightarrow b A e$
[5]	$A \rightarrow c$
[6]	$B \rightarrow c$



构造分析表示例

	$S' \rightarrow S EOF$
[1]	$S \rightarrow a A d$
[2]	$S \rightarrow b B d$
[3]	$S \rightarrow a B e$
[4]	$S \rightarrow b A e$
[5]	$A \rightarrow c$
[6]	$B \rightarrow c$



分析栈 \ 向前看	a	b	c	d	e	EOF
ϵ	移进	移进				
S						接受
$a c$				归约 [5]	归约 [6]	
$b c$				归约 [6]	归约 [5]	
...						

LR 文法

- ◎ 能够通过**无冲突的移进-归约预测分析**进行识别的 CFG 文法
 - ❖ **L**: 从左往右扫描 token
 - ❖ 把 token 依次**移进**分析栈
 - ❖ **R**: 构造最右推导的分析树
 - ❖ 从分析栈的栈顶识别模式, 按照产生规则进行**归约**
- ◎ **LR(k) 文法**: 部分分析状态可以记录 k 个「再向前看」的 token
 - ❖ 最常见的一种是 LR(1)
- ◎ **LR(k) 比 LL(k) 能表达的文法更多**
 - ❖ LL 文法要求在向前看 k 个符号时「猜出」用哪个产生规则进行推导
 - ❖ LR 文法是在完整的看到一个可归约的子串后, 还能「再向前看」 k 个符号来决定用哪个产生规则进行归约

LR 文法允许左递归

ϵ || $[[[]]] \text{ EOF}$
 S || $[[[]]] \text{ EOF}$
 $S[$ || $[[[]]] \text{ EOF}$
 $S[S$ || $[[[]]] \text{ EOF}$
 $S[S[$ || $[[[]]] \text{ EOF}$
 $S[S[S$ || $[[[]]] \text{ EOF}$
 $S[S[S]$ || $[[[]]] \text{ EOF}$
 $S[S$ || $[[[]]] \text{ EOF}$
 $S[S[$ || $[[[]]] \text{ EOF}$
 $S[S[S$ || $[[[]]] \text{ EOF}$
 $S[S[S]$ || $[[[]]] \text{ EOF}$
 $S[S$ || $[[[]]] \text{ EOF}$
 $S[S]$ || $[[[]]] \text{ EOF}$
 S || $[[[]]] \text{ EOF}$

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow S[S]$

分析栈 \ 向前看	[	]	EOF
ϵ	归约 [1]	归约 [1]	归约 [1]
S	移进		接受
$\beta S[$	归约 [1]	归约 [1]	归约 [1]
$\beta S[S$	移进	移进	
$\beta S[S]$	归约 [2]	归约 [2]	归约 [2]

LR 文法允许左公因子

$$S' \rightarrow S \text{ EOF}$$

$$[1] \quad S \rightarrow [S]$$

$$[2] \quad S \rightarrow []$$

ϵ || $[[[]]] \text{ EOF}$
 $[$ || $[[]]] \text{ EOF}$
 $[[$ || $[]]] \text{ EOF}$
 $[[[$ || $]]] \text{ EOF}$
 $[[[]$ || $]] \text{ EOF}$
 $[[[S$ || $]] \text{ EOF}$
 $[[[S]$ || $] \text{ EOF}$
 $[[S$ || $] \text{ EOF}$
 $[S]$ || EOF
 S || EOF

分析栈 \ 向前看	[	]	EOF
ϵ	移进		
S			接受
$\beta[$	移进	移进	
$\beta[S$		移进	
$\beta[S]$	归约 [1]	归约 [1]	归约 [1]
$\beta[]$	归约 [2]	归约 [2]	归约 [2]

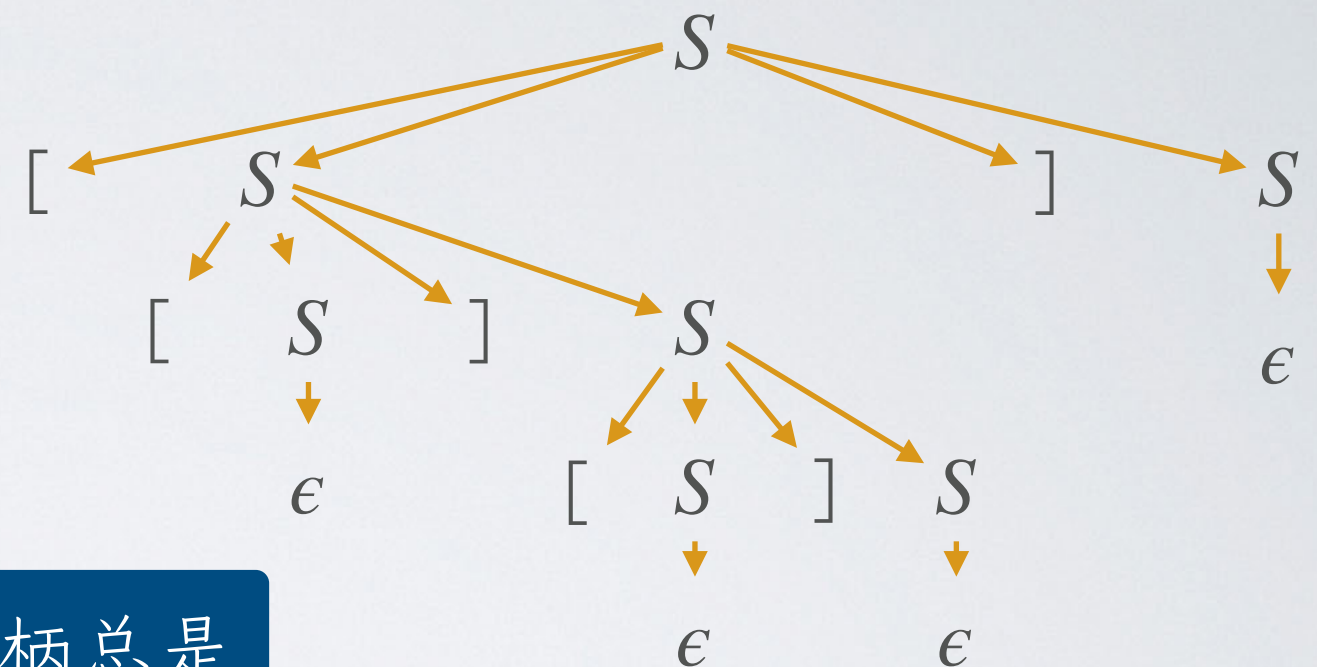
LR(1) 文法

- ◎ 允许一个「再向前看」符号
- ◎ 用于识别栈模式的自动机中，状态形式为 $\langle A \rightarrow \alpha \bullet \gamma; c \rangle$
 - ❖ $A \rightarrow \alpha \bullet \gamma$ 表示「部分分析状态」
 - ❖ c 表示「再向前看」符号
- ◎ 构造移进-归约分析表的方法：
 - ❖ 构造识别栈模式的 NFA，中间可能有非确定转移和 ϵ 转移
 - ❖ 把 NFA 转换为 DFA
 - ❖ 根据 DFA 确定可能的栈模式，再根据状态转移确定移进/归约动作
- ◎ LR(1) 的工作原理是什么？

LR(1) 的工作原理是什么？



句柄总是在栈顶



$S \Leftarrow [S]S$
 $\Leftarrow [S]$
 $\Leftarrow [[S]S]$
 $\Leftarrow [[S][S]S]$
 $\Leftarrow [[S][S]]$
 $\Leftarrow [[S][]]$
 $\Leftarrow [][]$

LR(1) 的工作原理是什么？

● LR(1) 能进行无冲突移进-归约分析的原因：

- ❖ 对任意 $S \Rightarrow^* \alpha c \beta$ 和 $S \Rightarrow^* \alpha c \gamma$, 其中 c 是终结符号, β, γ 中只有终结符号, 如果 $\alpha c \beta$ 的句柄是 α 的一个后缀, 那么 $\alpha c \gamma$ 的句柄是 α 的同一个后缀, 并且这两个句柄对应的归约规则相同

- ❖ α 表示分析栈, c 表示「再向前看」符号

● 构造识别句柄的正则文法：

- ❖ 部分分析状态: $\langle A; c \rangle$, A 为非终结符号, c 为「再向前看」符号
- ❖ 归约状态: $\langle [p] \rangle$, $[p]$ 为规则编号
- ❖ $\langle S; EOF \rangle \Rightarrow^* \alpha c \langle [p] \rangle$ 当且仅当存在 β 满足 $S \Rightarrow^* \alpha c \beta$ 且句柄是 α 的后缀且使用规则 $[p]$ 归约

[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow [S] S$

$\langle S; EOF \rangle \rightarrow EOF$	$\langle [1] \rangle$
$\langle S; EOF \rangle \rightarrow [$	$\langle S;] \rangle$
$\langle S; EOF \rangle \rightarrow [S]$	$\langle S; EOF \rangle$
$\langle S; EOF \rangle \rightarrow [S] S EOF$	$\langle [2] \rangle$
$\langle S;] \rangle \rightarrow]$	$\langle [1] \rangle$
$\langle S;] \rangle \rightarrow [$	$\langle S;] \rangle$
$\langle S;] \rangle \rightarrow [S]$	$\langle S;] \rangle$
$\langle S;] \rangle \rightarrow [S] S]$	$\langle [2] \rangle$

LR(1) 的工作原理是什么？

$[[\quad] \quad] \text{ EOF}$	句柄: $\langle S; \text{EOF} \rangle \Rightarrow^* [\quad] \langle [1] \rangle$
$[[S] [\quad] \text{ EOF}$	句柄: $\langle S; \text{EOF} \rangle \Rightarrow^* [[S] [\quad] \langle [1] \rangle$
$[[S] [S] \quad] \text{ EOF}$	句柄: $\langle S; \text{EOF} \rangle \Rightarrow^* [[S] [S] [\quad] \langle [1] \rangle$
$[[S] [S] S \quad] \text{ EOF}$	句柄: $\langle S; \text{EOF} \rangle \Rightarrow^* [[S] [S] S] \langle [2] \rangle$
$[[S] S \quad] \text{ EOF}$	句柄: $\langle S; \text{EOF} \rangle \Rightarrow^* [[S] S] \langle [2] \rangle$
$[S] \quad \text{ EOF}$	句柄: $\langle S; \text{EOF} \rangle \Rightarrow^* [S] \text{ EOF} \langle [1] \rangle$
$[S] S \quad \text{ EOF}$	句柄: $\langle S; \text{EOF} \rangle \Rightarrow^* [S] S \text{ EOF} \langle [2] \rangle$

[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow [S] S$

- LR(1) 能进行无冲突移进-归约分析的原因:

❖ 如果 $\langle S; \text{EOF} \rangle \Rightarrow^* \alpha \langle [p] \rangle$ 且 $\langle S; \text{EOF} \rangle \Rightarrow^* \alpha \beta \langle [q] \rangle$, 则一定有 $\beta = \epsilon$ 且 $p = q$

$\langle S; \text{EOF} \rangle \rightarrow \text{EOF}$	$\langle [1] \rangle$
$\langle S; \text{EOF} \rangle \rightarrow [$	$\langle S;] \rangle$
$\langle S; \text{EOF} \rangle \rightarrow [S]$	$\langle S; \text{EOF} \rangle$
$\langle S; \text{EOF} \rangle \rightarrow [S] S \text{ EOF}$	$\langle [2] \rangle$
$\langle S;] \rangle \rightarrow]$	$\langle [1] \rangle$
$\langle S;] \rangle \rightarrow [$	$\langle S;] \rangle$
$\langle S;] \rangle \rightarrow [S]$	$\langle S;] \rangle$
$\langle S;] \rangle \rightarrow [S] S]$	$\langle [2] \rangle$

LR(1) 文法的变种

◎ LR(0)

- ❖ 不使用「再向前看」符号
- ❖ 识别栈模式的自动机中状态形式为 $\langle A \rightarrow \alpha \bullet \gamma \rangle$
- ❖ 前面已介绍过

◎ SLR(1), Simple LR

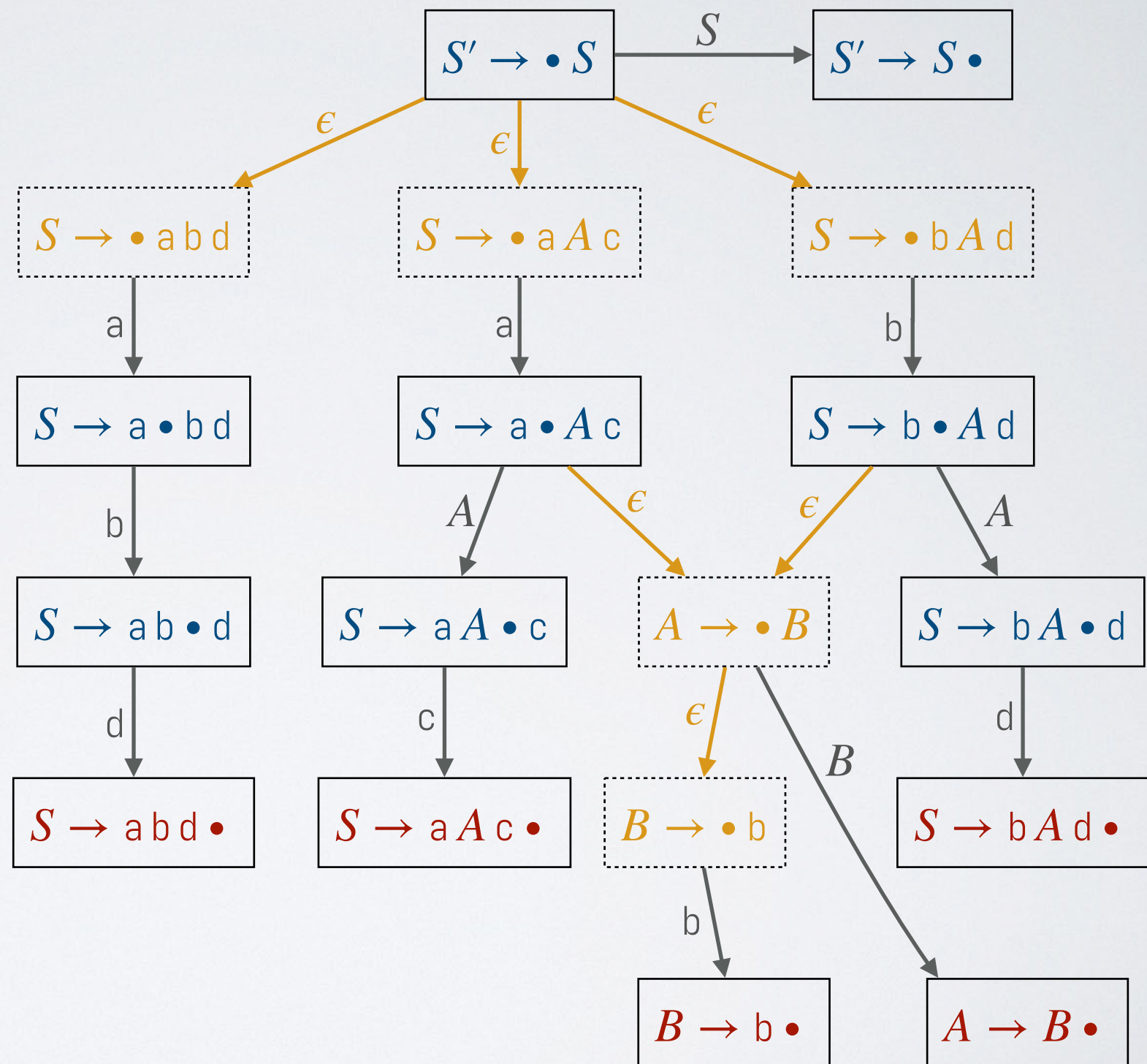
- ❖ 自动机与 LR(0) 相同
- ❖ 每个部分分析状态通过 FOLLOW 集合确定「再向前看」符号
- ❖ 前面已介绍过

◎ LALR(1), Look-Ahead LR

- ❖ 自动机与 LR(0) 同构, 但状态形式为 $\langle A \rightarrow \alpha \bullet \gamma; c \rangle$
- ❖ 可以认为是 LR(1) 的自动机把只有「再向前看」符号不同的状态进行合并

LALR(1) 示例

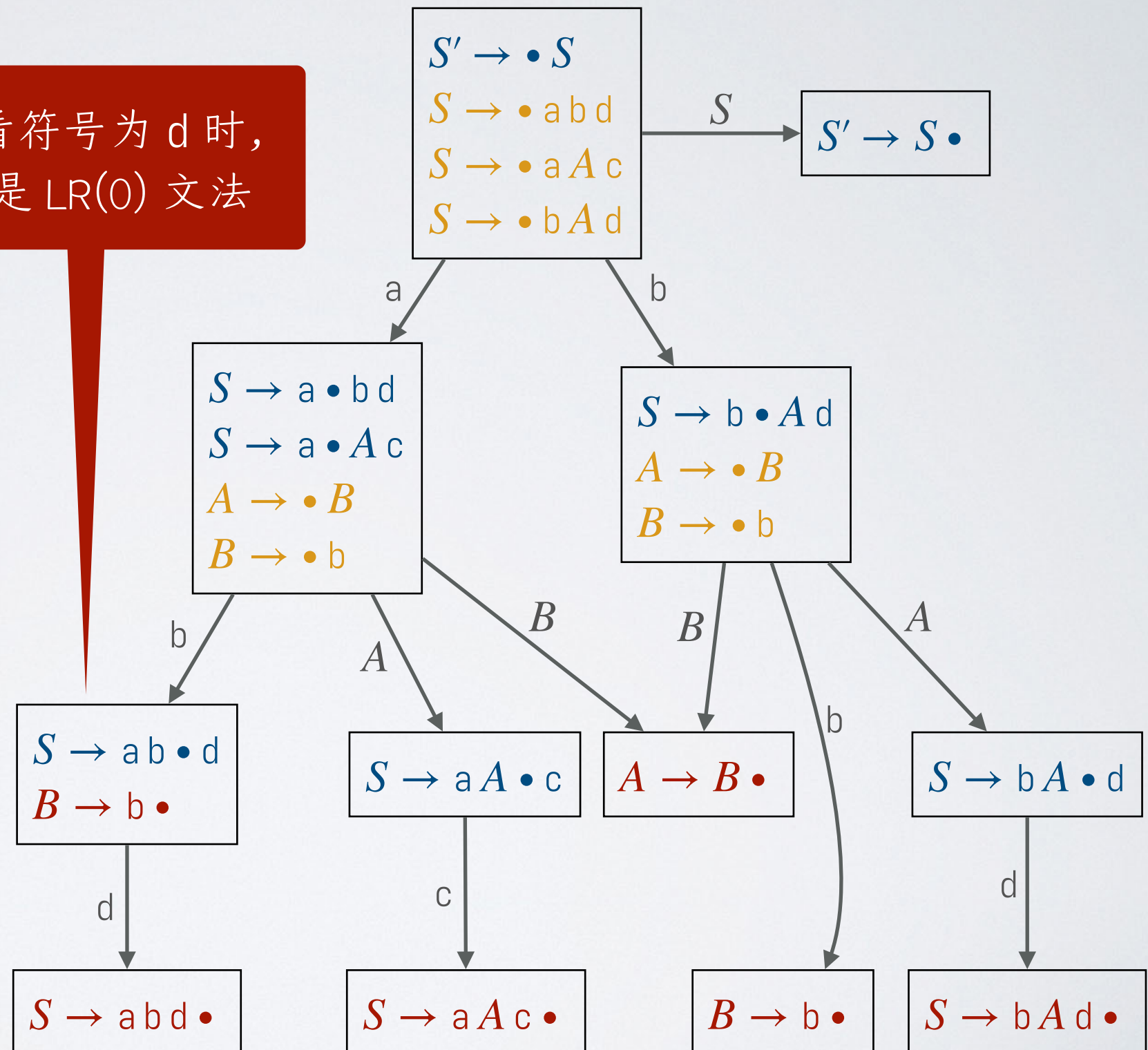
	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow a b d$
[2]	$S \rightarrow a A c$
[3]	$S \rightarrow b A d$
[4]	$A \rightarrow B$
[5]	$B \rightarrow b$



LALR(1) 示例

当分析栈「模式」为 ab 、向前看符号为 d 时，存在移进-归约冲突，所以不是 LR(0) 文法

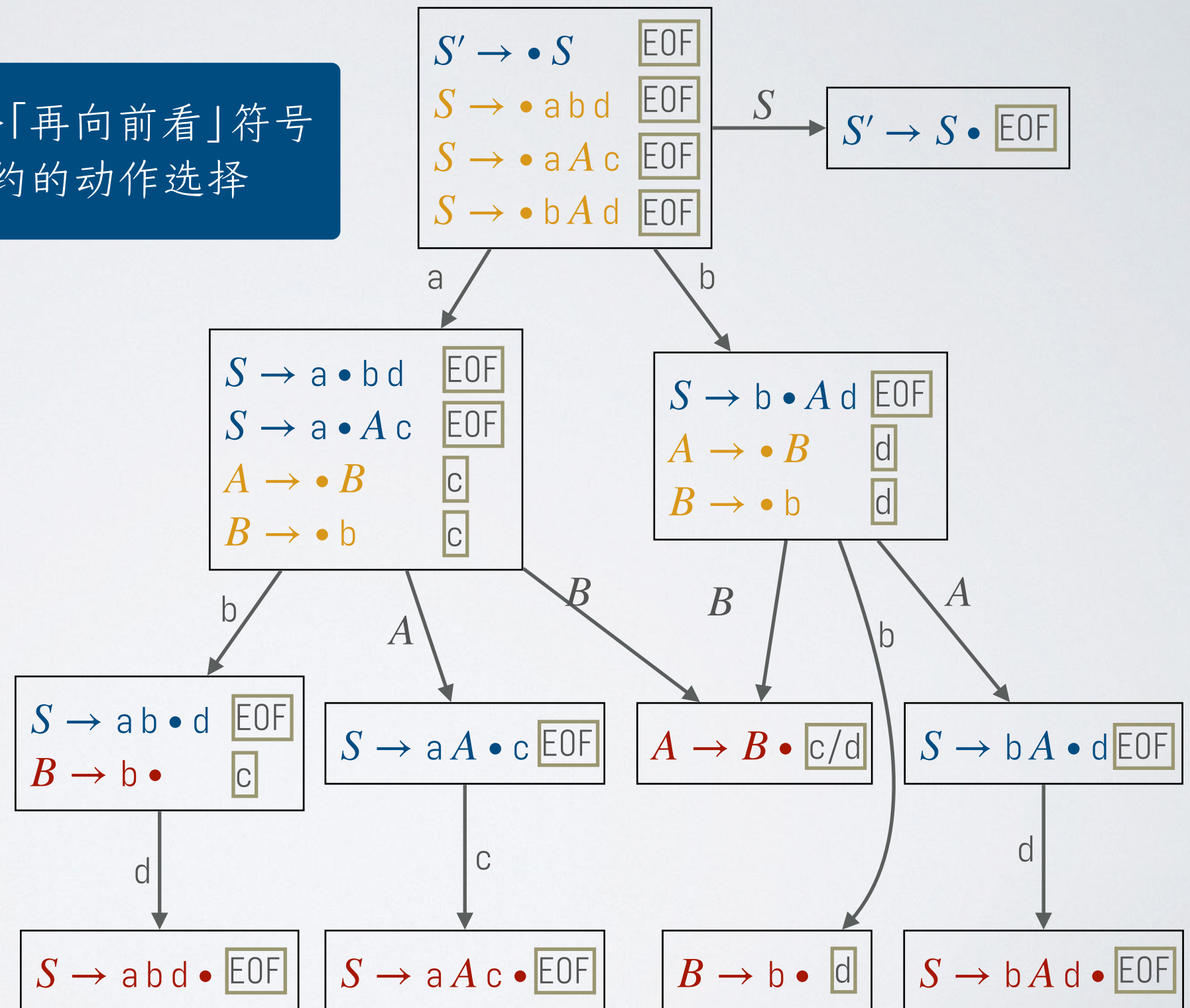
	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow a b d$
[2]	$S \rightarrow a A c$
[3]	$S \rightarrow b A d$
[4]	$A \rightarrow B$
[5]	$B \rightarrow b$



LALR(1) 示例

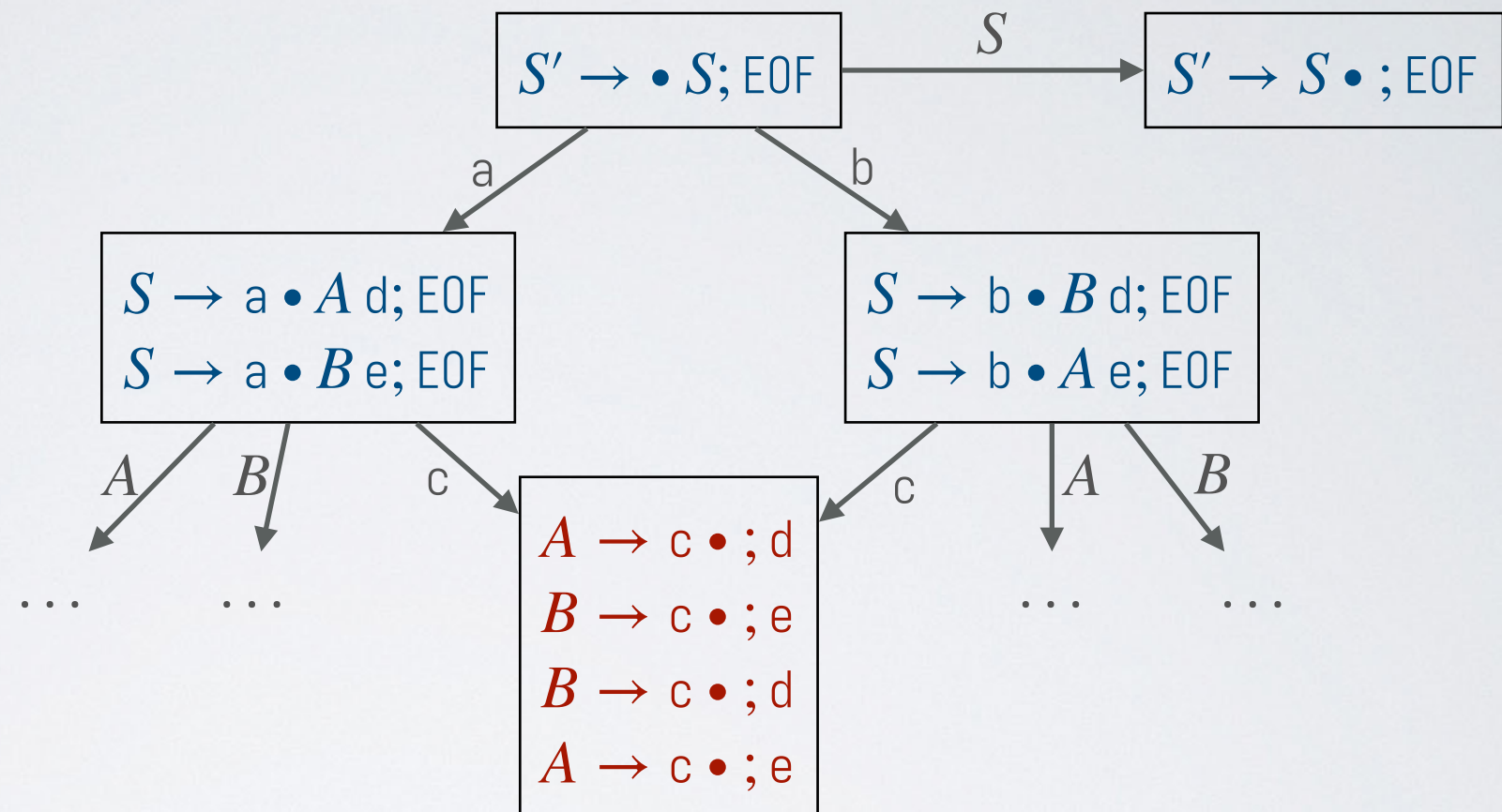
LALR(1) 通过一个「再向前看」符号
决定移进/归约的动作选择

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow a b d$
[2]	$S \rightarrow a A c$
[3]	$S \rightarrow b A d$
[4]	$A \rightarrow B$
[5]	$B \rightarrow b$



LALR 可能引入归约-归约冲突

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow a A d$
[2]	$S \rightarrow b B d$
[3]	$S \rightarrow a B e$
[4]	$S \rightarrow b A e$
[5]	$A \rightarrow c$
[6]	$B \rightarrow c$

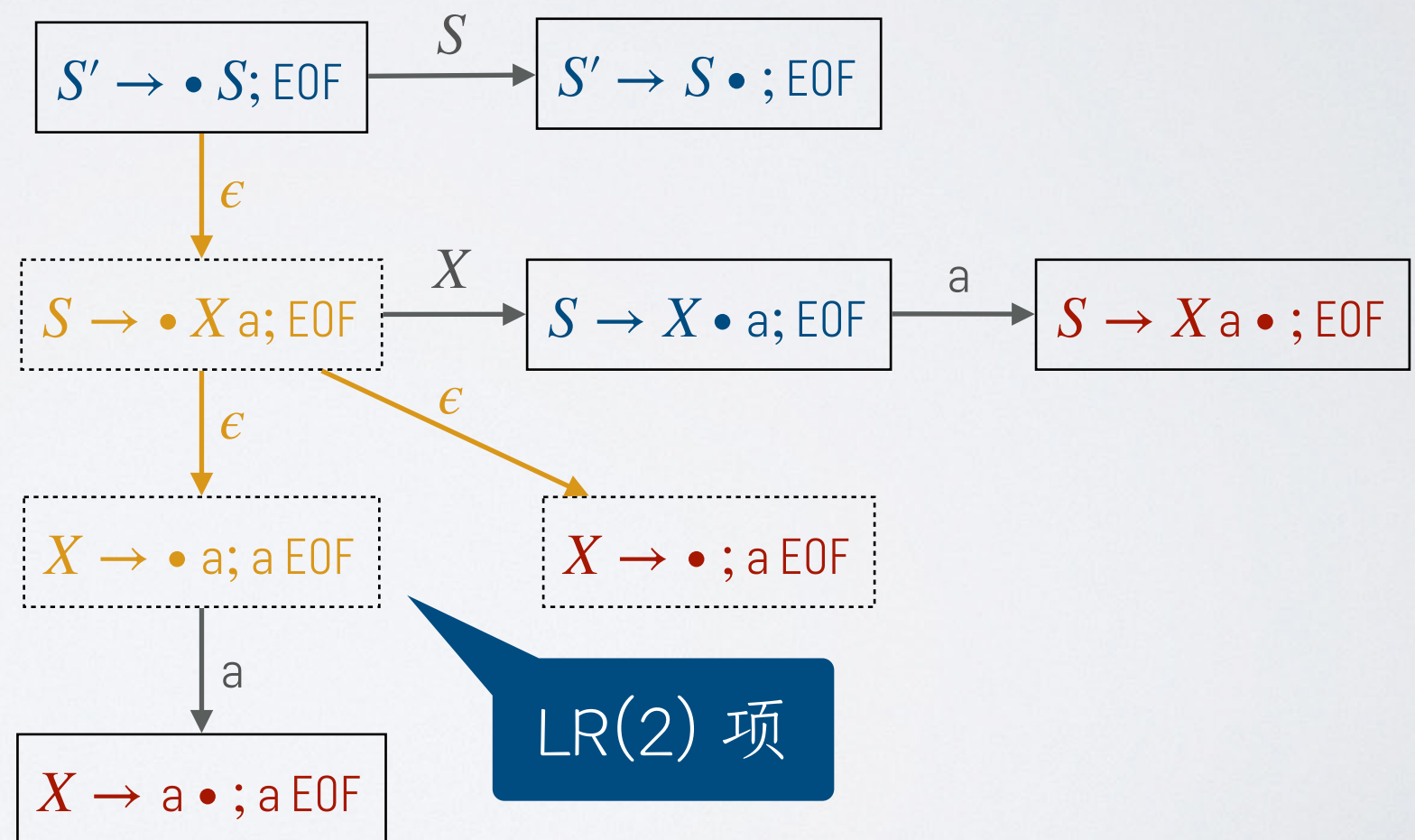


分析栈 \ 向前看	a	b	c	d	e	EOF
ϵ	移进	移进				
S						接受
ac 或 bc				归约 [5]/[6]???		

注意：「向前看」和「再向前看」

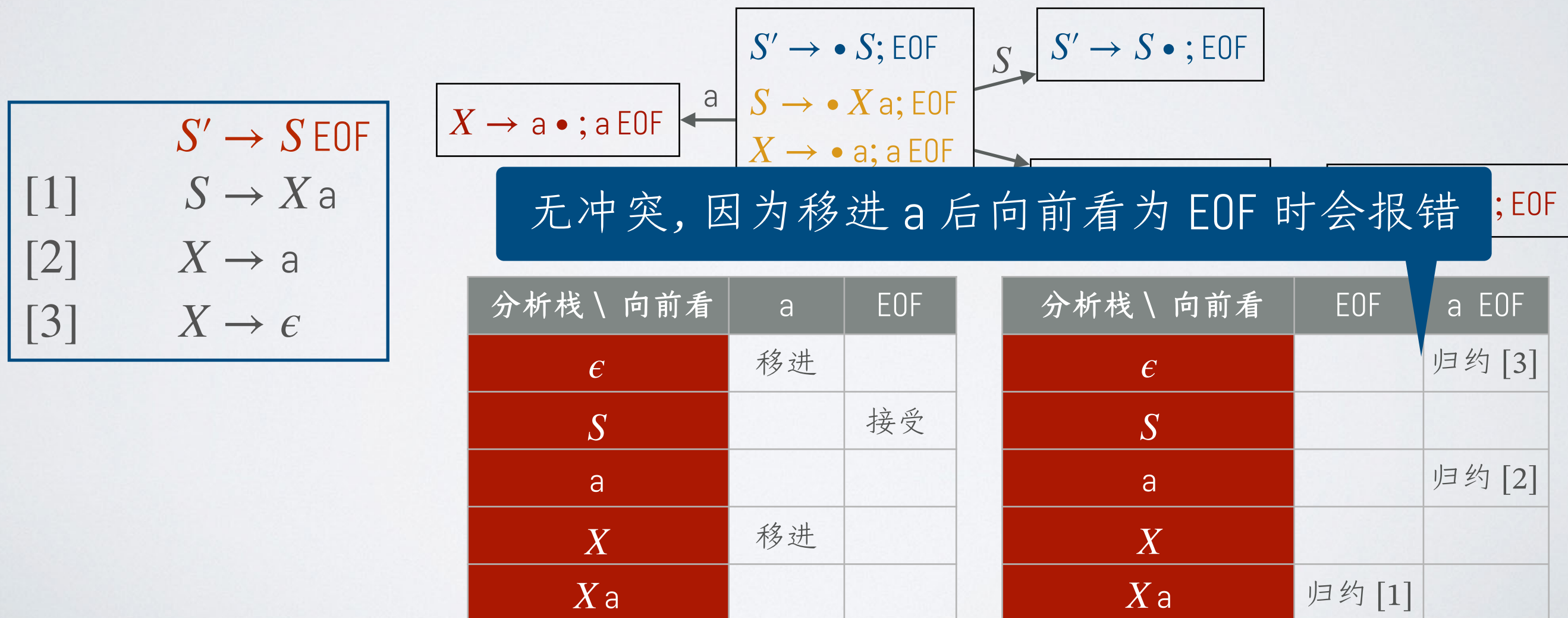
- **LL(k)**: 向前看 k 个符号, 确定产生规则
- **LR(k)**: 移进时, 向前看 1 个符号; 归约时, 向前看 k 个符号并与部分分析状态中的「再向前看」符号比较并确定归约规则

	$S' \rightarrow S EOF$
[1]	$S \rightarrow X a$
[2]	$X \rightarrow a$
[3]	$X \rightarrow \epsilon$



注意：「向前看」和「再向前看」

- ◎ **LL(k)**: 向前看 k 个符号, 确定产生规则
- ◎ **LR(k)**: 移进时, 向前看 1 个符号; 归约时, 向前看 k 个符号并与部分分析状态中的「再向前看」符号比较并确定归约规则

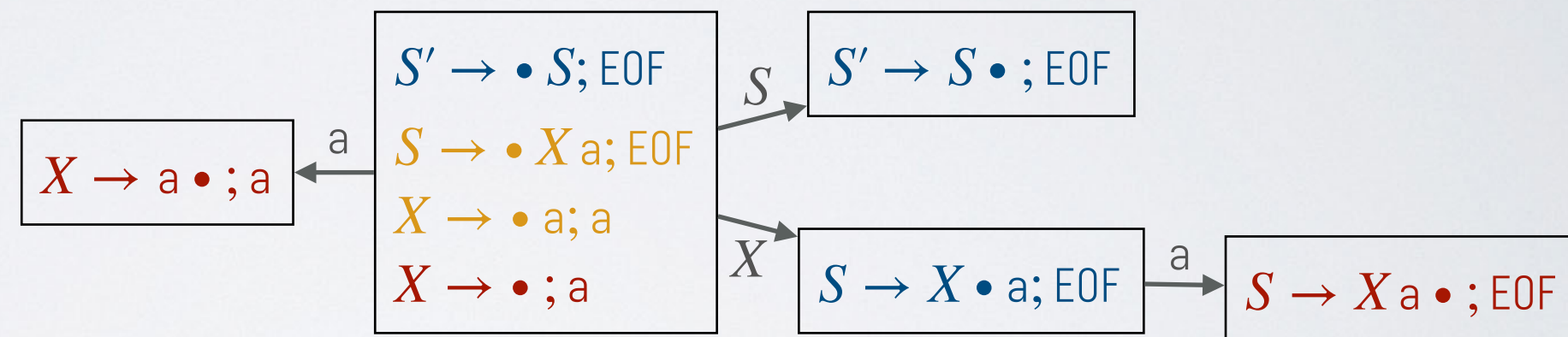


注意：「向前看」和「再向前看」

- ◎ **LL(k)**: 向前看 k 个符号, 确定产生规则
- ◎ **LR(k)**: 移进时, 向前看 1 个符号; 归约时, 向前看 k 个符号并与部分分析状态中的「再向前看」符号比较并确定归约规则

非 LR(1) 文法

$S' \rightarrow S EOF$
 [1] $S \rightarrow X a$
 [2] $X \rightarrow a$
 [3] $X \rightarrow \epsilon$



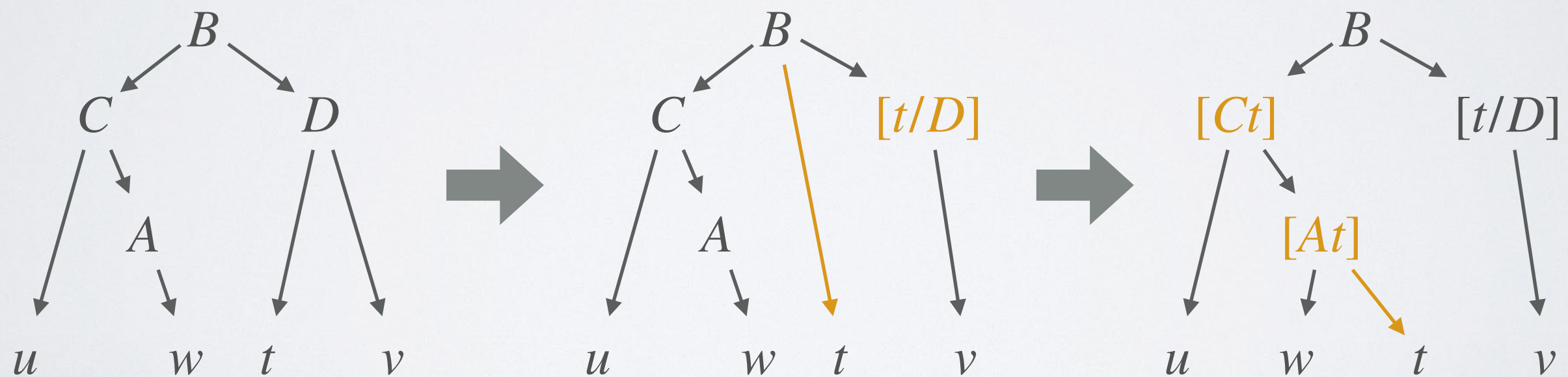
分析栈 \ 向前看	a	EOF
ϵ	移进/归约 [3]	
S		接受
a	归约 [2]	
X	移进	
$X a$		归约 [1]

但这个语言明显可以
写一个 LR(1) 文法:

$S ::= a a \mid a$

LR(k) 的「能力」不超过 LR(1)

- 定理: 对于一个能用 LR(k) 文法识别的语言, 一定存在一个 LR(1) 文法识别相同的语言 [MLS76]
- 思路: 如果 $A \rightarrow w$ 是造成非 LR(1) 的产生规则, 比如需要 2 个向前看符号 t, v 来产生一个 $A \Leftarrow w$ 的归约, 那么可以对 t 进行 **right-context extraction** 和 **premature scanning**



[MLS76] M. D. Mickunas, R. L. Lancaster, and V. B. Schneider. Transforming LR(k) Grammars to LR(1), SLR(1), and (1,1) Bounded Right-Context Grammars. J. ACM, 23(3):511-533, 1976.

LR(k) 的「能力」不超过 LR(1)

$X \rightarrow \epsilon$ 是导致非 LR(1) 的问题来源

[1] $S \rightarrow X a$
[2] $X \rightarrow a$
[3] $X \rightarrow \epsilon$

对 a 应用
premature scanning

[1] $S \rightarrow [X a]$
[2] $[X a] \rightarrow a a$
[3] $[X a] \rightarrow a$

$A \rightarrow a, B \rightarrow a$ 是导致非 LR(1) 的问题来源

[1] $S \rightarrow A b b$
[2] $S \rightarrow B b c$
[3] $A \rightarrow A a$
[4] $A \rightarrow a$
[5] $B \rightarrow a$

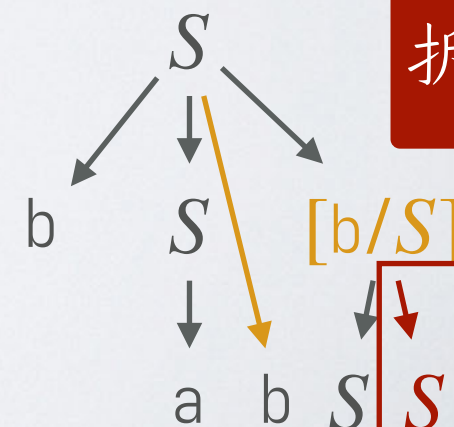
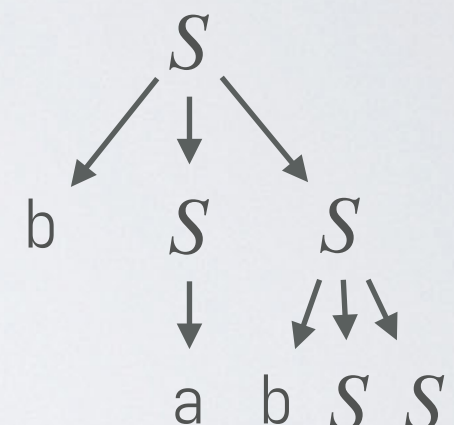
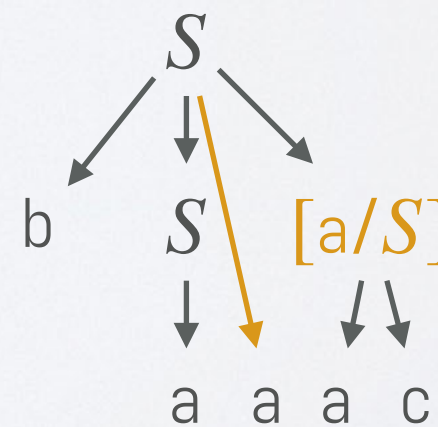
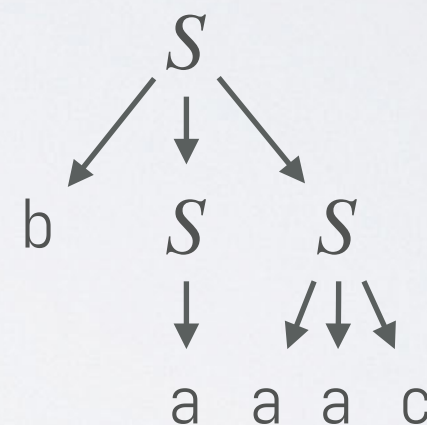
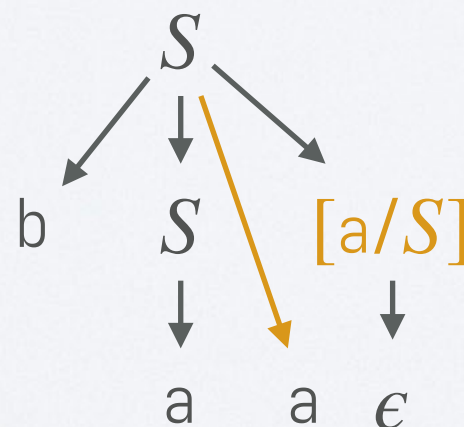
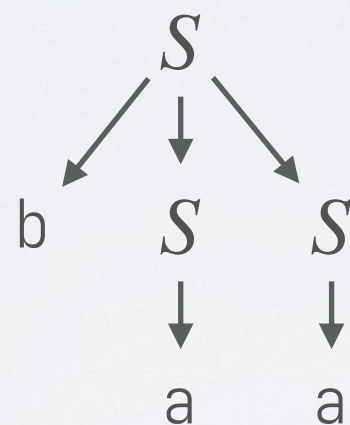
对 b 应用 pre. scan.
对 b 应用 pre. scan.
对 a 应用 pre. scan.

[1] $S \rightarrow [A b] b$
[2] $S \rightarrow [B b] c$
[3] $[A b] \rightarrow [A a] b$
[4] $[A b] \rightarrow a b$
[5] $[A a] \rightarrow [A a] a$
[6] $[A a] \rightarrow a a$
[7] $[B b] \rightarrow a b$

LR(k) 的「能力」不超过 LR(1)

$S \rightarrow a$ 是导致非 LR(1) 的问题来源

对第二个 S 应用
right-context
extraction

$$\begin{array}{ll} [1] & S \rightarrow b S S \\ [2] & S \rightarrow a \\ [3] & S \rightarrow a a c \end{array}$$
$$\begin{aligned} S &\rightarrow b S a [a/S] \\ S &\rightarrow b S b [b/S] \\ S &\rightarrow a \\ S &\rightarrow a a c \\ [a/S] &\rightarrow \epsilon \\ [a/S] &\rightarrow a c \\ [b/S] &\rightarrow S a [a/S] \\ [b/S] &\rightarrow S b [b/S] \end{aligned}$$


也要
拆分

LR(k) 的「能力」不超过 LR(1)

$S \rightarrow a$ 是导致非 LR(1) 的问题来源

- [1] $S \rightarrow b S S$
- [2] $S \rightarrow a$
- [3] $S \rightarrow a a c$

对 a 应用
premature scanning

$S \rightarrow b \boxed{S a} [a/S]$
 $S \rightarrow b S b [b/S]$
 $S \rightarrow a$
 $S \rightarrow a a c$
 $[a/S] \rightarrow \epsilon$
 $[a/S] \rightarrow a c$
 $[b/S] \rightarrow \boxed{S a} [a/S]$
 $[b/S] \rightarrow S b [b/S]$

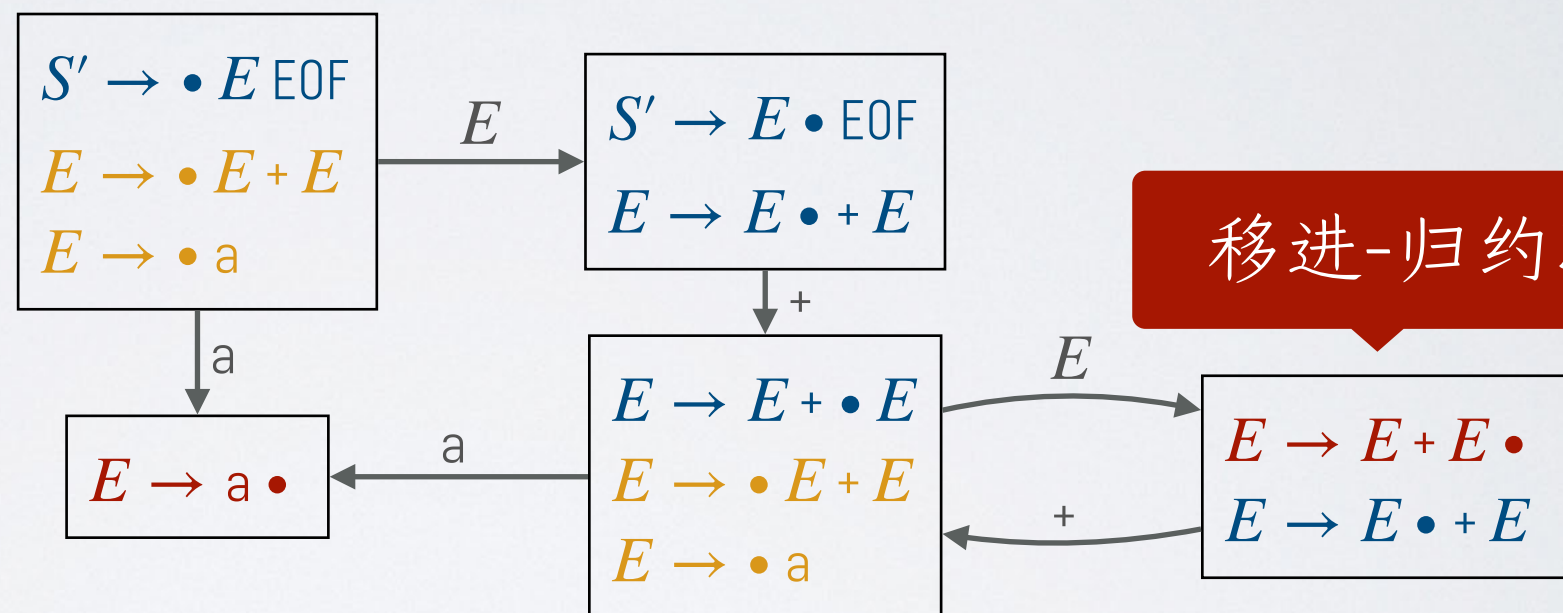
$S ::= b [S a] [a/S] \mid b S b [b/S] \mid a \mid a a c$
 $[S a] ::= b [S a] [\boxed{[a/S] a}] \mid b S b [\boxed{[b/S] a}] \mid a a \mid a a c a$
 $[a/S] ::= \epsilon \mid a c$
 $\boxed{[a/S] a} ::= a \mid a c a$
 $[b/S] ::= [S a] [a/S] \mid S b [b/S]$
 $\boxed{[b/S] a} ::= [S a] [\boxed{[a/S] a}] \mid S b [\boxed{[b/S] a}]$

Generalized LR (GLR)

- 当无法决定应该移进或归约时, 进行**广度优先搜索**
 - 进入某个部分分析状态时, 为每种可能的归约创建一个「复制」
 - 考虑创建出的所有「复制」(包括不归约的), 去掉不能进行移进的

二义性文法

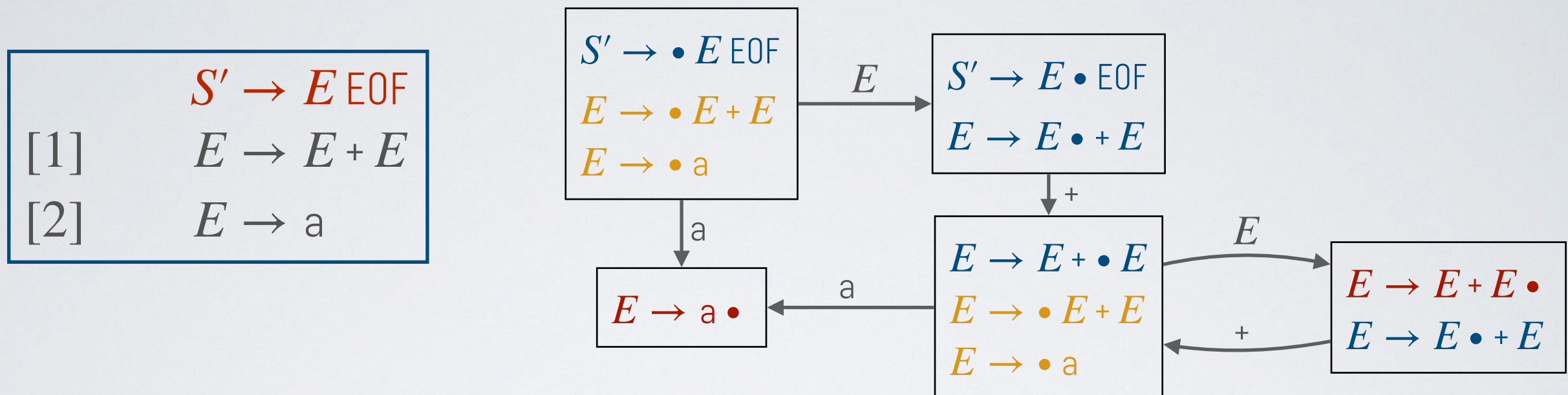
[1] $S' \rightarrow E \text{ EOF}$
 $E \rightarrow E + E$
 [2] $E \rightarrow a$



移进-归约冲突

向前看 \ 分析栈	ϵ	E	βa	$\beta E +$	$\beta E + E$
a	移进		归约 [2]	移进	归约 [1]
+		移进	归约 [2]		移进/归约
EOF		接受	归约 [2]		归约 [1]

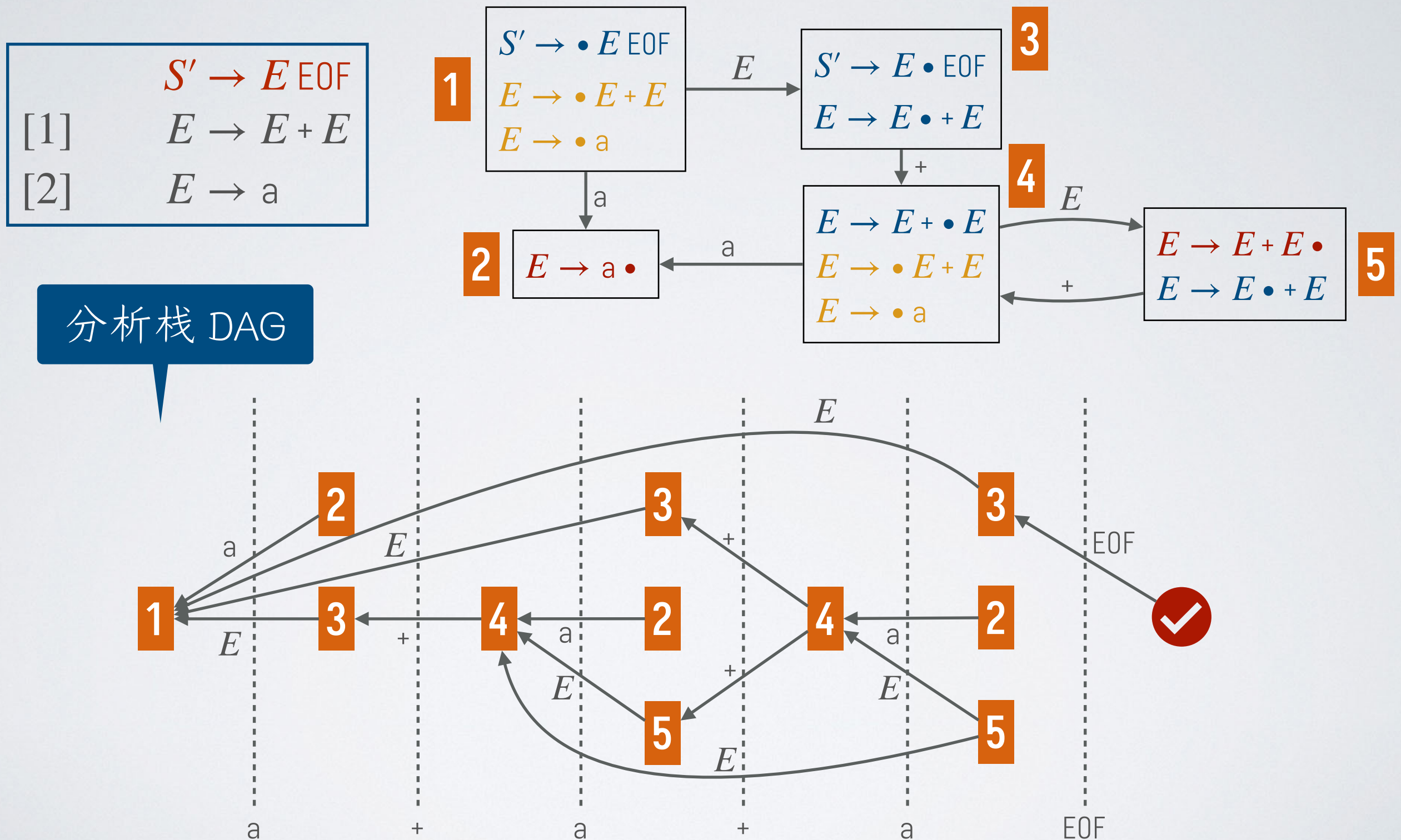
Generalized LR (GLR)



哪些状态是等价的？



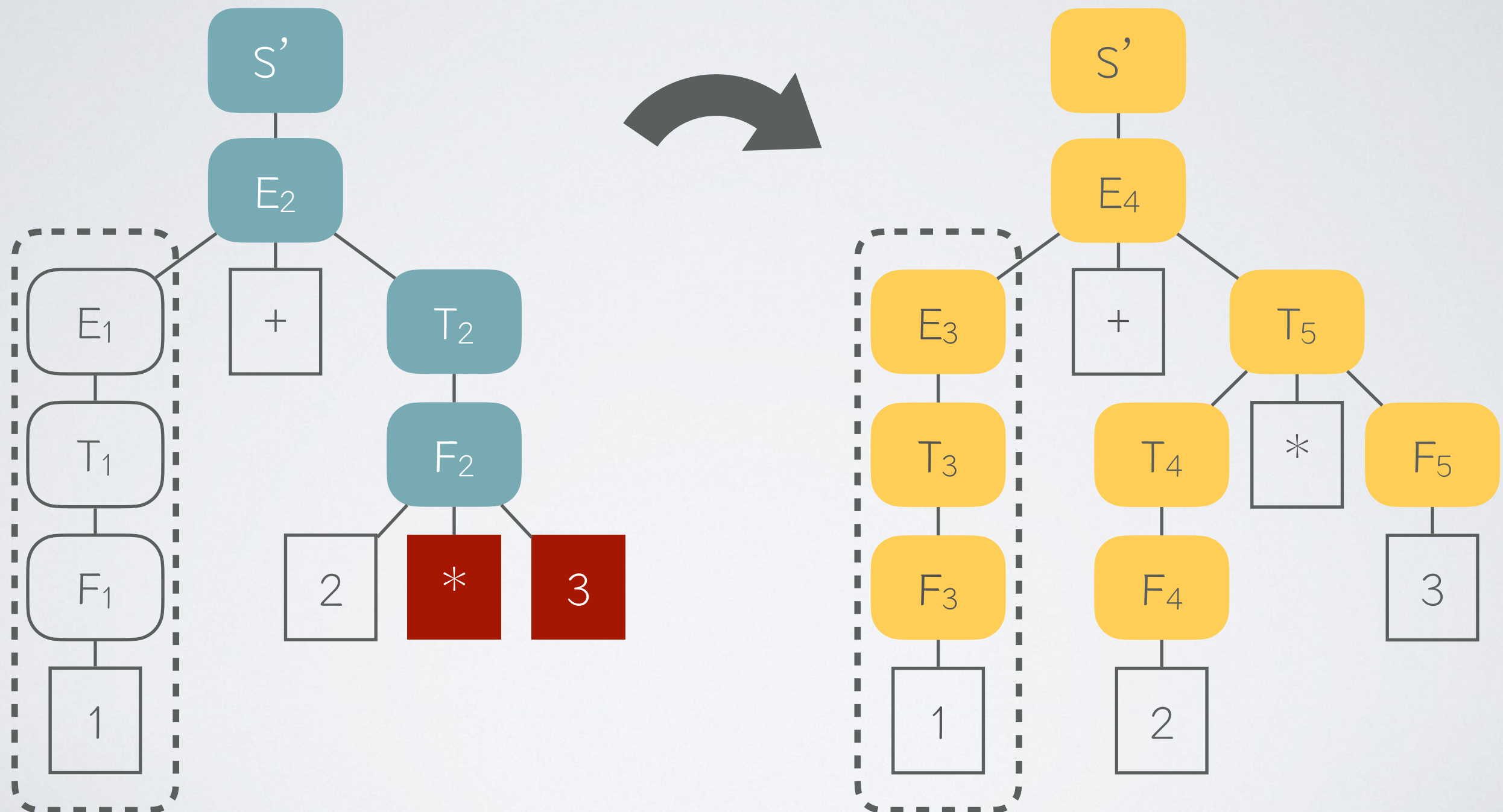
Generalized LR (GLR)

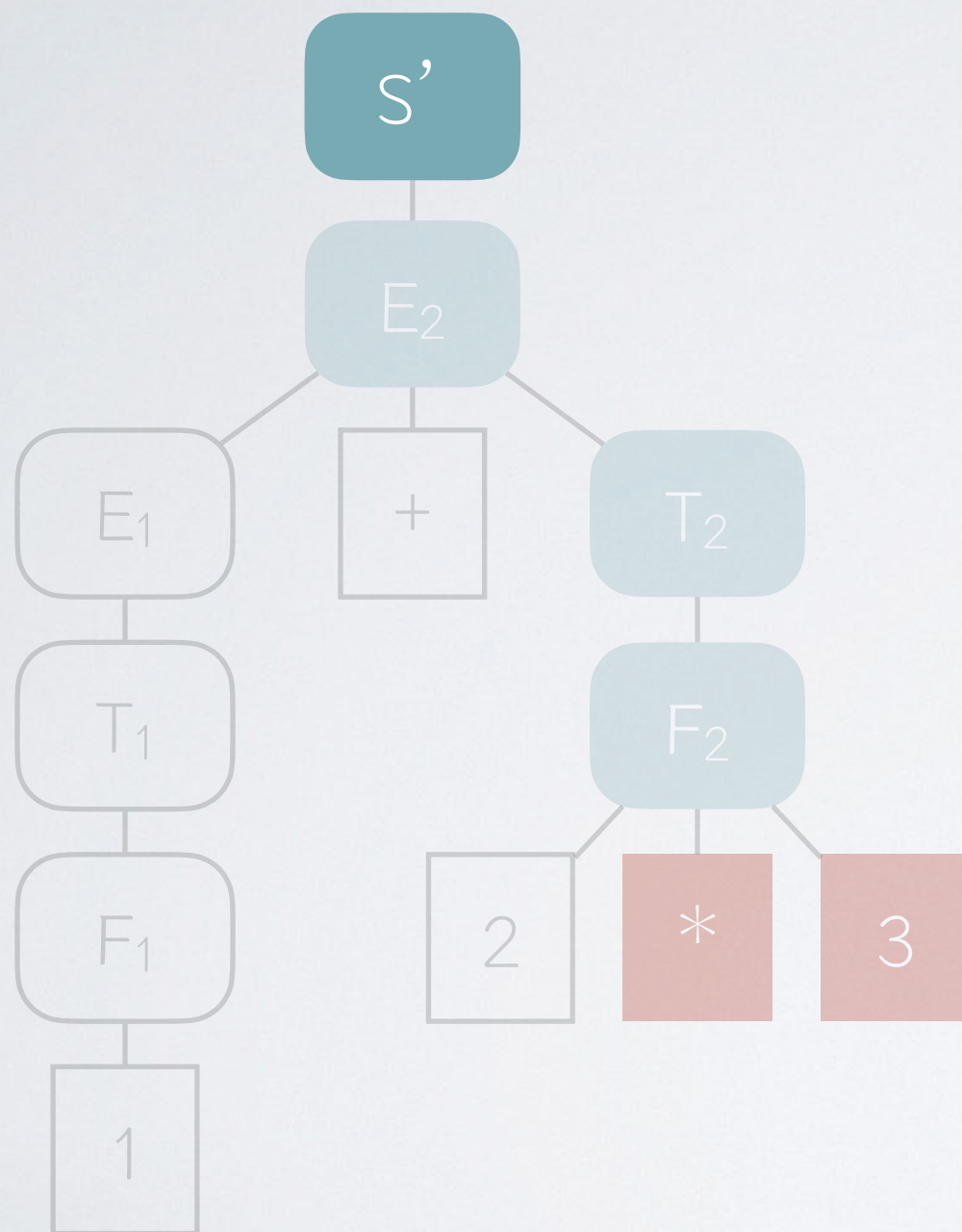


增量式语法分析

- ◎ 问题：IDE 中，是否每敲一个字符都要**重新进行语法分析**？
- ◎ 常用的语法分析算法时间复杂度是线性的
- ◎ GitHub：我每天要处理的代码太多了！
- ◎ **增量式语法分析**
 - ❖ Incremental Parsing
 - ❖ 重新分析时尽量**只分析代码的增/减量**
 - ❖ 基于编辑历史的错误信息生成
- ◎ GitHub 推出 tree-sitter 工具：自动生成增量式分析器

LR(1) 文法的增量式分析



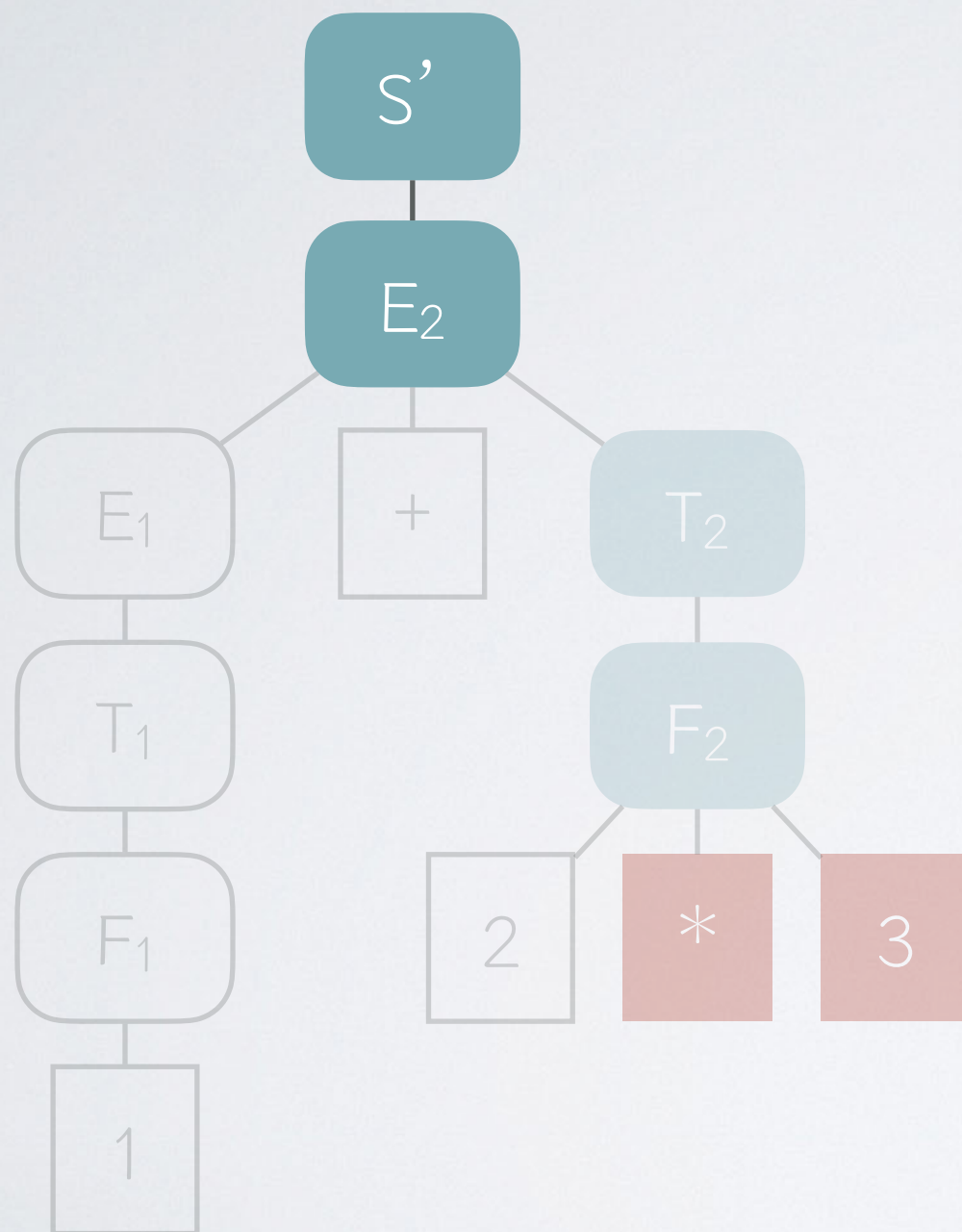


栈

输入

S' EOF

E_2 EOF



栈

输入

S' EOF

E_2 EOF

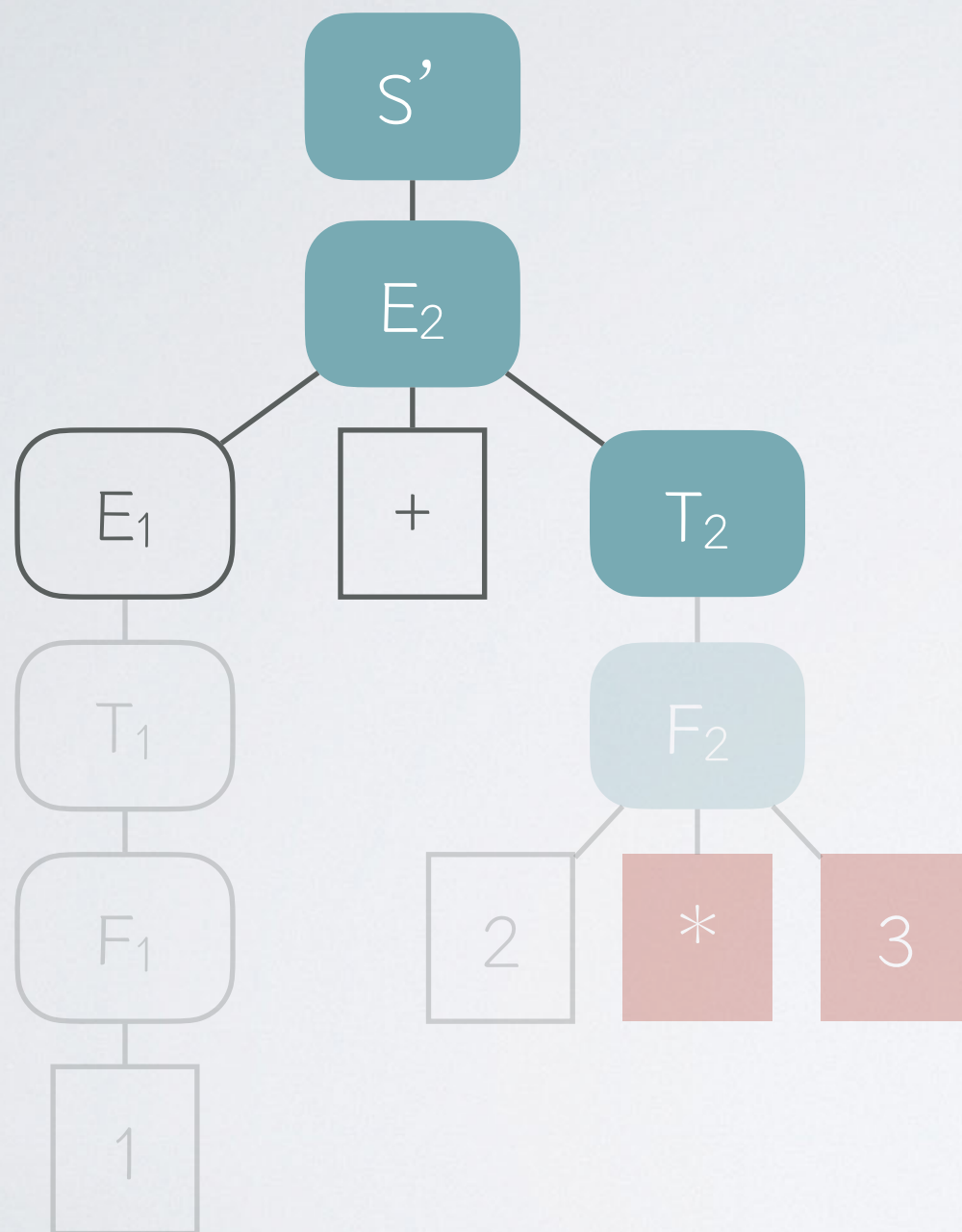
$E_1 + T_2$ EOF

$+ T_2$ EOF

T_2 EOF

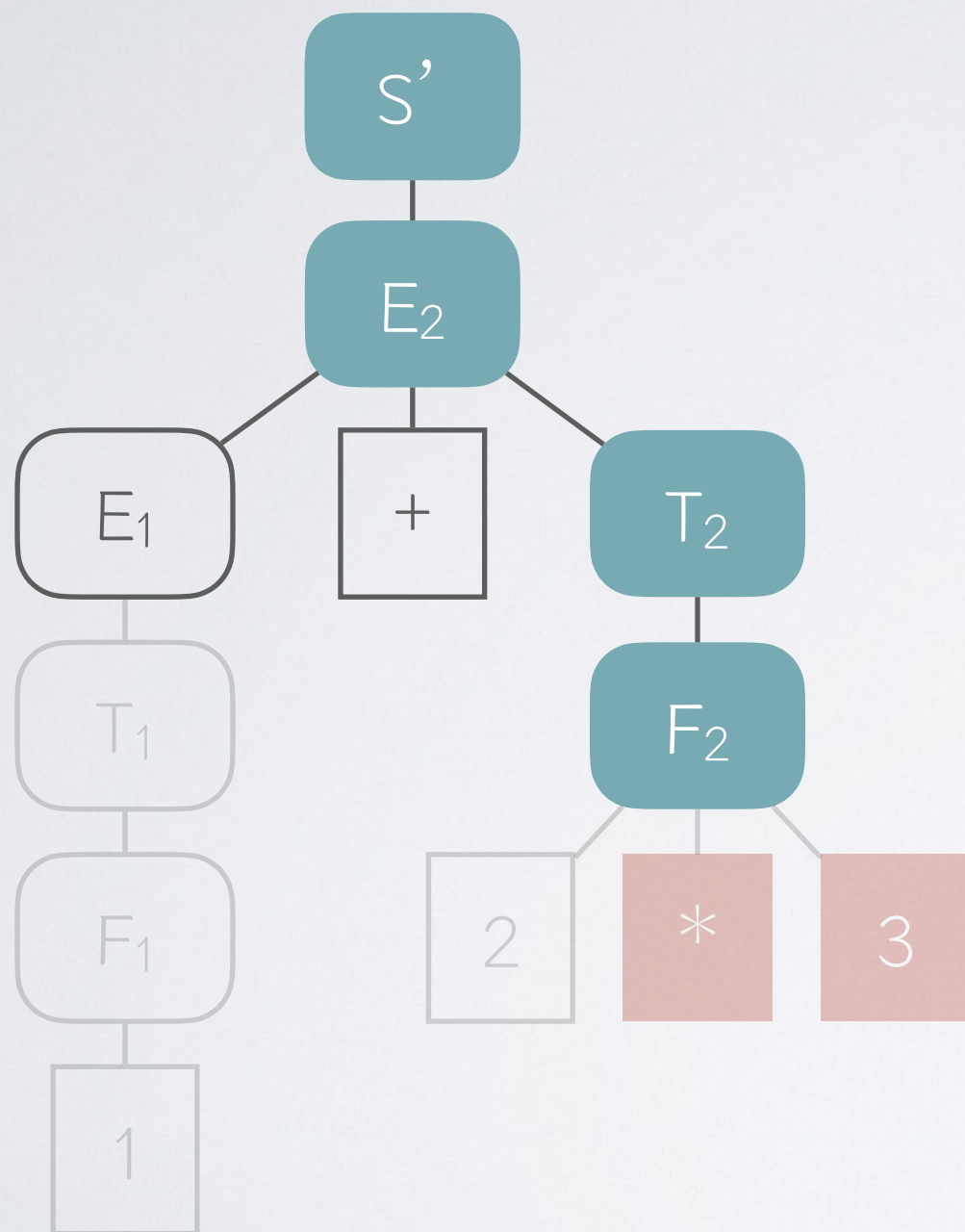
E_1

$E_1 +$



栈

输入



E_1

$E_1 +$

$E_1 +$

S' EOF

E_2 EOF

$E_1 + T_2$ EOF

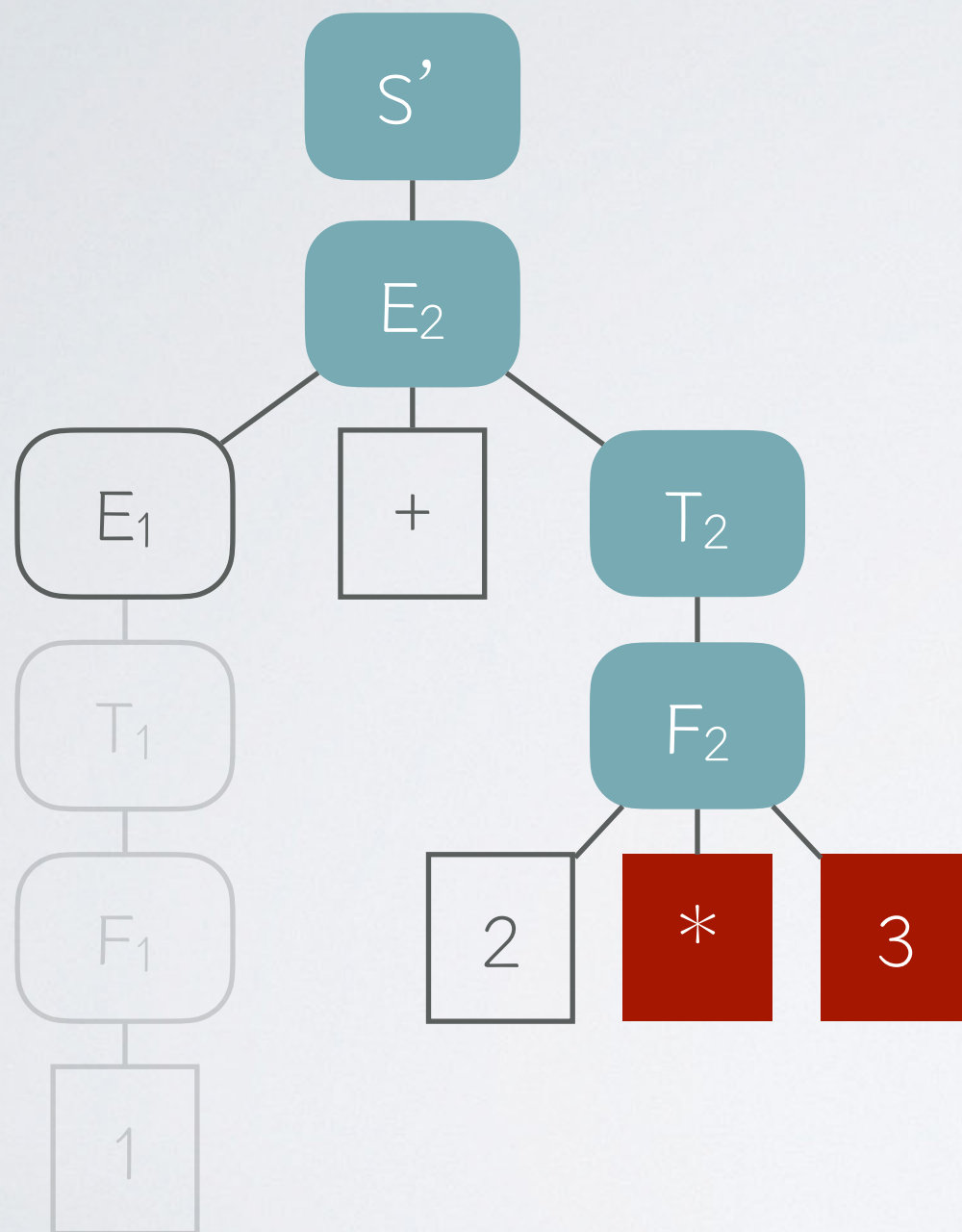
$+ T_2$ EOF

T_2 EOF

F_2 EOF

栈

输入



E_1

$E_1 +$

$E_1 +$

$E_1 +$

$E_1 + 2$

S' EOF

E_2 EOF

$E_1 + T_2$ EOF

$+ T_2$ EOF

T_2 EOF

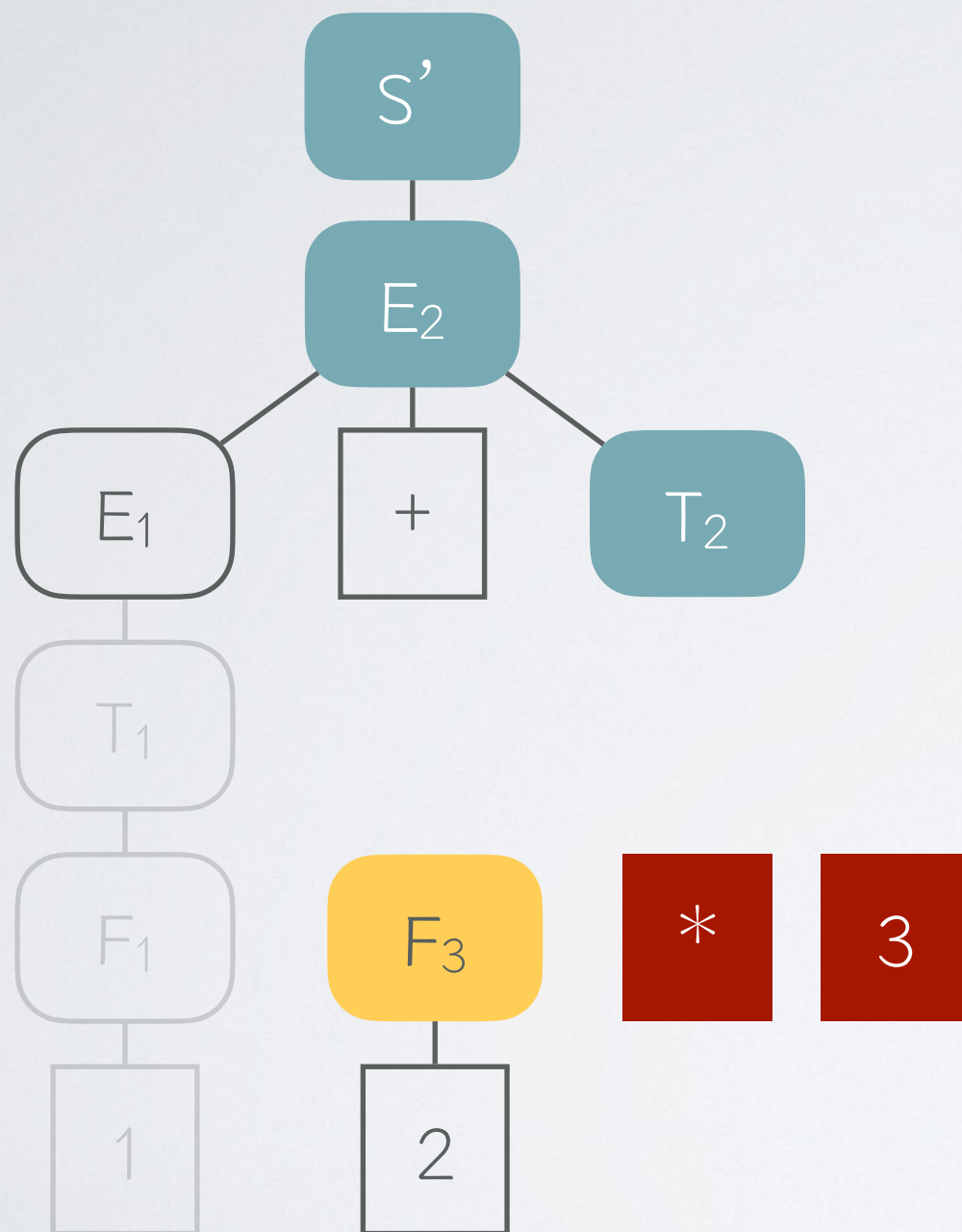
F_2 EOF

$2 * 3$ EOF

$* 3$ EOF

栈

输入



E_1

$E_1 +$

$E_1 +$

$E_1 +$

$E_1 + 2$

$E_1 + F_3$

S' EOF

E_2 EOF

$E_1 + T_2$ EOF

$+ T_2$ EOF

T_2 EOF

F_2 EOF

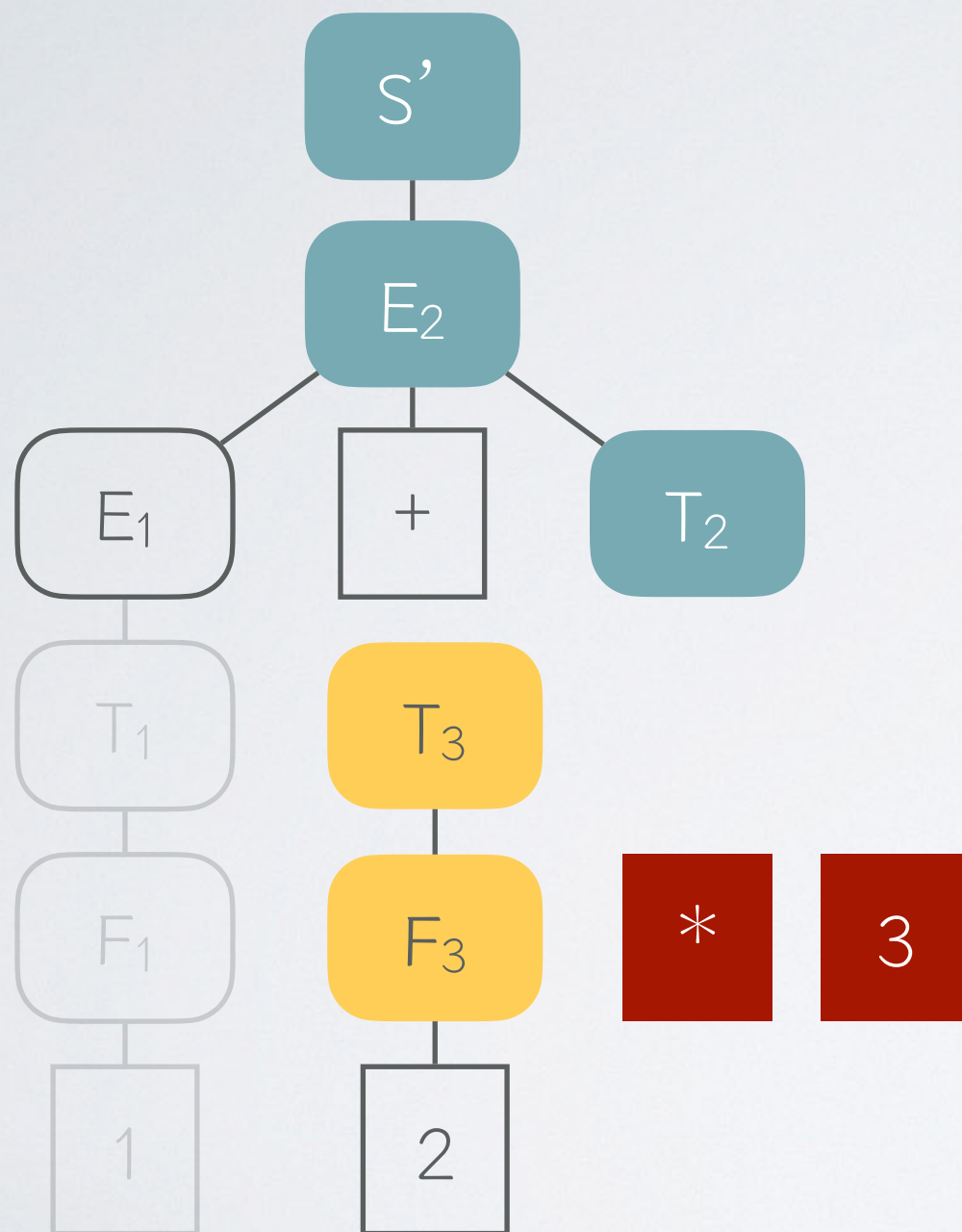
$2 * 3$ EOF

$* 3$ EOF

$* 3$ EOF

栈

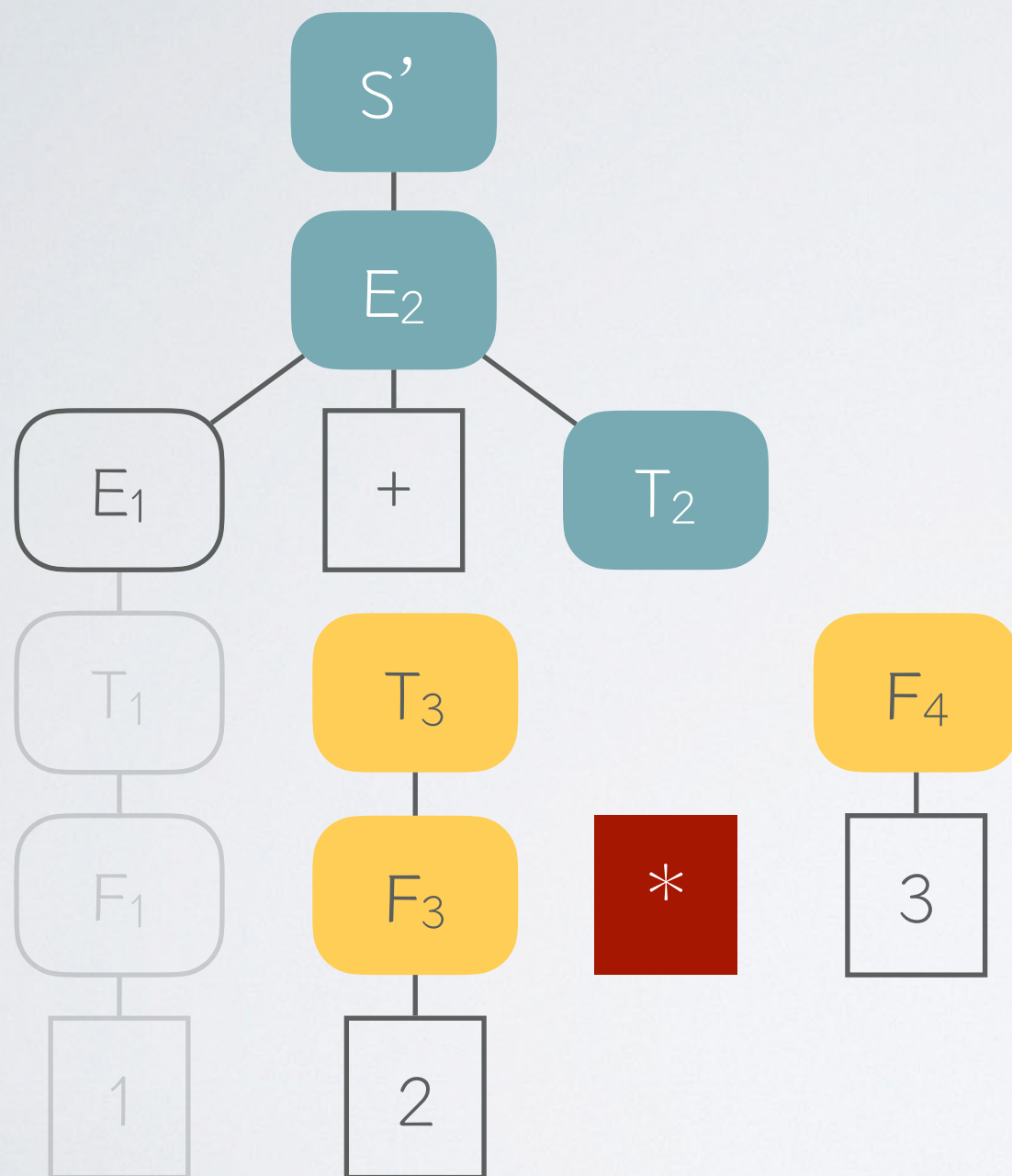
输入



	S' EOF
	E_2 EOF
	$E_1 + T_2$ EOF
E_1	$+ T_2$ EOF
$E_1 +$	T_2 EOF
$E_1 +$	F_2 EOF
$E_1 +$	$2 * 3$ EOF
$E_1 + 2$	$* 3$ EOF
$E_1 + F_3$	$* 3$ EOF
$E_1 + T_3$	$* 3$ EOF
$E_1 + T_3 *$	3 EOF
$E_1 + T_3 * 3$	EOF

栈

输入

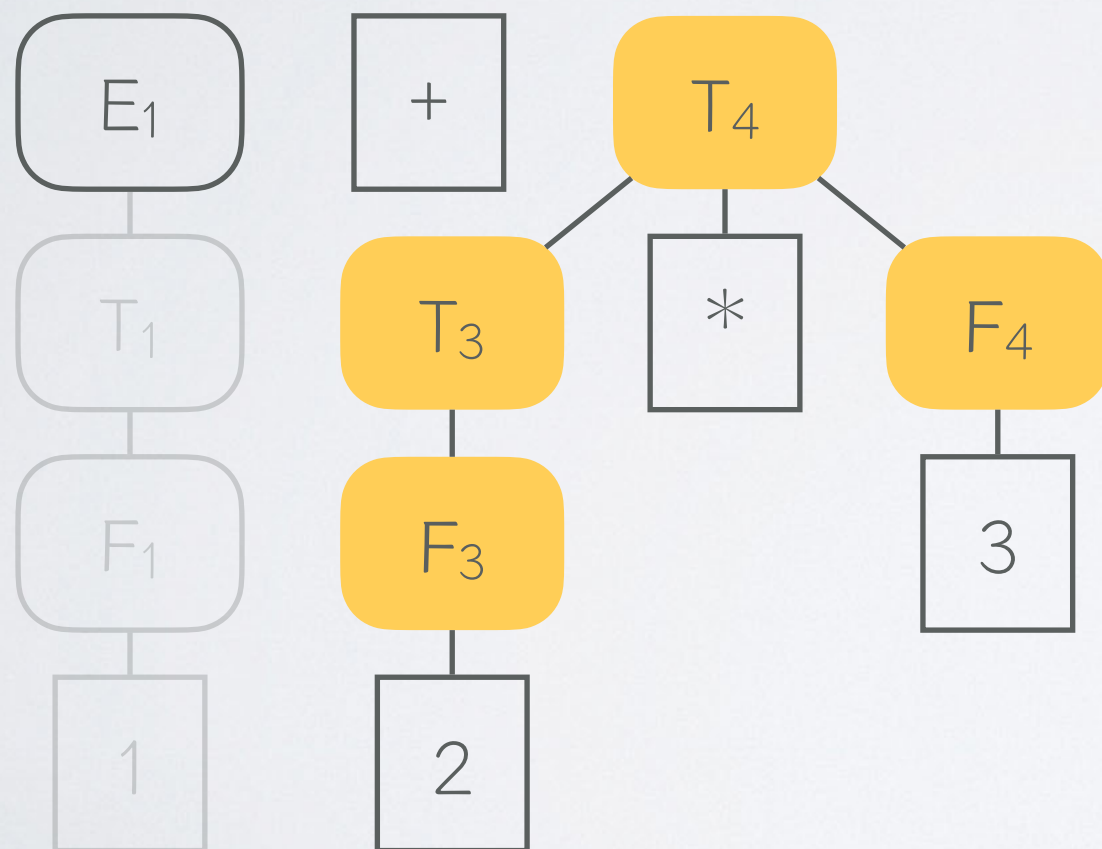


	S' EOF
	E_2 EOF
	$E_1 + T_2$ EOF
E_1	$+ T_2$ EOF
$E_1 +$	T_2 EOF
$E_1 +$	F_2 EOF
$E_1 +$	$2 * 3$ EOF
$E_1 + 2$	$* 3$ EOF
$E_1 + F_3$	$* 3$ EOF
$E_1 + T_3$	$* 3$ EOF
$E_1 + T_3 *$	3 EOF
$E_1 + T_3 * 3$	EOF
$E_1 + T_3 * F_4$	EOF

栈

输入

S'



S' EOF

E_2 EOF

$E_1 + T_2$ EOF

$+ T_2$ EOF

T_2 EOF

F_2 EOF

$2 * 3$ EOF

$* 3$ EOF

$* 3$ EOF

$* 3$ EOF

3 EOF

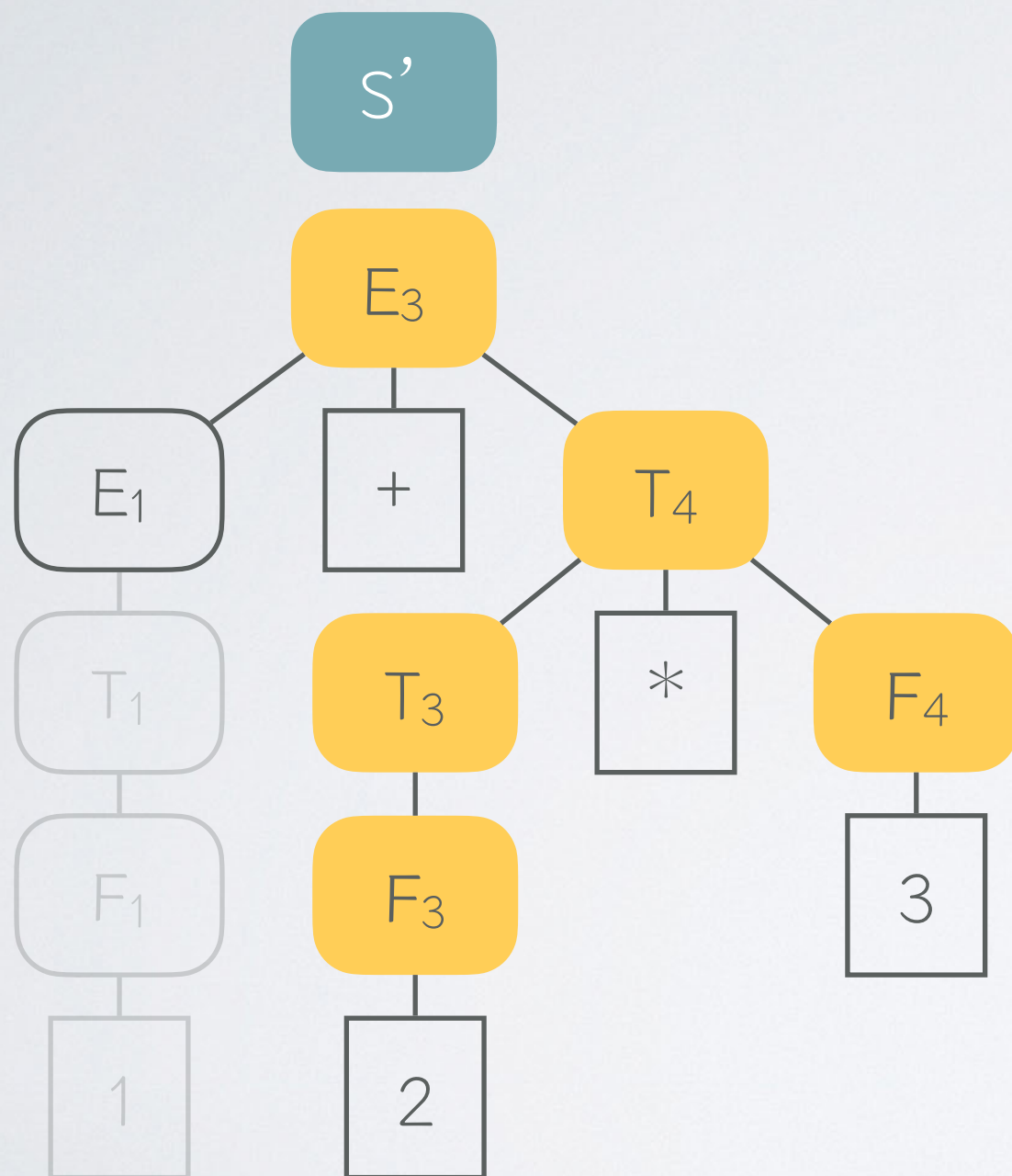
EOF

EOF

EOF

栈

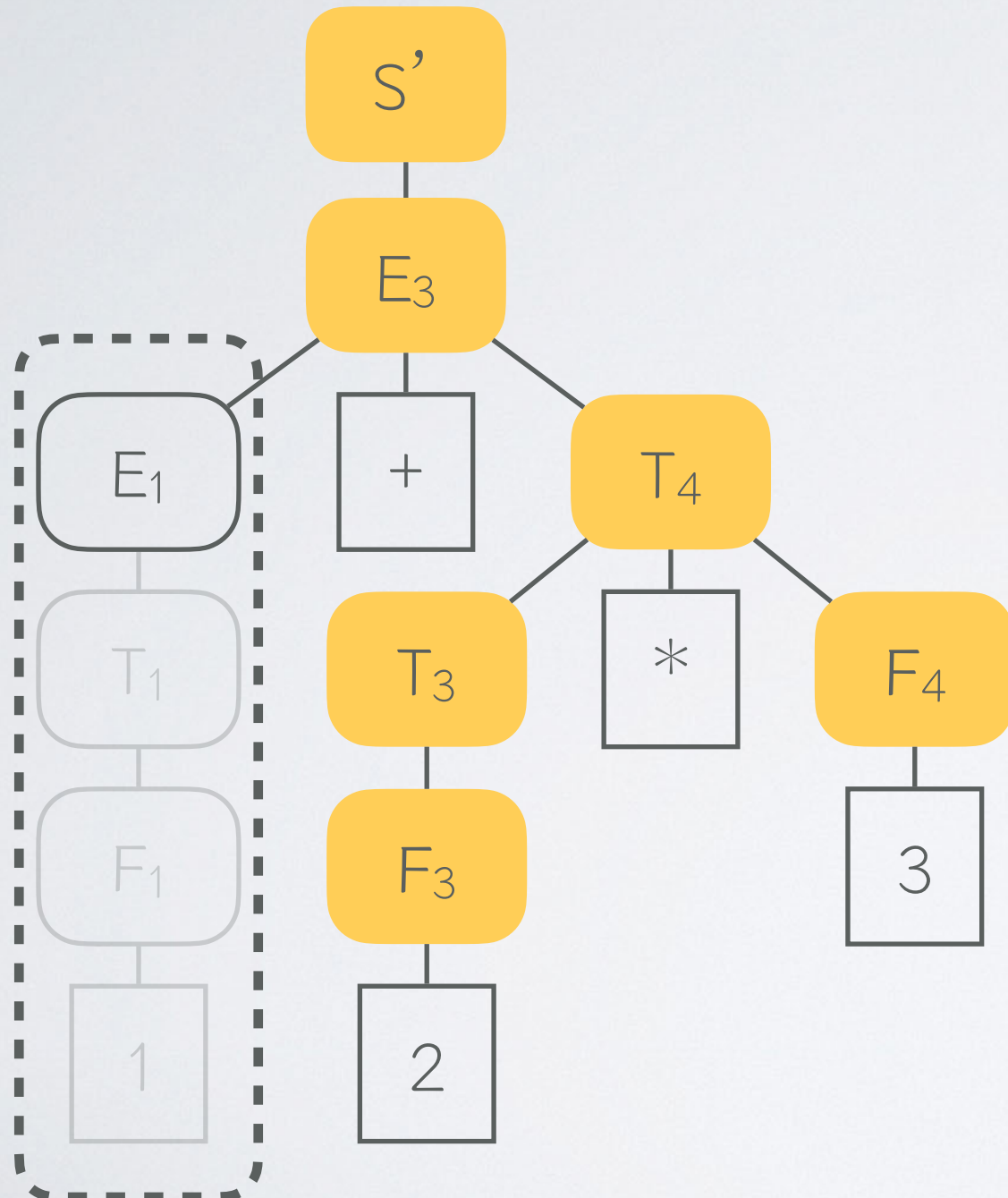
输入



	S'	EOF
	E_3	EOF
	$E_1 + T_2$	EOF
E_1	$+ T_2$	EOF
$E_1 +$	T_2	EOF
$E_1 +$	F_2	EOF
$E_1 +$	$2 * 3$	EOF
$E_1 + 2$	$* 3$	EOF
$E_1 + F_3$	$* 3$	EOF
$E_1 + T_3$	$* 3$	EOF
$E_1 + T_3 *$	3	EOF
$E_1 + T_3 * 3$		EOF
$E_1 + T_3 * F_4$		EOF
$E_1 + T_4$		EOF
E_3		EOF

栈

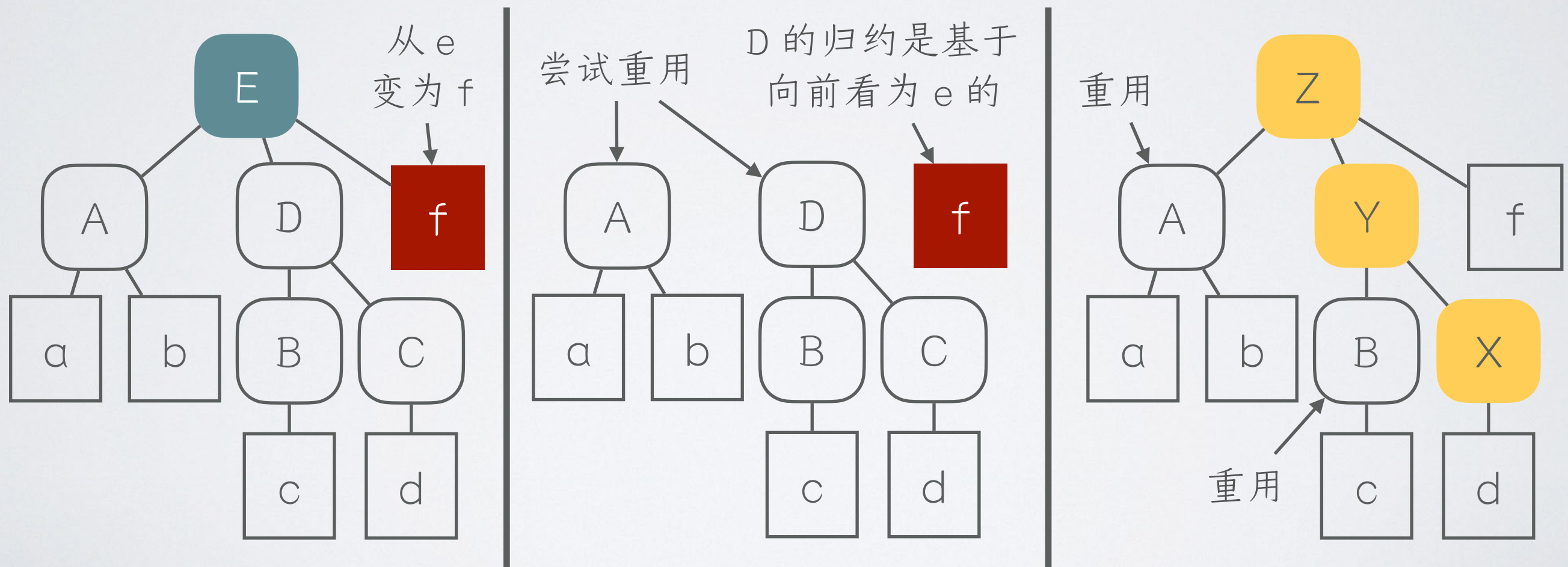
输入



	S'	EOF
	E_3	EOF
	$E_1 + T_2$	EOF
E_1	$+ T_2$	EOF
$E_1 +$	T_2	EOF
$E_1 +$	F_2	EOF
$E_1 +$	$2 * 3$	EOF
$E_1 + 2$	$* 3$	EOF
$E_1 + F_3$	$* 3$	EOF
$E_1 + T_3$	$* 3$	EOF
$E_1 + T_3 *$	3	EOF
$E_1 + T_3 * 3$		EOF
$E_1 + T_3 * F_4$		EOF
$E_1 + T_4$		EOF
E_3		EOF
S'		EOF

LR(1) 文法的增量式分析

- 保留语法分析树，下次分析时尽量重用 [WG98]
- 是否能够重用：**非终结符号对应的输入没有修改**
 - ❖ 可能的问题：**过于乐观**，向前看符号会影响归约



[WG98] Tim A. Wagner and Susan L. Graham. Efficient and Flexible Incremental Parsing. Trans. on Prog. Lang. and Syst., 20(5):980–1013, 1998.

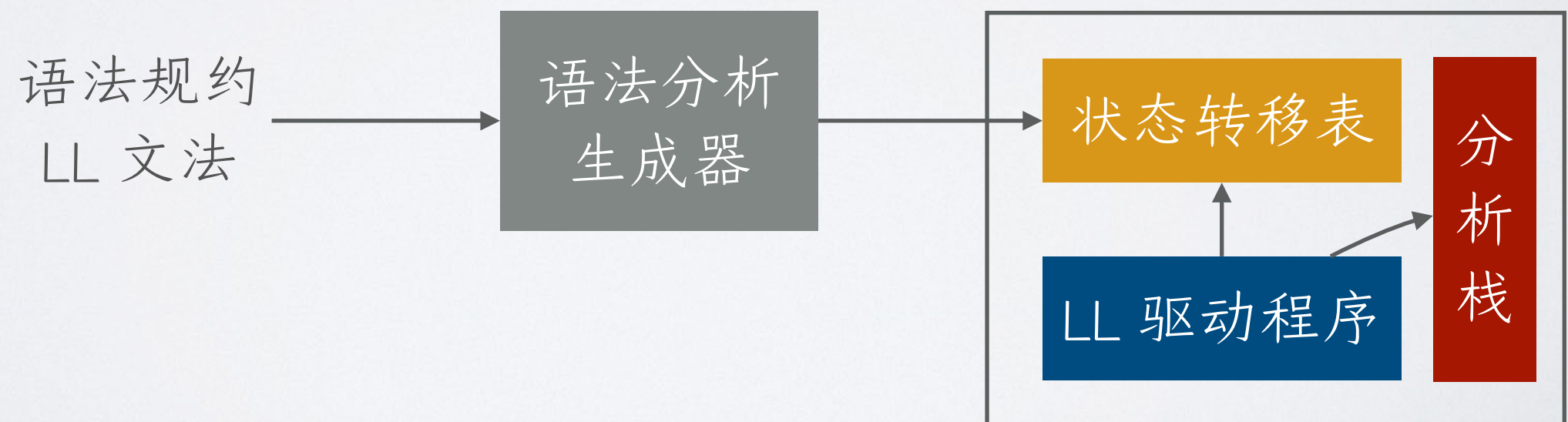
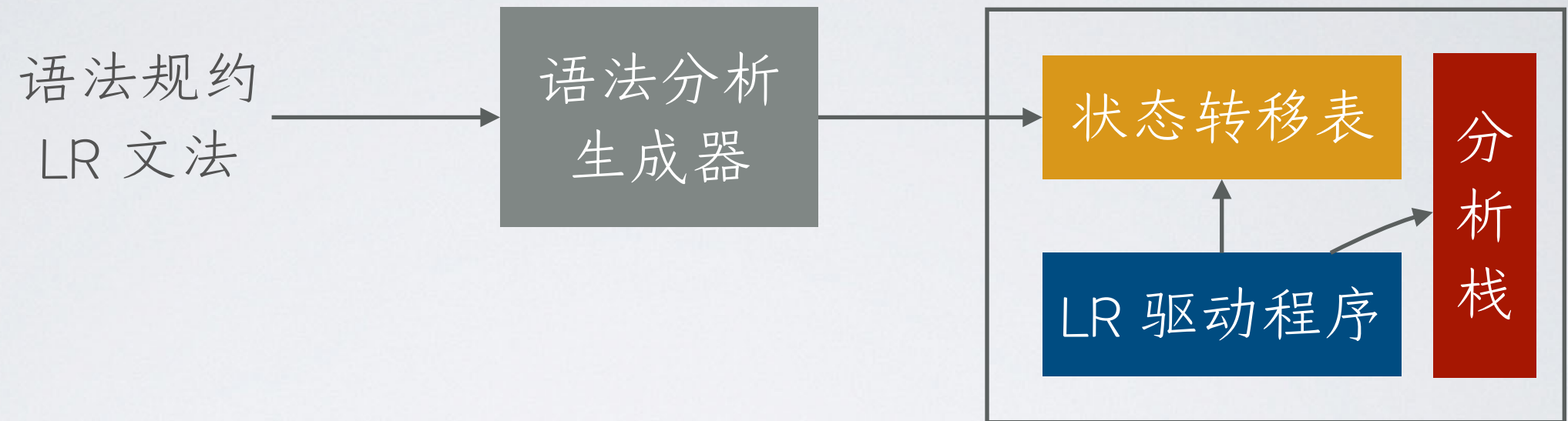
主要内容

- ◎ 自底向上的语法分析
- ◎ **语法分析的自动生成**
- ◎ 自顶向下的语法分析(进阶版)

从规约到实现的转换

- ◎ **语法分析的规约：**编程语言的文法
 - ❖ 通常为 CFG 文法，比如 LL 或 LR 文法
- ◎ **语法分析的实现：**
 - ❖ 自顶向下：递归下降分析，适用于 LL 文法
 - ❖ 自底向上：移进-归约分析，适用于 LR 文法
- ◎ **语法分析的自动生成：**自动化 LL/LR 文法到实现的转换

表驱动的语法分析



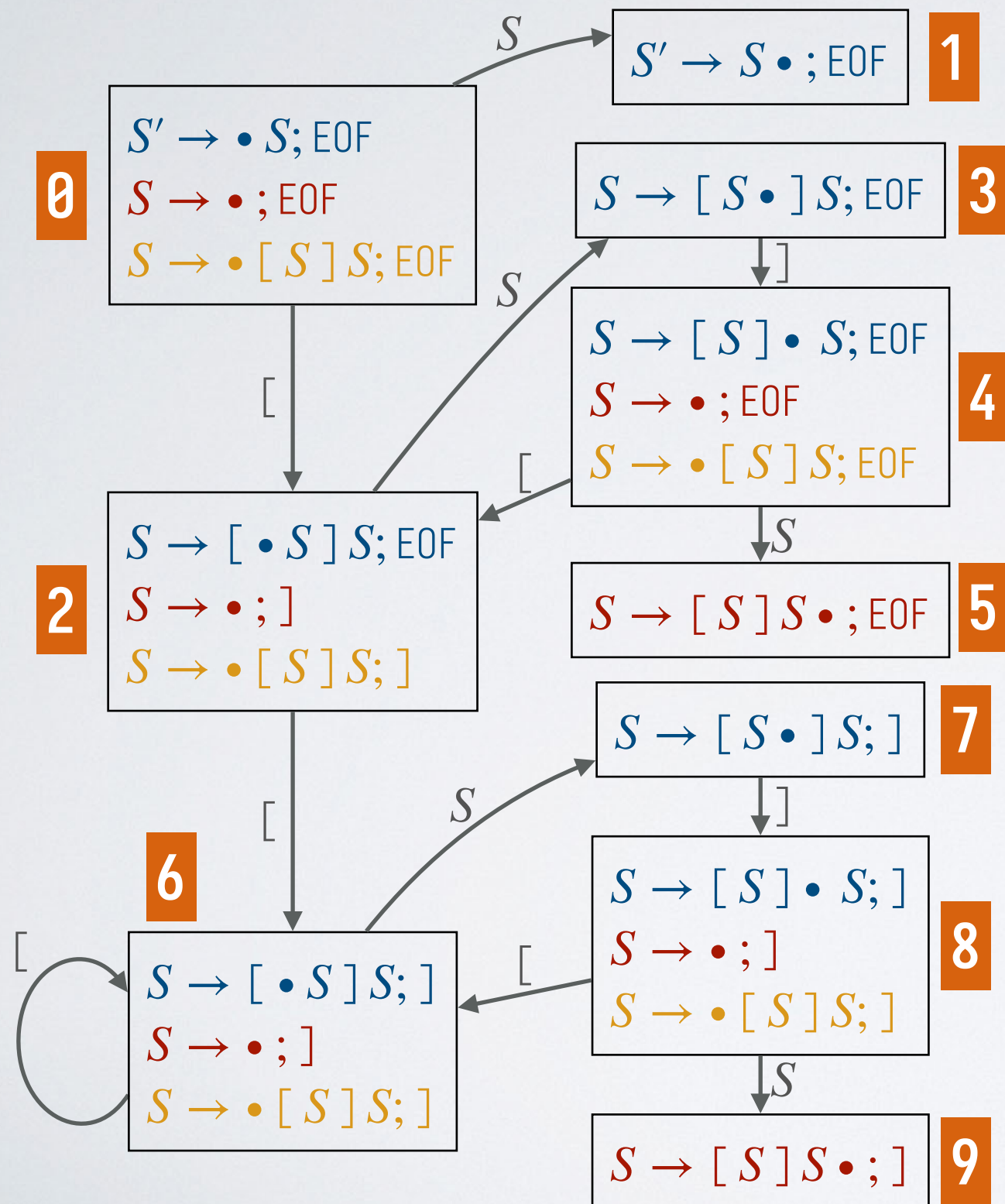
LR(1) 文法：移进-归约分析



	$S' \rightarrow S EOF$
[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow [S] S$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

LR(1) 文法：移进-归约分析

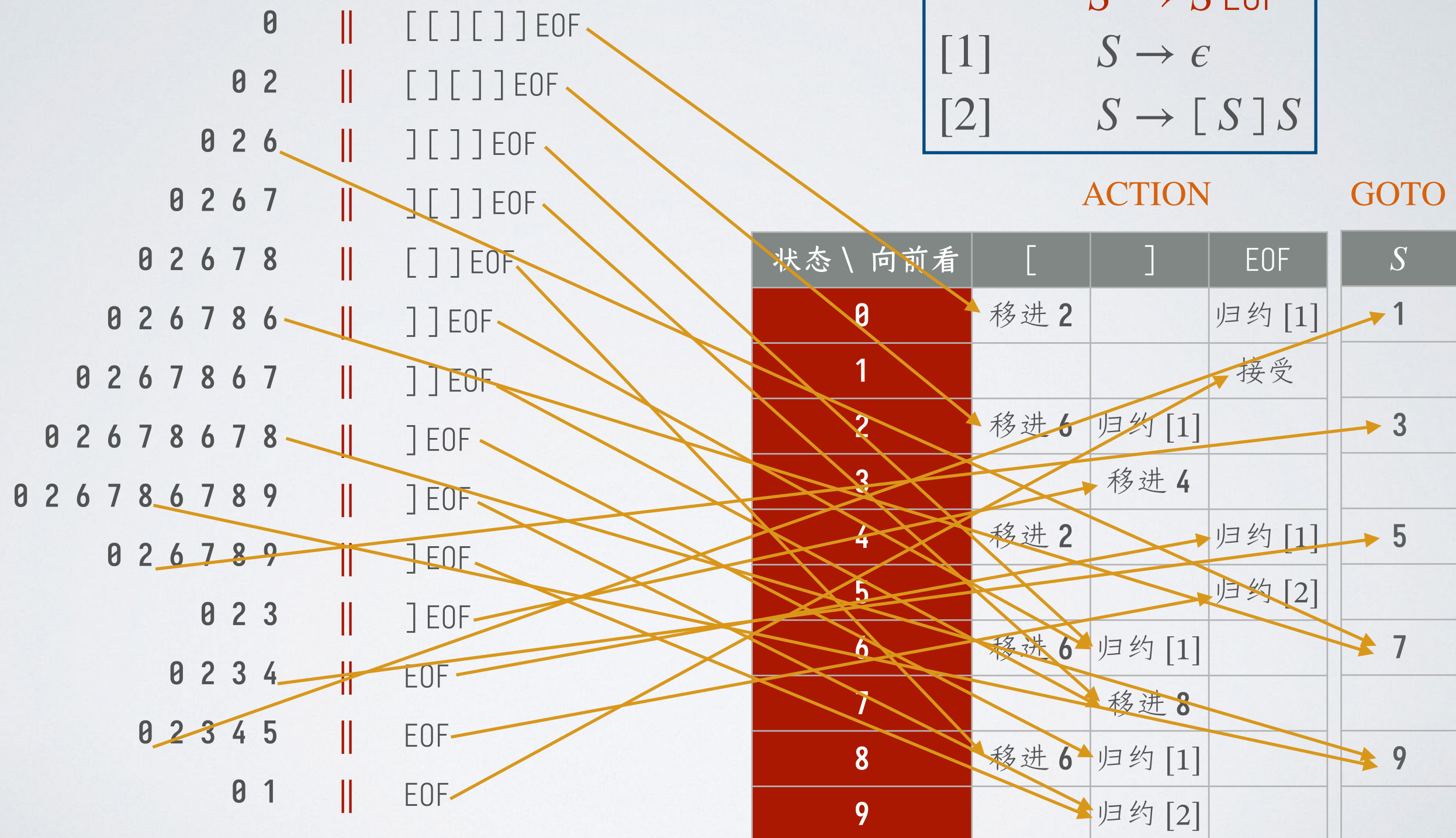


$S' \rightarrow S EOF$
 [1] $S \rightarrow \epsilon$
 [2] $S \rightarrow [S] S$

状态 \ 向前看	ACTION			GOTO
	[	]	EOF	
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

LR(1) 文法：移进-归约分析

$S' \rightarrow S \text{ EOF}$
 $[1] \quad S \rightarrow \epsilon$
 $[2] \quad S \rightarrow [S]S$



LR(1) 文法的驱动程序

```

token = next_token();
top = 1; stack[top] = I0;
while (true) {
    state = stack[top];
    if (ACTION[state][token] == “归约 A → α”) {
        top = top - |α|;
        state = stack[top];
        top++; stack[top] = GOTO[state][A];
    } else if (ACTION[state][token] == “移进 Ij”) {
        top++; stack[top] = Ij;
        token = next_token();
    } else if (ACTION[state][token] == “接受”) {
        return true;
    } else {
        return false;
    }
}

```

$$S' \rightarrow S \text{ EOF}$$

[1] $S \rightarrow \epsilon$

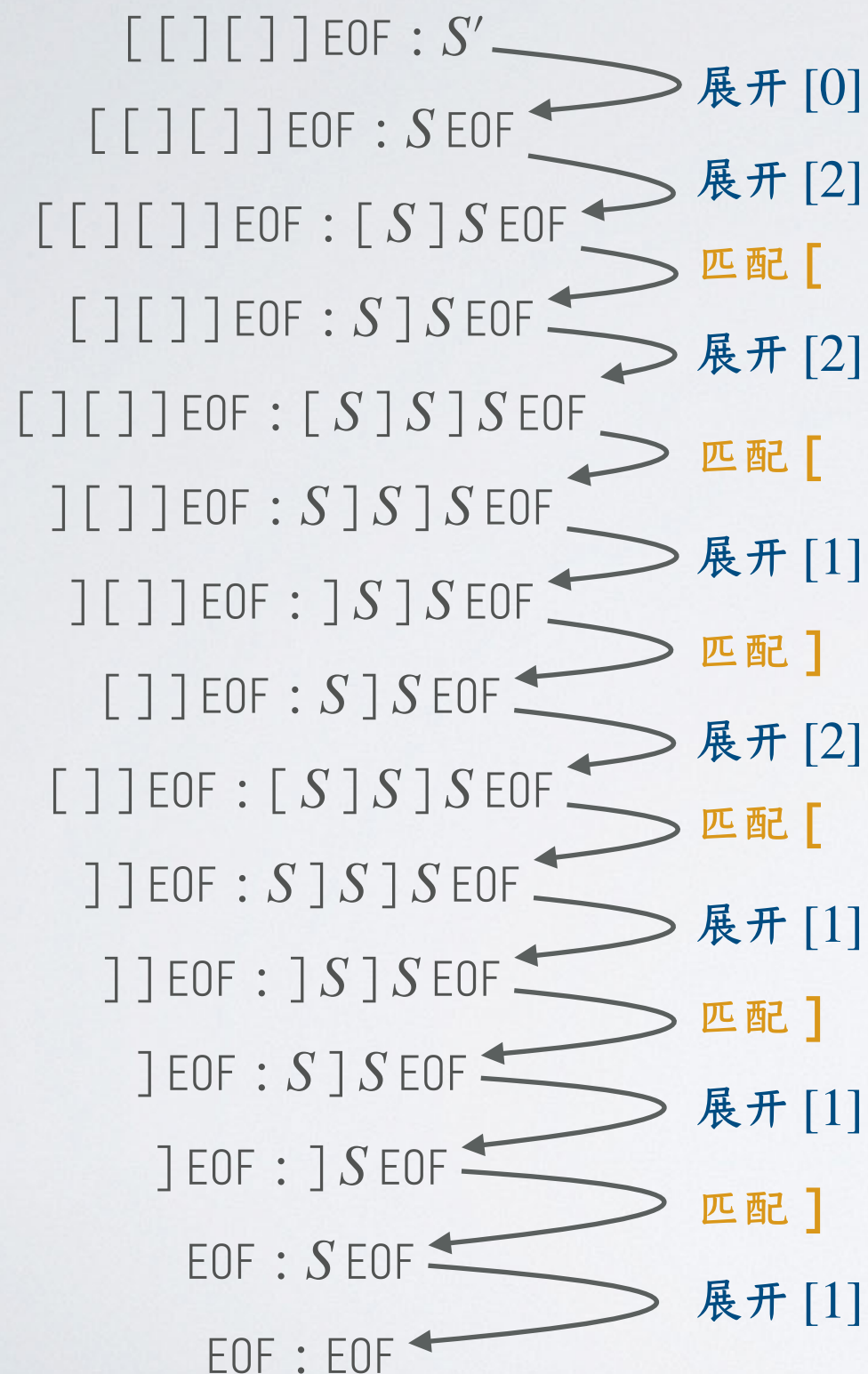
[2] $S \rightarrow [S] S$

ACTION				GOTO
状态 \ 向前看	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

LR(1) 文法的分析表

- ◎ 根据识别栈模式的自动机来构造 ACTION 和 GOTO 表
- ◎ 对于自动机状态 I_i , 考察其转移的情况
 - ❖ 如果对于终结符号 c 有转移到 I_j , 那么 $\text{ACTION}[I_i, c]$ 为“移进 I_j ”
 - ❖ 如果 I_i 中有可归约模式 $\langle A \rightarrow \alpha \bullet ; c \rangle$, 那么 $\text{ACTION}[I_i, c]$ 为“归约 $A \rightarrow \alpha$ ”
 - ❖ 如果 I_i 中有可归约模式 $\langle S' \rightarrow S \bullet ; \text{EOF} \rangle$, 那么 $\text{ACTION}[I_i, \text{EOF}]$ 为“接受”
 - ❖ 如果对于非终结符号 X 有转移到 I_j , 那么 $\text{GOTO}[I_i, X]$ 为 I_j
- ◎ 如果文法是 LR(1) 的, 上述构造过程不会引入冲突

LL(1) 文法：递归下降分析



[0]	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow [S] S$

使用一个栈来记录待匹配的终结符号和非终结符号

分析栈 \ 向前看	[	]	EOF
$\beta S'$	展开 [0]	错误	展开 [0]
βS	展开 [2]	展开 [1]	展开 [1]
$\beta [$	匹配	错误	错误
$\beta]$	错误	匹配	错误
EOF	错误	错误	接受
ϵ	错误	错误	错误

LL(1) 文法：递归下降分析

S' || $[[[]]] EOF$
 $EOF S$ || $[[[]]] EOF$
 $EOF S] S[$ || $[[[]]] EOF$
 $EOF S] S$ || $[[[]]] EOF$
 $EOF S] S] S[$ || $[[[]]] EOF$
 $EOF S] S] S$ || $] [[]] EOF$
 $EOF S] S]$ || $] [[]] EOF$
 $EOF S] S$ || $] [[]] EOF$
 $EOF S] S] S[$ || $] [[]] EOF$
 $EOF S] S] S$ || $] [[]] EOF$
 $EOF S] S]$ || $] [[]] EOF$
 $EOF S] S$ || $] [[]] EOF$
 $EOF S]$ || $] [[]] EOF$
 $EOF S$ || $] [[]] EOF$
 EOF || $] [[]] EOF$

$[0] \quad S' \rightarrow S EOF$
 $[1] \quad S \rightarrow \epsilon$
 $[2] \quad S \rightarrow [S] S$

使用一个栈来记录待匹配的
终结符号和非终结符号

分析栈 \ 向前看	[	]	EOF
$\beta S'$	展开 [0]	错误	展开 [0]
βS	展开 [2]	展开 [1]	展开 [1]
$\beta [$	匹配	错误	错误
$\beta]$	错误	匹配	错误
EOF	错误	错误	接受
ϵ	错误	错误	错误

LL(1) 文法的分析驱动程序

```

token = next_token();
top = 1; stack[top] = S';
while (true) {
    if (top == 1 &&
        stack[top] == EOF &&
        token == EOF) {
        return true;
    } else if (top > 0 && stack[top] == token) {
        top--; token = next_token();
    } else if (top > 0 &&
        table[stack[top]][token] ==
        A → B1 B2 ... Bk) {
        top--;
        for (int i = k; i >= 1; i--) {
            if (Bi != ε) {
                top++; stack[top] = Bi;
            }
        }
    } else {
        return false;
    }
}

```

[0]	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow [S] S$

使用一个栈来记录待匹配的
终结符号和非终结符号

分析栈 \ 向前看	[	]	EOF
$\beta S'$	展开 [0]	错误	展开 [0]
βS	展开 [2]	展开 [1]	展开 [1]
$\beta [$	匹配	错误	错误
$\beta]$	错误	匹配	错误
EOF	错误	错误	接受
ϵ	错误	错误	错误

LL(1) 文法的分析表

- 分析栈和向前看符号都是 EOF 时, 动作为接受
- 分析栈顶是终结符号时, 当向前看符号匹配时, 动作为匹配
- 分析栈是非终结符号 A 时:
 - 考虑规则 $A \rightarrow \alpha$, 对任意 $\text{FIRST}(\alpha)$ 中的终结符号 c , 向前看符号为 c 时, 动作为展开 $A \rightarrow \alpha$ 这条规则
 - 如果 $\epsilon \in \text{FIRST}(\alpha)$, 对任意 $\text{FOLLOW}(A)$ 中的终结符号 c , 向前看符号为 c 时, 动作为展开 $A \rightarrow \alpha$ 这条规则

[0]	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow [S] S$

$\text{FIRST}(S') = \{[, \text{EOF}\}$
 $\text{FIRST}(S) = \{[, \epsilon\}$
 $\text{FOLLOW}(S) = \{\text{EOF},]\}$

分析栈 \ 向前看	[	]	EOF
$\beta S'$	展开 [0]		展开 [0]
βS	展开 [2]	展开 [1]	展开 [1]
$\beta [$	匹配		
$\beta]$		匹配	
EOF			接受
ϵ			

主要内容

- ◎ 自底向上的语法分析
- ◎ 语法分析的自动生成
- ◎ 自顶向下的语法分析(进阶版)

为什么还是需要自顶向下?

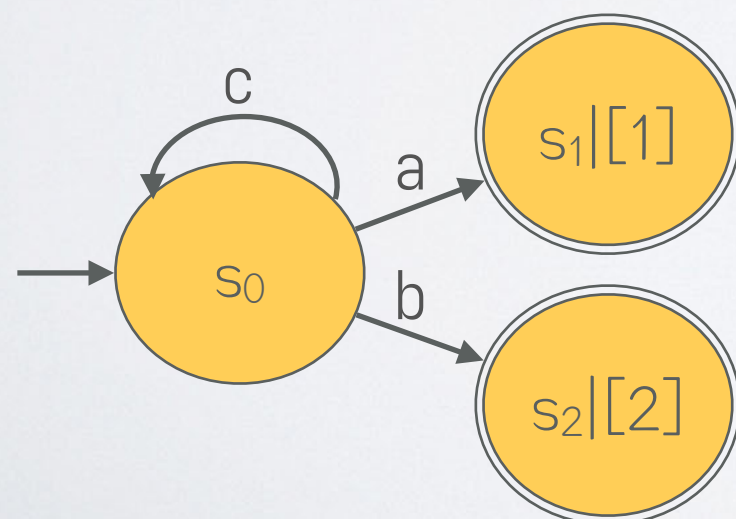
- ◎ GLR 和 PEG 等算法支持的文法已经足够广泛了
 - ❖ **问题:** GLR 接受有二义性的文法, 但无法探测二义性
 - ❖ **问题:** PEG 通过「贪心」策略绕过二义性, 但有时会违背直觉
 - ❖ 回顾: $A \leftarrow a / ab$ 的第二个选择永远不会被使用
- ◎ 自底向上分析算法难以手动实现, 通常依赖自动生成
 - ❖ **问题:** 语法分析过程需追踪分析栈模式, 难以调试
 - ❖ **问题:** 自动生成的方式难以生成高质量的错误信息
- ◎ **自顶向下分析算法的优势:**
 - ❖ 相比自底向上分析更加自然, 也更容易理解
 - ❖ 容易手动实现(递归下降), 从而方便进行调试、输出错误信息

LL(*) 算法

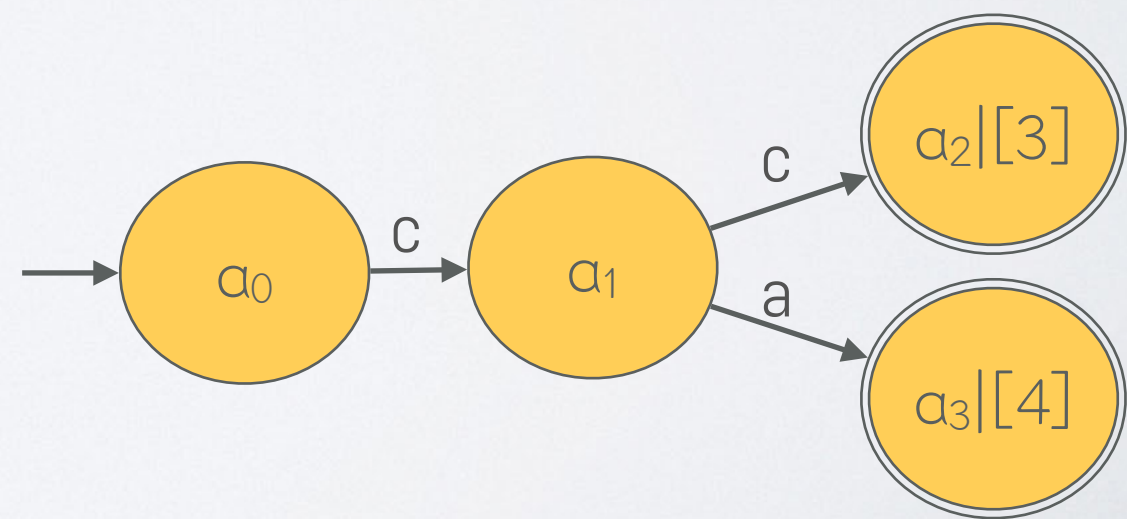
- LL(k) 算法的限制在于**固定长度**的向前看符号
- 例如：对任意的 k, 右侧文法都不是 LL(k) 的
 - ❖ 该文法有左公因子
 - ❖ 需要向前看**所有**的 c 之后才能决定产生规则
- LL(*) 的思路：用一个 DFA 来识别**向前看模式**

[0]	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow A$
[2]	$S \rightarrow B$
[3]	$A \rightarrow c A a$
[4]	$A \rightarrow c a$
[5]	$B \rightarrow c B b$
[6]	$B \rightarrow c b$

S 的向前看模式 DFA



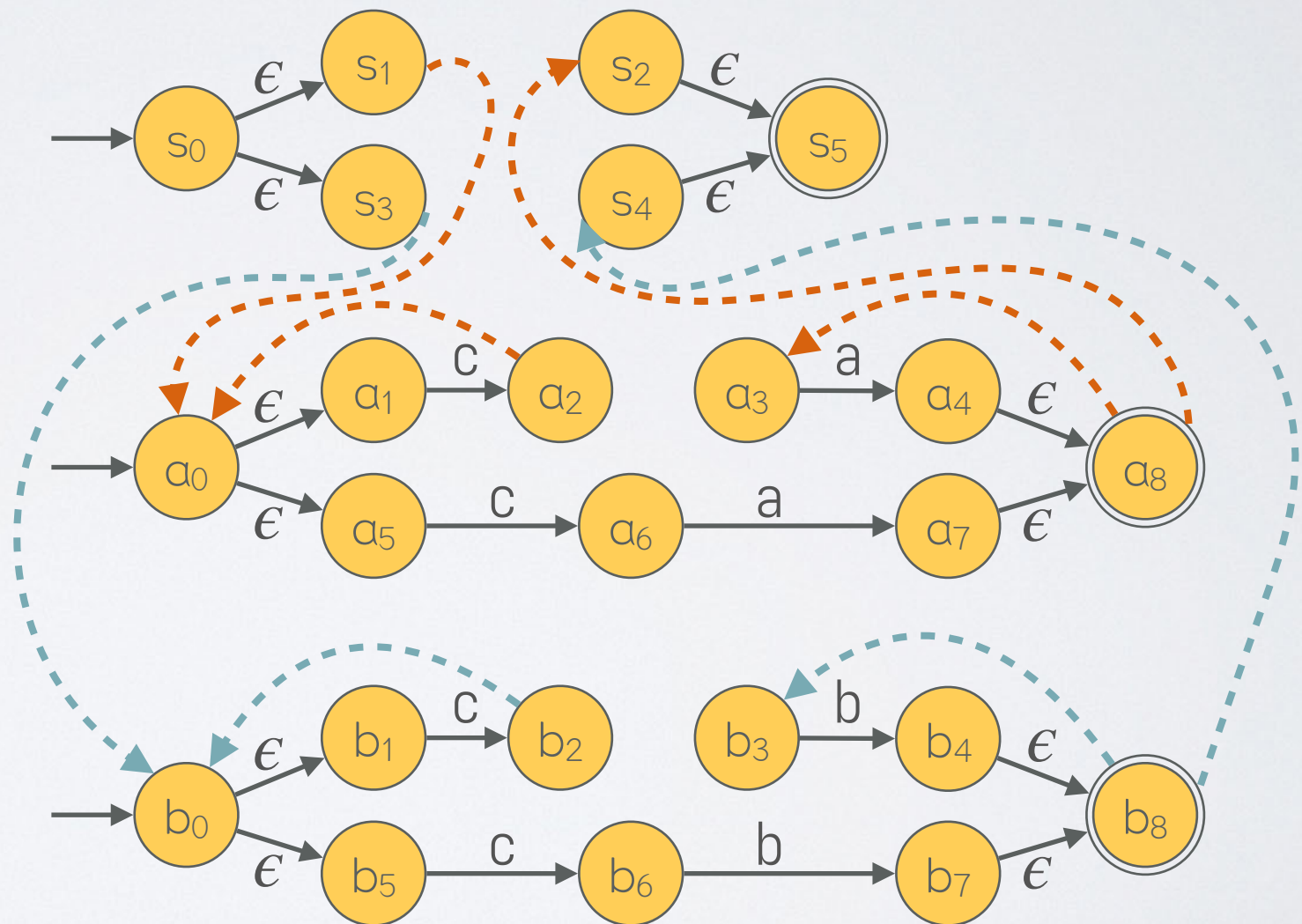
A 的向前看模式 DFA



LL(*) 算法

- 是否存在适用于 LL(*) 的正则表达式是不可判定问题
- ANTLR: 近似构造, 构造不出来就回去用 LL(k) 或者回溯[pF11]

[0]	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow A$
[2]	$S \rightarrow B$
[3]	$A \rightarrow c A a$
[4]	$A \rightarrow c a$
[5]	$B \rightarrow c B b$
[6]	$B \rightarrow c b$



[pF11] Terence Parr and Kathleen Fisher. 2011. LL(*): the foundation of the ANTLR parser generator. In PLDI'11.

LL(*) 算法

[1] $S \rightarrow A$
[2] $S \rightarrow B$

子集构造法变种:
 ◎ 记录产生规则
 ◎ 追踪「调用栈」的情况

<...> 里为「调用栈」的情况
 一种近似方法: 至多追踪一层

[1]: $s1\langle \$ \rangle, a0\langle s2 \rangle, a1\langle s2 \rangle, a5\langle s2 \rangle$
 [2]: $s3\langle \$ \rangle, b0\langle s4 \rangle, b1\langle s4 \rangle, b5\langle s4 \rangle$

[1]: $a2\langle s2 \rangle, a0\langle a3 \rangle, a1\langle a3 \rangle, a5\langle a3 \rangle, a6\langle s2 \rangle$
 [2]: $b2\langle s4 \rangle, b0\langle b3 \rangle, b1\langle b3 \rangle, b5\langle b3 \rangle, b6\langle s2 \rangle$

[1]: $a2\langle a3 \rangle, a0\langle a3 \rangle, a1\langle a3 \rangle, a5\langle a3 \rangle, a6\langle a3 \rangle$
 [2]: $b2\langle b3 \rangle, b0\langle b3 \rangle, b1\langle b3 \rangle, b5\langle b3 \rangle, b6\langle b3 \rangle$

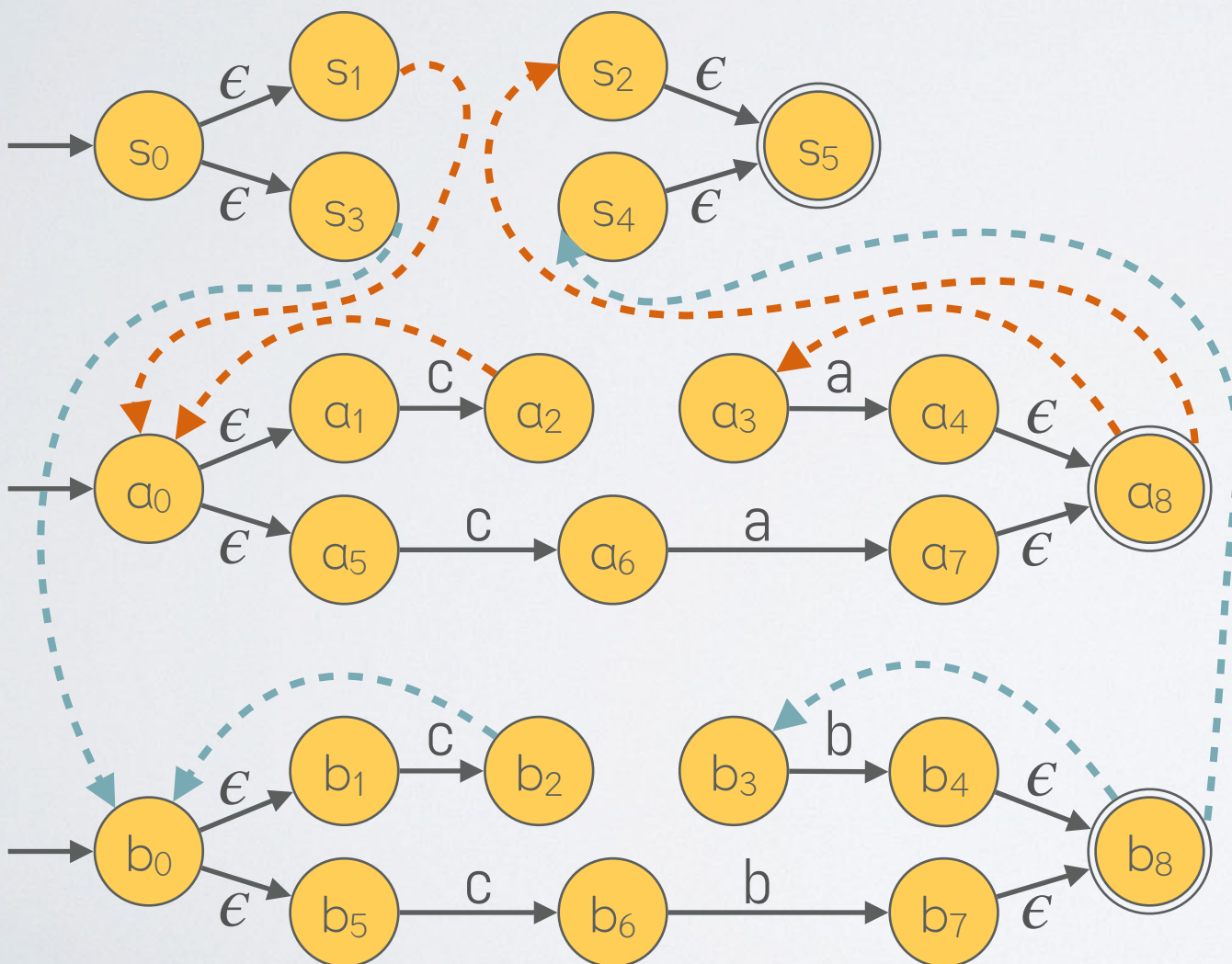
[1]: $a7\langle s2 \rangle, a8\langle s2 \rangle, s2\langle \$ \rangle, s5\langle \$ \rangle$

[2]: $b7\langle s4 \rangle, b8\langle s4 \rangle, s4\langle \$ \rangle, s5\langle \$ \rangle$

由于近似, 返回后不知道之前的「调用栈」

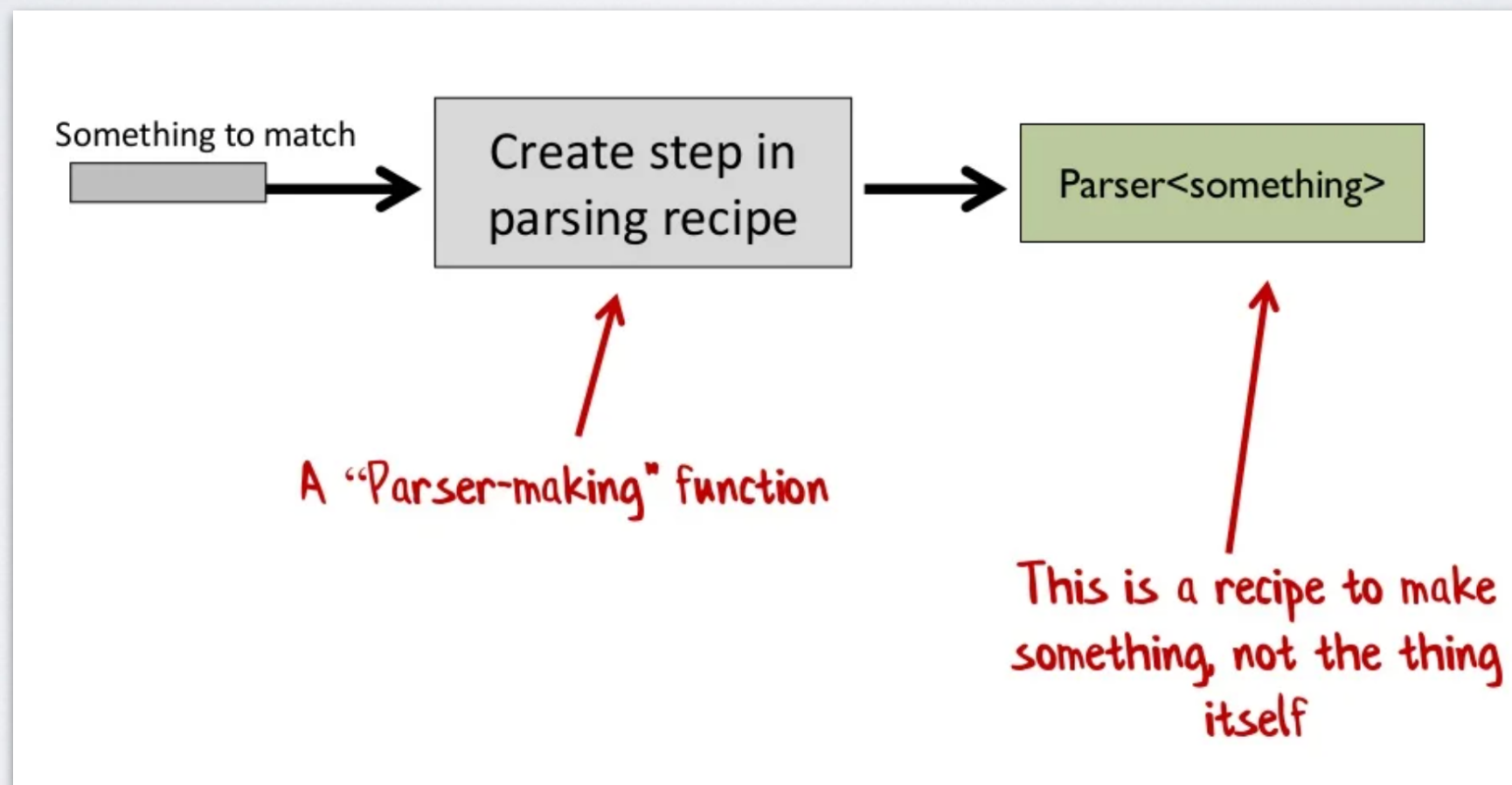
[1]: $a7\langle a3 \rangle, a8\langle a3 \rangle, a3\langle ? \rangle$

[2]: $b7\langle b3 \rangle, b8\langle b3 \rangle, b3\langle ? \rangle$



Parser Combinators

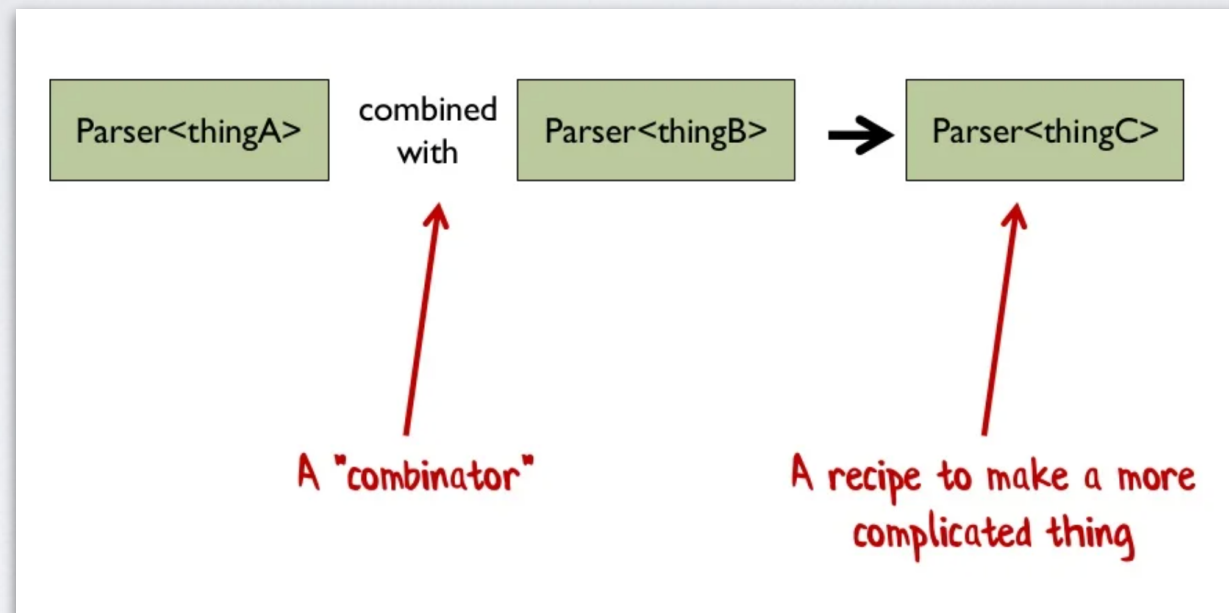
- 问题：有没有一种方式介于手动实现和自动生成之间？
- 观察：不同语言的分析器有很多类似的「构造方式」
 - ❖ 比如「零个或多个」、「用逗号隔开的多个」、「中缀运算符表达式」等



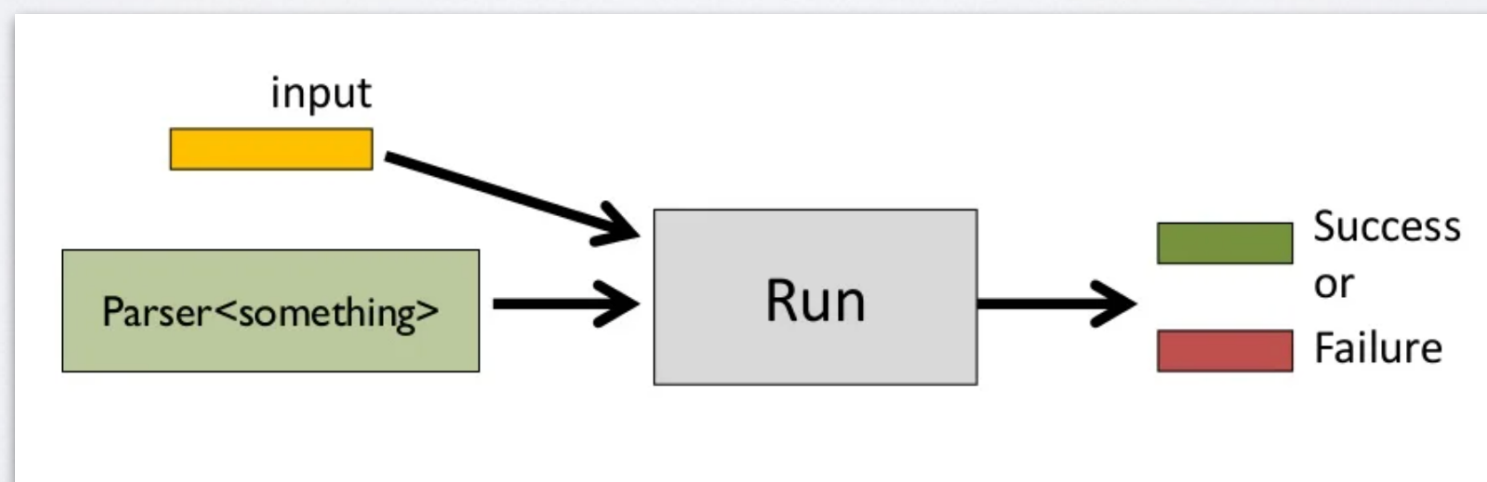
本部分课件参考了<https://fsharpforfunandprofit.com/parser/>。

Parser Combinators

- **组合子(combinator)**: 把多个简单 parser 组合成一个复杂的

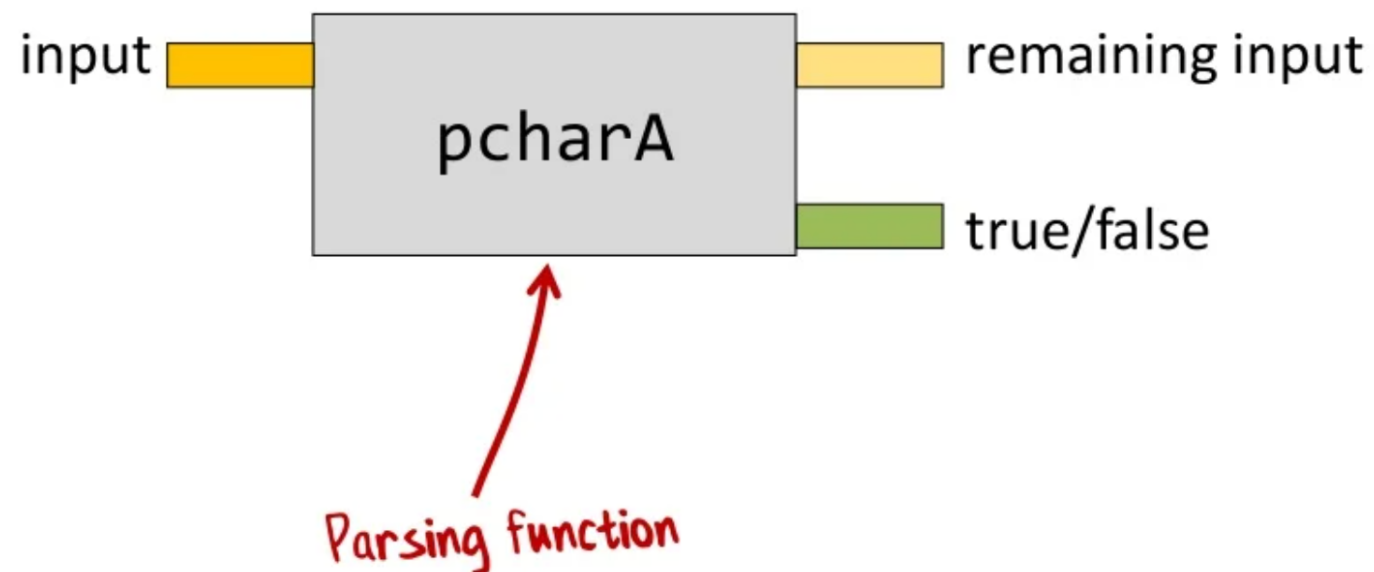


- 语法分析的过程示意图:



一个简单的 parser：第一版

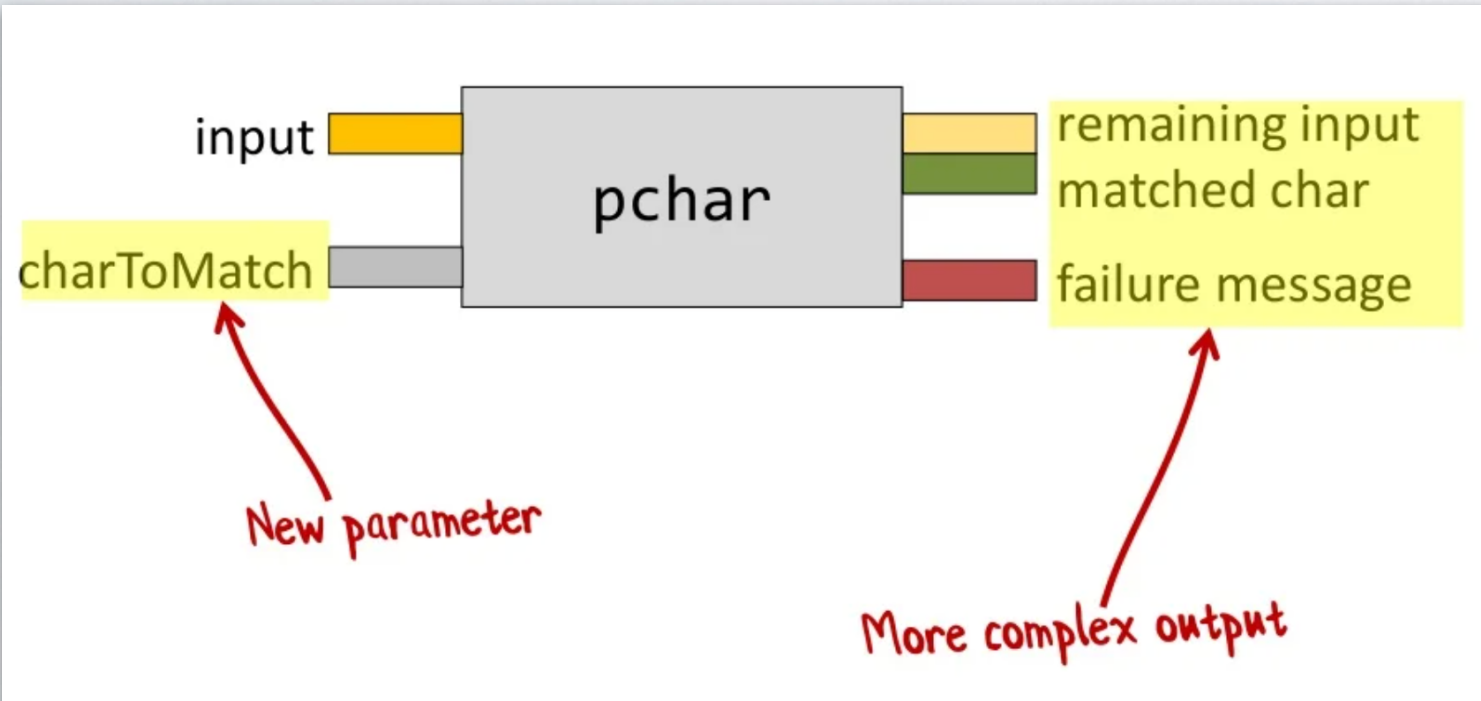
「函数式」建模：无副作用，读取一个字符串，
返回分析完成后剩余的字符串



```
pair<bool, string_view> pcharA(  
    string_view input  
) {  
    if (!input.empty() && input[0] == 'A') {  
        return {true, input.substr(1)};  
    } else {  
        return {false, input};  
    }  
}
```

一个简单的 parser：第二版

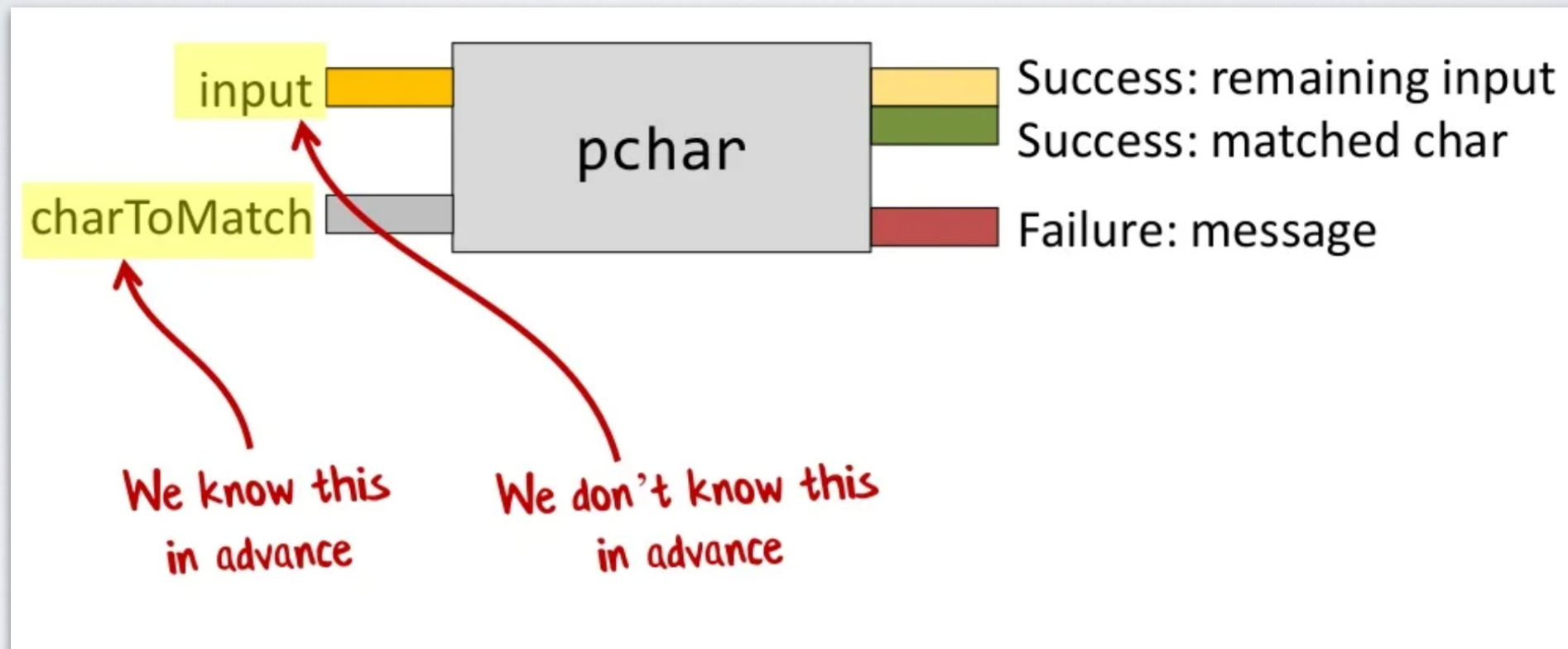
新增参数表示语法要求识别的字符
新增返回值表示识别到的字符, 如果出错则抛出异常



```
pair<char, string_view> pchar(
    char char_to_match,
    string_view input
) {
    if (!input.empty() &&
        input[0] == char_to_match) {
        return {input[0], input.substr(1)};
    } else {
        throw runtime_error("...");
    }
}
```

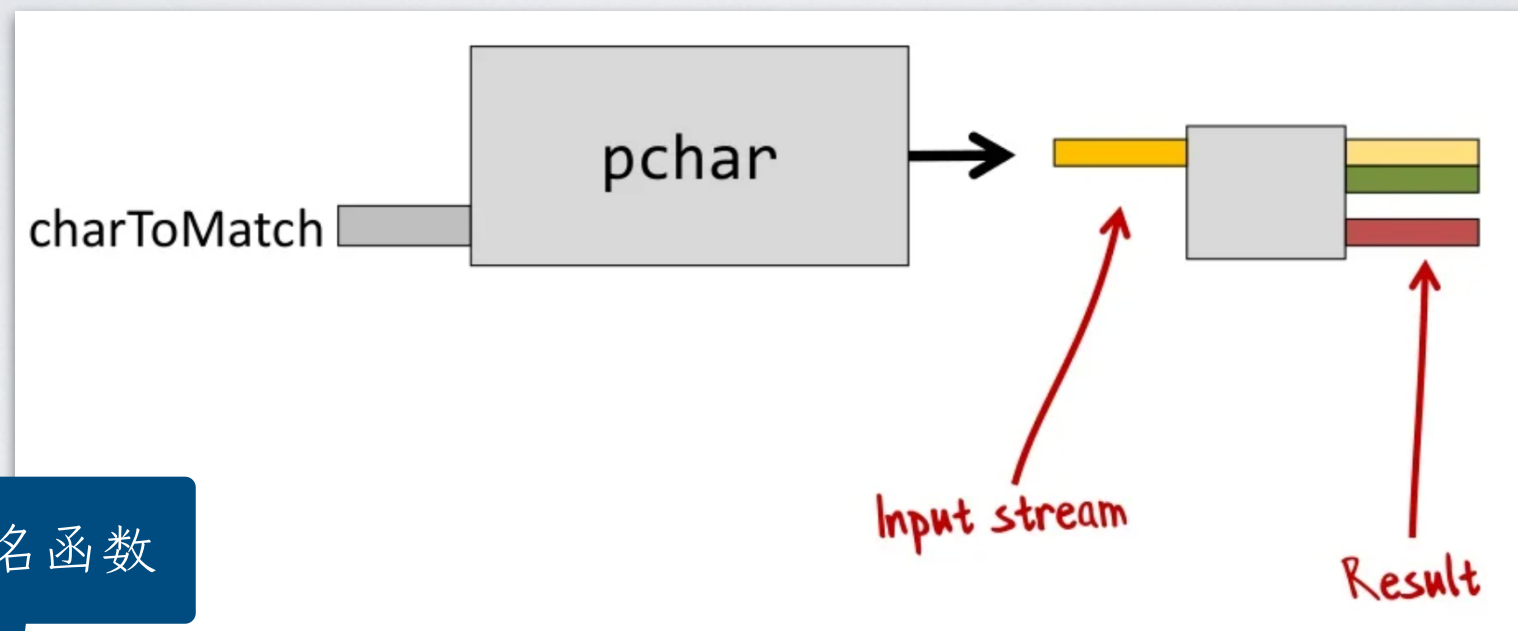
一个简单的 parser：第三版

对语法分析的计算进行「分阶段」：
在定义文法时，charToMatch 是已知的，input 是还不知道的



标准做法是在「函数式」编程中，让 `pchar` 函数只接受 `charToMatch` 一个参数，然后返回另一个函数处理 `input`

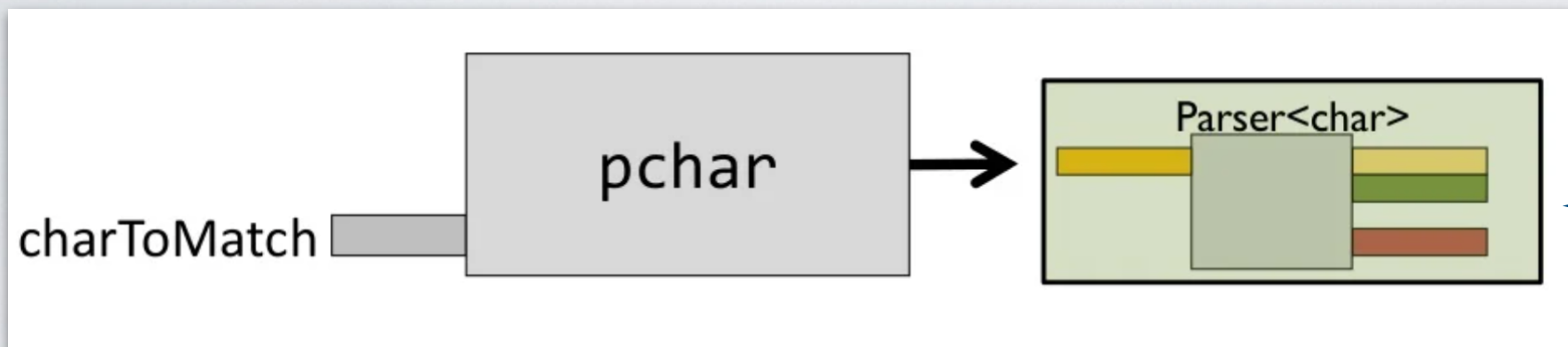
一个简单的 parser：第三版



C++ 中支持匿名函数

```
function<pair<char, string_view>(string_view)> pchar(char char_to_match) {
    return [=](string_view input) {
        if (!input.empty() && input[0] == char_to_match) {
            return pair<char, string_view>{input[0], input.substr(1)};
        } else {
            throw runtime_error("...");
        }
    };
}
```

一个简单的 parser：第四版



按照面向对象的方式进行封装

```
template <typename A>
struct Parser {
    virtual pair<A, string_view>
    parse(string_view) = 0;
};
```

```
Parser<char>* pchar(char c) {
    struct CharParser : public Parser<char> {
        char char_to_match;
        CharParser(char char_to_match)
            : char_to_match(char_to_match) {}
        pair<char, string_view> parse(string_view input) override {
            if (!input.empty() && input[0] == char_to_match) {
                return {input[0], input.substr(1)};
            } else {
                throw runtime_error("...");
            }
        }
    };
    return new CharParser(c);
}
```

Combinator 是什么？

- A “combinator” library is a library designed around combining things to get more complex values of the same type.
- `integer + integer = integer`
- `list @ list = list` *// @ is list concat*
- `Parser ?? Parser = Parser` *What could go here?*

一些基本的 Parser Combinators

- ◎ parser **and_then** parser $\Rightarrow$ parser
 - ❖ 先运行第一个 parser 进行语法分析
 - ❖ 再运行第二个 parser 在剩余的输入上进行语法分析
 - ❖ 如果两个都成功了, 则将它们的分析结果组成一个 pair 返回
- ◎ parser **or_else** parser $\Rightarrow$ parser
 - ❖ 先运行第一个 parser 进行语法分析, 若成功则返回其分析结果
 - ❖ 若失败, 则运行第二个 parser 进行语法分析并返回其分析结果
- ◎ parser **map** (transformer) $\Rightarrow$ parser
 - ❖ 运行 parser 进行语法分析
 - ❖ 若成功, 则在其分析结果上应用变换函数 transformer

and_then 组合子

```
template <typename B>
Parser<pair<A, B>>* and_then(Parser<B>* other) {
    struct AndThenParser : public Parser<pair<A, B>> {
        Parser<A>* parser1;
        Parser<B>* parser2;
        AndThenParser(Parser<A>* parser1, Parser<B>* parser2)
            : parser1(parser1), parser2(parser2) {}
        pair<pair<A, B>, string_view> parse(string_view input) override {
            auto res1 = parser1->parse(input);
            auto res2 = parser2->parse(res1.second);
            return {{res1.first, res2.first}, res2.second};
        }
    };
    return new AndThenParser(this, other);
}
```

先运行 parser1

res1.second 表示剩余的输入

组成 pair 返回

or_else 组合子

```
Parser<A>* or_else(Parser<A>* other) {  
    struct OrElseParser : public Parser<A> {  
        Parser<A>* parser1;  
        Parser<A>* parser2;  
        OrElseParser(Parser<A>* parser1, Parser<A>* parser2)  
            : parser1(parser1), parser2(parser2) {}  
        pair<A, string_view> parse(string_view input) override {  
            try {  
                return parser1->parse(input);  
            } catch (runtime_error&) {  
                return parser2->parse(input);  
            }  
        }  
    };  
    return new OrElseParser(this, other);  
}
```

先运行 parser1

若失败, 则在同样的
输入上运行 parser2

map 组合子

从 A 类型变换到 B 类型

```
template <typename B>
Parser<B>* map(function<B(A)> transform) {
    struct MapParser : public Parser<B> {
        Parser<A>* inner_parser;
        function<B(A)> transform;
        MapParser(Parser<A>* inner_parser, function<B(A)> transform)
            : inner_parser(inner_parser), transform(transform) {}
        pair<B, string_view> parse(string_view input) override {
            auto res = inner_parser->parse(input);
            return {transform(res.first), res.second};
        }
    };
    return new MapParser(this, transform);
}
```

分析完成后应用变换

组合这些组合子

```
Parser<char>* one_of(char start, char final) {  
    auto parser = pchar(start);  
    for (char c = start + 1; c <= final; ++c) {  
        parser = parser->or_else(pchar(c));  
    }  
    return parser;  
}
```

识别 start..final 区间
中的所有字符

```
one_of('0', '9')
```

识别一个数字; 类型为 Parser<char>*

```
one_of('0', '9')->map<int>([](char c) { return c - '0'; })
```

识别一个数字并变换为整型; 类型为 Parser<int>*

使用组合子构造四则运算求值器

many1 表示「一个或多个」

```
auto pdigit = one_of('0', '9')->map<int>(chr2int);
auto pnumber = pdigit->many1()->map<int>(vec2int);
auto pexpr = new RecParser<int>();
auto pterm = new RecParser<int>();
Parser<int>* atom = pexpr->between(pchar('('), pchar(')'))->or_else(pnumber);
pterm->inner_parser = atom->chain_left(
    pchar('*')
    ->map_throw<intop>(multiplies<int>())
    ->or_else(pchar('/')->map_throw<intop>(divides<int>())));
pexpr->inner_parser = pterm->chain_left(
    pchar('+')
    ->map_throw<intop>(plus<int>())
    ->or_else(pchar('-')->map_throw<intop>(minus<int>())));
```

RecParser 处理文法中的递归情况

between 表示「夹在两者之间」

chain_left 用来处理左结合的中缀表达式

map_throw 类似 map, 但不考虑本来分析出的结果

many 和 many1 组合子

```
Parser<vector<A>>* many() {  
    struct ManyParser : public Parser<vector<A>> {  
        Parser<A>* inner_parser;  
        ManyParser(Parser<A>* inner_parser)  
            : inner_parser(inner_parser) {}  
        pair<vector<A>, string_view> parse(  
            string_view input  
        ) override {  
            vector<A> vec;  
            while (true) {  
                try {  
                    auto res = inner_parser->parse(input);  
                    vec.push_back(res.first);  
                    input = res.second;  
                } catch (runtime_error&) {  
                    break;  
                }  
            }  
            return {vec, input};  
        }  
    };  
    return new ManyParser(this);  
};
```

many1 通过组合 and_then
和 many 来实现, 使用 map
变换分析结果

```
Parser<vector<A>>* many1() {  
    return this->and_then(this->many())  
        ->template map<vector<A>>(  
            [](pair<A, vector<A>> p) {  
                p.second.insert(  
                    p.second.begin(),  
                    p.first  
                );  
                return p.second;  
            }  
        );  
}
```

between 组合子

```
template <typename B>
Parser<A>* throw_right(Parser<B>* other) {
    return this->and_then(other)->template map<A>(
        [](pair<A, B> p) { return p.first; });
}
```

throw_right 表示依次进行 this 和 other, 但 other 的值不要

```
template <typename B>
Parser<B>* throw_left(Parser<B>* other) {
    return this->and_then(other)->template map<B>(
        [](pair<A, B> p) { return p.second; });
}
```

throw_left 表示依次进行 this 和 other, 但 this 的值不要

```
template <typename B, typename C>
Parser<A>* between(Parser<B>* before, Parser<C>* after) {
    return before->throw_left(this)->throw_right(after);
}
```

between 通过组合 throw_left 和 throw_right 来实现

chain_left 组合子

运算符的类型是接受 2 个 A 类型的参数, 返回 1 个 A 类型的值

```
Parser<A>* chain_left(Parser<function<A(A, A)>>* op_parser) {  
    struct ChainLeftParser : public Parser<A> {  
        Parser<A>* inner_parser;  
        Parser<function<A(A, A)>>* op_parser;  
        ChainLeftParser(Parser<A>* inner_parser,  
                        Parser<function<A(A, A)>>* op_parser)  
            : inner_parser(inner_parser), op_parser(op_parser) {}  
        pair<A, string_view> parse(string_view input) override {  
            auto res = inner_parser->parse(input);  
            auto acc = res.first;  
            input = res.second;  
            while (true) {  
                try {  
                    auto op = op_parser->parse(input);  
                    auto res = inner_parser->parse(op.second);  
                    acc = op.first(acc, res.first);  
                    input = res.second;  
                } catch (runtime_error&) {  
                    break;  
                }  
            }  
            return {acc, input};  
        }  
    };  
    return new ChainLeftParser(this, op_parser);  
}
```

先分析第一个运算分量

反复使用 op_parser 和 inner_parser 分析运算符和下一个运算分量, 并通过 op_parser 的分析结果进行合并

也可以使用 and_then、many、map 组合子来实现

使用组合子构造四则运算求值器

RecParser 是对 Parser 的简单包装, 用来实现「延迟绑定」, 从而实现文法的递归情况

```
template <typename A>
struct RecParser : public Parser<A> {
    Parser<A>* inner_parser;
    pair<A, string_view> parse(string_view input) override {
        return inner_parser->parse(input);
    }
};
```

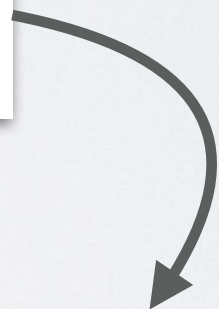
```
auto pdigit = one_of('0', '9')->map<int>(chr2int);
auto pnumber = pdigit->many1()->map<int>(vec2int);
auto pexpr = new RecParser<int>();
auto pterm = new RecParser<int>();
Parser<int>* atom = pexpr->between(pchar('('), pchar(''))->or_else(pnumber);
pterm->inner_parser = atom->chain_left(
    pchar('*')
    ->map_throw<intop>(multiplies<int>())
    ->or_else(pchar('/')->map_throw<intop>(divides<int>())));
pexpr->inner_parser = pterm->chain_left(
    pchar('+')
    ->map_throw<intop>(plus<int>())
    ->or_else(pchar('-')->map_throw<intop>(minus<int>())));
```

Pratt Parsing

- 问题：处理运算符的优先级和左右结合性通常比较繁琐
- **「Binding Power」**：给每个运算符一对数值 (lp, rp)，对于每个运算分量，如果它左边算符的 rp 比右边算符的 lp 大，则向左结合；反之，则向右结合
- 例如，给 +、- 的 BP 为 (1,2)，给 *、/ 的 BP 为 (3,4)：

a	+	b	*	c	*	d	+	e
1	2	3	4	3	4	1	2	

- ❖ 对第二个运算分量 b，应该向右结合
- ❖ 对第三个运算分量 c，应该向左结合
- ❖ 对第四个运算分量 d，应该向左结合



$a + ((b * c) * d) + e$

支持 Pratt Parsing 的组合子

```
Parser<A>* chain(  
    Parser<tuple<function<A(A, A)>, int, int>>* op_parser  
) {  
    struct ChainParser : public Parser<A> {  
        ...  
        pair<A, string_view> parse_bp(string_view input, int min_bp) {  
            auto res = inner_parser->parse(input);  
            auto acc = res.first;  
            input = res.second;  
            while (true) {  
                try {  
                    auto op = op_parser->parse(input);  
                    auto op_func = get<0>(op.first);  
                    int l_bp = get<1>(op.first), r_bp = get<2>(op.first);  
                    if (l_bp < min_bp) { break; }  
                    auto res = parse_bp(op.second, r_bp);  
                    acc = op_func(acc, res.first);  
                    input = res.second;  
                } catch (runtime_error&) {  
                    break;  
                }  
            }  
            return {acc, input};  
        }  
    };  
    return new ChainParser(this, op_parser);  
}
```

额外返回运算符的 Binding Power
(两个整数)

记录当前正在分析的部分「之前」
的运算符的「右侧」 Binding Power

如果 min_bp 更大, 则意味着前面的运算分
量要向左结合, 所以终止当前分析

否则, 意味这它需要向右结合, 以当前算符
的 r_bp 为新的 min_bp 往后进行分析

支持 Pratt Parsing 的组合子

无需区分 Expr 和 Term, 也方便扩展更多运算符

```
pexpr->inner_parser =  
  atom->chain(pchar('+')  
    ->map_throw<tuple<intop, int, int>>({plus<int>(), 1, 2})  
    ->or_else(pchar('-')->map_throw<tuple<intop, int, int>>(  
      {minus<int>(), 1, 2}))  
    ->or_else(pchar('*')->map_throw<tuple<intop, int, int>>(  
      {multiplies<int>(), 3, 4}))  
    ->or_else(pchar('/')->map_throw<tuple<intop, int, int>>(  
      {divides<int>(), 3, 4})));
```