北京大学计算机学院

2024 年春季学期　　《编译原理》

第三讲　语法分析

Syntax Analysis

主要内容

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

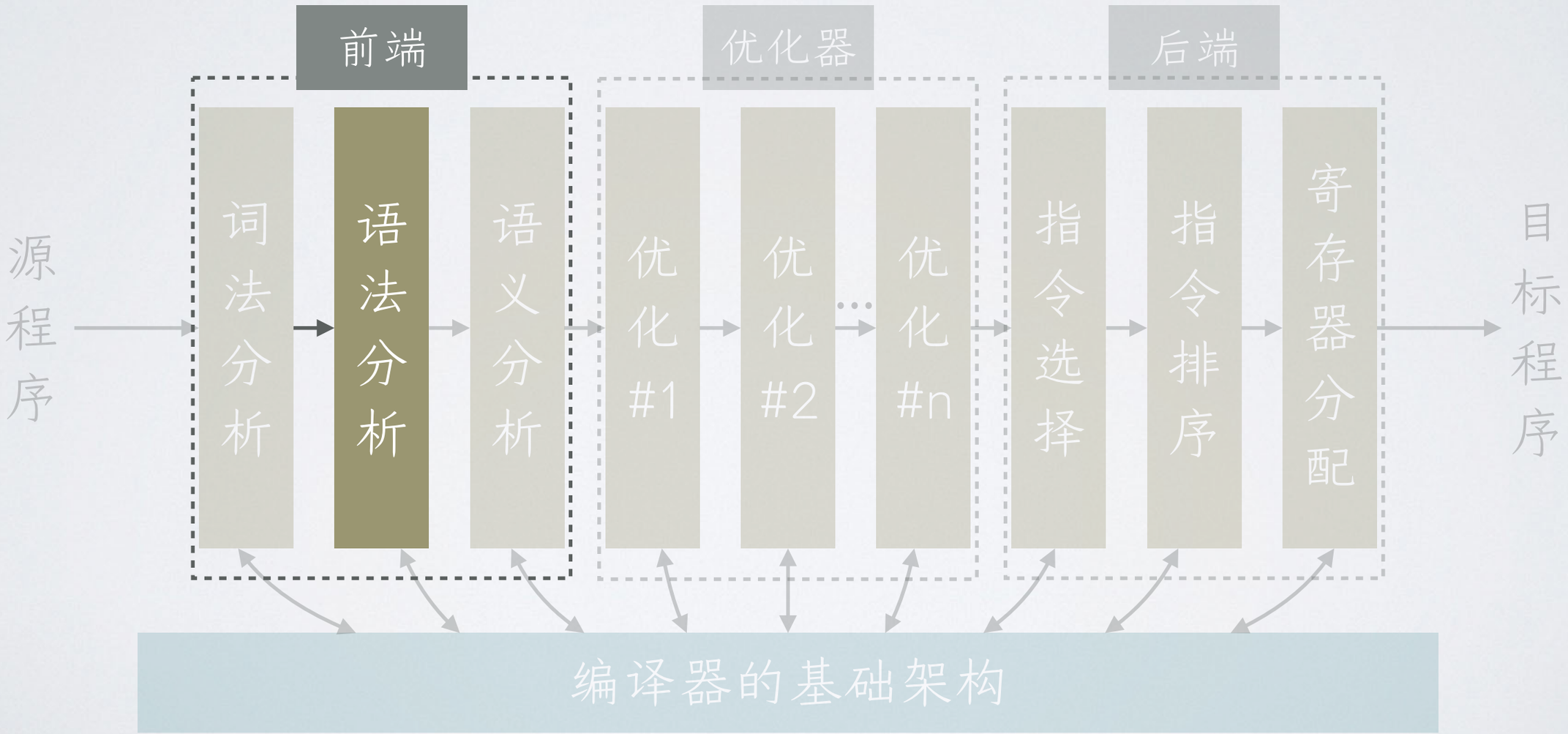
- 对应章节：第 4 章

主要内容

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

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」

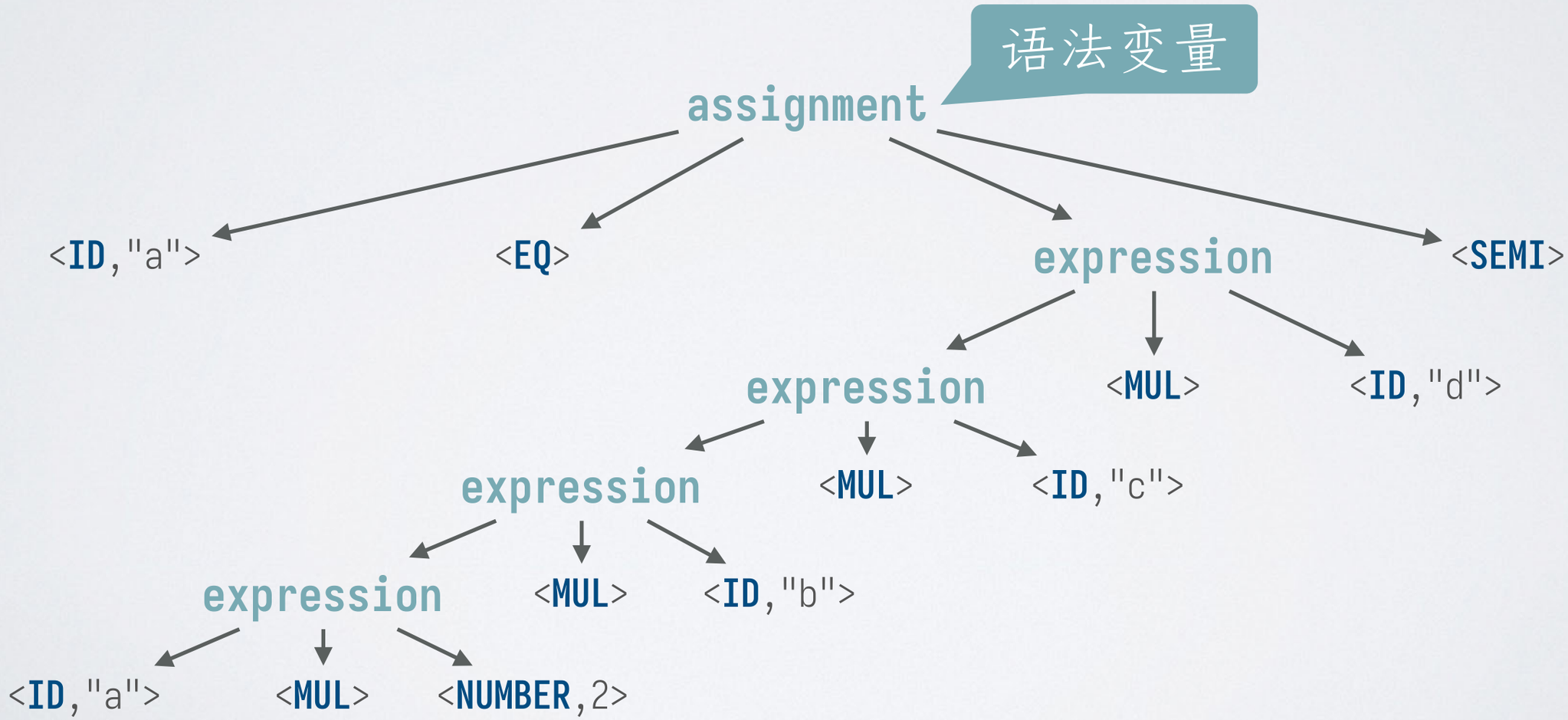


语法分析示例

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

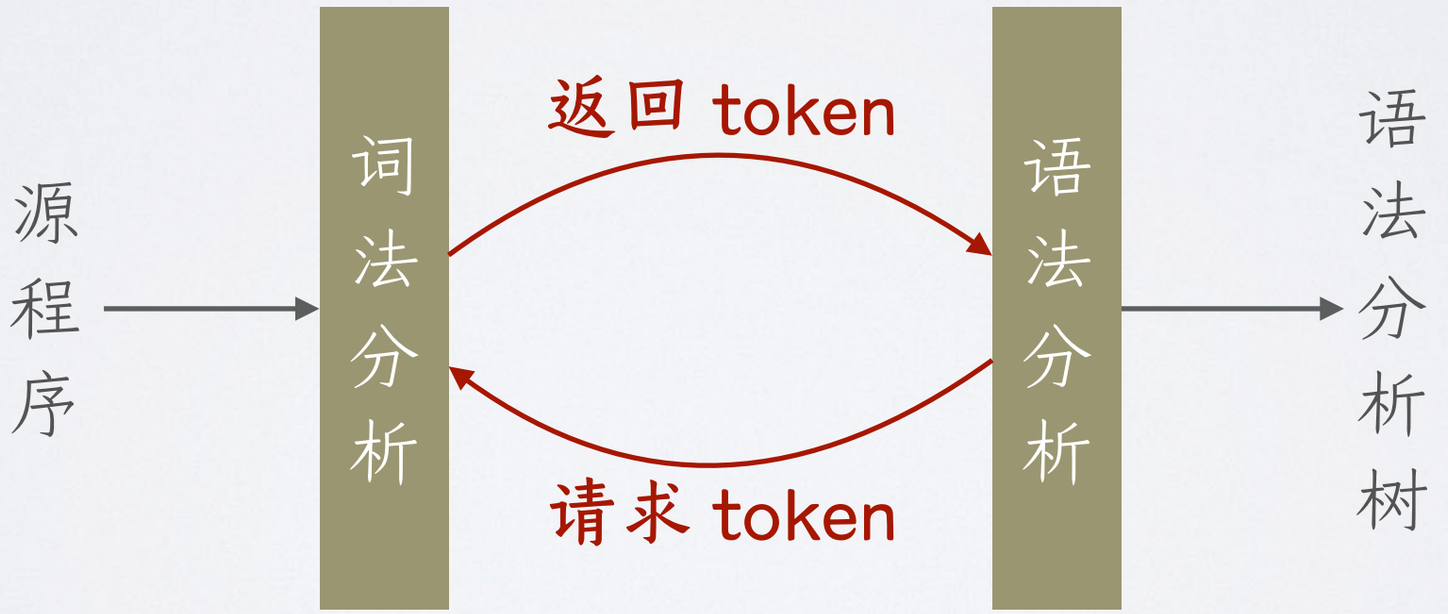
<ID,"a"> <EQ> <ID,"a"> <MUL> <NUMBER,2> <MUL> <ID,"b"> <MUL> <ID,"c"> <MUL> <ID,"d"> <SEMI>

- 从上面的 token 序列构造语法分析树：



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



为什么需要语法分析？

- **什么样的语言不需要语法分析？**

- B****F***(BF)语言
  - 其程序操作一个指针，该指针指向一个全局数组中的位置

```
[->+<]
```

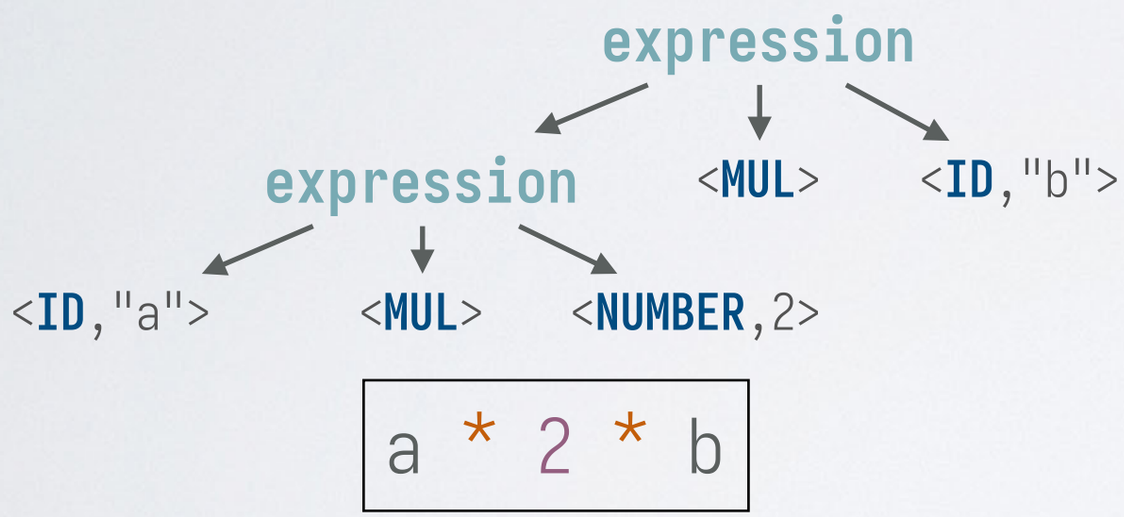
这个 BF 程序把指针指向位置的值加到下一个位置

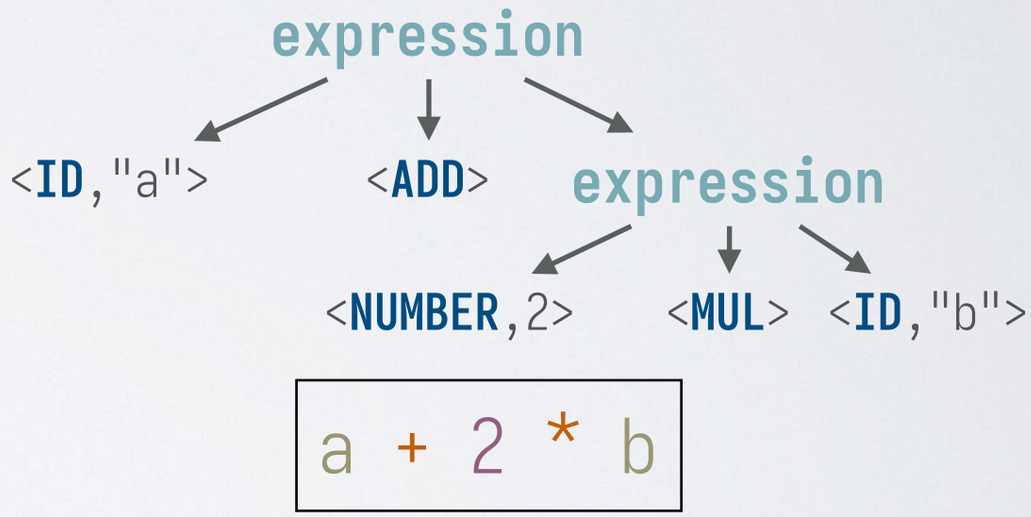
字符	语义
>	把指针往右移一个位置
<	把指针往左移一个位置
+	把指针指向位置的值加一
-	把指针指向位置的值减一
[	如果指针指向位置的值为零，跳转到匹配的] 之后
]	如果指针指向位置的值非零，跳转到匹配的 [之后

- 除了遇到 []，按照源程序字符流从左到右执行
- **不需要语法分析**
- **其实词法分析也不需要**

为什么需要语法分析？

- 常见语言不具有 BF 语言那种**从左到右执行**的语义模型
- 例：四则运算中，我们需要考虑**括号**和运算的**优先级**
- **语法分析：从线性结构（token 序列）到非线性结构（分析树）**





- 语法中蕴含一些**语义信息**

主要内容

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

回顾：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	`!=`

- 每个正则表达式 r 表示了符号表 Σ 上的一个语言 $L(r)$
- 语法分析的规约需要描述**由 token 组成的语言**

文法（Grammar）

- **终结符号**集
  - V_T = {<**动词**, “are”>,<**结束符**, “.”>,<**名词**, “objects”>,<**形容词**, “engineered”>, ……}
- **非终结符号**(语法变量)集
  - V_N = {句子,主语,宾语,修饰}
- **初始符号**
  - S = 句子
- **产生规则**集
  - P = {句子 → 主语 **动词** 宾语 **结束符**,……}

「可以具有如下形式」
「推导」

语法变量

句子 → 主语 **动词** 宾语 **结束符**
主语 → **名词**
宾语 → **名词**
宾语 → 修饰 **名词**
修饰 → **形容词**
……

基于文法进行推导

句子 → 主语 动词 宾语 结束符
主语 → 名词
宾语 → 名词
宾语 → 修饰 名词
修饰 → 形容词
……

句子 ⇒ 主语<动词, “are”> 宾语 <结束符, “.”>

⇒ <名词, “Compilers”><动词, “are”> 宾语 <结束符, “.”>

⇒ <名词, “Compilers”><动词, “are”> 修饰 <名词, “objects”><结束符, “.”>

⇒ <名词, “Compilers”><动词, “are”><形容词, “engineered”><名词, “objects”><结束符, “.”>

Compilers are engineered objects.

文法的形式化定义

- 一个文法 $G = (V_T, V_N, S, P)$ 是一个四元组
  - V_T 是一个非空有限的**终结符号**(terminal symbol)集合
    - 可以理解为该文法的「字母表」
  - V_N 是一个非空有限的**非终结符号**(nonterminal symbol)集合
    - $V_T \cap V_N = \varnothing$
  - $S \in V_N$ 为**初始符号**(start symbol)
  - $P = \{\alpha \to \beta \mid \alpha$ 和 β 是由 $V_T \cup V_N$ 构成的符号串，且 α 中至少有一个非终结符号$\}$ 是一个**产生规则**(production rule)集合

- **推导**表示从初始符号 S 出发，反复应用 P 中的产生规则，直到符号串中只有终结符号
  - 用 $L(G)$ 表示文法能推导出的符号串集合，即其表示的语言

文法示例

- 文法 $G=(V_T, V_N, S, P)$，其中 $V_T=\{\mathsf{a},\mathsf{b},\mathsf{c}\}$，$V_N=\{S,B\}$，$P$ 中有 4 条产生规则(如右侧所示)

$$
\begin{aligned}
S &\Rightarrow \mathsf{a}\,B\,\boxed{S}\,\mathsf{c} && [1]\\
&\Rightarrow \mathsf{a}\,\boxed{B}\,\mathsf{a}\,B\,S\,\mathsf{c}\,\mathsf{c} && [1]\\
&\Rightarrow \mathsf{a}\,\mathsf{a}\,B\,B\,\boxed{S}\,\mathsf{c}\,\mathsf{c} && [3]\\
&\Rightarrow \mathsf{a}\,\mathsf{a}\,B\,\boxed{B}\,\mathsf{a}\,\mathsf{b}\,\mathsf{c}\,\mathsf{c}\,\mathsf{c} && [2]\\
&\Rightarrow \mathsf{a}\,\mathsf{a}\,\boxed{B}\,\mathsf{a}\,B\,\mathsf{b}\,\mathsf{c}\,\mathsf{c}\,\mathsf{c} && [3]\\
&\Rightarrow \mathsf{a}\,\mathsf{a}\,\mathsf{a}\,B\,\boxed{B}\,\mathsf{b}\,\mathsf{c}\,\mathsf{c}\,\mathsf{c} && [3]\\
&\Rightarrow \mathsf{a}\,\mathsf{a}\,\mathsf{a}\,\boxed{B}\,\mathsf{b}\,\mathsf{b}\,\mathsf{c}\,\mathsf{c}\,\mathsf{c} && [4]\\
&\Rightarrow \mathsf{a}\,\mathsf{a}\,\mathsf{a}\,\mathsf{b}\,\mathsf{b}\,\mathsf{c}\,\mathsf{c}\,\mathsf{c} && [4]
\end{aligned}
$$

[1]	$S \to \mathsf{a}\,B\,S\,\mathsf{c}$
[2]	$S \to \mathsf{a}\,\mathsf{b}\,\mathsf{c}$
[3]	$B\,\mathsf{a} \to \mathsf{a}\,B$
[4]	$B\,\mathsf{b} \to \mathsf{b}\,\mathsf{b}$

一般约定：

- 第一条规则的左部是初始符号
- 大写字母表示非终结符号
- 小写字母表示终结符号
- 可以用 $G[S]$ 明确表达 G 的初始符号是 S

上下文无关文法

- 文法的表达能力很强
  - 判断符号串 w 是否属于文法 G 表示的语言是**图灵不可判定**问题
- 上下文无关文法
  - **Context-Free Grammar, CFG**

- 所有产生规则的左边有且仅有一个非终结符号
  - 每个产生规则的形式为 $A \to \beta$, 其中 A 是非终结符号
  - 在推导时不需要知道 A 的前后状况(上下文)
- 能被 CFG 表示的语言被称为上下文无关语言
  - **Context-Free Language, CFL**

CFG 示例

- 上下文无关文法 $G = (V_T, V_N, S, P)$，其中 $V_T = \{[,]\}$，$V_N = \{S\}$，P 中有 3 条产生规则(如右侧所示)

[1]	$S \to \epsilon$
[2]	$S \to [\ S\ ]$
[3]	$S \to S\,S$

- 该文法描述了配平的括号串构成的语言

$$
\begin{aligned}
S &\Rightarrow S\boxed{S} && [3]\\
&\Rightarrow S[\boxed{S}] && [2]\\
&\Rightarrow S[\boxed{S}S] && [3]\\
&\Rightarrow S[[\boxed{S}]S] && [2]\\
&\Rightarrow \boxed{S}[[\,]S] && [1]\\
&\Rightarrow [S][[\,]\boxed{S}] && [2]\\
&\Rightarrow [\boxed{S}][[\,][S]] && [2]\\
&\Rightarrow [\,][[\,][\boxed{S}]] && [1]\\
&\Rightarrow [\,][[\,][\,]] && [1]
\end{aligned}
$$

为什么不在语法分析中使用正则表达式作为规约？

为什么不使用正则表达式？

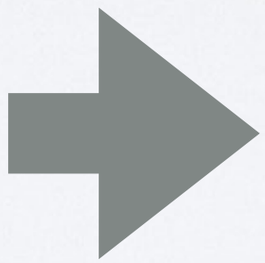
- **能用正则表达式表示配平的括号串构成的语言吗？**
- **很遗憾，不能**
- 但是在编程语言语法中，类似「配平的括号串」的结构很常见

$$
\begin{aligned}
Expr &\rightarrow (\ Expr\ ) \\
Expr &\rightarrow Expr\ Op\ \texttt{ID} \\
Expr &\rightarrow \texttt{ID} \\
Op &\rightarrow + \\
Op &\rightarrow - \\
Op &\rightarrow * \\
Op &\rightarrow /
\end{aligned}
$$

BNF 范式

- Backus-Naur Form, BNF
- 把 $A \to \beta$ 写作 $A ::= \beta$
- 如果 A 对应多个产生规则，可放在一起写作 $A ::= \beta_1 \mid \beta_2$

$$
\begin{aligned}
Expr &\to (\ Expr \ ) \\
Expr &\to Expr \ Op \ \texttt{ID} \\
Expr &\to \texttt{ID} \\
Op &\to \texttt{+} \\
Op &\to \texttt{-} \\
Op &\to \texttt{*} \\
Op &\to \texttt{/}
\end{aligned}
$$



$$
\begin{aligned}
Expr ::= &(\ Expr \ ) \\
\mid \ &Expr \ Op \ \texttt{ID} \\
\mid \ &\texttt{ID} \\
Op ::= &\texttt{+} \\
\mid \ &\texttt{-} \\
\mid \ &\texttt{*} \\
\mid \ &\texttt{/}
\end{aligned}
$$

推导和归约

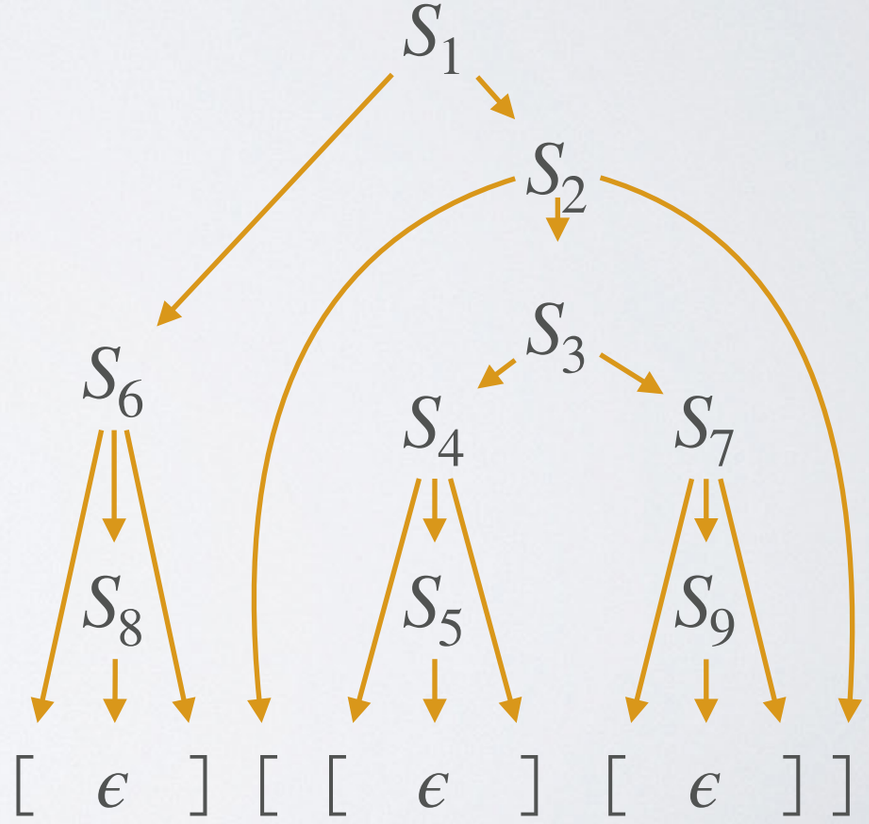
- 考虑文法 $G = (V_T, V_N, S, P)$，若 $\alpha \to \beta$ 是 P 中的产生规则，且 γ、δ 是由 $V_T \cup V_N$ 构成的符号串，则称 $\gamma\alpha\delta$ **直接推导** $\gamma\beta\delta$，表示为 $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$
- 多个直接推导构成的序列被称为**推导**
  - $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$（$n$ 为自然数）可以简记为 $\alpha_0 \Rightarrow^* \alpha_n$
  - 若要求至少一步直接推导（$n > 0$），可记为 $\alpha_0 \Rightarrow^+ \alpha_n$
- 文法 G 表示的语言为 $L(G) = \{w \mid S \Rightarrow^* w, w \text{ 由终结符号构成}\}$
- **归约(reduction)**是**推导(derivation)**的逆过程
  - $\gamma\beta\delta$ 可以被直接归约到 $\gamma\alpha\delta$，可记为 $\gamma\alpha\delta \Leftarrow \gamma\beta\delta$

语法分析树

- **语法分析树**(parse tree)是对 CFG 的推导过程的可视化
  - 根结点是文法的初始符号
  - 每个内部结点(非叶子结点)的标号是非终结符号
  - 每个内部结点表示应用某个产生规则，其孩子结点对应规则的右侧

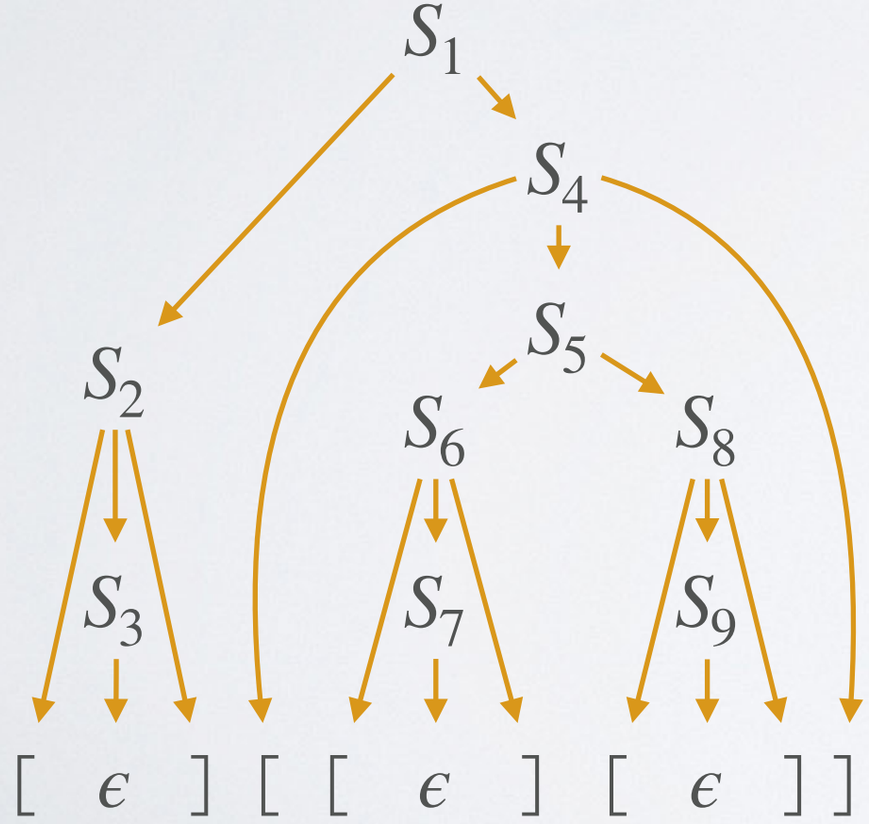
$$
\begin{aligned}
S &\Rightarrow SS && [3]\\
&\Rightarrow S[S] && [2]\\
&\Rightarrow S[SS] && [3]\\
&\Rightarrow S[[S]S] && [2]\\
&\Rightarrow S[[]S] && [1]\\
&\Rightarrow [S][[]S] && [2]\\
&\Rightarrow [S][[][S]] && [2]\\
&\Rightarrow [][[][S]] && [1]\\
&\Rightarrow [][[][]] && [1]
\end{aligned}
$$

[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]$
[3]	$S \to SS$



最左推导和最右推导

- **最左推导**：每一步选择**最左边**的非终结符号进行直接推导
- **最右推导**：每一步选择**最右边**的非终结符号进行直接推导
- 同一棵语法分析树的最左、最右推导序列可能不同

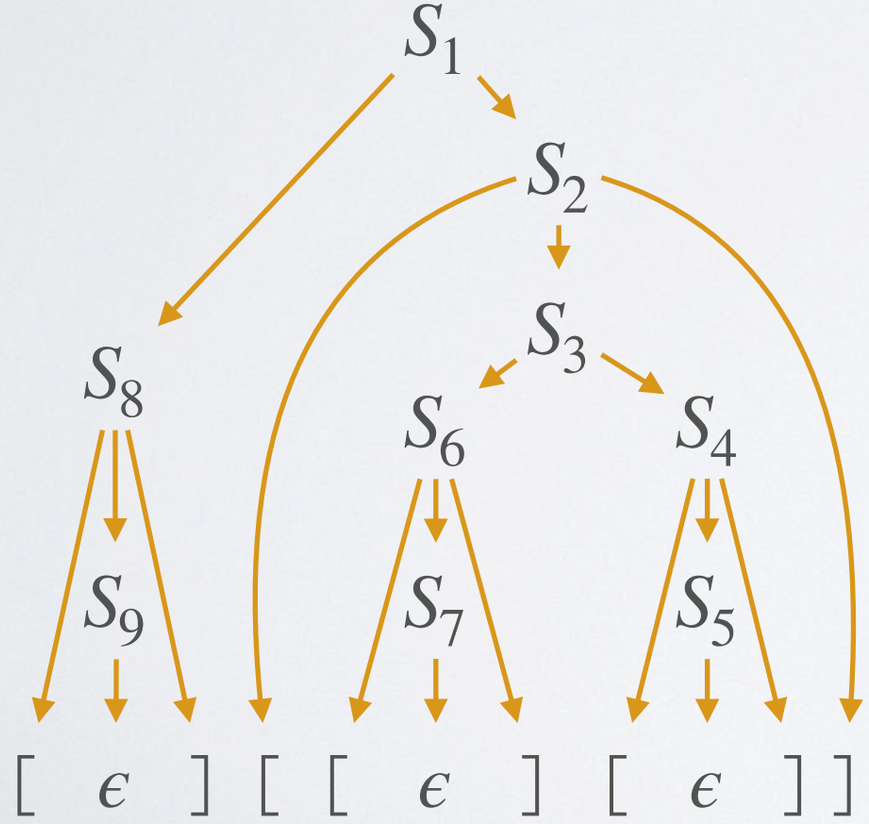


$$
\begin{aligned}
S &\Rightarrow SS \\
&\Rightarrow [S]S \\
&\Rightarrow [\,]S \\
&\Rightarrow [\,][S] \\
&\Rightarrow [\,][SS] \\
&\Rightarrow [\,][[S]S] \\
&\Rightarrow [\,][[\,]S] \\
&\Rightarrow [\,][[\,][S]] \\
&\Rightarrow [\,][[\,][\,]]
\end{aligned}
$$

最左推导

最左推导和最右推导

- **最左推导**：每一步选择**最左边**的非终结符号进行直接推导
- **最右推导**：每一步选择**最右边**的非终结符号进行直接推导
- 同一棵语法分析树的最左、最右推导序列可能不同

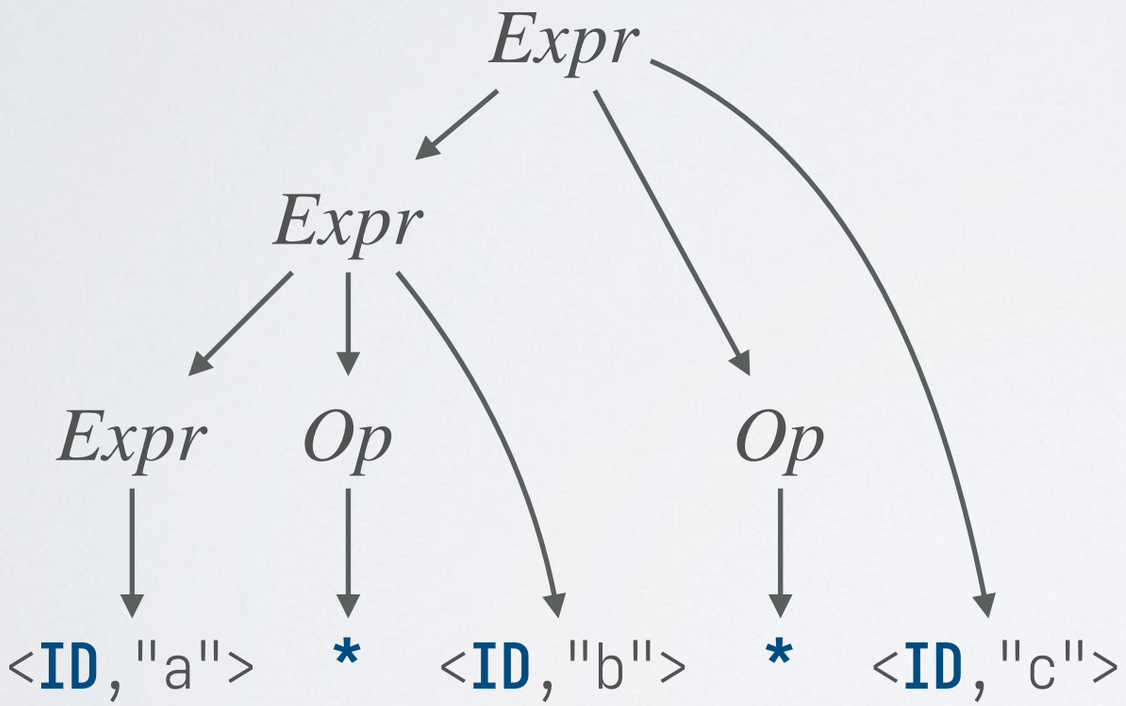


最右推导

$$
\begin{aligned}
S &\Rightarrow SS \\
&\Rightarrow S[S] \\
&\Rightarrow S[SS] \\
&\Rightarrow S[S[S]] \\
&\Rightarrow S[S[]] \\
&\Rightarrow S[[S][]] \\
&\Rightarrow S[[][]] \\
&\Rightarrow [S][[][]] \\
&\Rightarrow [][[][]]
\end{aligned}
$$

语法分析树示例

```
a * b * c
```

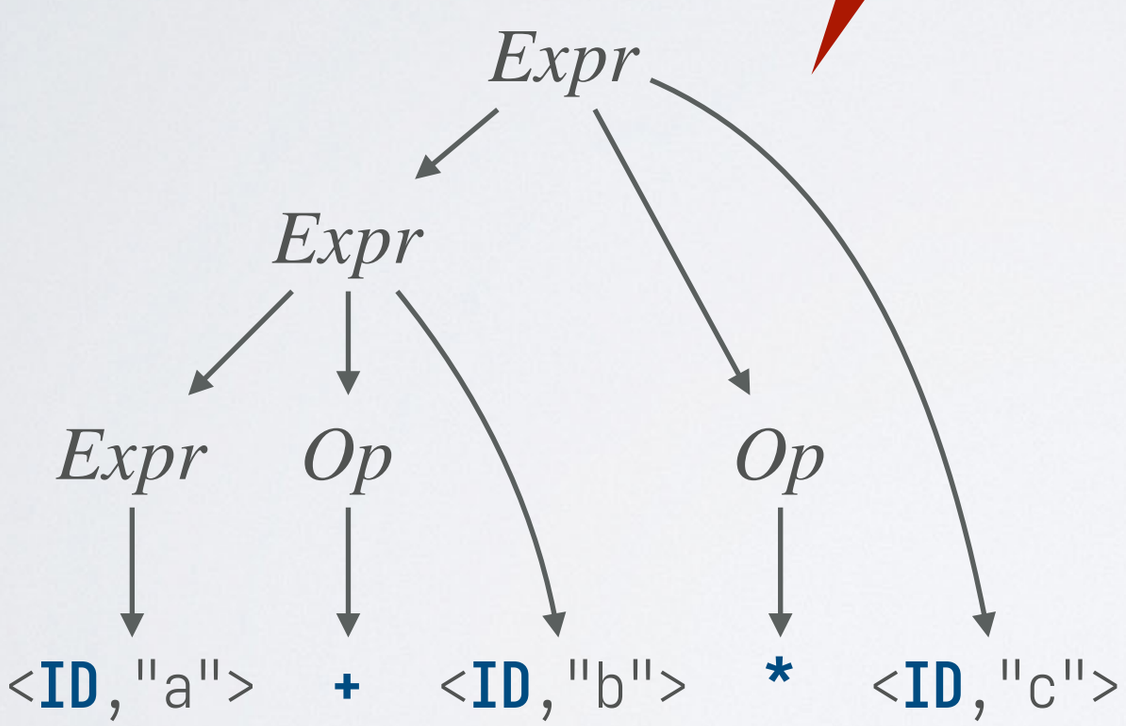


$$
\begin{aligned}
Expr &::= (\ Expr\ ) \\
&\mid\ Expr\ Op\ \texttt{ID} \\
&\mid\ \texttt{ID} \\
Op &::= \texttt{+} \\
&\mid\ \texttt{-} \\
&\mid\ \texttt{*} \\
&\mid\ \texttt{/}
\end{aligned}
$$

语法分析树示例

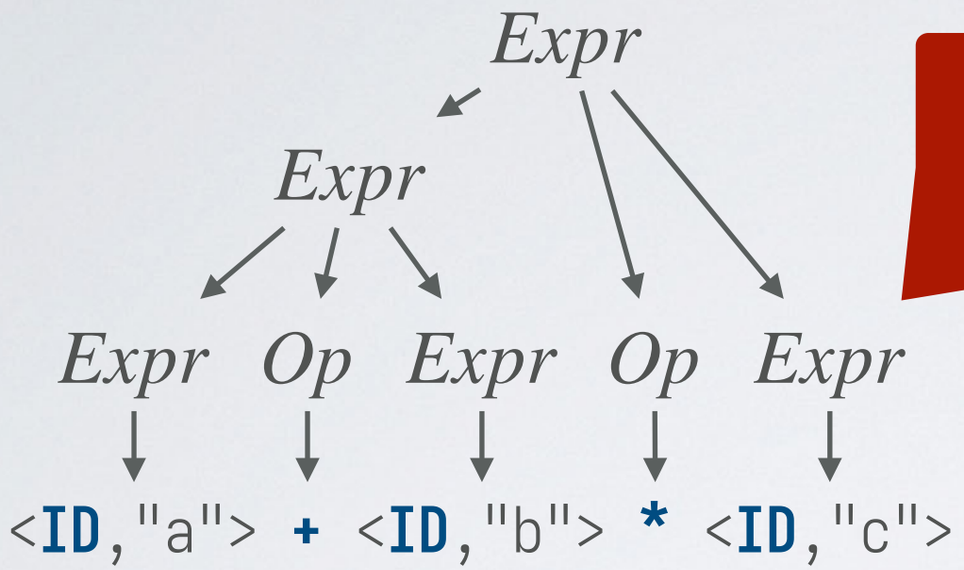
```
a + b * c
```

但语义一般要求先计算乘法



$$
\begin{aligned}
Expr &::= (\ Expr\ ) \\
&\mid\ Expr\ Op\ \texttt{ID} \\
&\mid\ \texttt{ID} \\
Op &::= \texttt{+} \\
&\mid\ \texttt{-} \\
&\mid\ \texttt{*} \\
&\mid\ \texttt{/}
\end{aligned}
$$

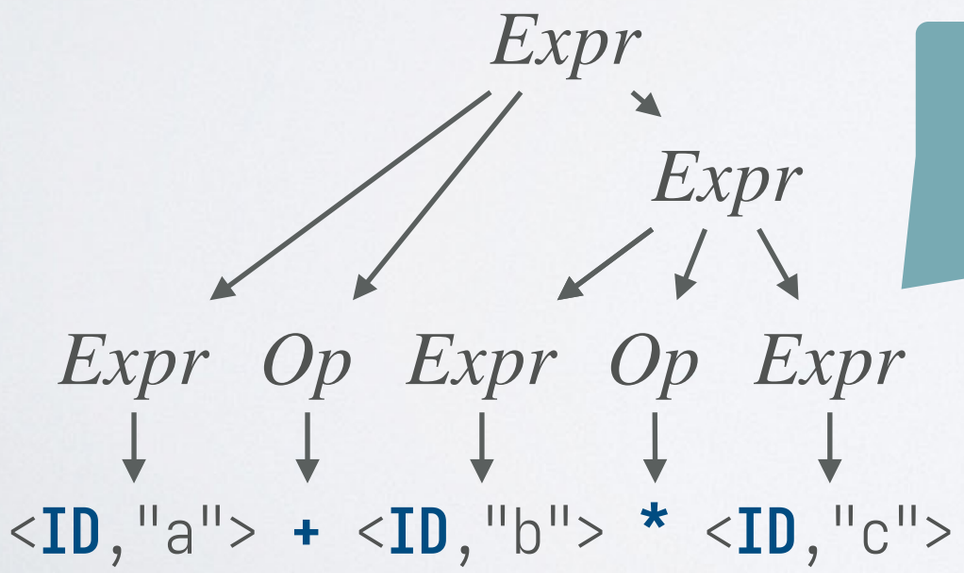
语法分析树示例



先计算加法
再计算乘法

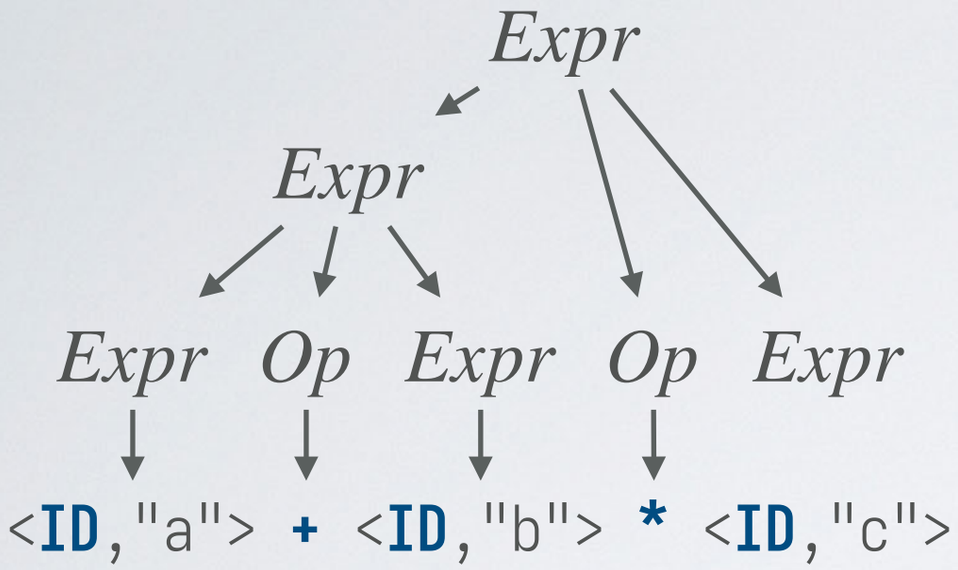
$$
\begin{aligned}
Expr &::= (\ Expr\ ) \\
&\mid\ Expr\ Op\ Expr \\
&\mid\ \texttt{ID} \\
Op &::= \texttt{+} \\
&\mid\ \texttt{-} \\
&\mid\ \texttt{*} \\
&\mid\ \texttt{/}
\end{aligned}
$$

```
a + b * c
```

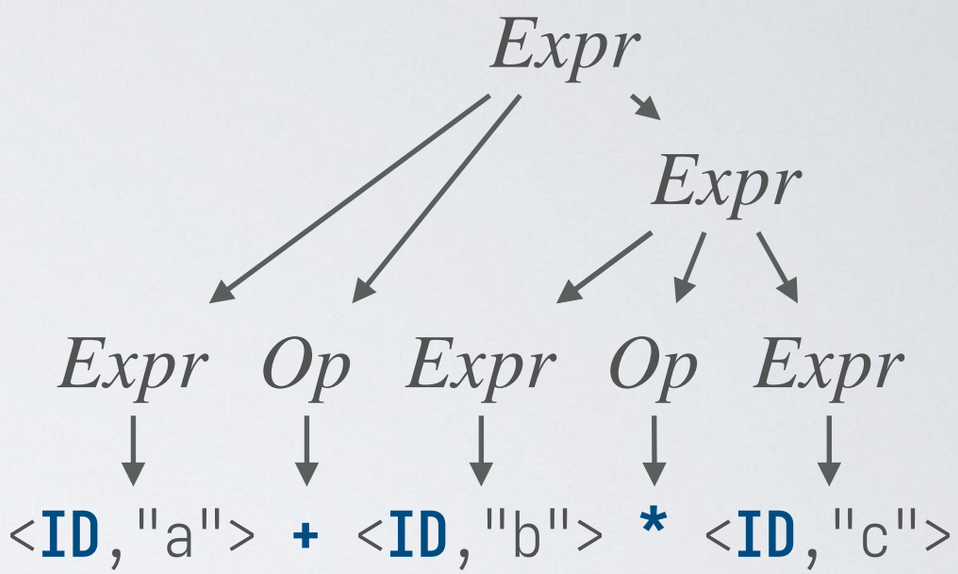


先计算乘法
再计算加法

二义性（ambiguity）



a + b * c



- 如果一个串有两棵不同的分析树，那么该串是**二义性**的
- 如果一个文法产生二义性的串，那么该文法是**二义性**的
- 一般来说，语法分析过程需要无二义性文法
- 对于**一些语言**来说，可以把二义性文法转换为无二义性的

消除二义性

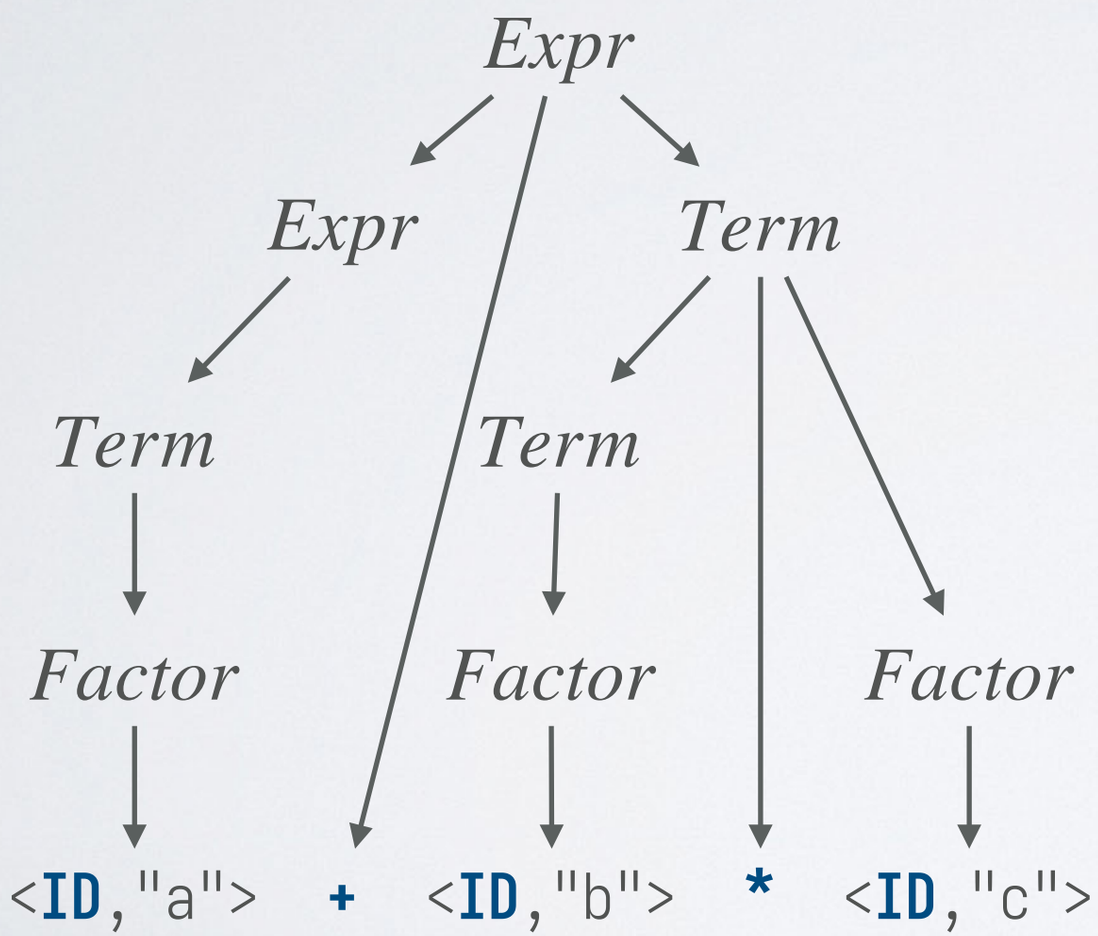
- *Expr* 的文法有二义性的原因是没有在文法中遵循四则运算规则
- 转换文法以体现**运算符优先级**
  - 括号的优先级最高
  - 乘法(*)和除法(/)的优先级次之
  - 加法(+)和减法(-)的优先级最低
- 给每种优先级引入一个**非终结符号**
  - *Factor* 对应括号
  - *Term* 对应乘、除法
  - *Expr* 对应加、减法

$$
\begin{aligned}
Expr &::= (\ Expr\ ) \\
&\mid\ Expr\ Op\ Expr \\
&\mid\ \texttt{ID} \\
Op &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
Expr &::= Expr + Term \\
&\mid\ Expr - Term \\
&\mid\ Term \\
Term &::= Term * Factor \\
&\mid\ Term\ /\ Factor \\
&\mid\ Factor \\
Factor &::= (\ Expr\ ) \\
&\mid\ \texttt{ID}
\end{aligned}
$$

消除二义性

```
a + b * c
```



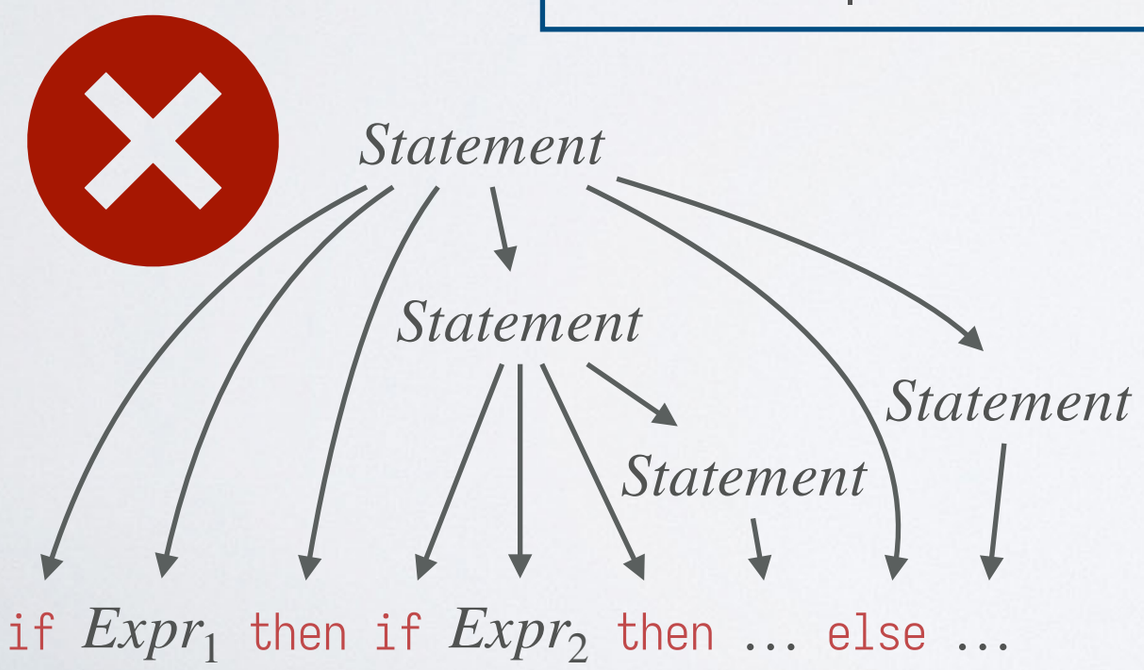
$$
\begin{aligned}
Expr &::= Expr + Term \\
&\mid Expr - Term \\
&\mid Term \\
Term &::= Term * Factor \\
&\mid Term / Factor \\
&\mid Factor \\
Factor &::= (\ Expr\ ) \\
&\mid \texttt{ID}
\end{aligned}
$$

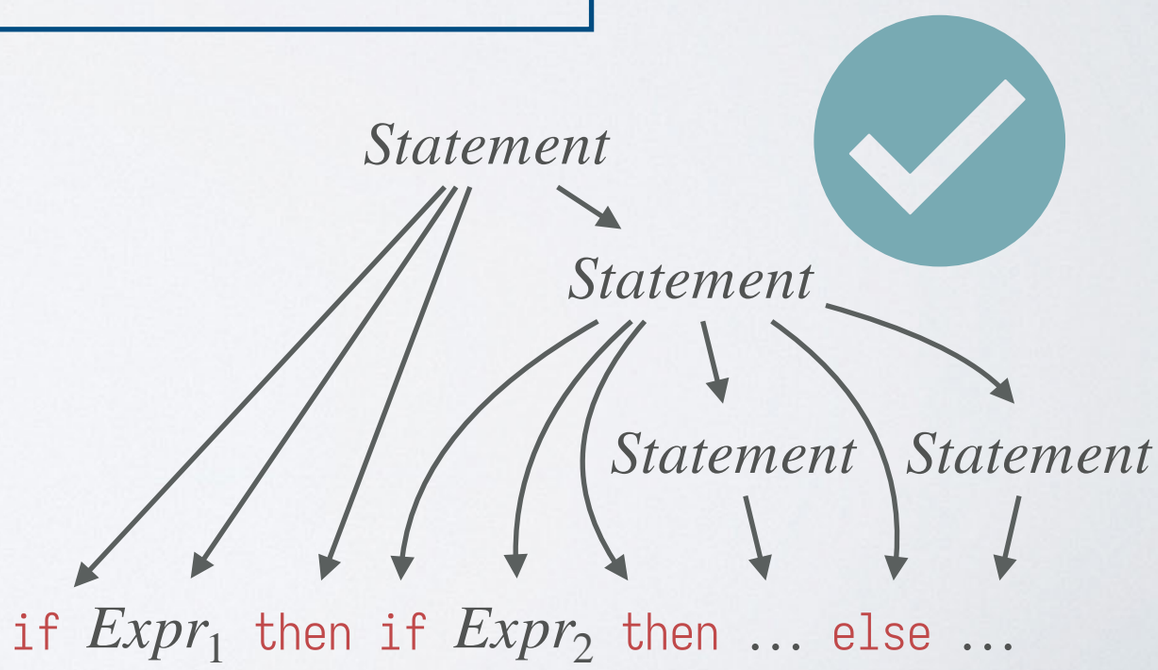
消除二义性

- 另一个经典的二义性例子是 if-then-else 语句

- 二义性文法：

$$
\begin{aligned}
Statement ::= & \texttt{if}\ Expr\ \texttt{then}\ Statement\ \texttt{else}\ Statement \\
| \ & \texttt{if}\ Expr\ \texttt{then}\ Statement \\
| \ & \ldots
\end{aligned}
$$





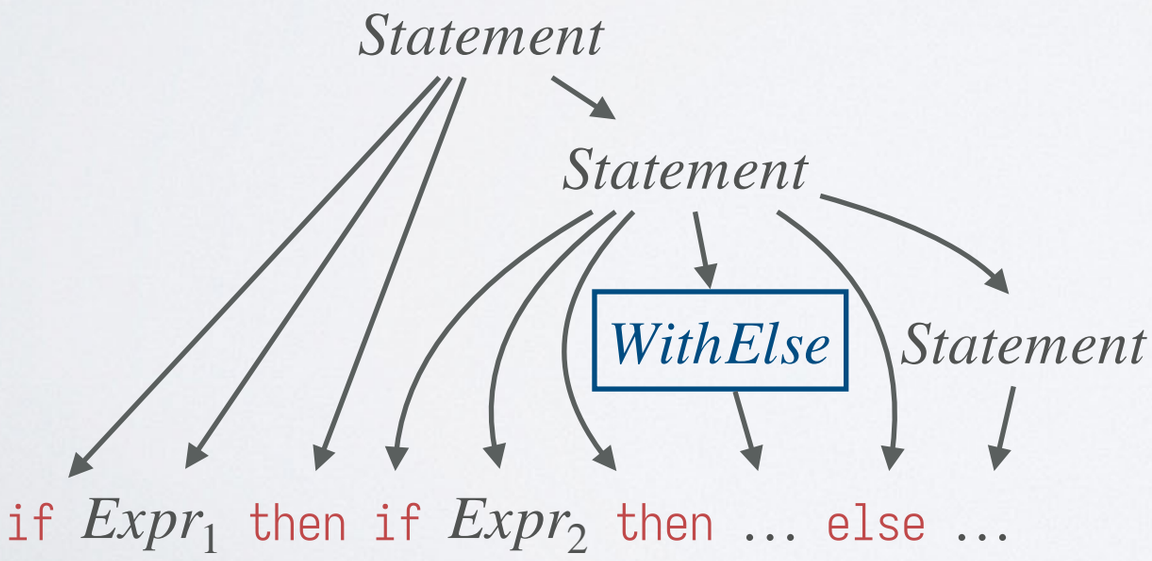
消除二义性

Statement ::= `if` *Expr* `then` *Statement* `else` *Statement*
| `if` *Expr* `then` *Statement*
| …

Statement ::= `if` *Expr* `then` *Statement*
| `if` *Expr* `then` *WithElse* `else` *Statement*
| …

WithElse ::= `if` *Expr* `then` *WithElse* `else` *WithElse*
| …

每个 else 与离它最近的一个能对应的 if 进行对应



如果 *WithElse* 继续产生出 if 语句，它必定会产生对应起来的 else

CFG 作为语法分析的规约

$$
\begin{aligned}
Expr &::= Expr + Term \\
&\mid Expr - Term \\
&\mid Term \\
Term &::= Term * Factor \\
&\mid Term / Factor \\
&\mid Factor \\
Factor &::= (Expr) \\
&\mid \texttt{ID} \mid \texttt{INT}
\end{aligned}
$$

$$
\begin{aligned}
BDisj &::= BDisj \texttt{ || } BConj \\
&\mid BConj \\
BConj &::= BConj \texttt{ \&\& } BCmp \\
&\mid BCmp \\
BCmp &::= Expr \texttt{ == } Expr \mid Expr \texttt{ <= } Expr \mid \ldots \\
&\mid BAtom \\
BAtom &::= (BDisj) \mid \texttt{!} BAtom \\
&\mid \texttt{true} \mid \texttt{false}
\end{aligned}
$$

$$
\begin{aligned}
Stmt &::= \{ Block \} \\
&\mid \texttt{ID} = Expr \texttt{ ;} \\
&\mid \texttt{if (} BDisj \texttt{)} Stmt \texttt{ else } Stmt \\
&\mid \texttt{while (} BDisj \texttt{)} Stmt \\
&\mid \texttt{return } Expr \texttt{ ;} \\
Block &::= \epsilon \\
&\mid Stmt\ Block
\end{aligned}
$$

```
{
  n = 10; a = 1; b = 1;
  while (!(n == 0)) {
    t = a + b; a = b; b = t;
    n = n - 1;
  }
  return a;
}
```

CFG 作为语法分析的规约

文言 / wenyan-lang

$$
\begin{aligned}
Stmt &::= IfStmt \mid \ldots \\
Stmts &::= \epsilon \\
&\mid\ Stmt\ Stmts \\
IfStmt &::= \text{若}\ IfExpr\ \text{者}\ Stmts\ \text{若非}\ Stmts\ IfEnd \\
IfEnd &::= \text{云云} \mid \text{也} \\
IfExpr &::= UnaryIfExpr\ Op\ UnaryIfExpr \mid \ldots \\
UnaryIfExpr &::= \text{ID} \mid \text{INT} \mid \ldots \\
Op &::= \text{等於} \mid \text{不等於} \mid \ldots
\end{aligned}
$$

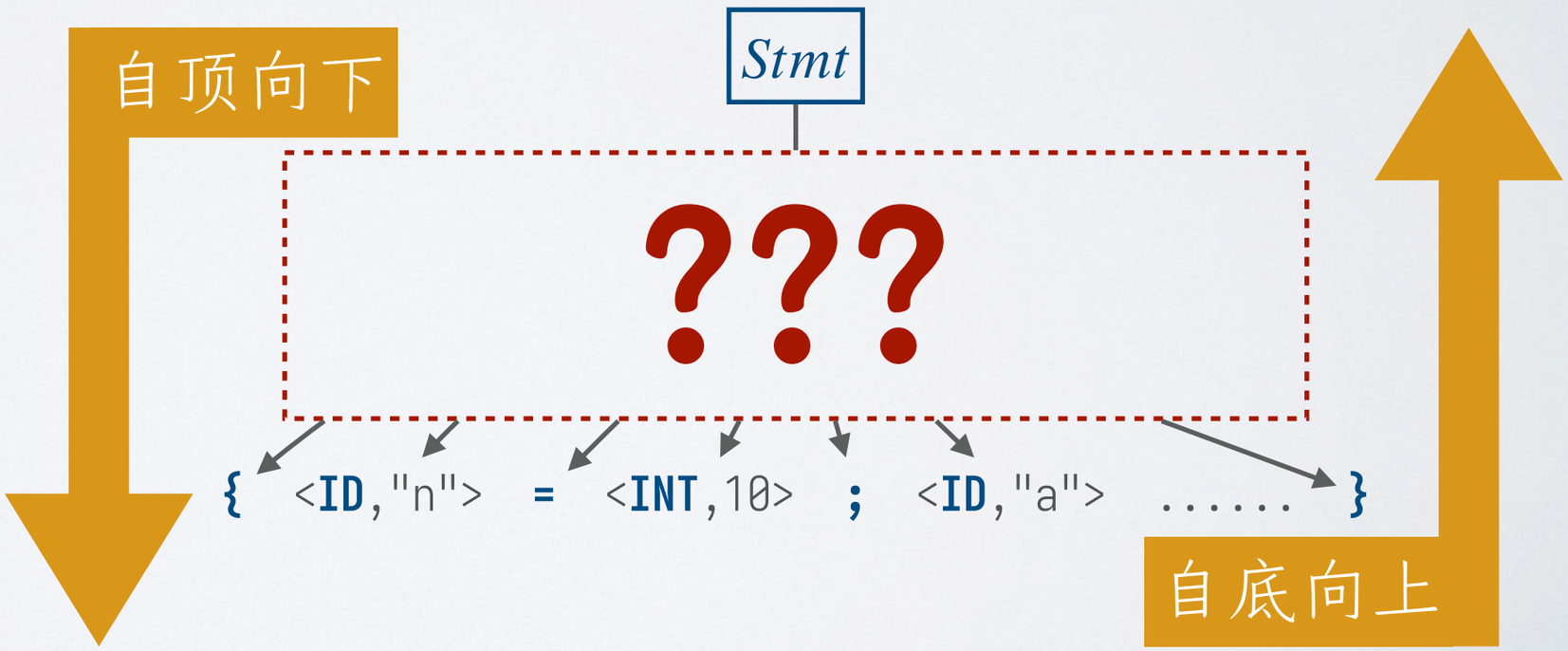
若「甲」等於一者
……
若非
……
也

主要内容

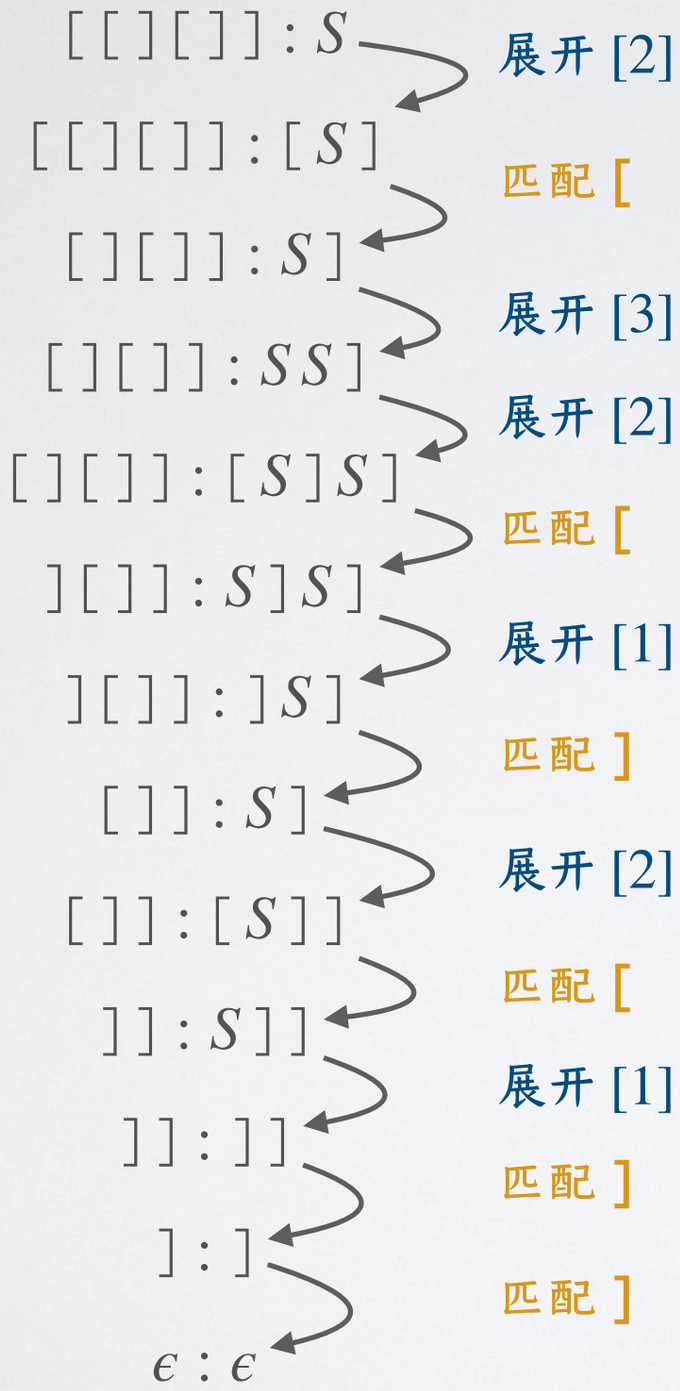
- 语法分析的作用
- 语法分析的规约
- **语法分析的手动实现**
- 语法分析的自动生成

如何构造语法分析树？


```
{
  n = 10; a = 1; b = 1;
  while (!(n == 0)) {
    t = a + b; a = b; b = t;
    n = n - 1;
  }
  return a;
}
```



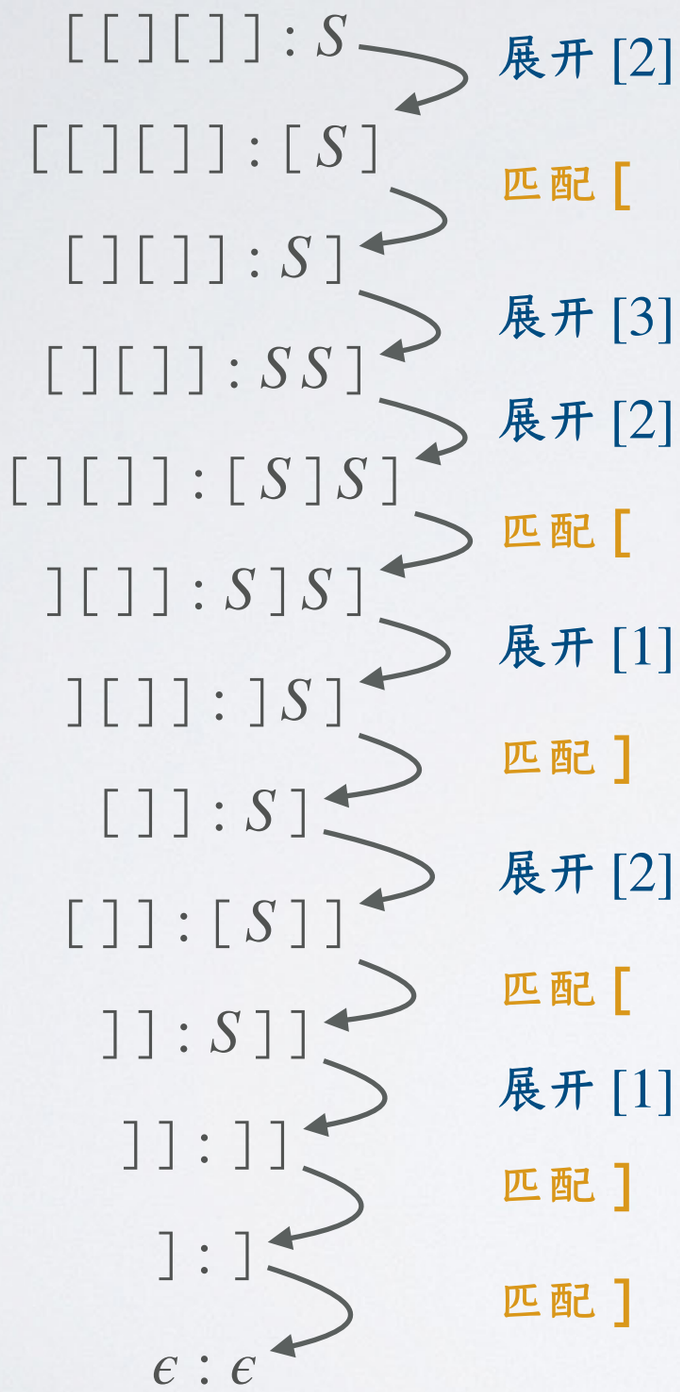
自顶向下语法分析



[1]	$S \to \epsilon$
[2]	$S \to [\, S \,]$
[3]	$S \to SS$

- 用 $w:\beta$ 表示存在推导关系 $\beta \Rightarrow^* w$，其中 w 表示 token 流，β 表示分析目标
- 从 β **最左边**的非终结符号出发，展开产生规则
- 如果 β **最左边**出现了终结符号，则把它与输入的 token 匹配

自顶向下分析构造最左推导



[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]$
[3]	$S \to SS$

$$
\begin{aligned}
S &\Rightarrow [\,S\,] \\
&\Rightarrow [\,SS\,] \\
&\Rightarrow [\,[\,S\,]S\,] \\
&\Rightarrow [\,[\,]S\,] \\
&\Rightarrow [\,[\,][S\,]\,] \\
&\Rightarrow [\,[\,][\,]\,]
\end{aligned}
$$

基于递归下降实现自顶向下分析

```
S'() {
  if (S()) {
    if (token == EOF) {
      return true;
    }
  }
  return false;
}
```

全局变量记录当前的 token

```
S1() {
  return true;
}
```

```
S2() {
  if (token == LSQUARE) {
    token = next_token();
    if (S()) {
      if (token == RSQUARE) {
        token = next_token();
        return true;
      }
    }
  }
  return false;
}
```

调用词法分析获取下一个 token

```
S3() {
  if (S()) {
    if (S()) {
      return true;
    }
  }
  return false;
}
```

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]$
[3]	$S \to S\,S$

问题： S 的文法有二义性，比如 $S \Rightarrow SS \Rightarrow S$

基于递归下降实现自顶向下分析

```
S'() {
  if (S()) {
    if (token == EOF) {
      return true;
    }
  }
  return false;
}
```

```
S1() {
  return true;
}
```

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

```
S2() {
  if (token == LSQUARE) {
    token = next_token();
    if (S()) {
      if (token == RSQUARE) {
        token = next_token();
        if (S()) return true;
      }
    }
  }
  return false;
}
```

问题：S 有两条规则，不知道分析时要用哪一条

解决方案：

- 回溯尝试多种方案
- 进行预测分析

预测分析

- 通过「向前看」(lookahead)符号来选择产生规则
  - 在尝试分析 S 时，考察当前的 token：
    - 如果是 [，则应该选择 [2]
    - 如果是 EOF 或]，则应该选择 [1]
    - 否则，则出现了语法错误

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

```
S'() {
  if (S()) {
    if (token == EOF) {
      return true;
    }
  }
  return false;
}
```

```
S() {
  if (token == LSQUARE) {
    token = next_token();
    if (S()) {
      if (token == RSQUARE) {
        token = next_token();
        if (S()) return true;
      }
    }
  } else if (token == EOF || token == RSQUARE) return true;
  return false;
}
```

自顶向下分析的问题：左递归

$$Expr ::= Expr + Term \mid Expr - Term \mid \ldots$$

```
Expr() {
  if (Expr()) {
    ......
  }
}
```

- **直接左递归：**文法中有形如 $A \to A\alpha$ 的产生规则
- **间接左递归：**文法中能导出形如 $A \Rightarrow^{+} A\alpha$ 的推导

例：$S ::= Aa \mid b$

$A ::= Sd \mid \epsilon$

消除直接左递归

- 把 $A ::= A\,\alpha \mid \beta$ 转换为 $A ::= \beta\,A'$，$A' ::= \alpha\,A' \mid \epsilon$

$$
\begin{aligned}
Expr &::= Expr + Term \\
&\mid\ Expr - Term \\
&\mid\ Term \\
Term &::= Term * Factor \\
&\mid\ Term\ /\ Factor \\
&\mid\ Factor \\
Factor &::= (\ Expr\ ) \\
&\mid\ \texttt{ID}
\end{aligned}
$$

$$
\begin{aligned}
Expr &::= Term\ Expr' \\
Expr' &::= + Term\ Expr' \\
&\mid\ - Term\ Expr' \\
&\mid\ \epsilon \\
Term &::= Factor\ Term' \\
Term' &::= * Factor\ Term' \\
&\mid\ /\ Factor\ Term' \\
&\mid\ \epsilon \\
Factor &::= (\ Expr\ ) \\
&\mid\ \texttt{ID}
\end{aligned}
$$

消除间接左递归

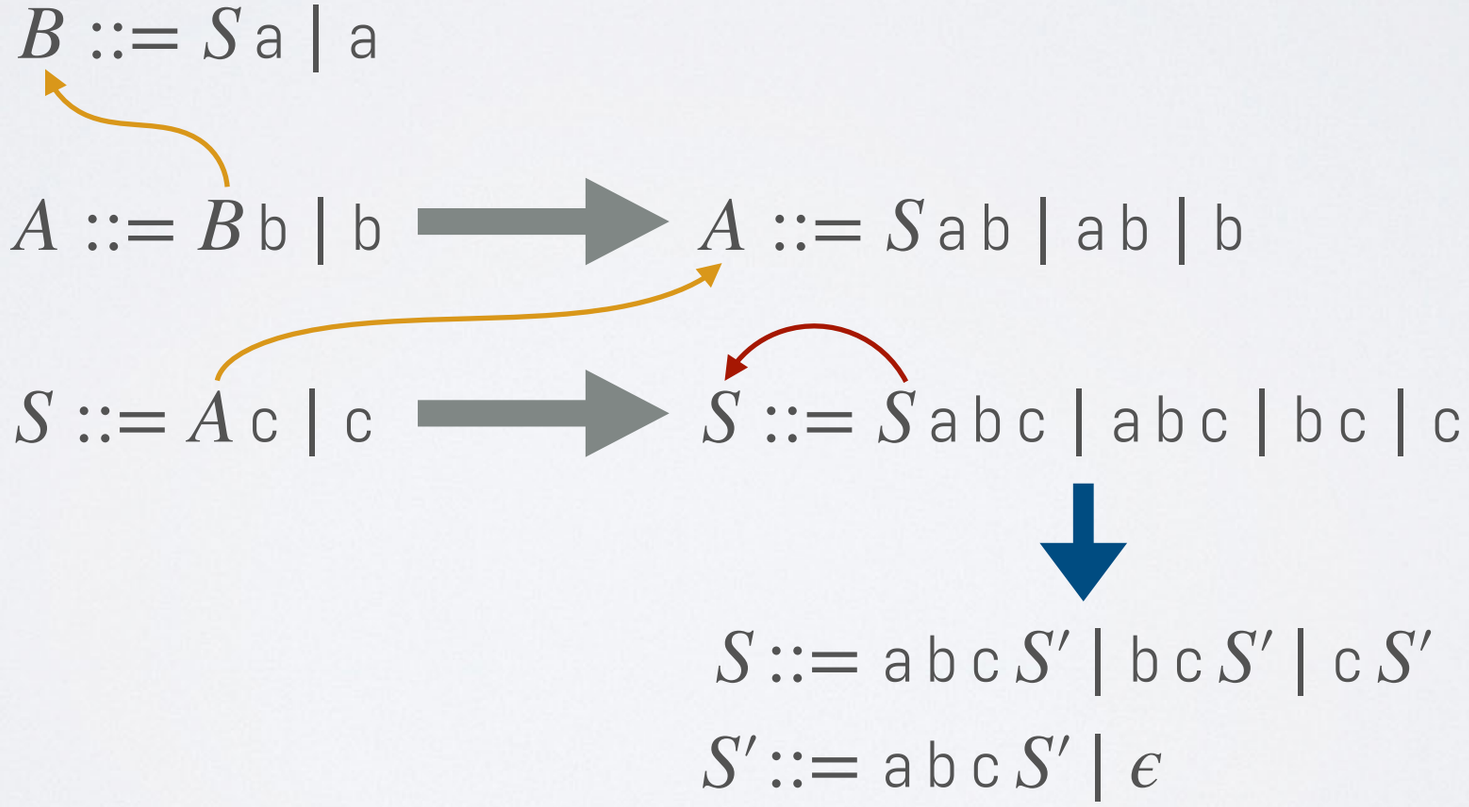
- 把文法的非终结符号整理成某种顺序 $A_1, A_2, \ldots, A_n$
- 依次遍历每个非终结符号 A_i
  - 对每个 $j < i$ 和形如 $A_i ::= A_j \gamma$ 的规则替换为 $A_i ::= \delta_1 \gamma \mid \ldots \mid \delta_k \gamma$
    - 其中 $A_j ::= \delta_1 \mid \ldots \mid \delta_k$ 是 A_j 的产生规则
  - 消除 $A_i ::= \delta_1 \gamma \mid \ldots \mid \delta_k \gamma$ 中的直接左递归
- 最终得到的文法依赖于非终结符号的处理顺序，但表示的语言都是等价的

左递归消除示例

$$
\begin{aligned}
S &::= Ac \mid c \\
A &::= Bb \mid b \\
B &::= Sa \mid a
\end{aligned}
$$

按照 B、A、S 的顺序进行处理

$$
\begin{aligned}
S &::= abcS' \mid bcS' \mid cS' \\
S' &::= abcS' \mid \epsilon \\
A &::= Sab \mid ab \mid b \\
B &::= Sa \mid a
\end{aligned}
$$



自顶向下分析的问题：预测的确定性

- 若非终结符号 A 有多条产生规则 $A ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$，如何确定应该选择哪条规则进行推导？

- 预测分析中可以知道当前的 token 是什么
  - **「向前看」(lookahead)**：也可以多看几个 token

- 如果当前 token 是 a，而且只有 β_i 能以 a 开头，那么就可以确定应该选择 $A \to \beta_i$ 这条规则进行推导

- **问题：计算 FIRST(β) 集合**，即 β 推导出的符号串的首个终结符号可能是哪些

计算 FIRST 集合

$$
\begin{aligned}
S &::= Expr\ \texttt{EOF} \\
Expr &::= Term\ Expr' \\
Expr' &::= \texttt{+}\ Term\ Expr' \\
&\mid\ \texttt{-}\ Term\ Expr' \\
&\mid\ \epsilon \\
Term &::= Factor\ Term' \\
Term' &::= \texttt{*}\ Factor\ Term' \\
&\mid\ \texttt{/}\ Factor\ Term' \\
&\mid\ \epsilon \\
Factor &::= \texttt{(}\ Expr\ \texttt{)} \\
&\mid\ \texttt{ID}
\end{aligned}
$$

$\text{FIRST}(S) = \{\texttt{(},\texttt{ID}\}$

$\text{FIRST}(Expr) = \{\texttt{(},\texttt{ID}\}$

$\text{FIRST}(Expr') = \{\texttt{+},\texttt{-},\epsilon\}$

$\text{FIRST}(Term) = \{\texttt{(},\texttt{ID}\}$

$\text{FIRST}(Term') = \{\texttt{*},\texttt{/},\epsilon\}$

$\text{FIRST}(Factor) = \{\texttt{(},\texttt{ID}\}$

- 如果 $\beta \Rightarrow^* \epsilon$，规定有 $\epsilon \in \text{FIRST}(\beta)$

计算 FIRST 集合

- 首先有 $\mathrm{FIRST}(a) = \{a\}$，其中 a 为终结符号
- 对于每个非终结符号 A，执行以下操作：
  - 如果有产生规则 $A \to \epsilon$，则把 ϵ 加入 $\mathrm{FIRST}(A)$
  - 如果有产生规则 $A \to X_1 X_2 \ldots X_k$，其中每个 X 是一个符号，那么
    - 如果 $a \in \mathrm{FIRST}(X_i)$ 且对任意 $j < i$ 有 $\epsilon \in \mathrm{FIRST}(X_j)$，则把 a 加入 $\mathrm{FIRST}(A)$
    - 如果对所有 j 都有 $\epsilon \in \mathrm{FIRST}(X_j)$，则把 ϵ 加入 $\mathrm{FIRST}(A)$
- 重复上一步直到所有的 FIRST 集合都不变

计算 FIRST 集合

$$
\begin{aligned}
S &::= Expr\ \texttt{EOF} \\
Expr &::= Term\ Expr' \\
Expr' &::= \texttt{+}\ Term\ Expr' \\
&\mid\ \texttt{-}\ Term\ Expr' \\
&\mid\ \epsilon \\
Term &::= Factor\ Term' \\
Term' &::= \texttt{*}\ Factor\ Term' \\
&\mid\ \texttt{/}\ Factor\ Term' \\
&\mid\ \epsilon \\
Factor &::= \texttt{(}\ Expr\ \texttt{)} \\
&\mid\ \texttt{ID}
\end{aligned}
$$

$\text{FIRST}(S) = \{\texttt{(}, \texttt{ID}\}$

$\text{FIRST}(Expr) = \{\texttt{(}, \texttt{ID}\}$

$\text{FIRST}(Expr') = \{\texttt{+}, \texttt{-}, \epsilon\}$

$\text{FIRST}(Term) = \{\texttt{(}, \texttt{ID}\}$

$\text{FIRST}(Term') = \{\texttt{*}, \texttt{/}, \epsilon\}$

$\text{FIRST}(Factor) = \{\texttt{(}, \texttt{ID}\}$

计算 FIRST 集合

- 对于 $\beta = X_1 X_2 \ \ldots \ X_k$, 计算 $\text{FIRST}(\beta)$ 的方法如下:
  - 如果 $a \in \text{FIRST}(X_i)$ 且对任意 $j < i$ 有 $\epsilon \in \text{FIRST}(X_j)$, 则把 a 加入 $\text{FIRST}(\beta)$
  - 如果对所有 j 都有 $\epsilon \in \text{FIRST}(X_j)$, 则把 ϵ 加入 $\text{FIRST}(\beta)$

$$
\begin{aligned}
\mathit{Expr}' &::= + \ \mathit{Term}\ \mathit{Expr}' \\
&\mid \ - \ \mathit{Term}\ \mathit{Expr}' \\
&\mid \ \epsilon \\
\mathit{Term}' &::= * \ \mathit{Factor}\ \mathit{Term}' \\
&\mid \ / \ \mathit{Factor}\ \mathit{Term}' \\
&\mid \ \epsilon \\
\mathit{Factor} &::= (\ \mathit{Expr} \ ) \\
&\mid \ \texttt{ID}
\end{aligned}
$$

$$
\begin{aligned}
&\text{FIRST}(+\ \mathit{Term}\ \mathit{Expr}') = \{+\} \\
&\text{FIRST}(-\ \mathit{Term}\ \mathit{Expr}') = \{-\} \\
&\text{FIRST}(\epsilon) = \{\epsilon\} \\
&\text{FIRST}(*\ \mathit{Factor}\ \mathit{Term}') = \{*\} \\
&\text{FIRST}(/\ \mathit{Factor}\ \mathit{Term}') = \{/\} \\
&\text{FIRST}(\epsilon) = \{\epsilon\} \\
&\text{FIRST}(\ (\ \mathit{Expr}\ )\ ) = \{(\} \\
&\text{FIRST}(\texttt{ID}) = \{\texttt{ID}\}
\end{aligned}
$$

对 ϵ 如何进行预测分析?

预测分析中的 ϵ

$$
\begin{array}{rcl}
Expr' & ::= & + \; Term \; Expr' \\
& | & - \; Term \; Expr' \\
& | & \epsilon
\end{array}
$$

$$\text{FIRST}(+ \; Term \; Expr') = \{+\}$$
$$\text{FIRST}(- \; Term \; Expr') = \{-\}$$
$$\text{FIRST}(\epsilon) = \{\epsilon\}$$

- 在推导 $Expr'$ 时，如果当前 token 为 + 或 −，我们可以根据 FIRST 集合来选择产生规则
- **问题：什么情况下应该选择 $Expr' \to \epsilon$？**
- 如果当前的 token 可以**紧接着 $Expr'$ 推导的串之后**！
- 需要计算每个非终结符号的 **FOLLOW 集合**

计算 FOLLOW 集合

$$
\begin{aligned}
S &::= Expr\ \texttt{EOF} \\
Expr &::= Term\ Expr' \\
Expr' &::= \texttt{+}\ Term\ Expr' \\
&\mid\ \texttt{-}\ Term\ Expr' \\
&\mid\ \epsilon \\
Term &::= Factor\ Term' \\
Term' &::= \texttt{*}\ Factor\ Term' \\
&\mid\ \texttt{/}\ Factor\ Term' \\
&\mid\ \epsilon \\
Factor &::= \texttt{(}\ Expr\ \texttt{)} \\
&\mid\ \texttt{ID}
\end{aligned}
$$

$$\text{FOLLOW}(Expr) = \{\texttt{EOF},\texttt{)}\}$$

$$\text{FOLLOW}(Expr') = \{\texttt{EOF},\texttt{)}\}$$

$$\text{FOLLOW}(Term) = \{\texttt{+},\texttt{-},\texttt{EOF},\texttt{)}\}$$

$$\text{FOLLOW}(Term') = \{\texttt{+},\texttt{-},\texttt{EOF},\texttt{)}\}$$

$$\text{FOLLOW}(Factor) = \{\texttt{*},\texttt{/},\texttt{+},\texttt{-},\texttt{EOF},\texttt{)}\}$$

计算 FOLLOW 集合

- 对于每个非终结符号 X, 执行以下操作:
  - 如果有产生规则 $A \to \alpha X \beta$, 则把 FIRST(β) 中的非 ϵ 符号加入 FOLLOW(X)
  - 如果有产生规则 $A \to \alpha X \beta$ 且 $\epsilon \in$ FIRST(β), 则把 FOLLOW(A) 中的符号加入 FOLLOW(X)
- 重复上一步直到所有的 FOLLOW 集合都不变

计算 FOLLOW 集合

$$
\begin{aligned}
S &::= Expr\ \texttt{EOF} \\
Expr &::= Term\ Expr' \\
Expr' &::= \texttt{+}\ Term\ Expr' \\
&\mid\ \texttt{-}\ Term\ Expr' \\
&\mid\ \epsilon \\
Term &::= Factor\ Term' \\
Term' &::= \texttt{*}\ Factor\ Term' \\
&\mid\ \texttt{/}\ Factor\ Term' \\
&\mid\ \epsilon \\
Factor &::= \texttt{(}\ Expr\ \texttt{)} \\
&\mid\ \texttt{ID}
\end{aligned}
$$

$$\text{FOLLOW}(Expr) = \{\texttt{EOF}, \texttt{)}\}$$

$$\text{FOLLOW}(Expr') = \{\texttt{EOF}, \texttt{)}\}$$

$$\text{FOLLOW}(Term) = \{\texttt{+}, \texttt{-}, \texttt{EOF}, \texttt{)}\}$$

$$\text{FOLLOW}(Term') = \{\texttt{+}, \texttt{-}, \texttt{EOF}, \texttt{)}\}$$

$$\text{FOLLOW}(Factor) = \{\texttt{*}, \texttt{/}, \texttt{+}, \texttt{-}, \texttt{EOF}, \texttt{)}\}$$

无回溯的预测分析

- 若非终结符号 A 有多条产生规则 $A ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$，如何确定应该选择哪条规则进行推导？
  - 对任意不同的 i、j，有 $\mathrm{FIRST}(\beta_i) \cap \mathrm{FIRST}(\beta_j) = \varnothing$
  - 对任意不同的 i、j，如果 $\epsilon \in \mathrm{FIRST}(\beta_i)$，则 $\mathrm{FOLLOW}(A) \cap \mathrm{FIRST}(\beta_j) = \varnothing$

- **预测分析法：**
  - 如果当前 token 在集合 $\mathrm{FIRST}(\beta_i)$ 中，则使用规则 $A \to \beta_i$
  - 如果当前 token 在集合 $\mathrm{FOLLOW}(A)$ 中，且 β_i 能够推导出空串，则使用规则 $A \to \beta_i$
  - **上述要求确保了预测分析的确定性**

无回溯的预测分析

$$
\begin{aligned}
S &::= Expr\ \texttt{EOF} \\
Expr &::= Term\ Expr' \\
Expr' &::= \texttt{+}\ Term\ Expr' \\
&\mid\ \texttt{-}\ Term\ Expr' \\
&\mid\ \epsilon
\end{aligned}
$$

$$\text{FOLLOW}(Expr') = \{\texttt{EOF}, \texttt{)}\}$$

```
Expr() {
  if (Term()) {
    if (Expr'()) {
      return true;
    }
  }
  return false;
}
```

```
S() {
  if (Expr()) {
    if (token == EOF) {
      return true;
    }
  }
  return false;
}
```

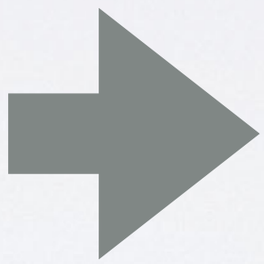
```
Expr'() {
  if (token == PLUS) {
    token = next_token();
    if (Term()) { if (Expr'()) return true; }
  } else if (token == MINUS) {
    token = next_token();
    if (Term()) { if (Expr'()) return true; }
  } else if (token == EOF || token == RPAREN) return true;
  return false;
}
```

自顶向下分析的问题：左公因子

- 若非终结符号 A 有两条产生规则 $A ::= \beta_1 \mid \beta_2$，无回溯的预测分析要求 $\mathbf{FIRST}(\beta_1) \cap \mathbf{FIRST}(\beta_2) = \varnothing$

- 若 β_1 和 β_2 有**相同的前缀**，则无法满足该要求

$$
\begin{array}{rcl}
\mathit{Factor} & ::= & (\ \mathit{Expr}\ ) \\
& \mid & \texttt{ID} \\
& \mid & \texttt{ID}[\ \mathit{Expr}\ ] \\
& \mid & \texttt{ID}(\ \mathit{Expr}\ )
\end{array}
$$



$$
\begin{array}{rcl}
\mathit{Factor} & ::= & (\ \mathit{Expr}\ ) \\
& \mid & \texttt{ID}\ \mathit{Argument} \\
\mathit{Argument} & ::= & [\ \mathit{Expr}\ ] \\
& \mid & (\ \mathit{Expr}\ ) \\
& \mid & \epsilon
\end{array}
$$

- **提取左公因子：**
  - 把 $A ::= \alpha\beta_1 \mid \alpha\beta_2$ 转换为 $\begin{array}{l} A ::= \alpha A' \\ A' ::= \beta_1 \mid \beta_2 \end{array}$

LL 文法

- 能够通过**无回溯的递归下降预测分析**进行识别的 CFG 文法
  - 建立推导关系 $w : \beta$，即是否有 $\beta \Rightarrow^* w$，其中 w 是 token 流
  - **第一个 L**：从左往右扫描 token
    - 如果 β **最左边**出现了终结符号，则把它与输入的 token 匹配
  - **第二个 L**：构造最左推导的分析树
    - 从 β **最左边**的非终结符号出发，展开产生规则

- **LL(k) 文法**：预测分析中可以访问 k 个「向前看」的 token
  - 最常见的一种是 LL(1)

LL(1) 文法

- 判定 LL(1) 文法：
  - 对任意产生规则 $A ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ 和任意不同的 i、j，有 $\mathbf{FIRST}(\beta_i) \cap \mathbf{FIRST}(\beta_j) = \varnothing$，且如果 $\epsilon \in \mathbf{FIRST}(\beta_i)$，则 $\mathbf{FOLLOW}(A) \cap \mathbf{FIRST}(\beta_j) = \varnothing$

- LL(1) 文法的性质：
  - 无二义性
  - 无左递归
  - 无左公因子

严格来说，上述判定会认为 $S ::= S\,\mathrm{a}$ 是 LL(1) 的，这个问题不大，因为 S 对应的语言是空集

小结：自顶向下语法分析

- 从语法分析的初始符号出发，展开产生规则进行**推导**
- **无回溯的递归下降预测分析**
  - 从左往右处理输入 token 流，构造最左推导
  - 需要消除左递归
  - 需要提取左公因子
  - 需要计算 FIRST 和 FOLLOW 集合
- 对应 LL(k) 文法
  - k 表示「向前看」符号数目
  - 常见的是 LL(1)

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 $\{\mathrm{a}^i\,\mathrm{b}^j \mid i \geq j\}$

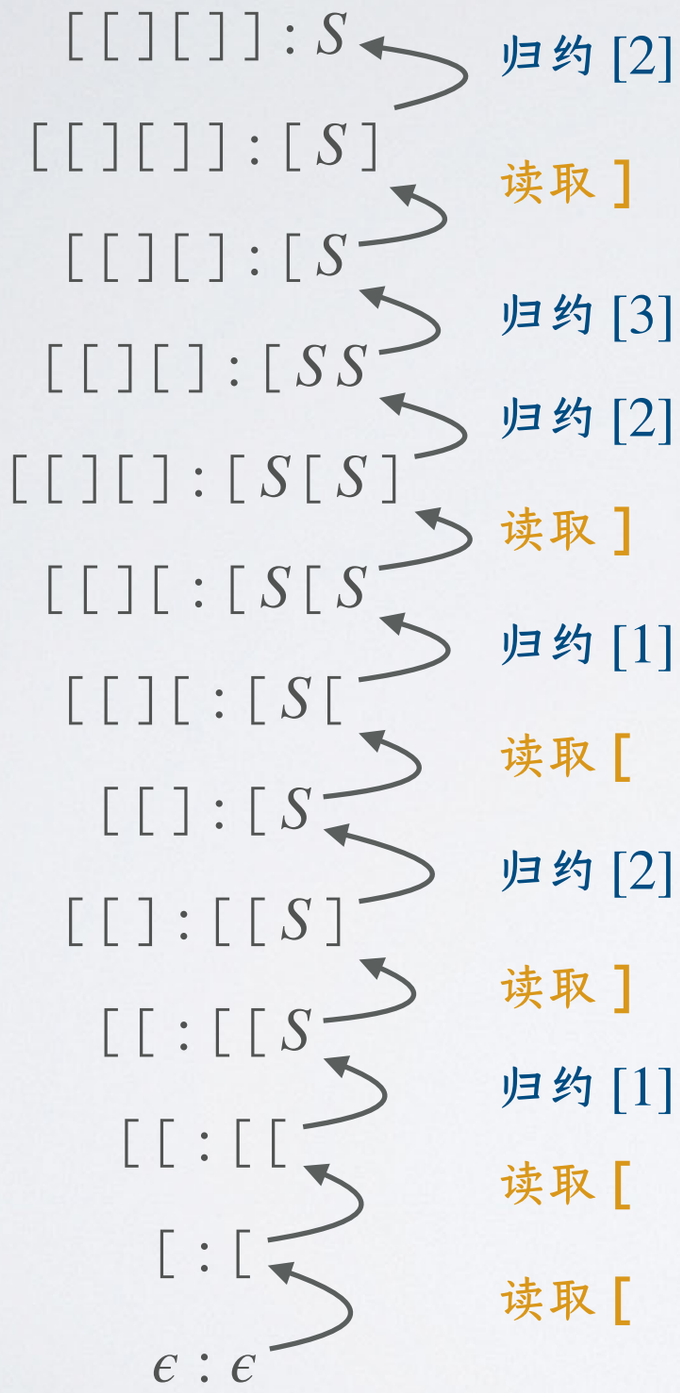
- 该文法不是 LL(1) 的
  - [1] 和 [2] 两条规则右侧的 FIRST 集合有交集

[1]	$S \to \mathrm{a}\,S$
[2]	$S \to P$
[3]	$P \to \mathrm{a}\,P\,\mathrm{b}$
[4]	$P \to \epsilon$

- 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 $\mathrm{a}^k\,\mathrm{b}^k$ 和 $\mathrm{a}^{k+1}\,\mathrm{b}^k$ 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**

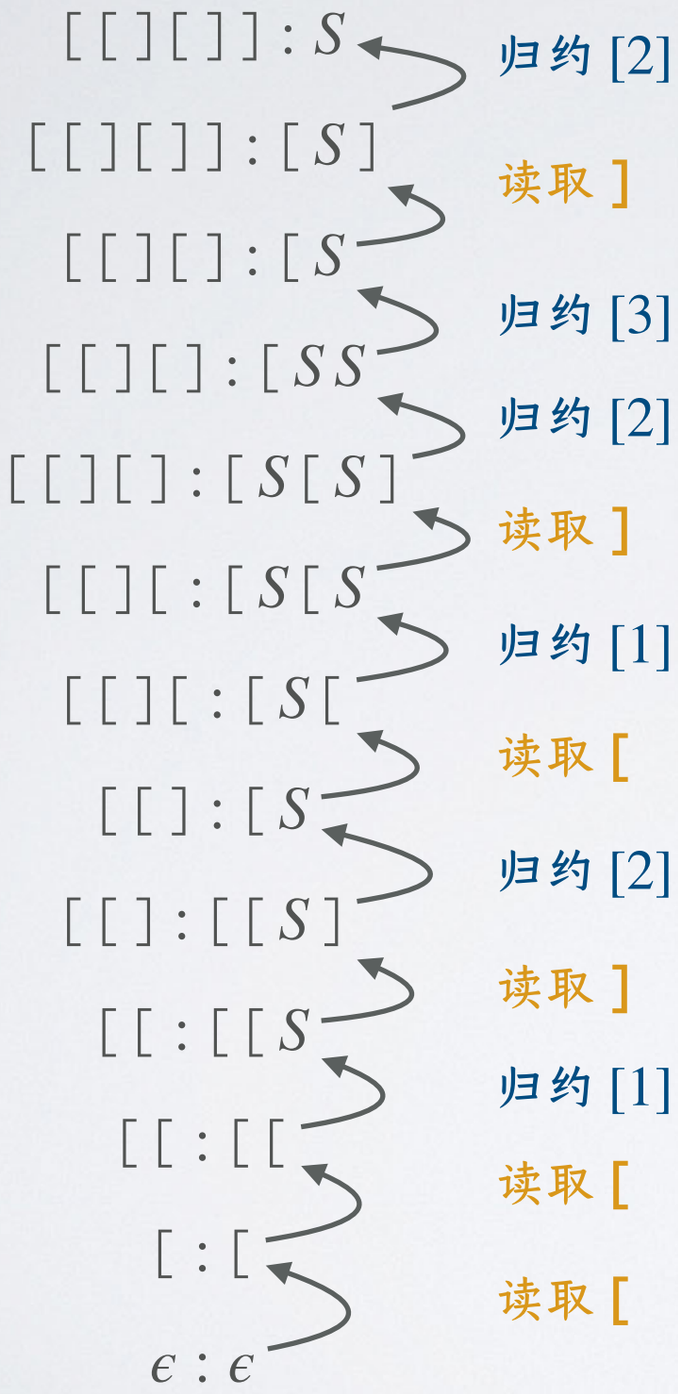
自底向上语法分析



[1]	$S \to \epsilon$
[2]	$S \to [\, S \,]$
[3]	$S \to SS$

- 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

自底向上分析构造最右推导



[1]	$S \to \epsilon$
[2]	$S \to [\, S \,]$
[3]	$S \to SS$

$$
\begin{aligned}
S &\Leftarrow [\, S \,] \\
&\Leftarrow [\, SS \,] \\
&\Leftarrow [\, S[\, S \,] \,] \\
&\Leftarrow [\, S[\,] \,] \\
&\Leftarrow [\, [\, S \,][\,] \,] \\
&\Leftarrow [\, [\,][\,] \,]
\end{aligned}
$$

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

$$
\begin{aligned}
& S' \to S\,\mathrm{EOF} \\
[1]\quad & S \to \epsilon \\
[2]\quad & S \to [\,S\,]S
\end{aligned}
$$

- **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**

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

$$\beta \quad \| \quad w$$

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

ϵ	‖	[[ ] [ ]] EOF
[	‖	[ ] [ ]] EOF
[[	‖	] [ ]] EOF
[[S	‖	] [ ]] EOF
[[S]	‖	[ ]] EOF
[[S] [	‖	]] EOF
[[S] [S	‖	]] EOF
[[S] [S]	‖	] EOF
[[S] [S] S	‖	] EOF
[[S] S	‖	] EOF
[S	‖	] EOF
[S]	‖	EOF
[S] S	‖	EOF
S	‖	EOF

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

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

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 ......
}
```

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]S$

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 ......
```

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

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 >= 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;
    }
  } else ......
```

$$
\begin{array}{ll}
 & S' \to S\,\mathrm{EOF} \\
[1] & S \to \epsilon \\
[2] & S \to [\,S\,]\,S
\end{array}
$$

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 $A \to \alpha\,\gamma$ 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

	$S' \to S$ EOF
[1]	$S \to [\,S\,]$
[2]	$S \to$ a

- **然后，确定每一种情况的动作**
  - ϵ: FIRST(S) 中的符号可以移进
  - S: EOF 符号可以接受
  - $\beta\,\alpha$ (对应规则 $A \to \alpha\,\gamma$):
    - FIRST(γ) 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
β [	移进		移进	
β [S		移进		
β [S]	归约 [1]	归约 [1]	归约 [1]	归约 [1]
β a	归约 [2]	归约 [2]	归约 [2]	归约 [2]

问题：移进-归约冲突

- $S \to \epsilon$ 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 $A \to \alpha$）时，如果向前看符号在 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

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

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

分析栈 \ 向前看	[	]	EOF
ϵ	移进/归约	归约 [1]	归约 [1]
S			接受
β [	移进	归约 [1]	归约 [1]
β [S		移进	
β [S]	移进	归约 [1]	归约 [1]
β [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

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \mathrm{a\,b\,d}$
[2]	$S \to \mathrm{a}\,A\,\mathrm{c}$
[3]	$S \to \mathrm{b}\,A\,\mathrm{d}$
[4]	$A \to \mathrm{b}$

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

分析栈 \ 向前看	a	b	c	d	EOF
ϵ	移进	移进			
S					接受
β 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 β a b 且向前看为 d 时，归约 [4]，分析栈变成 β 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 $A \to \alpha\,\gamma$ 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」：**

	$S' \to S\ \mathrm{EOF}$
[1]	$S \to [\ S\ ]$
[2]	$S \to \mathrm{a}$

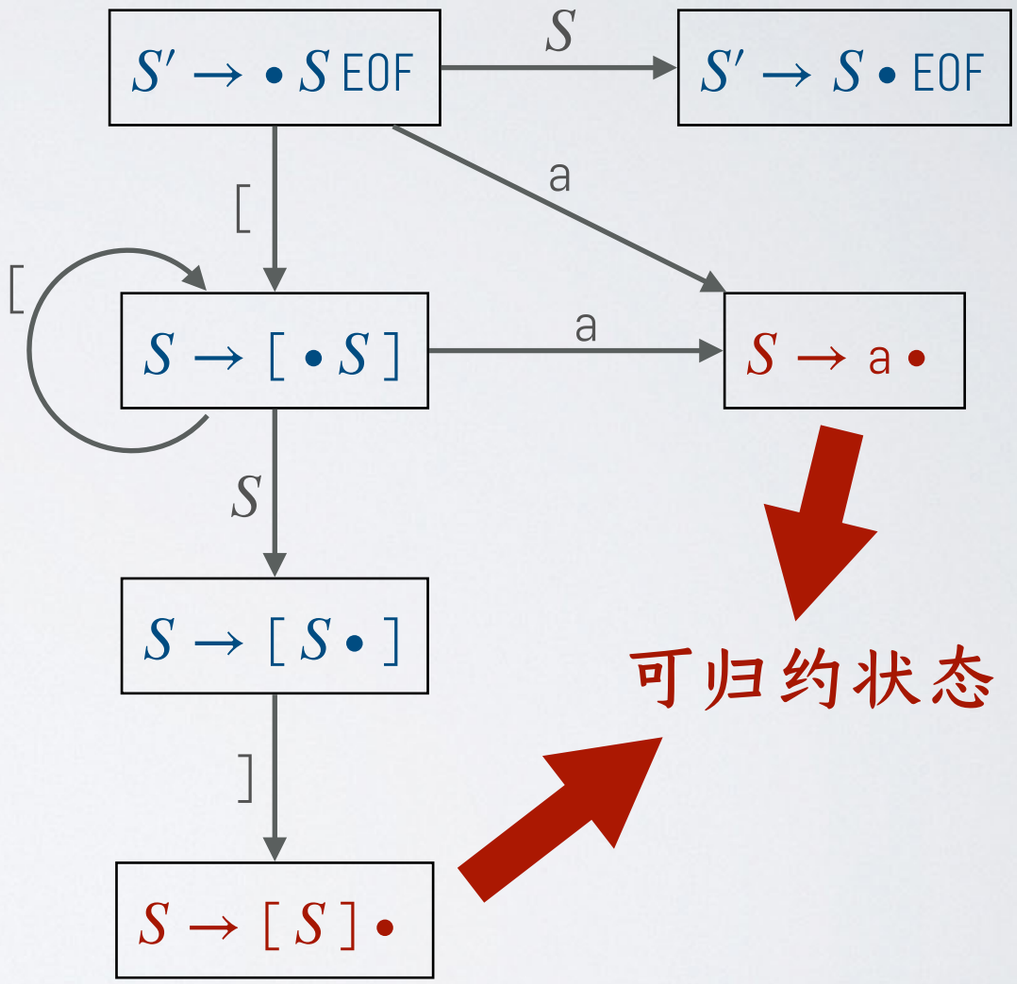
ϵ	$S' \to \bullet\, S\ \mathrm{EOF}$
S	$S' \to S \bullet \mathrm{EOF}$
$\beta\,[$	$S \to [\ \bullet\, S\ ]$
$\beta\,[\,S$	$S \to [\ S \bullet\ ]$
$\beta\,[\,S\,]$	$S \to [\ S\ ] \bullet$
$\beta\,\mathrm{a}$	$S \to \mathrm{a} \bullet$

圆点的左边表示分析栈顶内容

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

	$S' \to S\ \mathrm{EOF}$
[1]	$S \to [\ S\ ]$
[2]	$S \to \mathrm{a}$



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

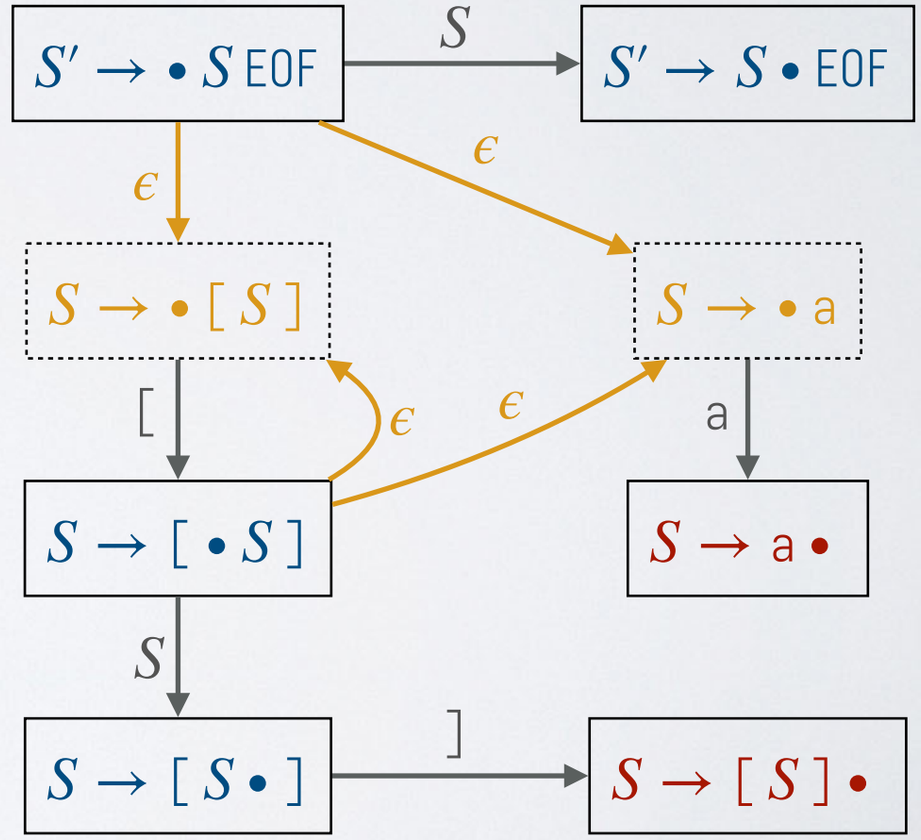
部分分析状态间的转移

- 为了方便构造转移，增加形如 $A \to \bullet\, \alpha$ 的「部分分析状态」

- 对于状态 $A \to \alpha \bullet \gamma$：
  - 如果 $\gamma = c\gamma'$，其中 c 是终结符号，则可以通过 c 转移到 $A \to \alpha c \bullet \gamma'$
  - 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 $A \to \alpha X \bullet \gamma'$
    - 表示栈 $\beta \alpha w$ **归约**成为栈 $\beta \alpha X$
  - 对任意规则 $X \to \delta$，可以 ϵ **转移到** $X \to \bullet\, \delta$

非确定性有限自动机(NFA)

	$S' \to S\ \mathrm{EOF}$
[1]	$S \to [\ S\ ]$
[2]	$S \to \mathrm{a}$



部分分析状态的自动机

$S' \to S\ \mathrm{EOF}$
[1] $S \to [\ S\ ]$
[2] $S \to \mathrm{a}$

非确定性有限自动机(NFA)

项

$S' \to \bullet S\ \mathrm{EOF}$ —S→ $S' \to S \bullet \mathrm{EOF}$

$S' \to \bullet S\ \mathrm{EOF}$ —ϵ→ $S \to \bullet [\ S\ ]$

$S' \to \bullet S\ \mathrm{EOF}$ —ϵ→ $S \to \bullet \mathrm{a}$

$S \to \bullet [\ S\ ]$ —$[$→ $S \to [\bullet S\ ]$

$S \to [\bullet S\ ]$ —ϵ→ $S \to \bullet [\ S\ ]$

$S \to \bullet \mathrm{a}$ —a→ $S \to \mathrm{a} \bullet$

$S \to [\bullet S\ ]$ —S→ $S \to [S \bullet]$

$S \to [S \bullet]$ —$]$→ $S \to [S] \bullet$

确定性有限自动机(DFA)

项集

$S' \to \bullet S\ \mathrm{EOF}$
$S \to \bullet [\ S\ ]$
$S \to \bullet \mathrm{a}$

S → $S' \to S \bullet \mathrm{EOF}$

$[$ → $S \to [\bullet S\ ]$
$S \to \bullet [\ S\ ]$
$S \to \bullet \mathrm{a}$

a → $S \to \mathrm{a} \bullet$

$[$ → $S \to [\bullet S\ ]$ (自环)

S → $S \to [S \bullet]$

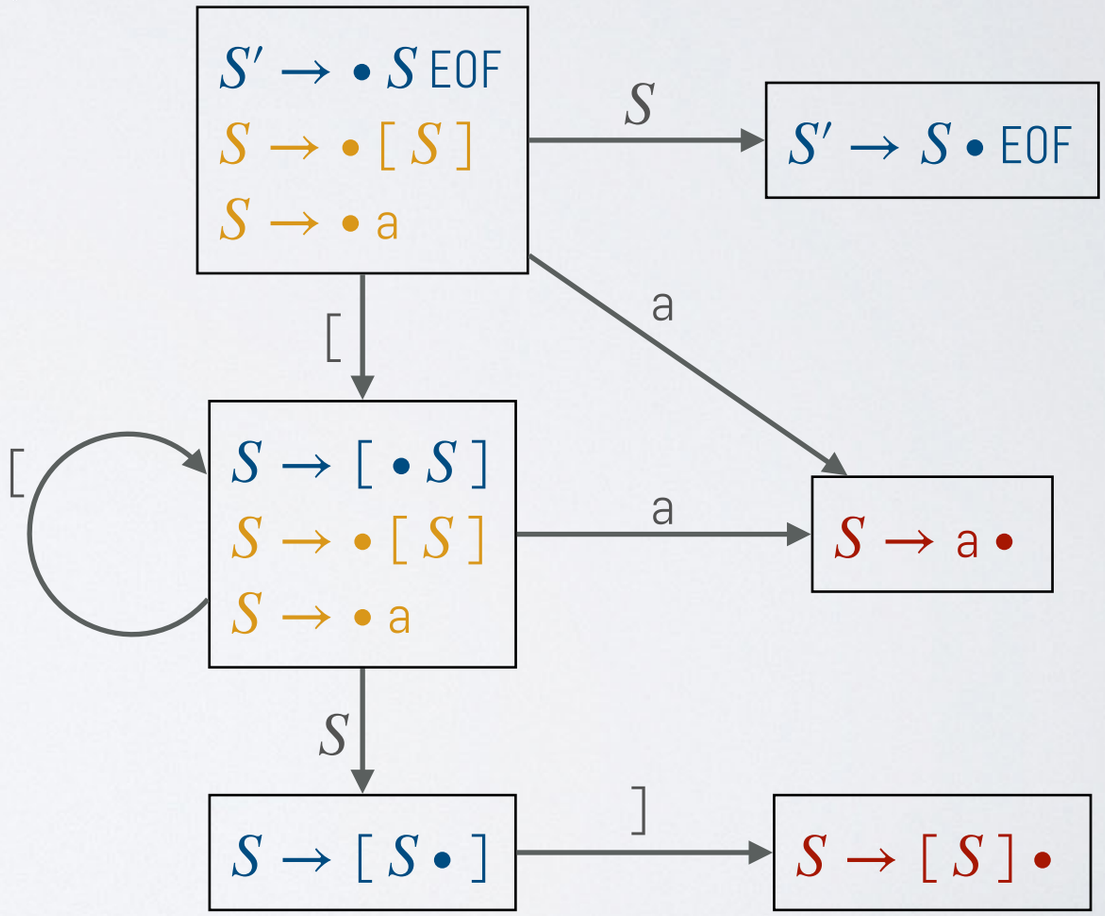
$]$ → $S \to [S] \bullet$

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 $A \to \alpha \bullet$)，则可以进行**归约**

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

	$S' \to S$ EOF
[1]	$S \to [\ S\ ]$
[2]	$S \to$ a



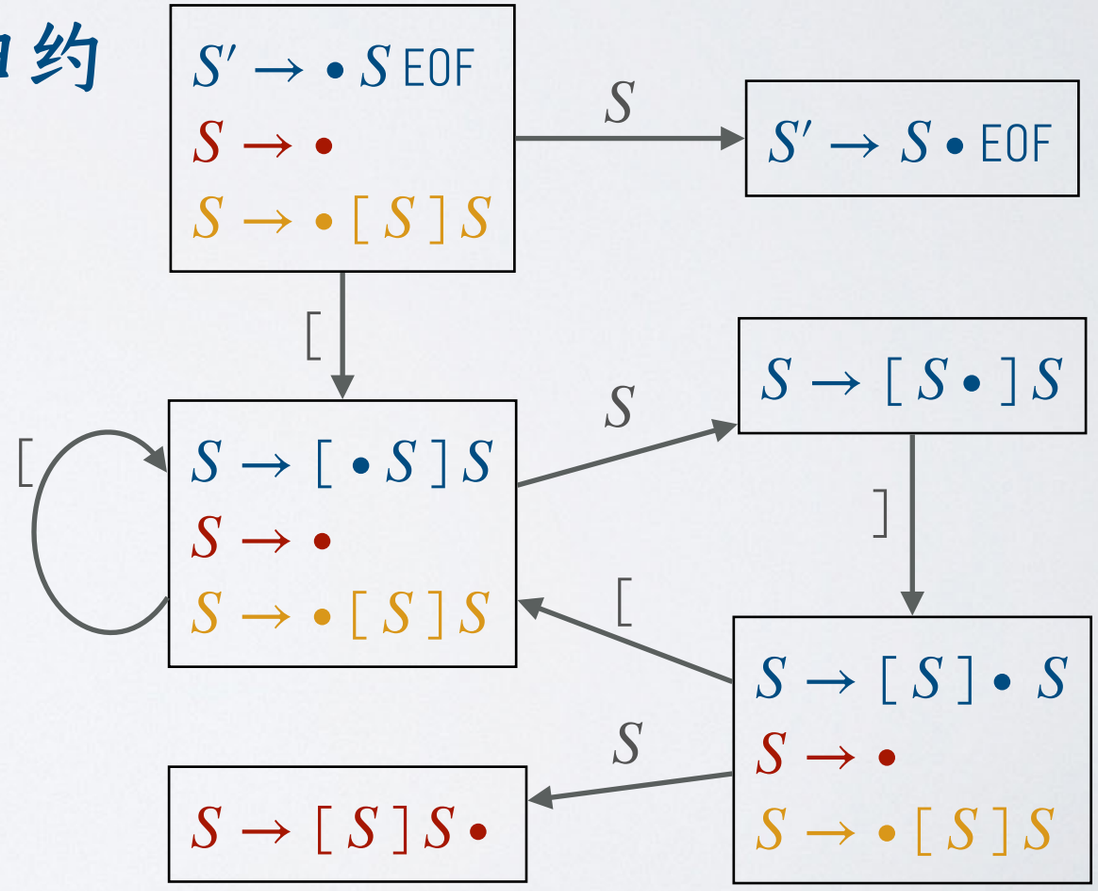
根据自动机构造移进-归约分析表

- 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 $A \to \alpha \bullet$)，且向前看符号在 $\mathrm{FOLLOW}(A)$ 中，则可以进行**归约**

	$S' \to S\,\mathtt{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

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

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]

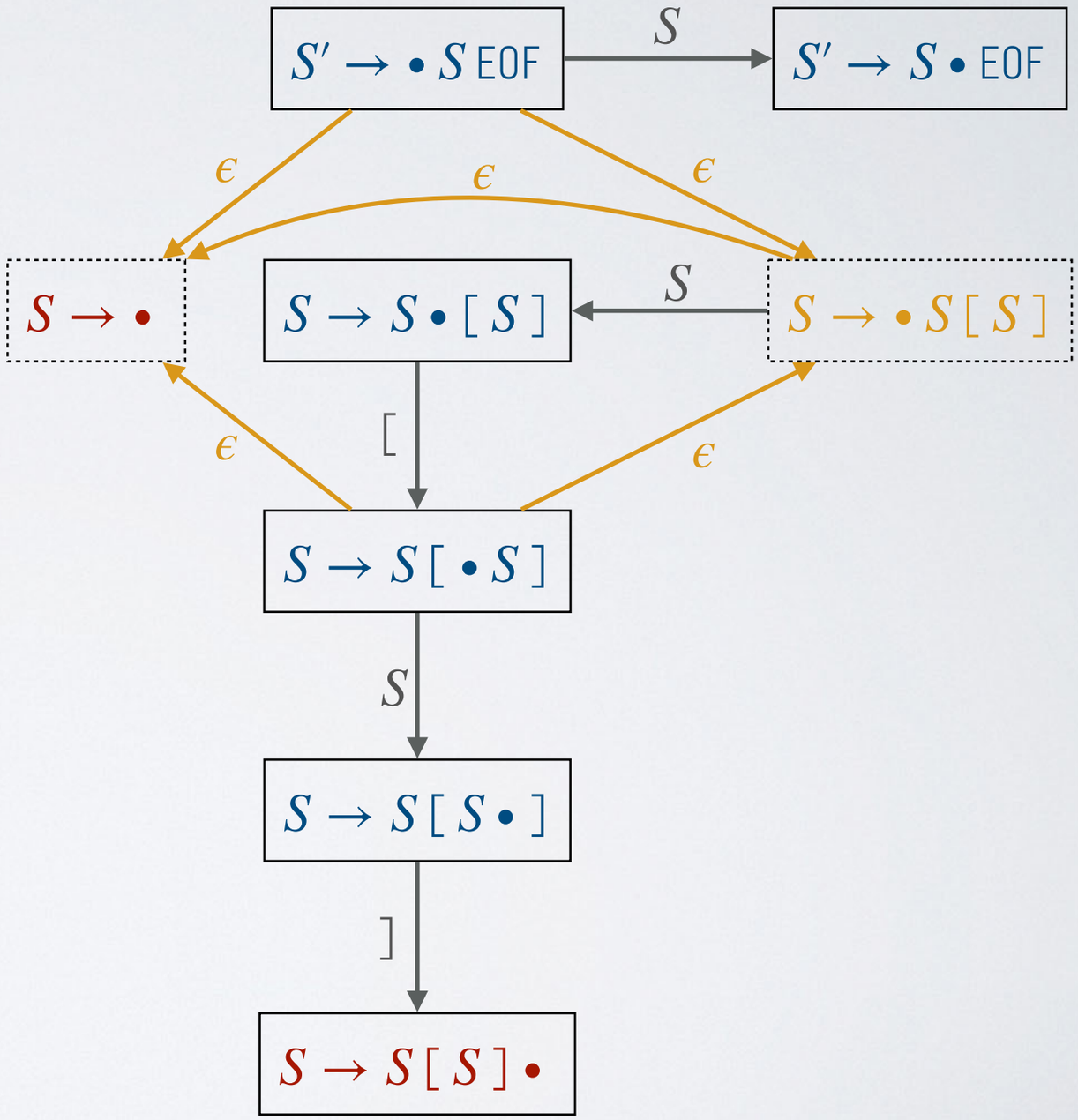


构造分析表示例

	$S' \to S\,\texttt{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to S\,[\,S\,]$

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

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

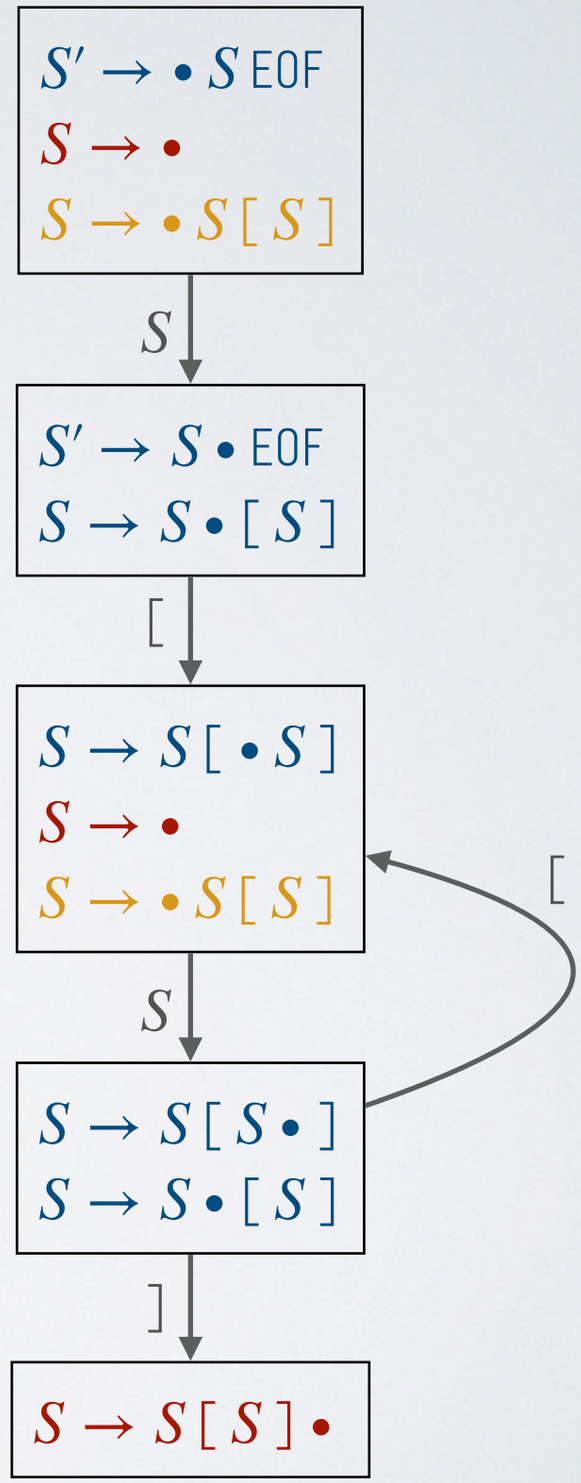


构造分析表示例

	$S' \to S\,\texttt{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to S\,[\,S\,]$

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

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

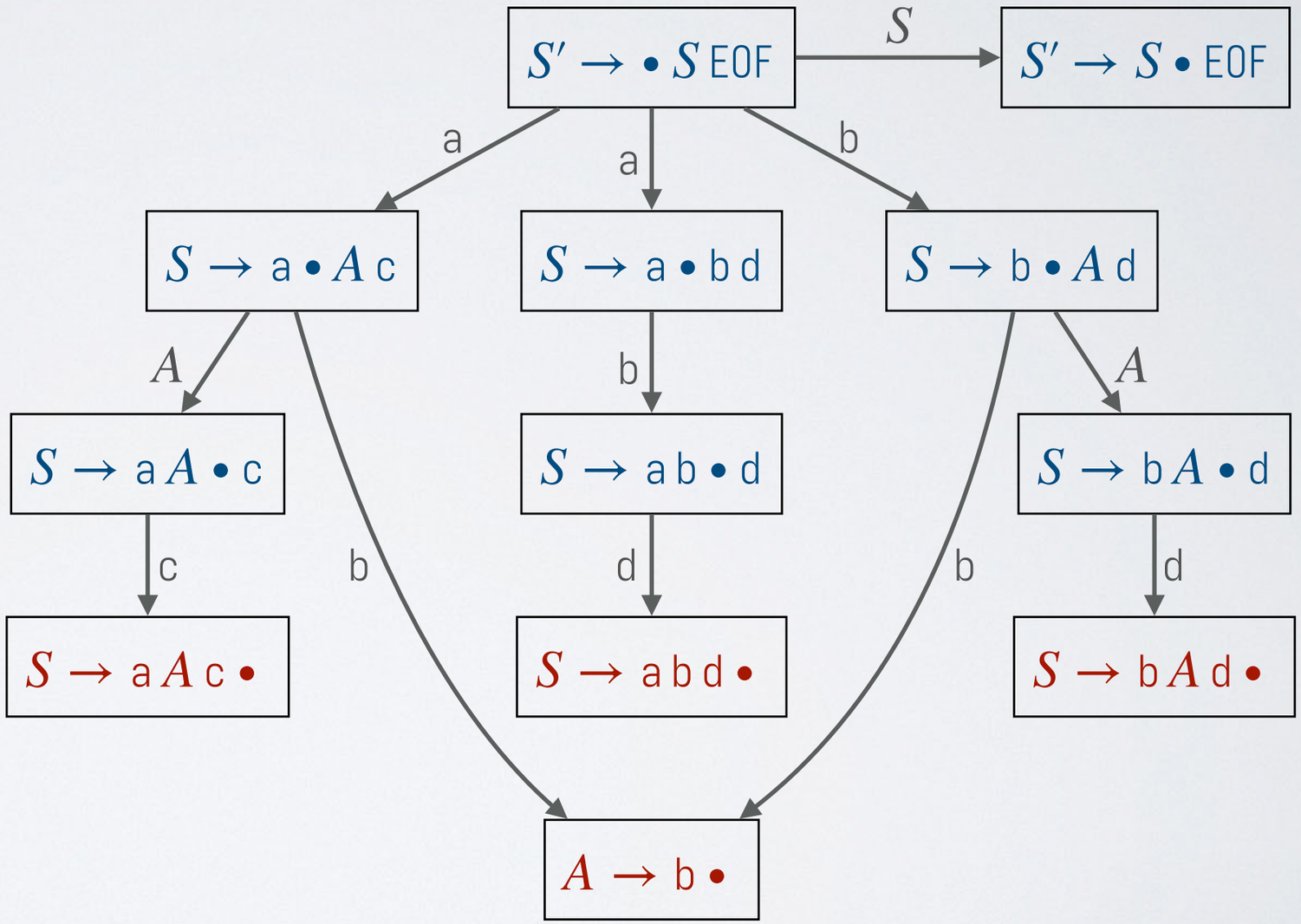
$S' \to S\,\mathrm{EOF}$

[1] $S \to \mathrm{a\,b\,d}$

[2] $S \to \mathrm{a}\,A\,\mathrm{c}$

[3] $S \to \mathrm{b}\,A\,\mathrm{d}$

[4] $A \to \mathrm{b}$

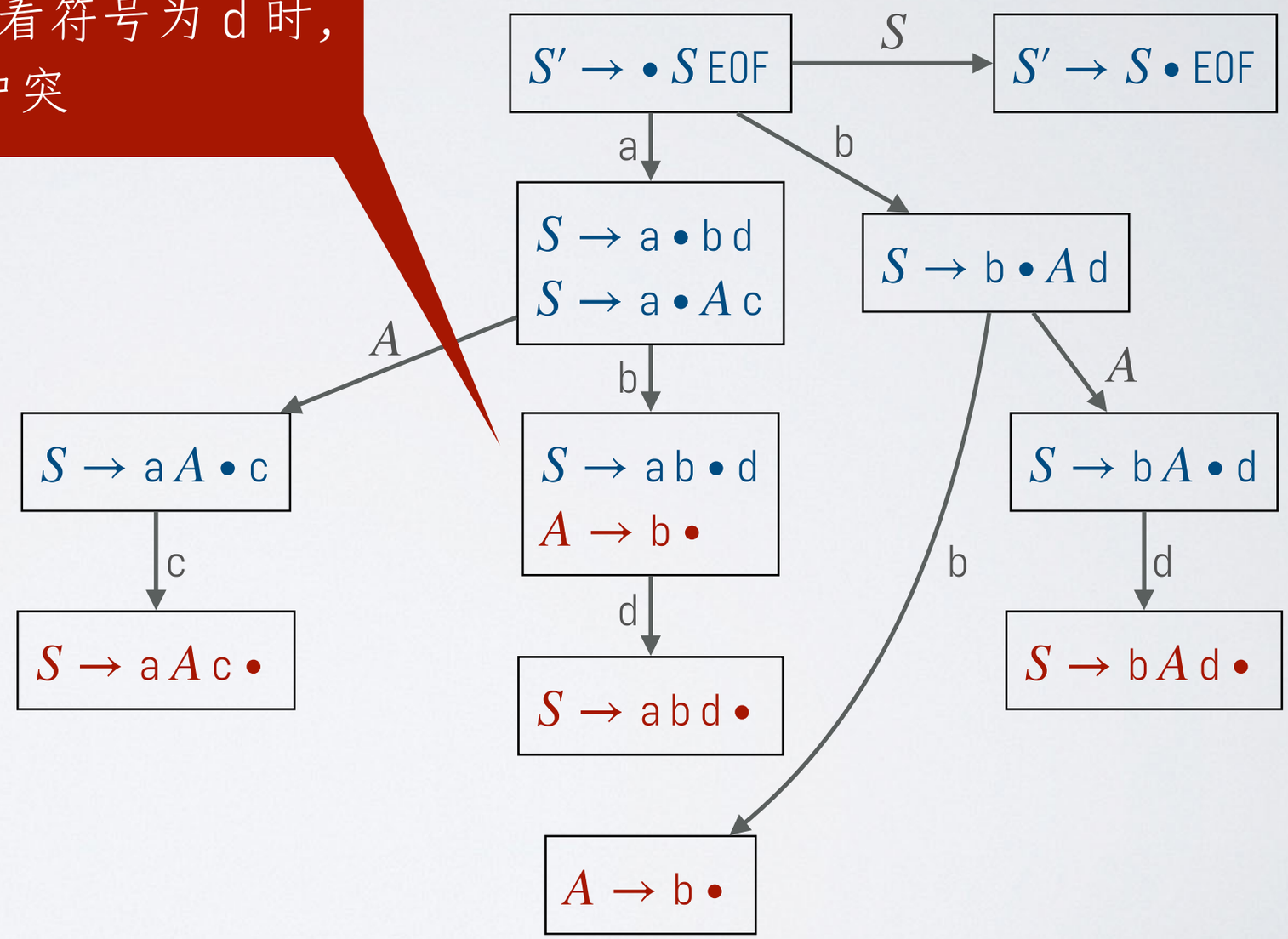


使用 FOLLOW 集合的局限性

当分析栈「模式」为 a b、向前看符号为 d 时，
存在移进-归约冲突

$S' \to S\,\mathrm{EOF}$
[1] $S \to \mathrm{a\,b\,d}$
[2] $S \to \mathrm{a}\,A\,\mathrm{c}$
[3] $S \to \mathrm{b}\,A\,\mathrm{d}$
[4] $A \to \mathrm{b}$

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

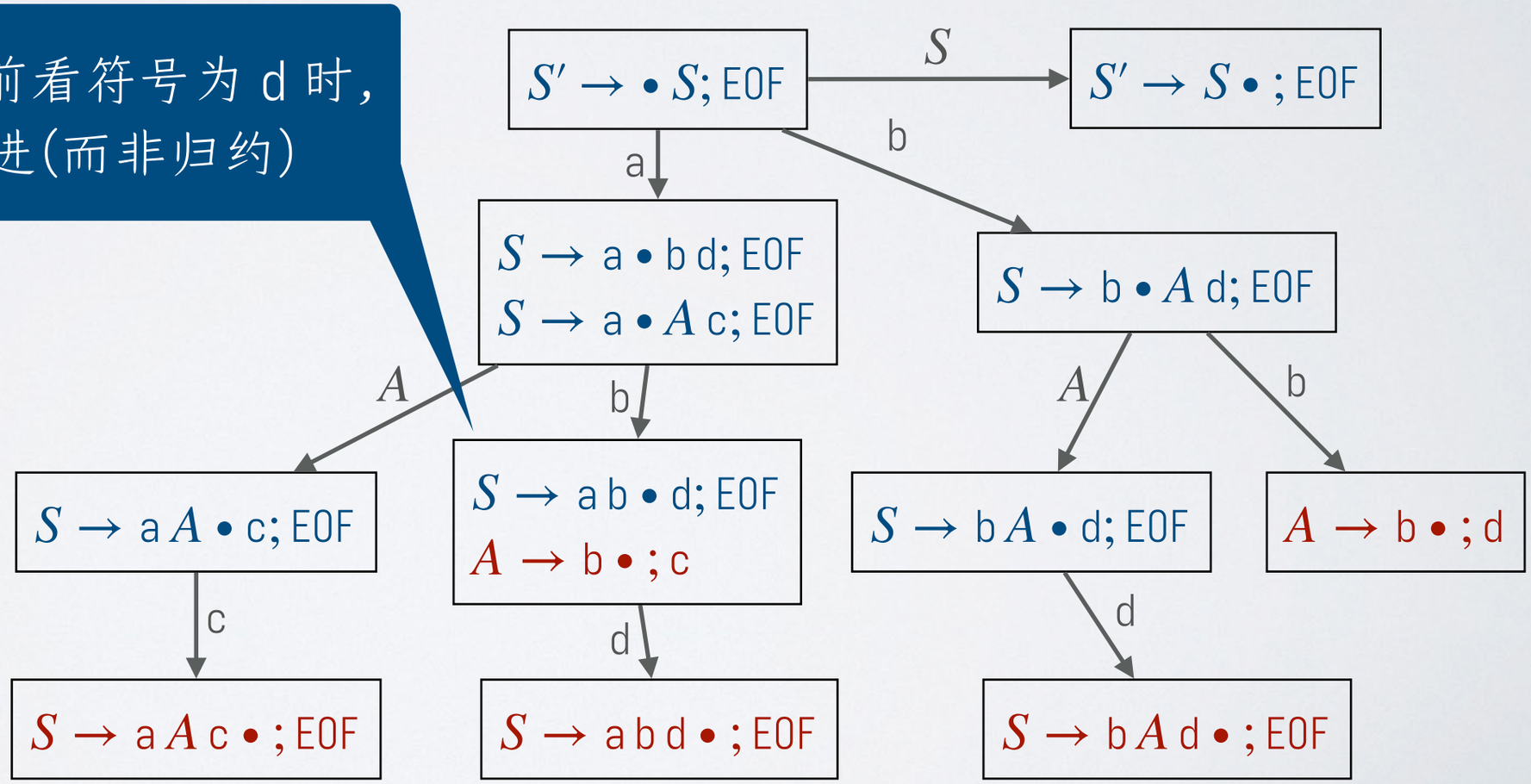


部分分析状态 +「向前看」

- 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 $A \to \alpha \bullet \gamma$ 为 $\langle A \to \alpha \bullet \gamma; c \rangle$
  - 从该部分分析状态完成 γ 的分析后，向前看为 c 时可以进行归约

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

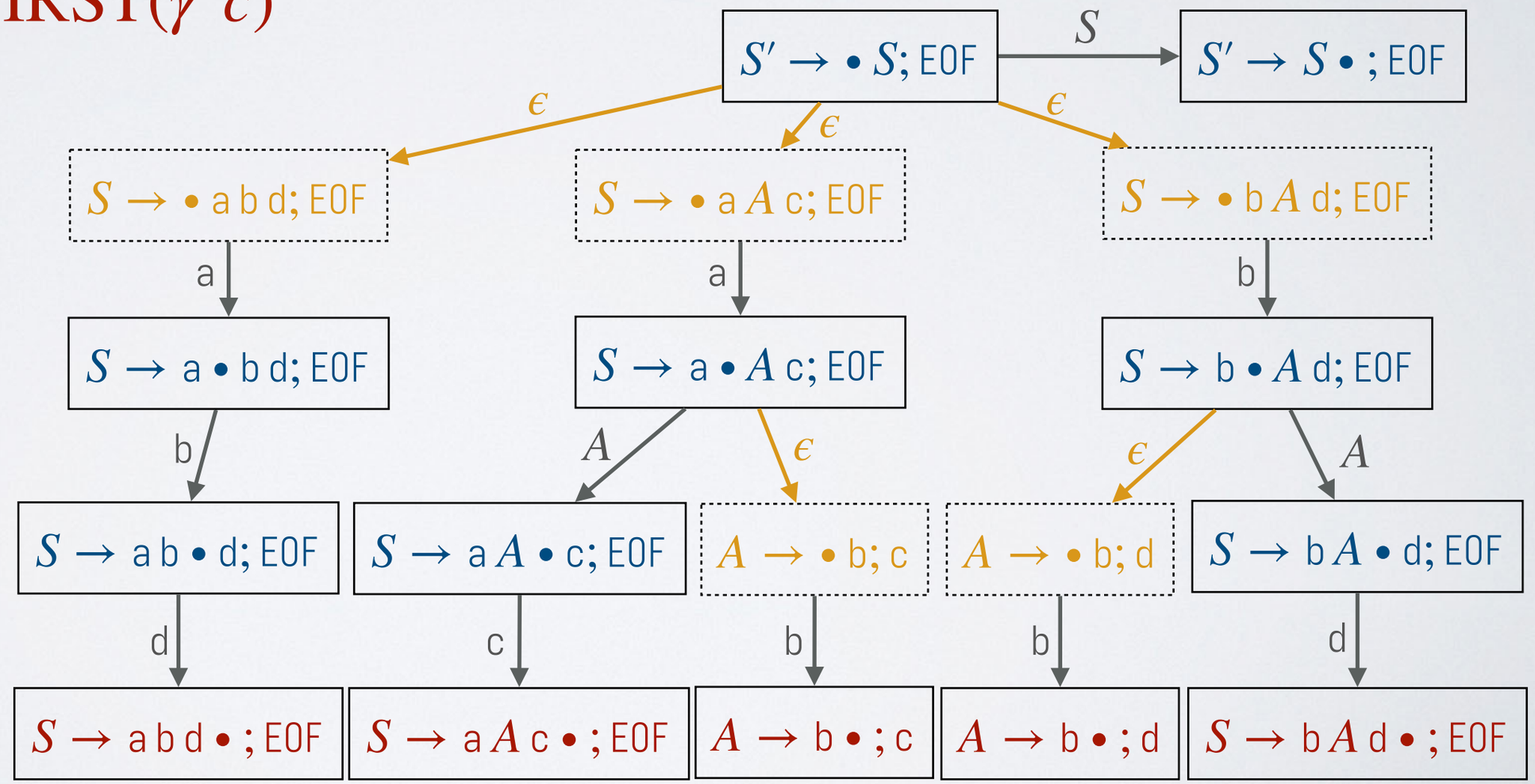
	$S' \to S$ EOF
[1]	$S \to$ a b d
[2]	$S \to$ a A c
[3]	$S \to$ b A d
[4]	$A \to$ b



部分分析状态 + 向前看

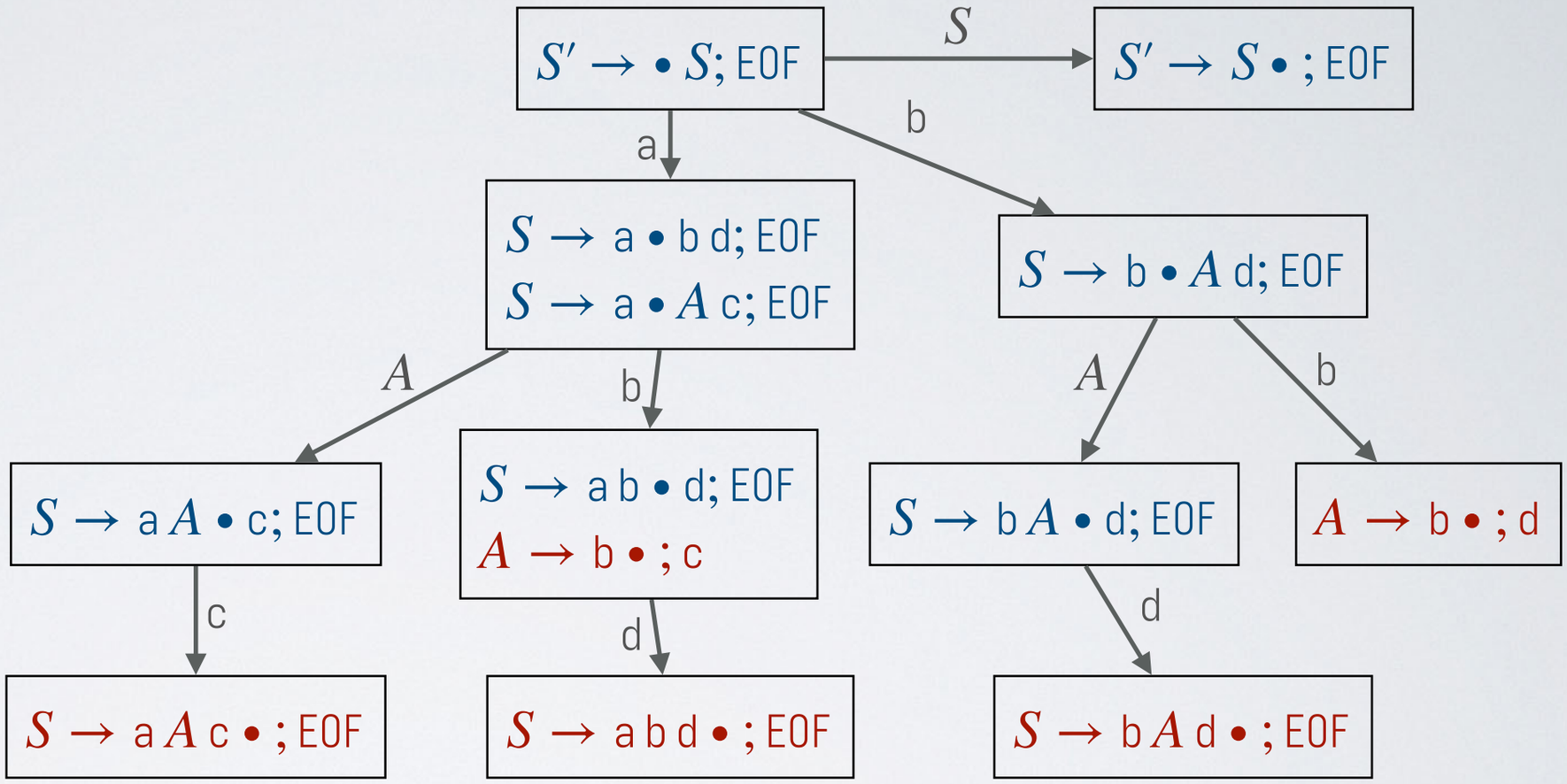
- 构造时，主要考虑 ϵ 转移对向前看的影响
  - 回顾：对于状态 $A \to \alpha \bullet X\gamma'$ 和规则 $X \to \delta$，ϵ 转移到 $X \to \bullet\, \delta$
  - 对于状态 $\langle A \to \alpha \bullet X\gamma'; c\rangle$ 和规则 $X \to \delta$，ϵ 转移到 $\langle X \to \bullet\, \delta; d\rangle$，其中 $d \in \mathrm{FIRST}(\gamma' c)$

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



部分分析状态 + 向前看

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



向前看 \ 分析栈	ϵ	S	a	a A	a A c	a b	a b d	b	b A	b A d	b b
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

$S' \to S$ EOF

[1] $S \to$ a A d

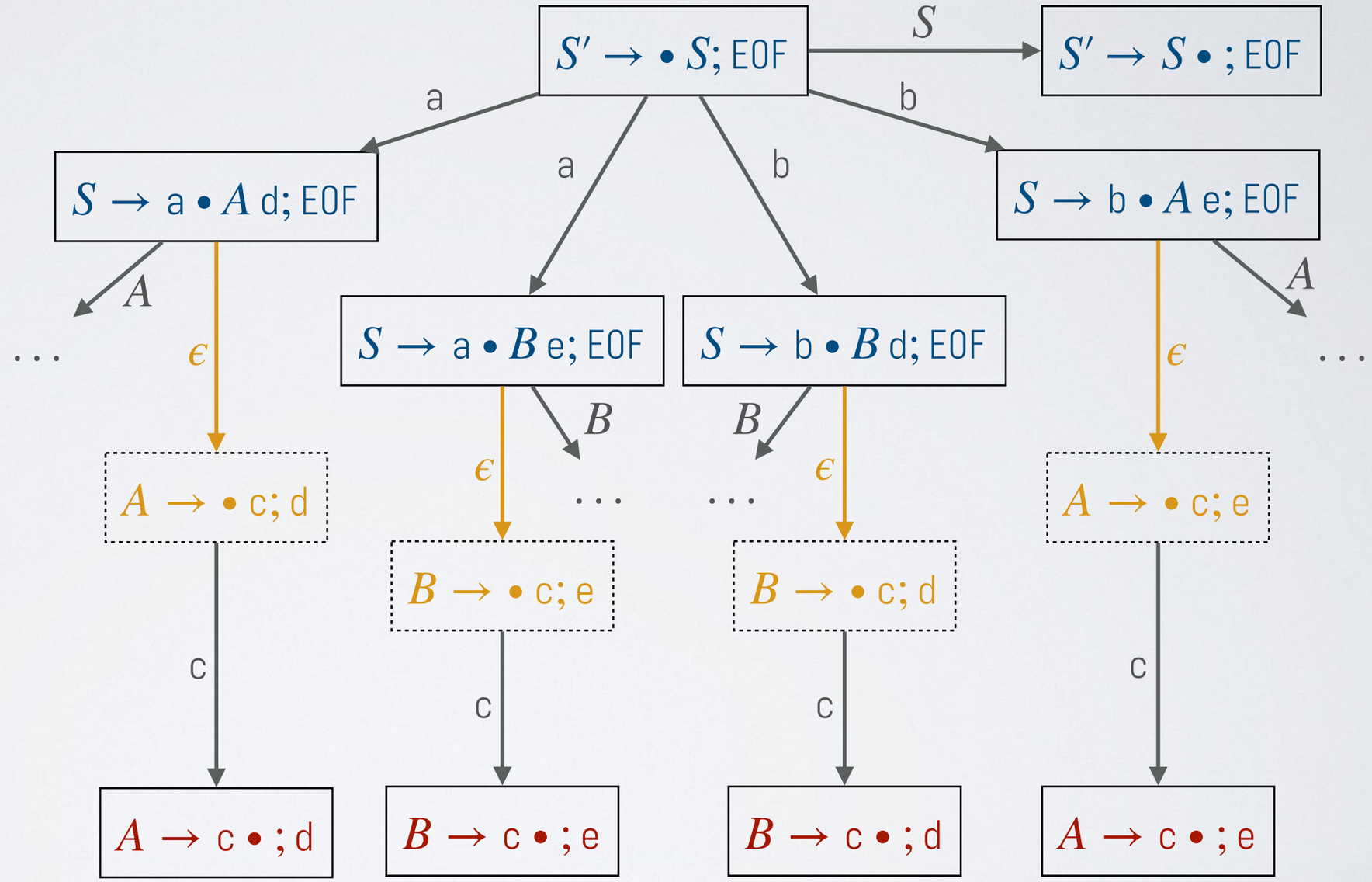
[2] $S \to$ b B d

[3] $S \to$ a B e

[4] $S \to$ b A e

[5] $A \to$ c

[6] $B \to$ c



构造分析表示例

$S' \to S$ EOF

[1] $S \to$ a A d

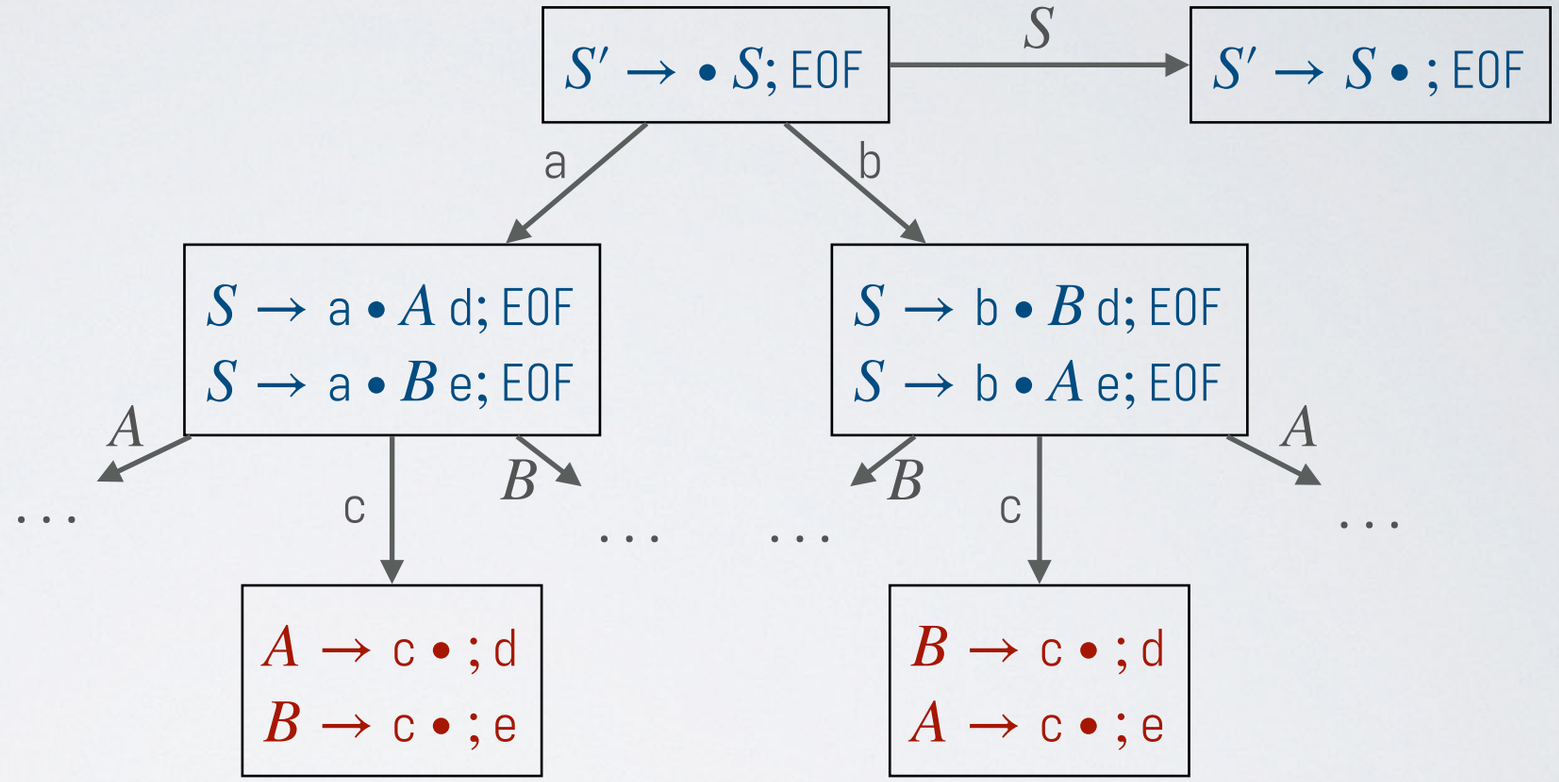
[2] $S \to$ b B d

[3] $S \to$ a B e

[4] $S \to$ b A e

[5] $A \to$ c

[6] $B \to$ 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**: 预测分析中可以访问 k 个向前看的 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

		$S' \to S\,\mathrm{EOF}$
[1]		$S \to \epsilon$
[2]		$S \to S\,[\,S\,]$

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]

ϵ	‖	[[ ] [ ]] EOF
S	‖	[[ ] [ ]] EOF
$S[$	‖	[ ] [ ]] EOF
$S[S$	‖	[ ] [ ] EOF
$S[S[$	‖	] [ ] EOF
$S[S[S$	‖	] [ ] EOF
$S[S[S]$	‖	[ ] EOF
$S[S$	‖	[ ] EOF
$S[S[$	‖	]] EOF
$S[S[S$	‖	]] EOF
$S[S[S]$	‖	] EOF
$S[S$	‖	] EOF
$S[S]$	‖	EOF
S	‖	EOF

LR 文法允许左公因子

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to [\,S\,]$
[2]	$S \to [\,]$

ϵ	‖	[[[]]]EOF
[	‖	[[]]]EOF
[[	‖	[]]]EOF
[[[	‖	]]]EOF
[[[]	‖	]]EOF
[[S	‖	]]EOF
[[S]	‖	]EOF
[S	‖	]EOF
[S]	‖	EOF
S	‖	EOF

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

LR(1) 文法

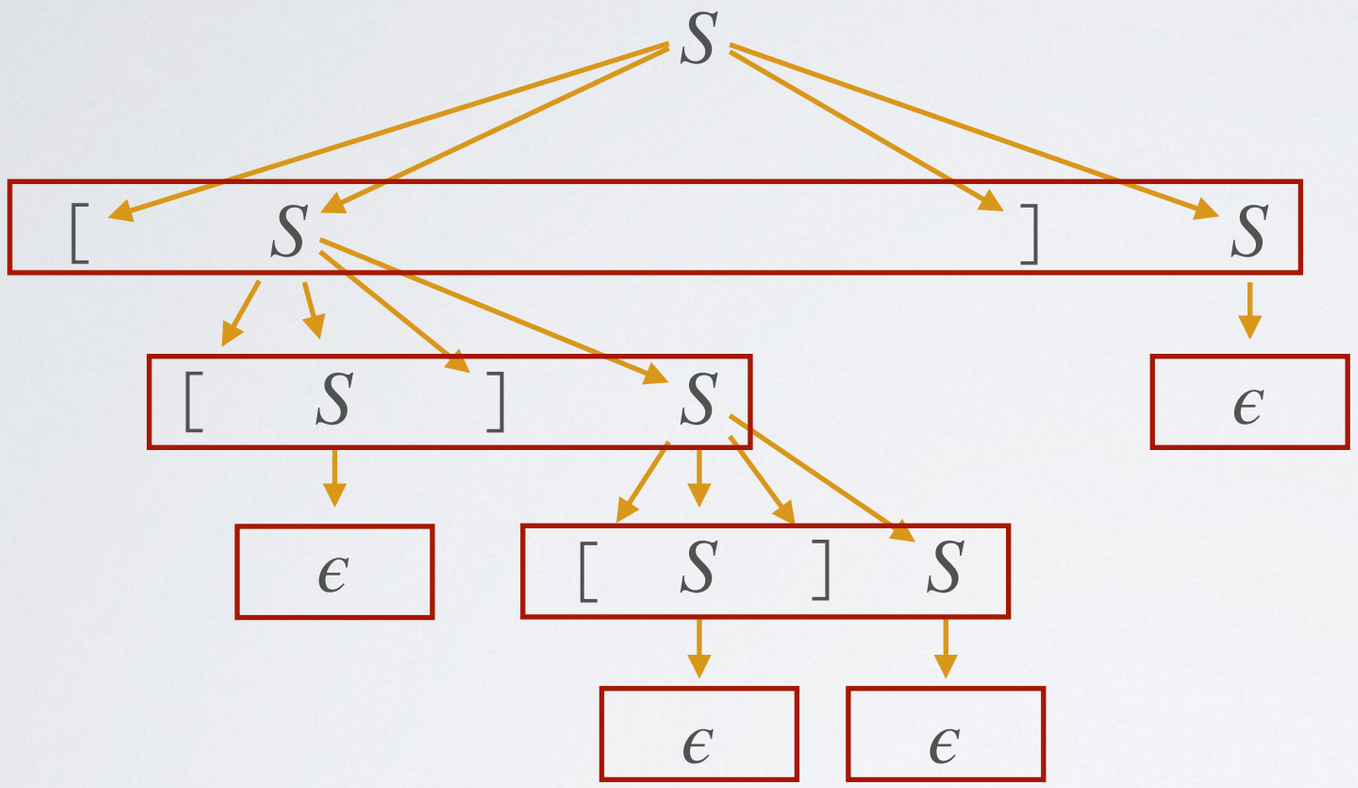
- 允许一个向前看符号
- 用于识别栈模式的自动机中，状态形式为 $\langle A \to \alpha \bullet \gamma; c \rangle$
  - $A \to \alpha \bullet \gamma$ 表示「部分分析状态」
  - c 表示向前看符号

- **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？

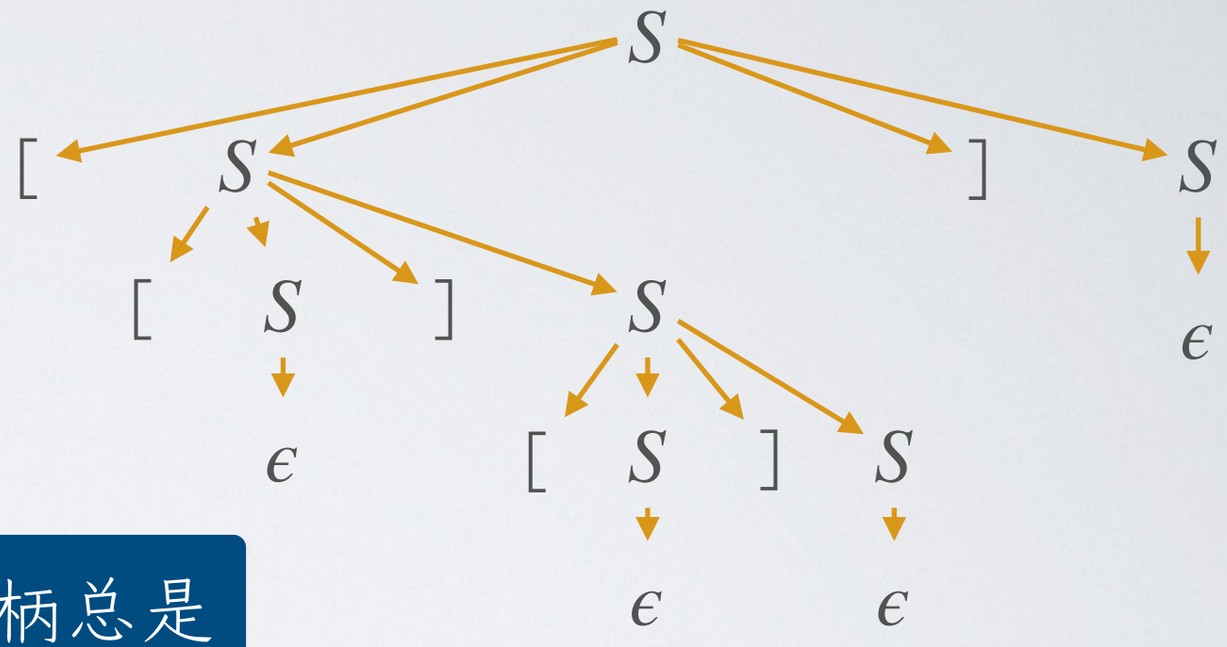
- 自底向上语法分析通过归约得出最右推导



$$
\begin{aligned}
S &\Leftarrow [S]S \\
&\Leftarrow [S] \\
&\Leftarrow [[S]S] \\
&\Leftarrow [[S][S]S] \\
&\Leftarrow [[S][S]] \\
&\Leftarrow [[S][]] \\
&\Leftarrow [[][]]
\end{aligned}
$$

- 分析过程是不断找出**句柄**并归约
  - **句柄(handle)**：分析树上最左侧的、构成直接推导的连续叶子结点

LR(1) 的工作原理是什么？



$$
\begin{aligned}
S &\Leftarrow [S]S \\
&\Leftarrow [S] \\
&\Leftarrow [[S]S] \\
&\Leftarrow [[S][S]S] \\
&\Leftarrow [[S][S]] \\
&\Leftarrow [[S][]] \\
&\Leftarrow [[][]]
\end{aligned}
$$

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「向前看」**
- 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 为向前看符号
  - **归约状态**：$\langle [p] \rangle$，$[p]$ 为规则编号
  - $\langle S; \mathrm{EOF} \rangle \Rightarrow^* \alpha c \langle [p] \rangle$ 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]	$S \to \epsilon$
[2]	$S \to [\ S\ ]\ S$

$\langle S; \mathrm{EOF} \rangle \to \mathrm{EOF}$	$\langle [1] \rangle$
$\langle S; \mathrm{EOF} \rangle \to [$	$\langle S;] \rangle$
$\langle S; \mathrm{EOF} \rangle \to [\ S\ ]$	$\langle S; \mathrm{EOF} \rangle$
$\langle S; \mathrm{EOF} \rangle \to [\ S\ ]\ S\ \mathrm{EOF}$	$\langle [2] \rangle$
$\langle S;] \rangle \to]$	$\langle [1] \rangle$
$\langle S;] \rangle \to [$	$\langle S;] \rangle$
$\langle S;] \rangle \to [\ S\ ]$	$\langle S,] \rangle$
$\langle S;] \rangle \to [\ S\ ]\ S\ ]$	$\langle [2] \rangle$

LR(1) 的工作原理是什么？

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

[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

- 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 $p = q$

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

LR(1) 文法的变种

- LR(0)
  - 不使用向前看符号
  - 识别栈模式的自动机中状态形式为 $\langle A \to \alpha \bullet \gamma \rangle$
  - 前面已介绍过

- SLR(1), Simple LR
  - 自动机与 LR(0) 相同
  - 使用一个向前看符号，通过 FOLLOW 集合判断是否归约
  - 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 $\langle A \to \alpha \bullet \gamma; c \rangle$
  - 可以认为是 LR(1) 的自动机把只有向前看符号不同的状态进行合并

LALR(1) 示例

$S' \to S\ \mathrm{EOF}$

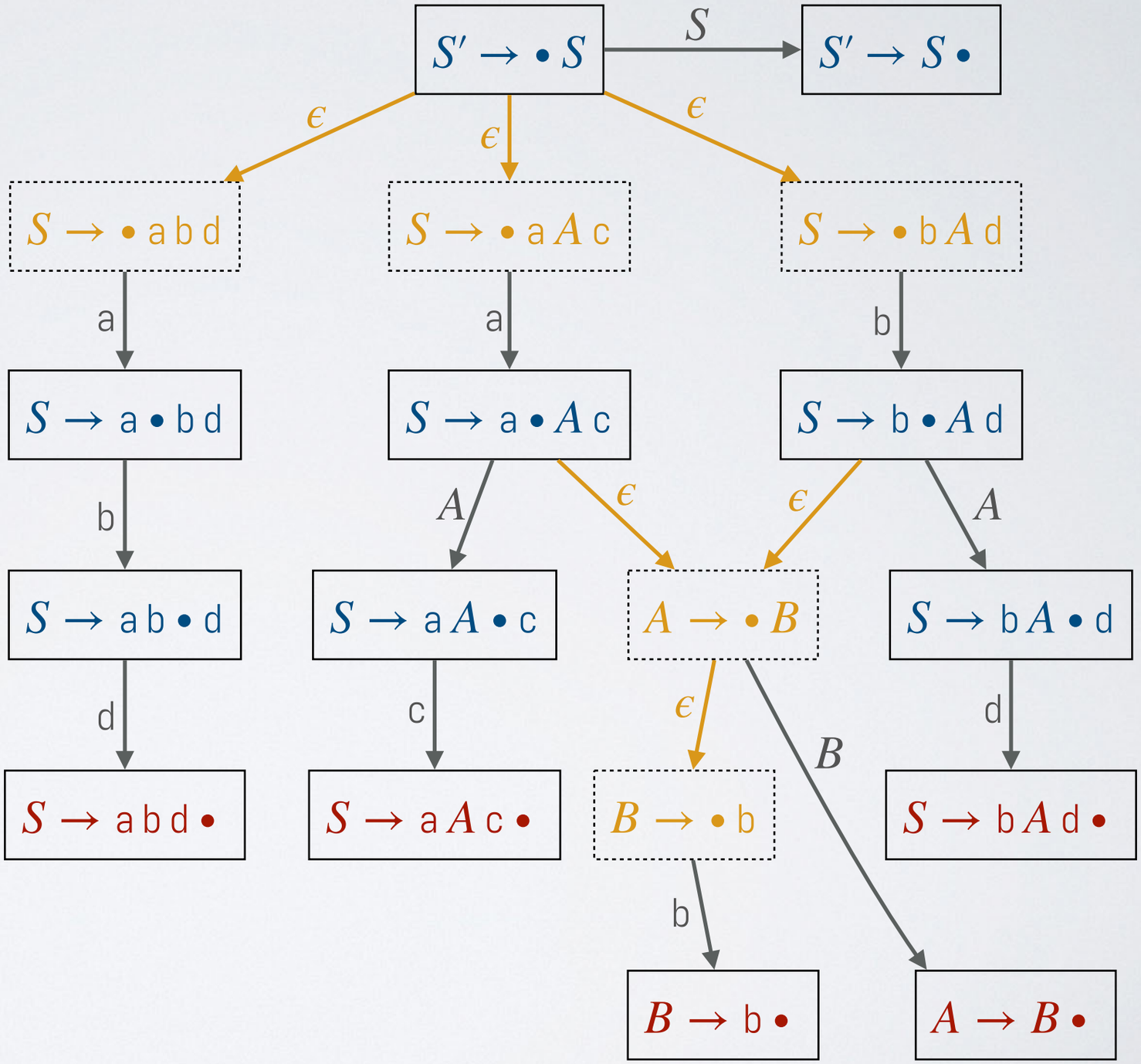
[1] $S \to \mathrm{a\,b\,d}$

[2] $S \to \mathrm{a}\,A\,\mathrm{c}$

[3] $S \to \mathrm{b}\,A\,\mathrm{d}$

[4] $A \to B$

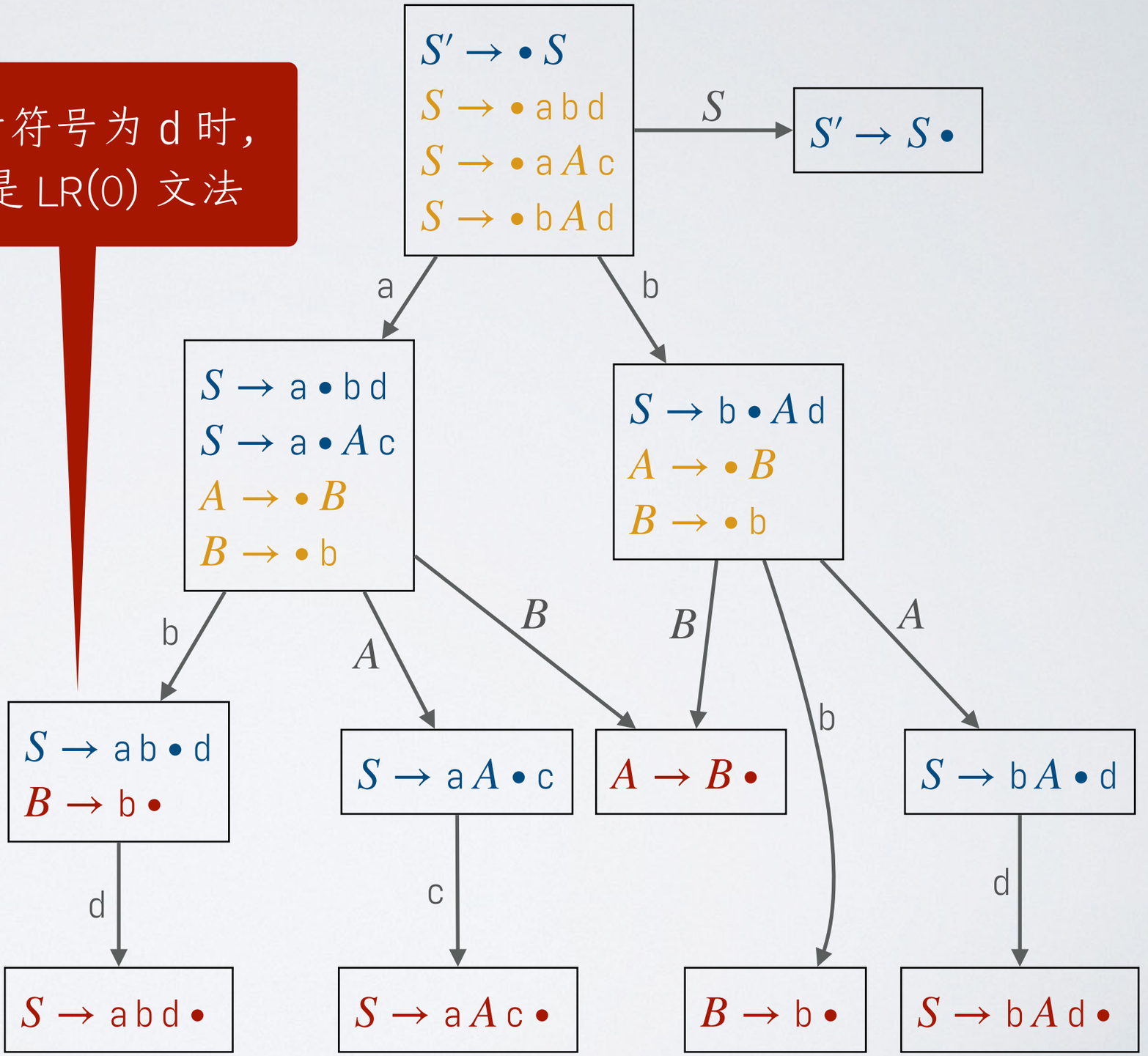
[5] $B \to \mathrm{b}$



LALR(1) 示例

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

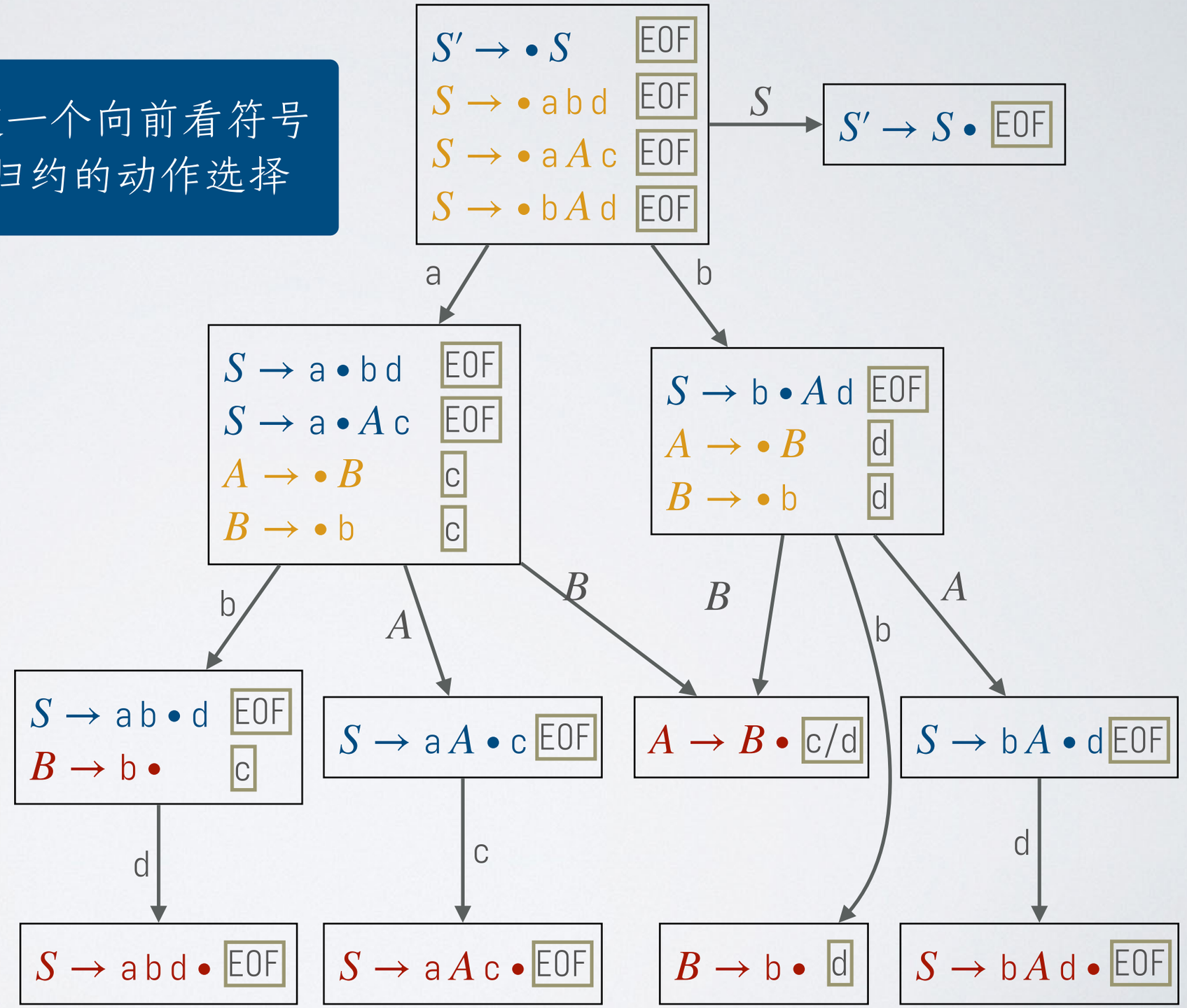
	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \mathrm{a}\,\mathrm{b}\,\mathrm{d}$
[2]	$S \to \mathrm{a}\,A\,\mathrm{c}$
[3]	$S \to \mathrm{b}\,A\,\mathrm{d}$
[4]	$A \to B$
[5]	$B \to \mathrm{b}$



LALR(1) 示例

LALR(1) 通过一个向前看符号
决定移进/归约的动作选择

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



LALR 可能引入归约-归约冲突

$S' \to S$ EOF

[1] $S \to$ a A d

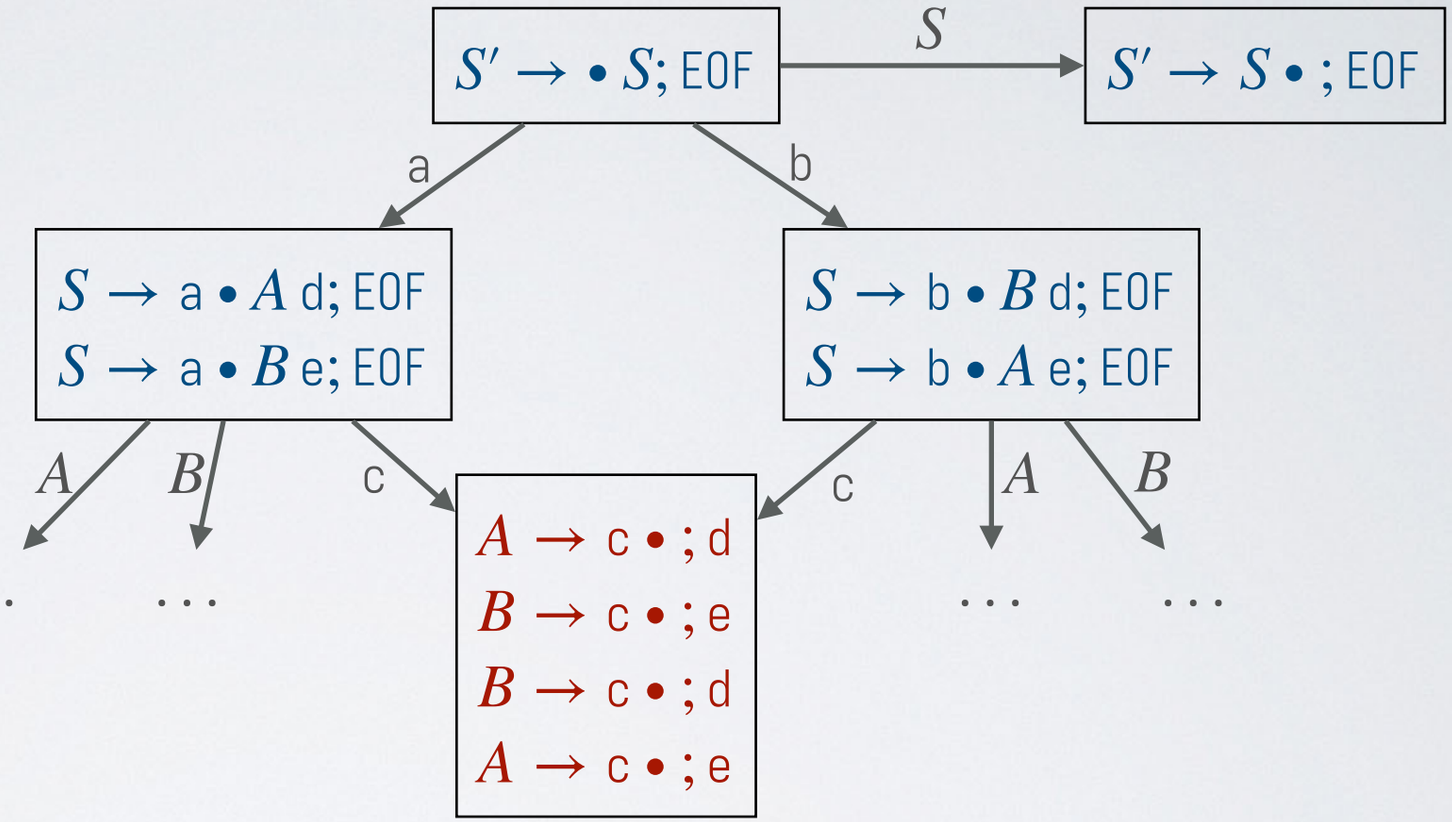
[2] $S \to$ b B d

[3] $S \to$ a B e

[4] $S \to$ b A e

[5] $A \to$ c

[6] $B \to$ 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]???		

小结：自底向上语法分析

- 从语法分析的终结符号序列出发，基于产生规则进行**归约**
- **无冲突的移进-归约预测分析**
  - 从左往右处理输入 token 流，构造最右推导
  - 构造识别分析栈模式(即识别句柄)的自动机
  - 根据自动机构造移进-归约分析表
- 对应 **LR(k)** 文法
  - k 表示向前看符号的数目
  - 常见的是 LR(1) 和 LR(0)、SLR(1)、LALR(1) 等变种

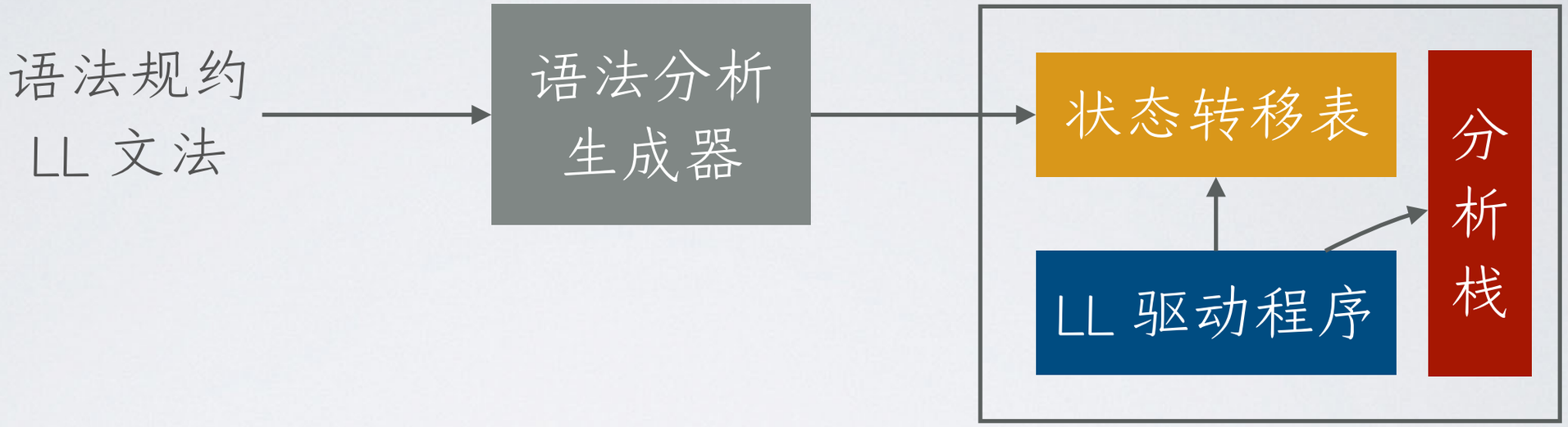
主要内容

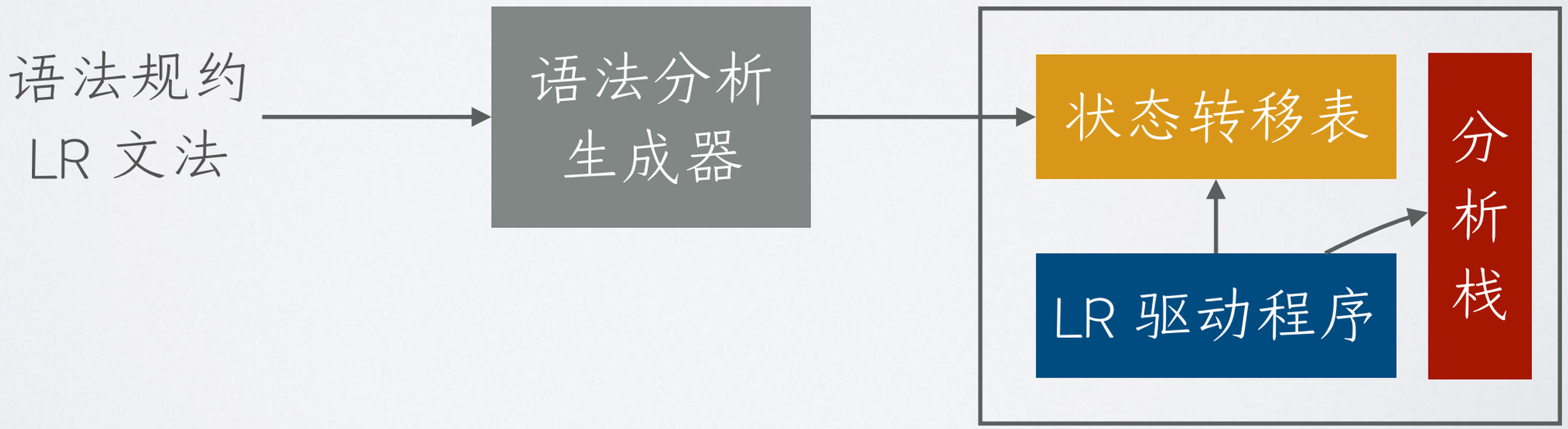
- 语法分析的作用
- 语法分析的规约
- 语法分析的手动实现
- **语法分析的自动生成**

从规约到实现的转换

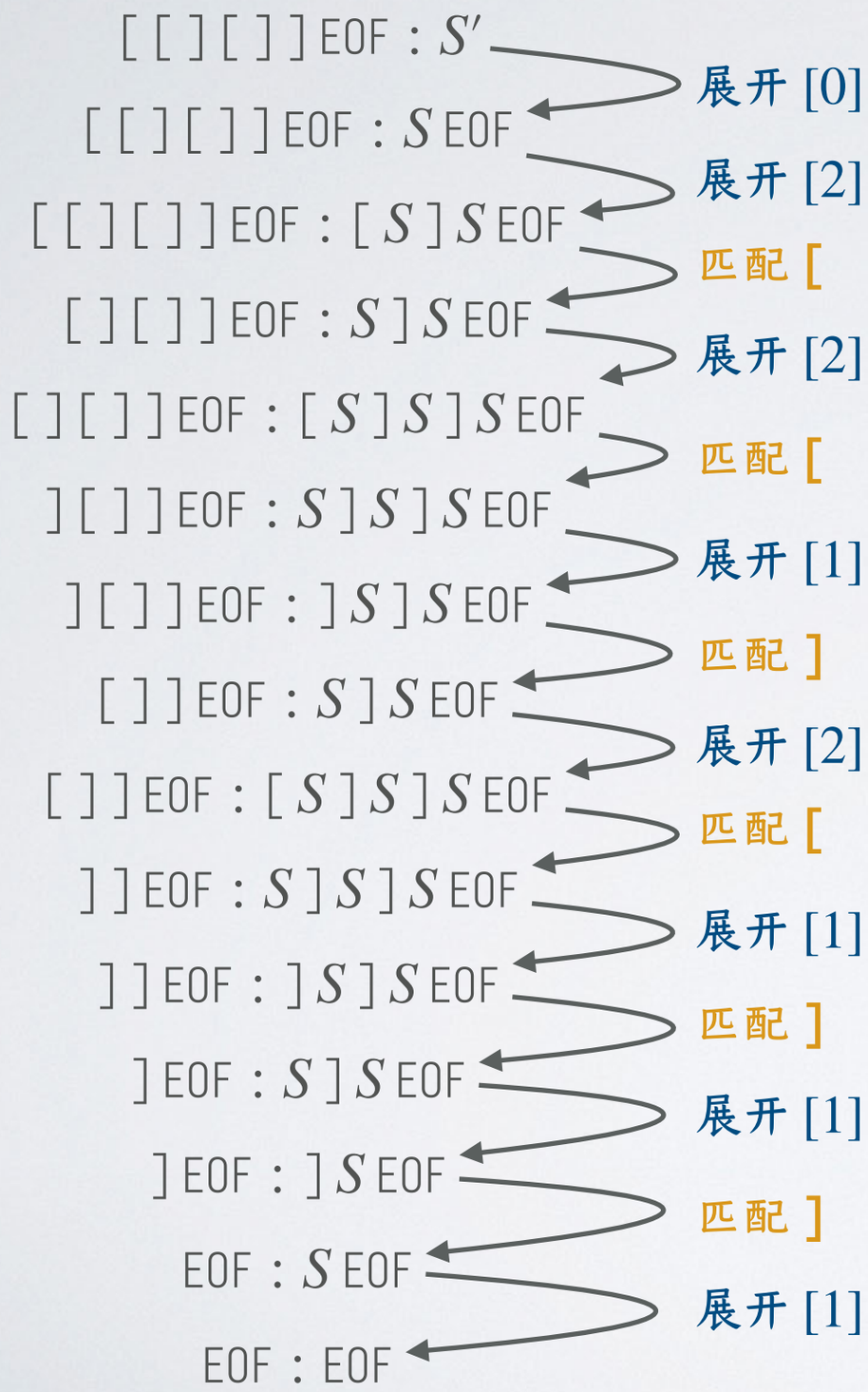
- **语法分析的规约：**编程语言的文法
  - 通常为 CFG 文法，比如 LL 或 LR 文法
- **语法分析的实现：**
  - 自顶向下：递归下降分析，适用于 LL 文法
  - 自底向上：移进-归约分析，适用于 LR 文法
- **语法分析的自动生成：**自动化 LL/LR 文法到实现的转换

表驱动的语法分析


语法规约
LL 文法
语法分析
生成器
状态转移表
分析栈
LL 驱动程序


语法规约
LR 文法
语法分析
生成器
状态转移表
分析栈
LR 驱动程序

LL(1) 文法：递归下降分析



[0]	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

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]
$\beta\,S$	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] [	‖	[ ]] EOF
EOF S] S] S	‖	]] EOF
EOF S] S]	‖	]] EOF
EOF S] S	‖	] EOF
EOF S]	‖	] EOF
EOF S	‖	EOF
EOF	‖	EOF

[0]	$S' \to S$ EOF
[1]	$S \to \epsilon$
[2]	$S \to [\ S\ ]\ S$

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]
β [	匹配	错误	错误
β]	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]	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

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]
$\beta\,S$	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 $A \to \alpha$，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 $A \to \alpha$ 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 $A \to \alpha$ 这条规则

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

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

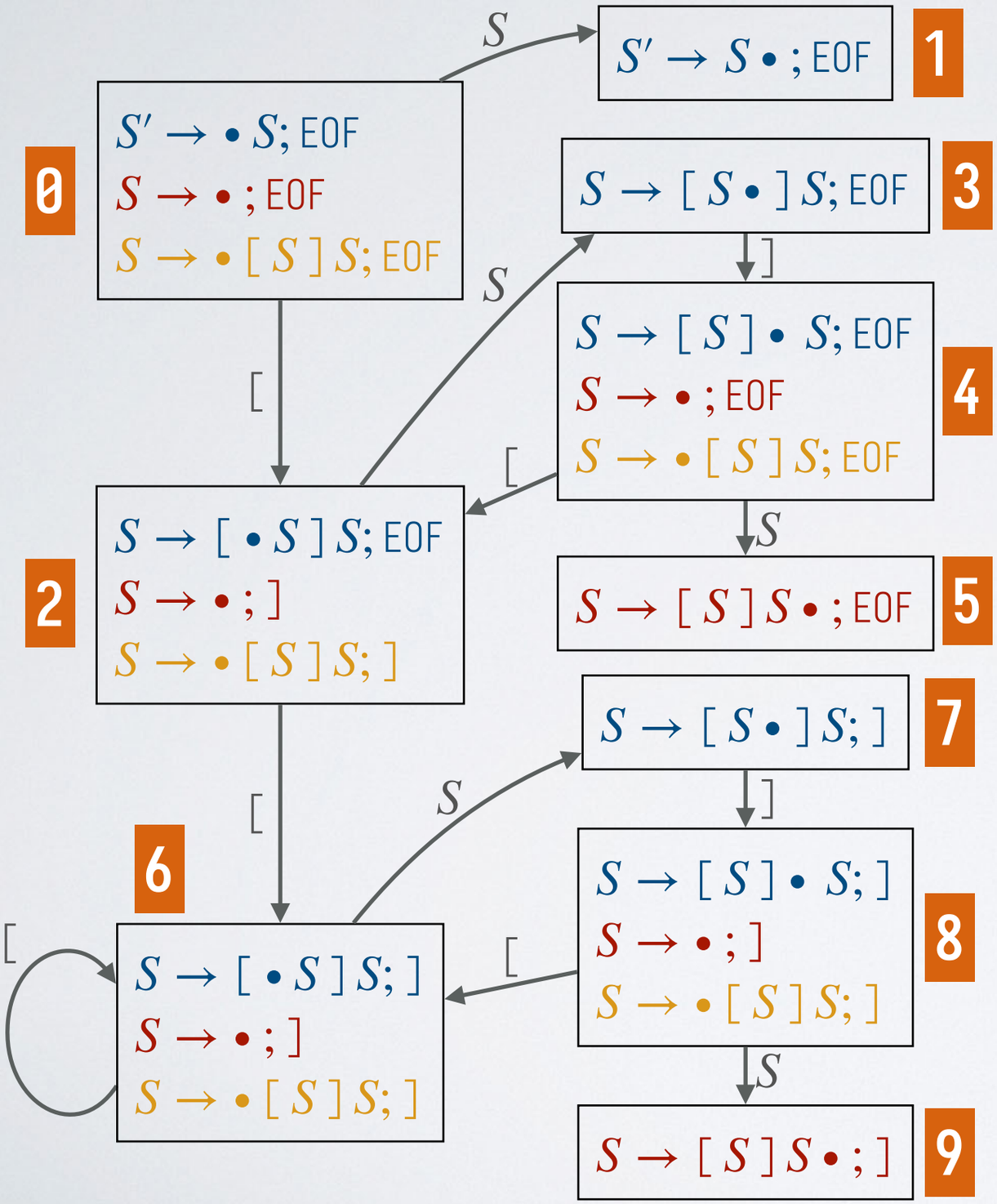
分析栈 \ 向前看	[	]	EOF
$\beta S'$	展开 [0]		展开 [0]
βS	展开 [2]	展开 [1]	展开 [1]
$\beta [$	匹配		
$\beta]$		匹配	
EOF			接受
ϵ			

LR(1) 文法：移进–归约分析

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

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

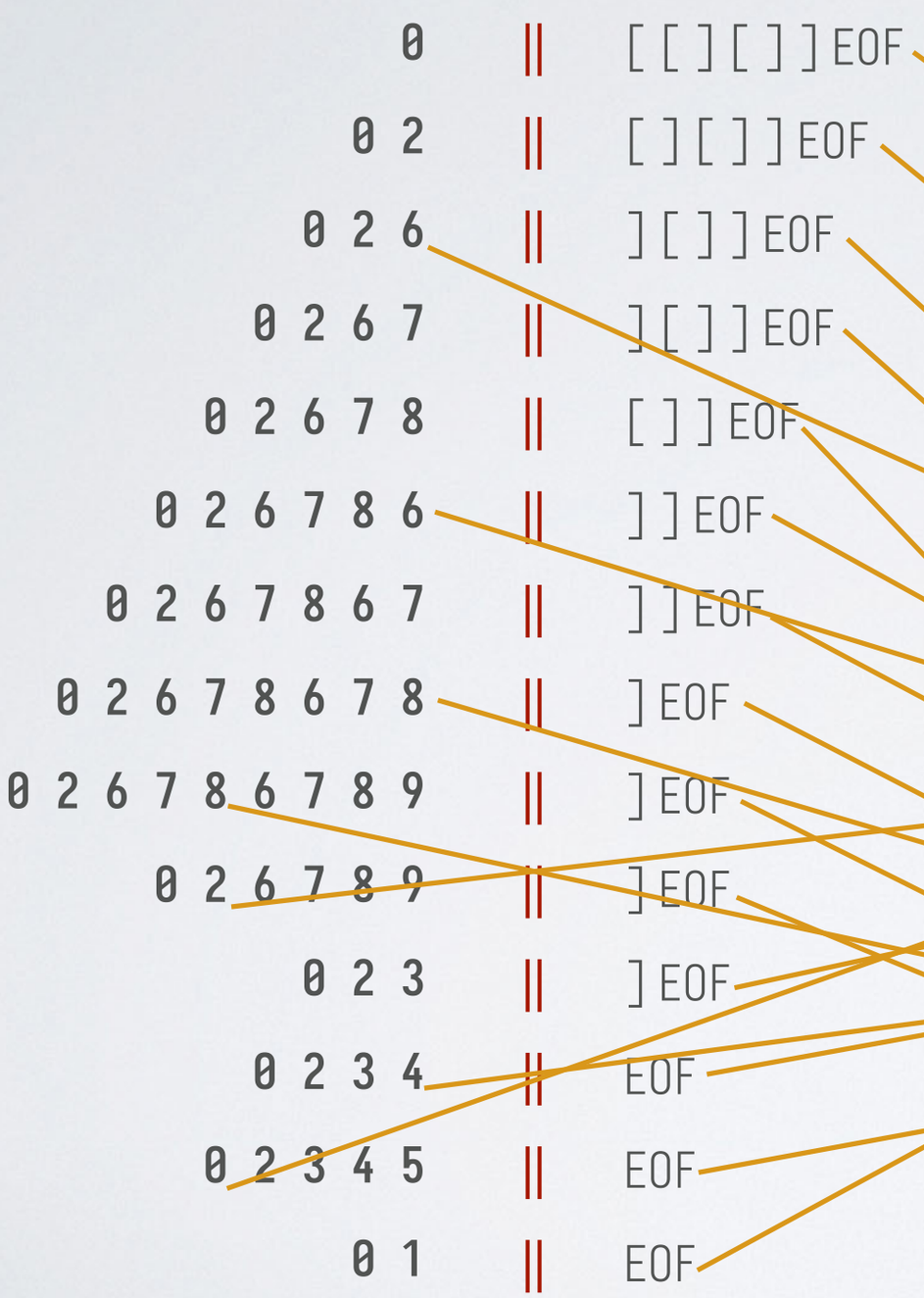
LR(1) 文法：移进-归约分析



$$
\begin{array}{ll}
 & S' \to S\ \mathrm{EOF} \\
[1] & S \to \epsilon \\
[2] & S \to [\ S\ ]\ S
\end{array}
$$

状态 \ 向前看	ACTION [	ACTION]	ACTION EOF	GOTO 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：移进-归约分析



$$
\begin{array}{ll}
 & S' \to S\,\mathrm{EOF} \\
[1] & S \to \epsilon \\
[2] & S \to [\,S\,]\,S
\end{array}
$$

	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) 文法的驱动程序

```
token = next_token();
top = 1; stack[top] = I_0;
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 I_j") {
    top++; stack[top] = I_j;
    token = next_token();
  } else if (ACTION[state][token] == "接受") {
    return true;
  } else {
    return false;
  }
}
```

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

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 ACTION$[I_i, c]$ 为“移进 I_j”
  - 如果 I_i 中有可归约模式 $\langle A \to \alpha \bullet ; c \rangle$, 那么 ACTION$[I_i, c]$ 为“归约 $A \to \alpha$”
  - 如果 I_i 中有可归约模式 $\langle S' \to S \bullet ; \texttt{EOF} \rangle$, 那么 ACTION$[I_i, \texttt{EOF}]$ 为“接受”
  - 如果对于非终结符号 X 有转移到 I_j, 那么 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

主要内容

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

产生式文法 vs 基于识别的文法

- 产生式文法(Generative Grammar)
  - 文法定义的语言通过产生式推导而来
  - 例子：正则表达式，上下文无关文法
  - **缺点：原本是为自然语言设计，二义性难以避免**
- **基于识别的文法(Recognition-Based Grammar)**
  - 编译注重**识别**符号串并构造语法分析树
  - 文法定义的语言通过**识别规则**来规定
- 例子：
  - $\{s \in 0^* \mid s = (00)^n\}$ 是产生式的
  - $\{s \in 0^* \mid (|s| \bmod 2 = 0)\}$ 是基于识别的

解析表达文法

- **Parsing Expression Grammar**，简称 PEG[For04]
- CFG 与 PEG 的关键语义区别：
  - CFG：产生式 $A \to \alpha_1 \mid \alpha_2$ 对 α_1、α_2 没有规定优先级
  - PEG：识别规则 $A \leftarrow \alpha_1 \ / \ \alpha_2$ **优先识别** α_1，失败后再考虑 α_2
- 例子：
  - CFG 中 $A \to 01 \mid 0$ 和 $A \to 0 \mid 01$ 是等价的
  - PEG 中 $A \leftarrow 01 \ / \ 0$ 和 $A \leftarrow 0 \ / \ 01$ 是**不等价**的
    - 后者的 01 永远不会被考虑

[For04] Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. Princ. Of Prog. Lang. (POPL'04), 2004.

解析表达文法

- **终结符号** a：识别符号 a
- $e_1\ e_2$：识别 e_1，若成功则继续识别 e_2，有一个失败则识别失败
- e_1 / e_2：先识别 e_1，若失败则尝试从相同位置识别 e_2
  - 例子：$S \leftarrow$ iSeS / iS / a 能正确识别悬空 else
- $e?, e^*, e^+$：类似正则表达式，但**贪心**地识别尽可能多的 e
  - 例子：0* 0 无法识别任何符号串
- PEG 支持**语法谓词**，在「向前看」符号串满足条件时识别空串
  - $\&e$：若「向前看」能识别 e，则成功
  - $!e$：若「向前看」不能识别 e，则成功
  - 例子：$C \leftarrow$ o (C / (!c a))* c 能正确识别嵌套注释

PEG 描述了自顶向下的语法分析

- PEG 的识别过程实质是在进行**带回溯**的递归下降分析

- **两个问题：**
  - **需要考虑如何支持左递归**
  - **回溯需要来回扫描，效率低**

- 通过左递归消除得到没有左递归的文法

- Packrat 分析算法：通过**记忆化**提高效率[For02]
  - 递归下降分析的结果只依赖于**当前的非终结符号**和**待处理的输入**
  - 通过动态规划用空间换时间，达到线性时间复杂度
  - CPython 把基于 LL(1) 文法的分析替换为了 Packrat 分析

[For02] Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. Int. Conf. on Functional Programming (ICFP'02), 2002.

Packrat 分析算法

$$E \leftarrow T + E \;/\; T$$
$$T \leftarrow F * T \;/\; F$$
$$F \leftarrow (E) \;/\; D$$
$$D \leftarrow 0 \;/\; \ldots \;/\; 9$$

C1	C2	C3	C4	C5	C6	C7	C8
2	*	(	3	+	4	)	EOF

	C1	C2	C3	C4	C5	C6	C7	C8
E	C8	✗	C8	C7	✗	C7	✗	✗
T	C8	✗	C8	C5	✗	C7	✗	✗
F	C2	✗	C8	C5	✗	C7	✗	✗
D	C2	✗	✗	C5	✗	C7	✗	✗

本讲小结

- 语法分析的规约给出编程语言的文法
  - 通常使用 CFG 来描述，比如 LL 或 LR 文法
- 语法分析可以通过递归下降法或移进-归约法进行实现
  - 递归下降：自顶向下，最左推导，消除左递归，提取左公因子，计算 FIRST、FOLLOW 集合，LL(1)
  - 移进-归约：自底向上，最右推导，构造识别栈模式(本质是识别句柄)的自动机，LR(1)、LR(0)、SLR(1)、LALR(1)
- 语法分析的自动生成
  - 表驱动的语法分析
  - 递归下降：构造 LL(1) 文法的分析表
  - 移进-归约：构造 LR(1) 文法的分析表

思考问题

- 为什么编译过程需要语法分析?
- 语法分析需要知道编程语言的哪些信息?
- 语法分析前先进行词法分析有什么好处?
- 语法分析可以和词法分析合二为一吗?
- 可以用移进-归约法分析二义性文法吗?有什么好处?
- 构造文法:
  - LR(0) 但不是 LL(0), SLR(1) 但不是 LR(0), LALR(1) 但不是 SLR(1) , LR(1) 但不是 LALR(1), LR(1) 但不是 LL(1), ……
  - LL(1) 但不是 LR(0), LR(0) 但不是 LL(1), ……