北京大学计算机学院

2025 年春季学期　　《编译原理(实验班)》

第三讲　语法分析（基础版）

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

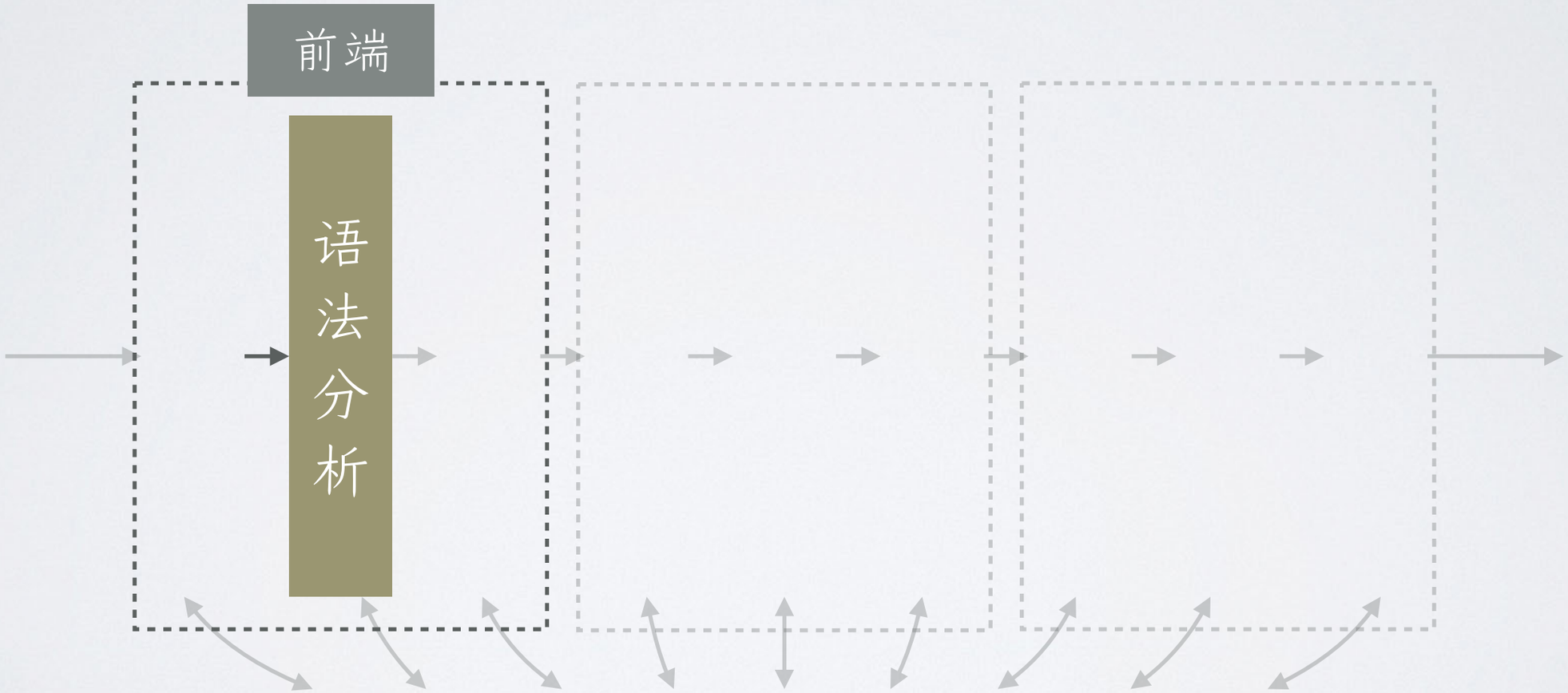
- 对应龙书章节：第 4 章

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」

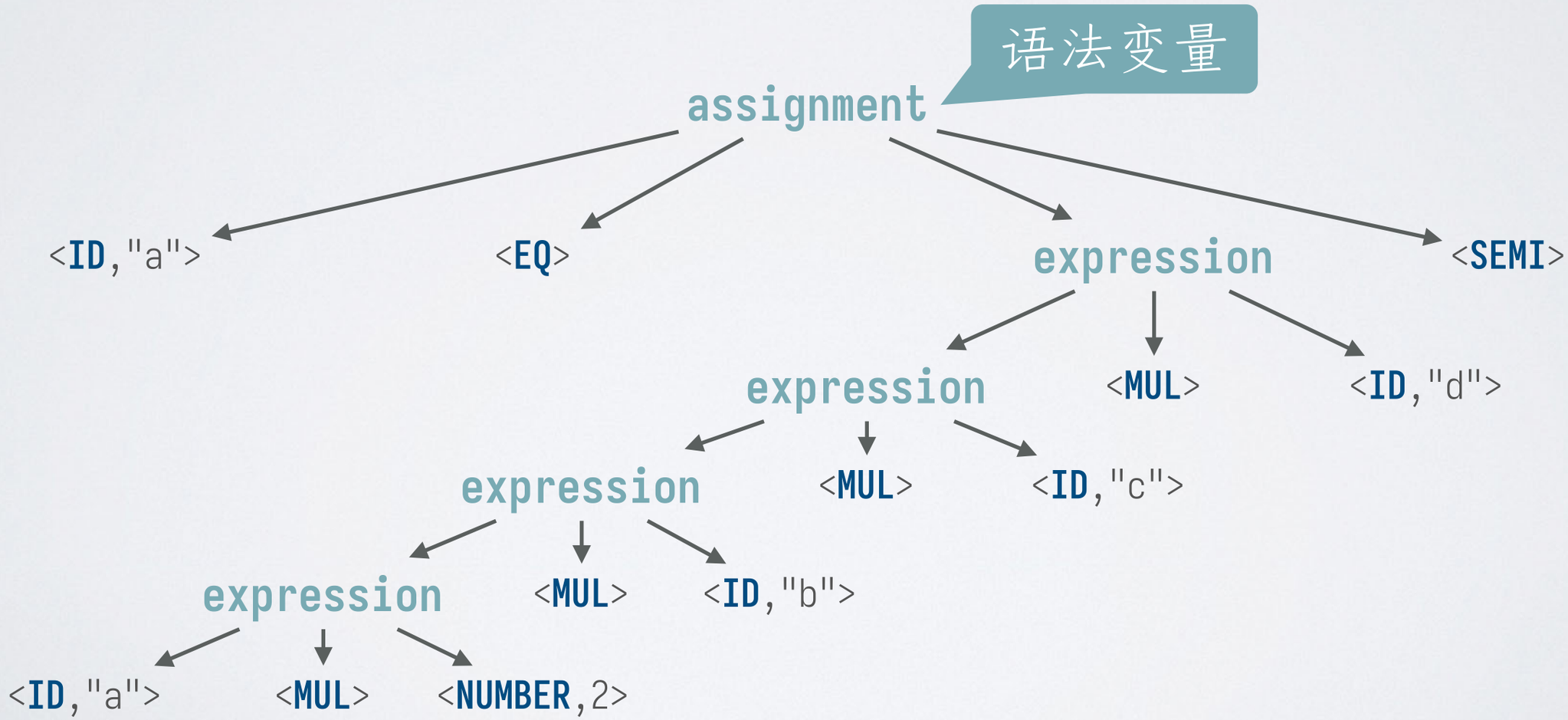


语法分析示例

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

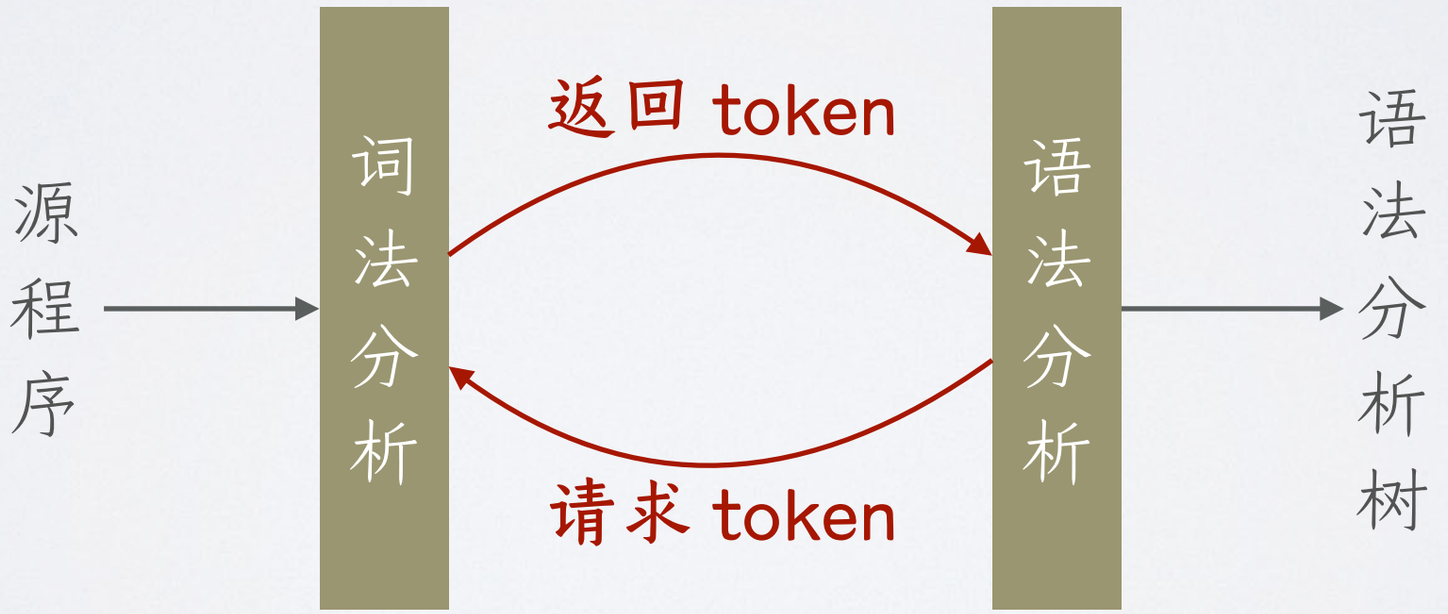
<ID,"a"> <EQ> <ID,"a"> <MUL> <NUMBER,2> <MUL> <ID,"b"> <MUL> <ID,"c"> <MUL> <ID,"d"> <SEMI>

- 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

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

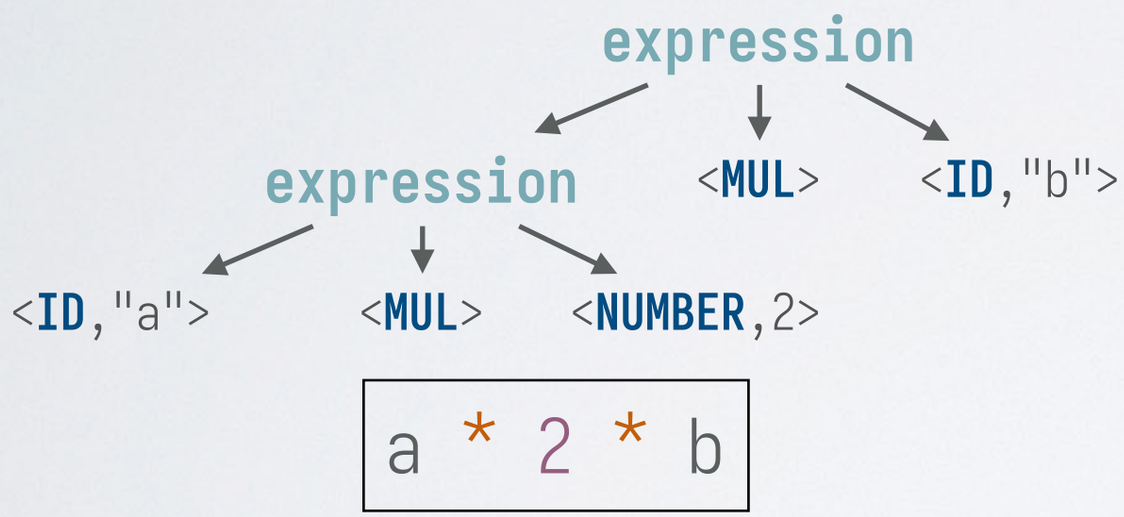
这个 BF 程序把指针指向位置的值加到下一个位置

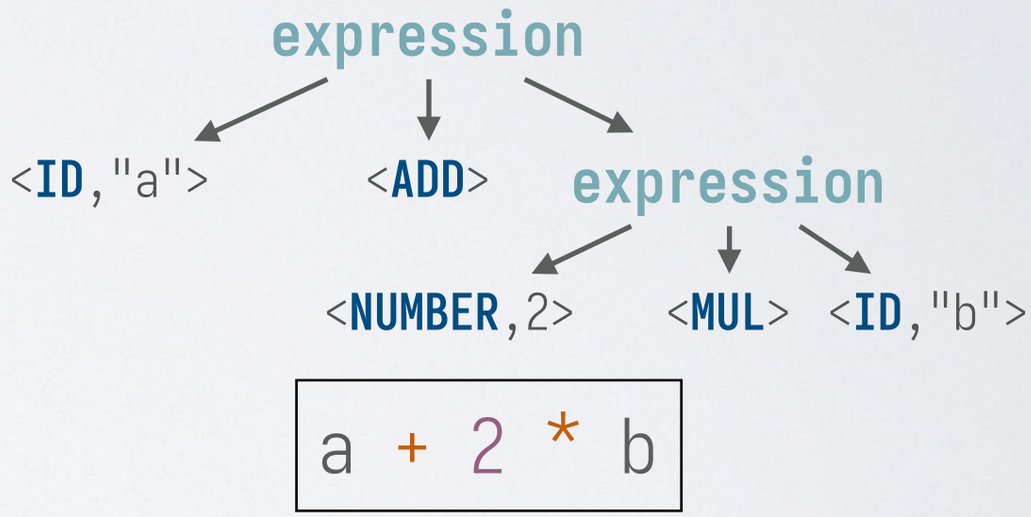
字符	语义
>	把指针往右移一个位置
<	把指针往左移一个位置
+	把指针指向位置的值加一
-	把指针指向位置的值减一
[	如果指针指向位置的值为零，跳转到匹配的] 之后
]	如果指针指向位置的值非零，跳转到匹配的 [之后

- 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

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)

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]

[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？

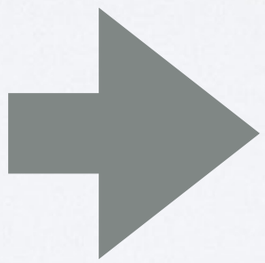
- **能用正则表达式表示配平的括号串构成的语言吗？**
- **很遗憾，不能**
- 但是在编程语言语法中，类似「配平的括号串」的结构很常见

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

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

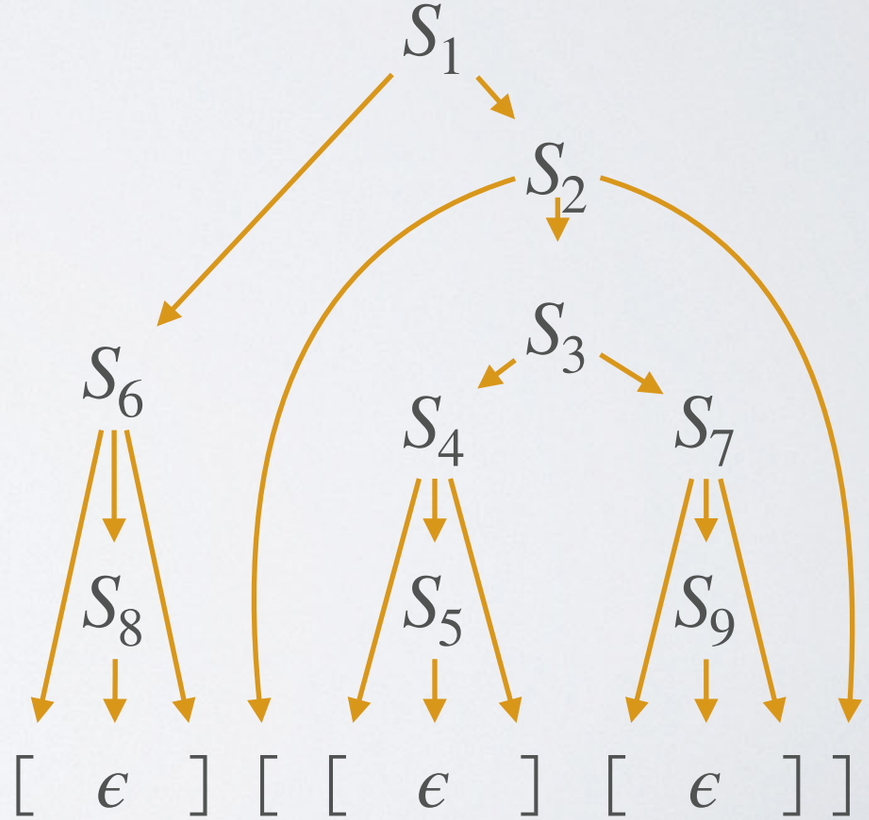
- 考虑文法 $G = (V_T, V_N, S, P)$，若 $\alpha \to \beta$ 是 P 中的产生规则，且 γ、δ 是由 $V_T \cup V_N$ 构成的符号串，则称 $\gamma\alpha\delta$ **直接推导** $\gamma\beta\delta$，表示为 $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$
- 多个直接推导构成的序列被称为**推导**
  - $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$（$n$ 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

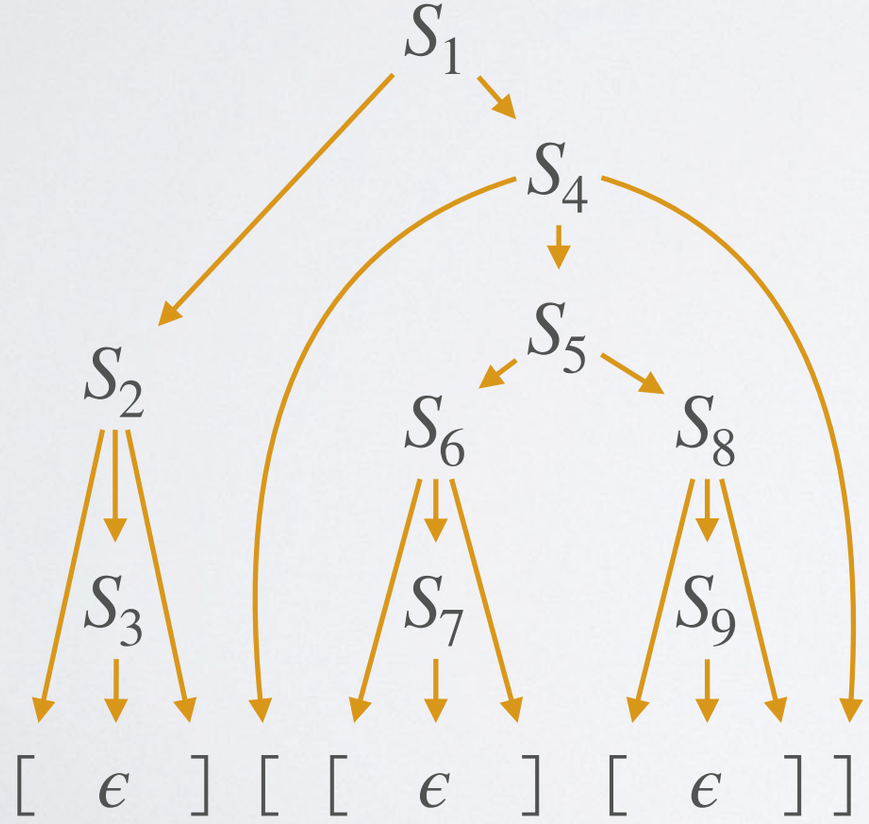
$$
\begin{aligned}
S &\Rightarrow SS && [3]\\
&\Rightarrow S[S] && [2]\\
&\Rightarrow S[SS] && [3]\\
&\Rightarrow S[[S]S] && [2]\\
&\Rightarrow S[[]S] && [1]\\
&\Rightarrow [S][[]S] && [2]\\
&\Rightarrow [S][[][S]] && [2]\\
&\Rightarrow [][[][S]] && [1]\\
&\Rightarrow [][[][]] && [1]
\end{aligned}
$$

[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

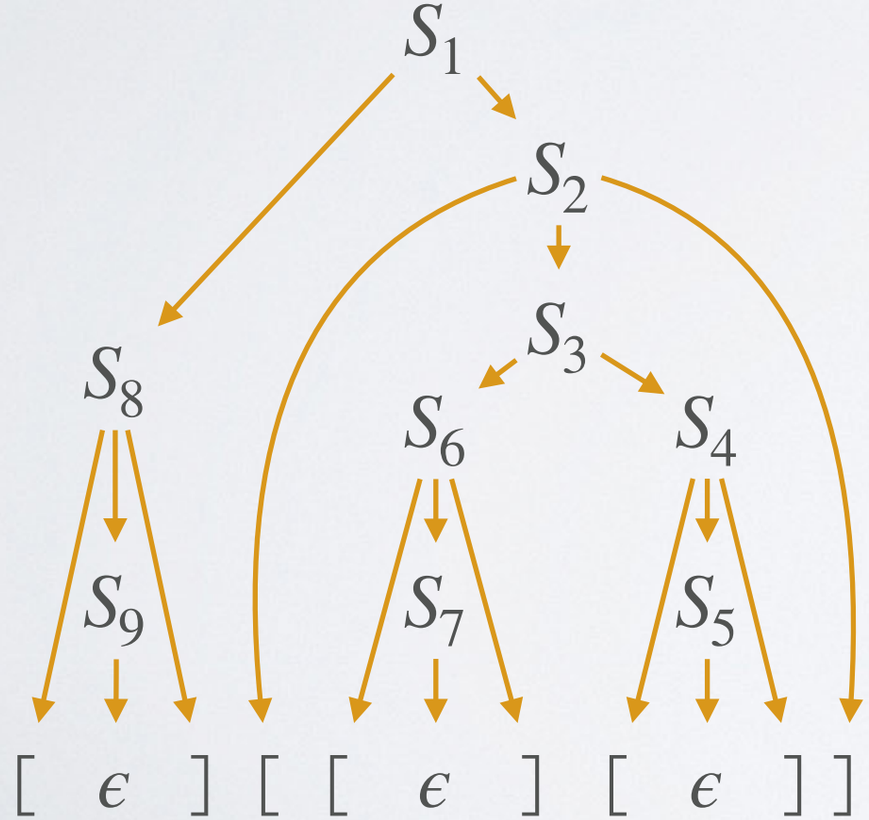


$$
\begin{aligned}
S &\Rightarrow SS \\
&\Rightarrow [S]S \\
&\Rightarrow [\,]S \\
&\Rightarrow [\,][S] \\
&\Rightarrow [\,][SS] \\
&\Rightarrow [\,][[S]S] \\
&\Rightarrow [\,][[\,]S] \\
&\Rightarrow [\,][[\,][S]] \\
&\Rightarrow [\,][[\,][\,]]
\end{aligned}
$$

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

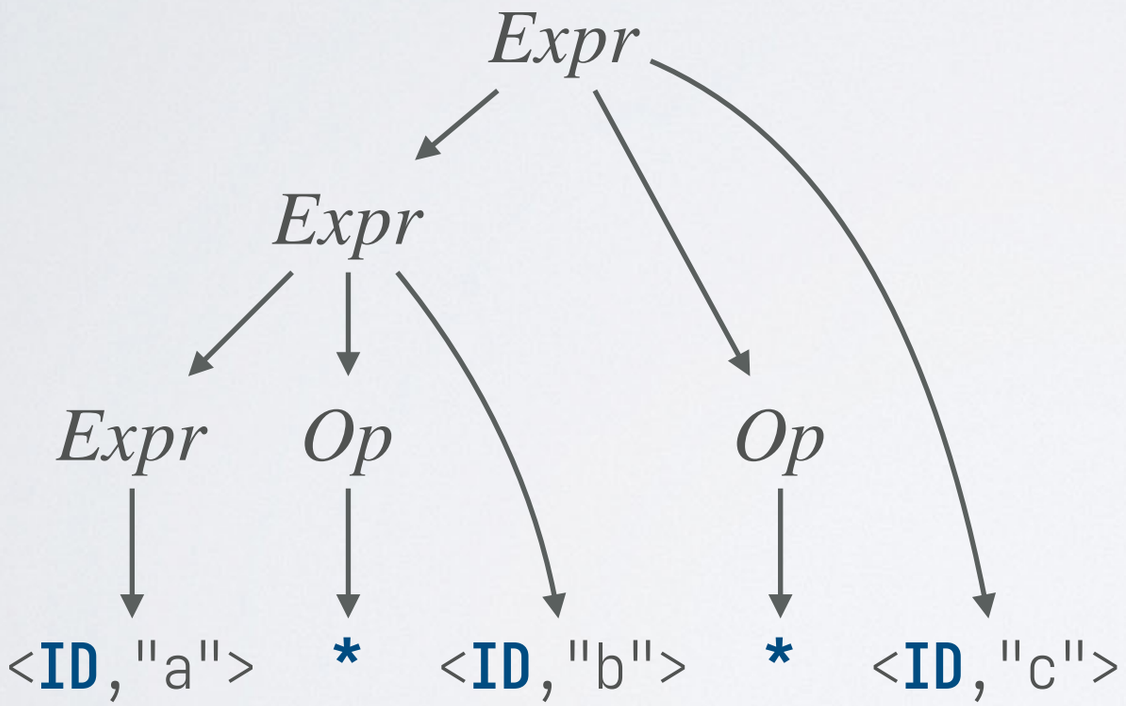


最右推导

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

语法分析树示例

```
a * b * c
```

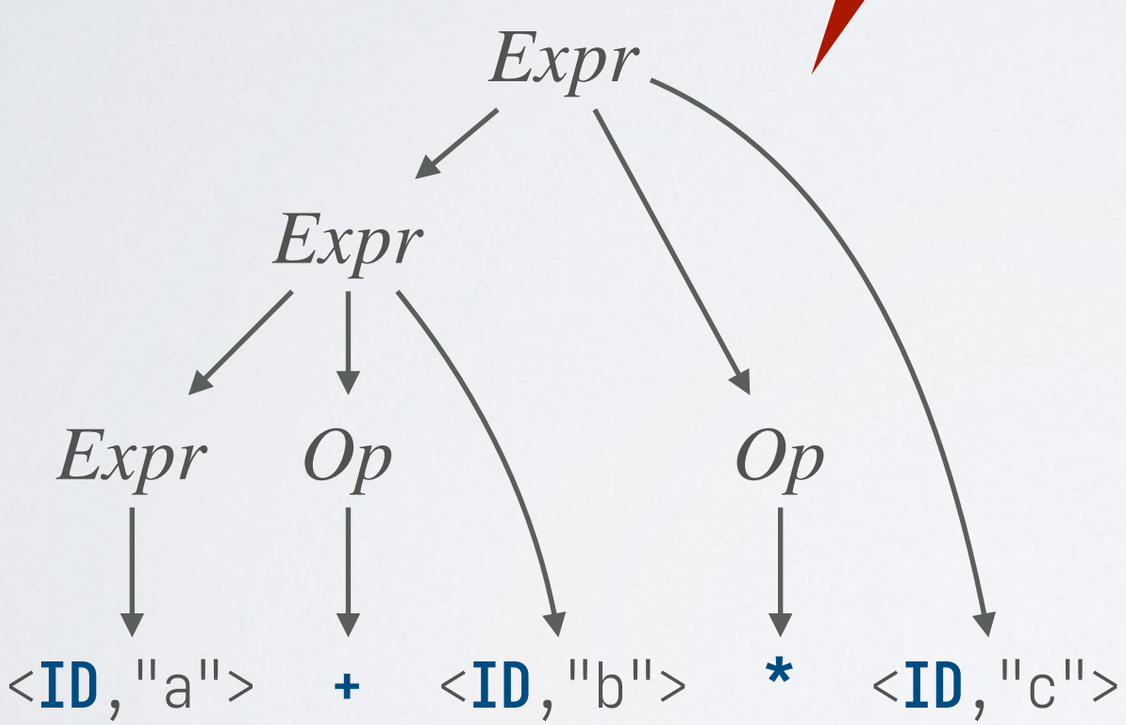


$$
\begin{aligned}
Expr &::= (\ Expr\ ) \\
&\mid\ Expr\ Op\ \texttt{ID} \\
&\mid\ \texttt{ID} \\
Op &::= \texttt{+} \\
&\mid\ \texttt{-} \\
&\mid\ \texttt{*} \\
&\mid\ \texttt{/}
\end{aligned}
$$

语法分析树示例

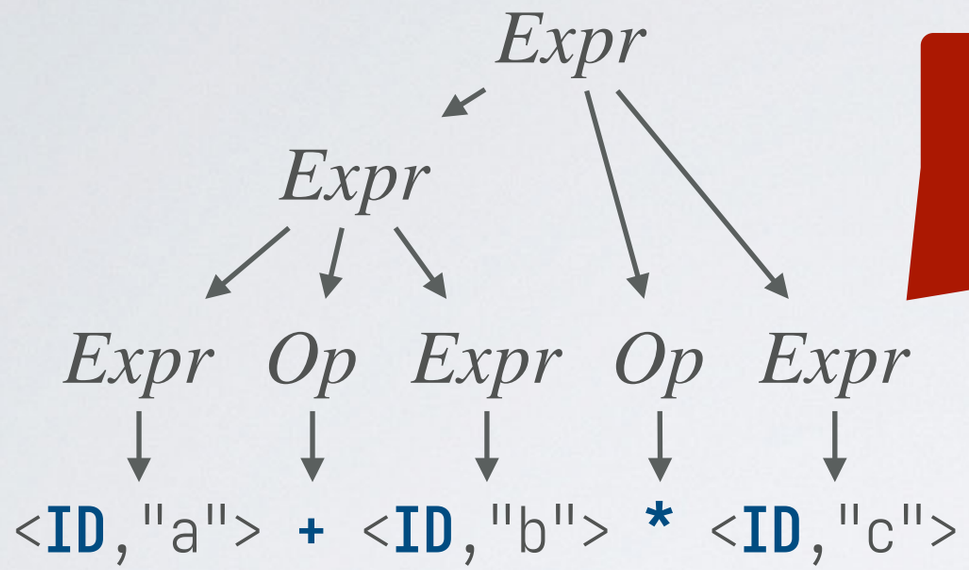
```
a + b * c
```

但语义一般要求先计算乘法



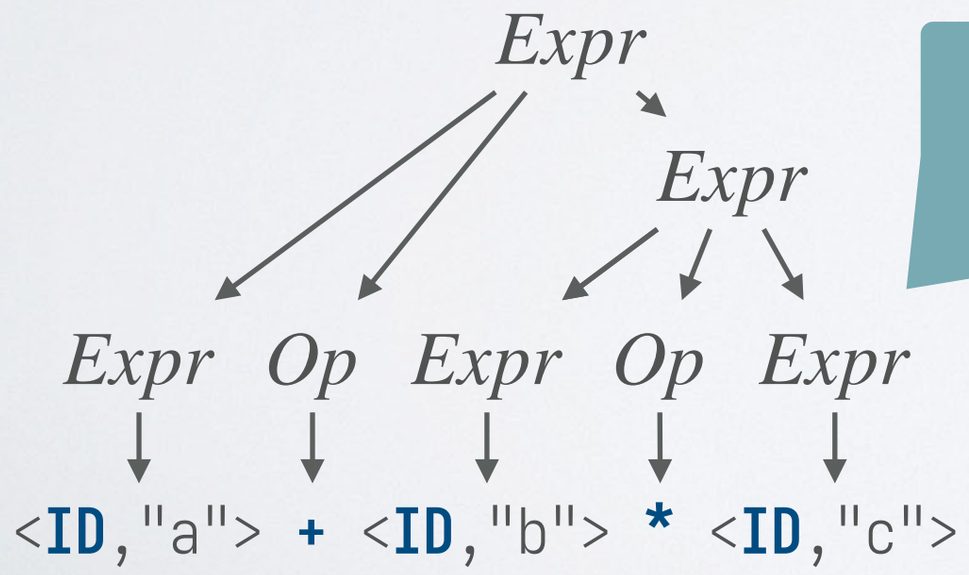
$$
\begin{aligned}
Expr &::= (\ Expr\ ) \\
&\mid\ Expr\ Op\ \texttt{ID} \\
&\mid\ \texttt{ID} \\
Op &::= \texttt{+} \\
&\mid\ \texttt{-} \\
&\mid\ \texttt{*} \\
&\mid\ \texttt{/}
\end{aligned}
$$

语法分析树示例



先计算加法
再计算乘法

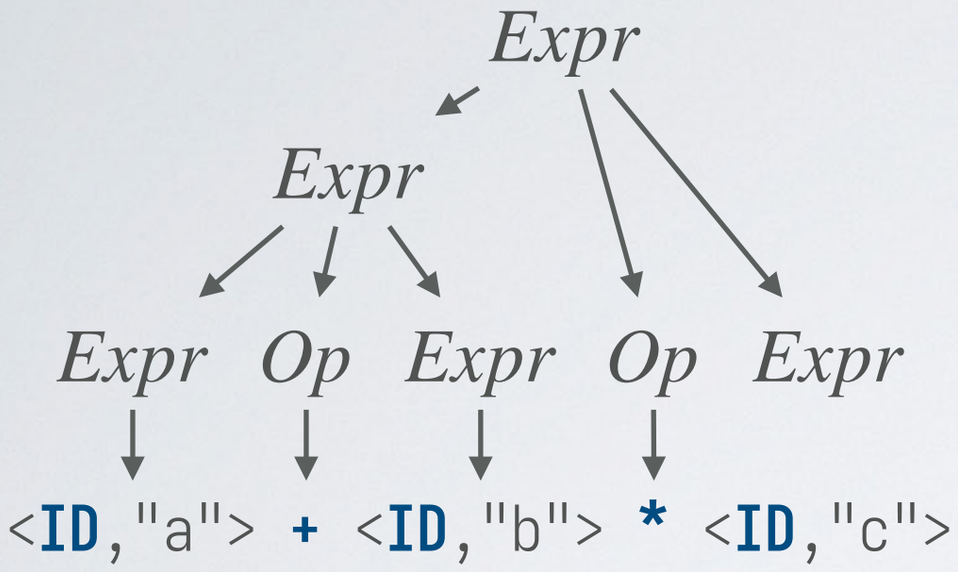
```
a + b * c
```



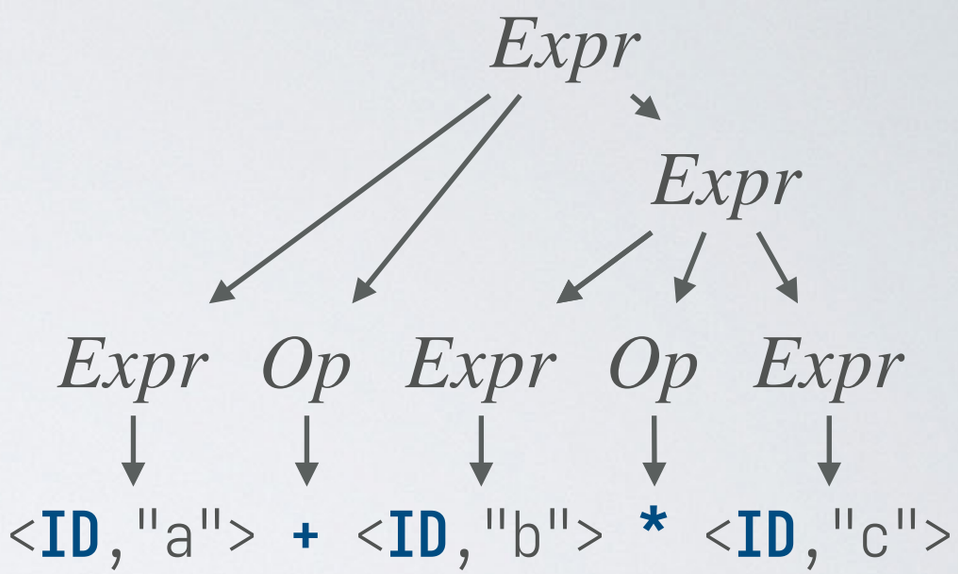
先计算乘法
再计算加法

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

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

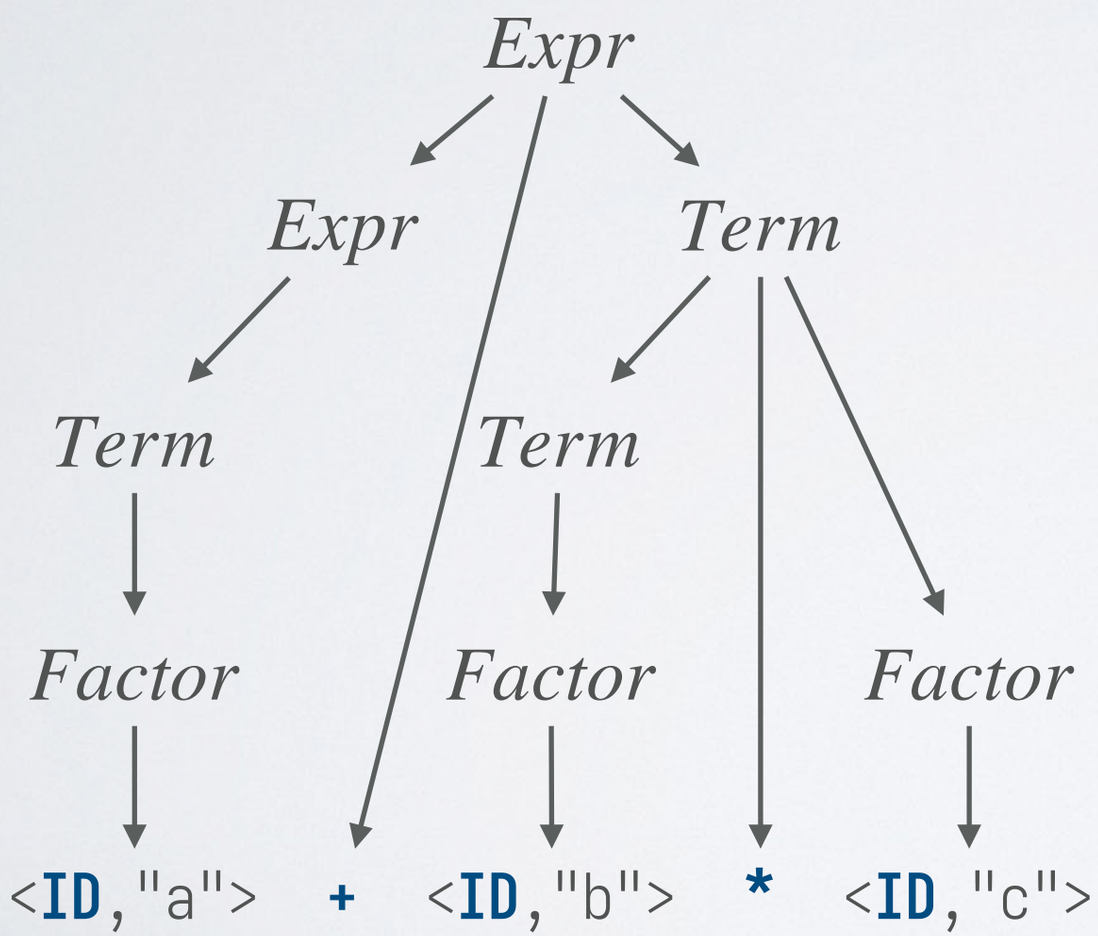
- *Expr* 的文法有二义性的原因是没有在文法中遵循四则运算规则
- 转换文法以体现**运算符优先级**
  - 括号的优先级最高
  - 乘法(*)和除法(/)的优先级次之
  - 加法(+)和减法(-)的优先级最低
- 给每种优先级引入一个**非终结符号**
  - *Factor* 对应括号
  - *Term* 对应乘、除法
  - *Expr* 对应加、减法

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

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

消除二义性

```
a + b * c
```



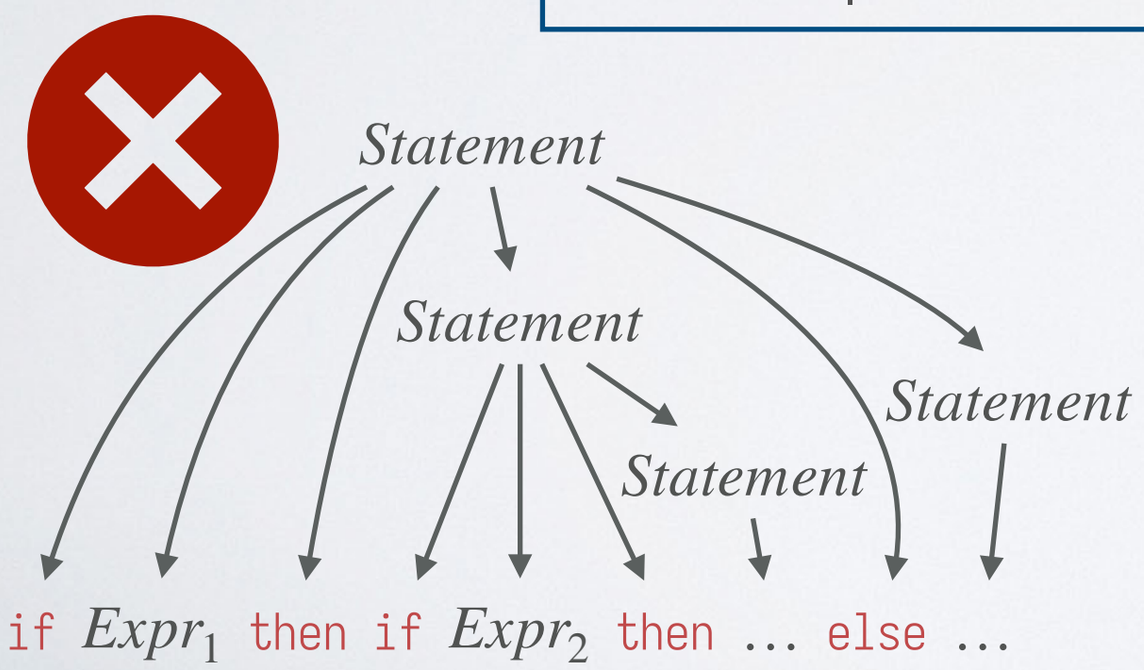
$$
\begin{aligned}
Expr &::= Expr + Term \\
&\mid Expr - Term \\
&\mid Term \\
Term &::= Term * Factor \\
&\mid Term / Factor \\
&\mid Factor \\
Factor &::= (\ Expr\ ) \\
&\mid \texttt{ID}
\end{aligned}
$$

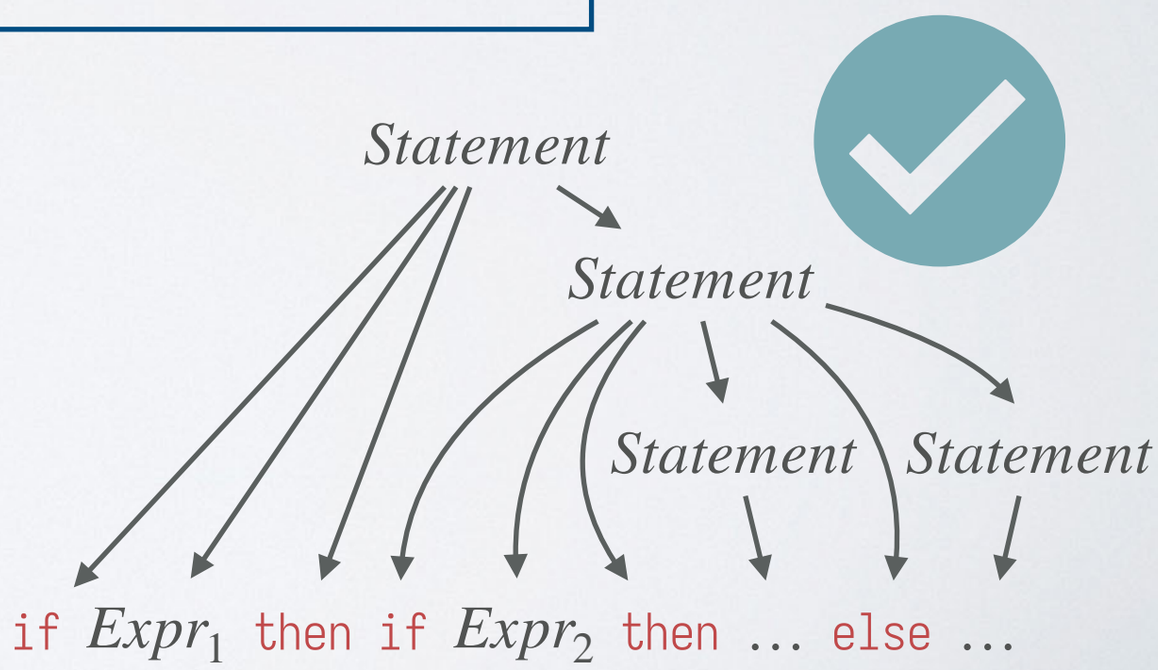
消除二义性

- 另一个经典的二义性例子是 if-then-else 语句

- 二义性文法：

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





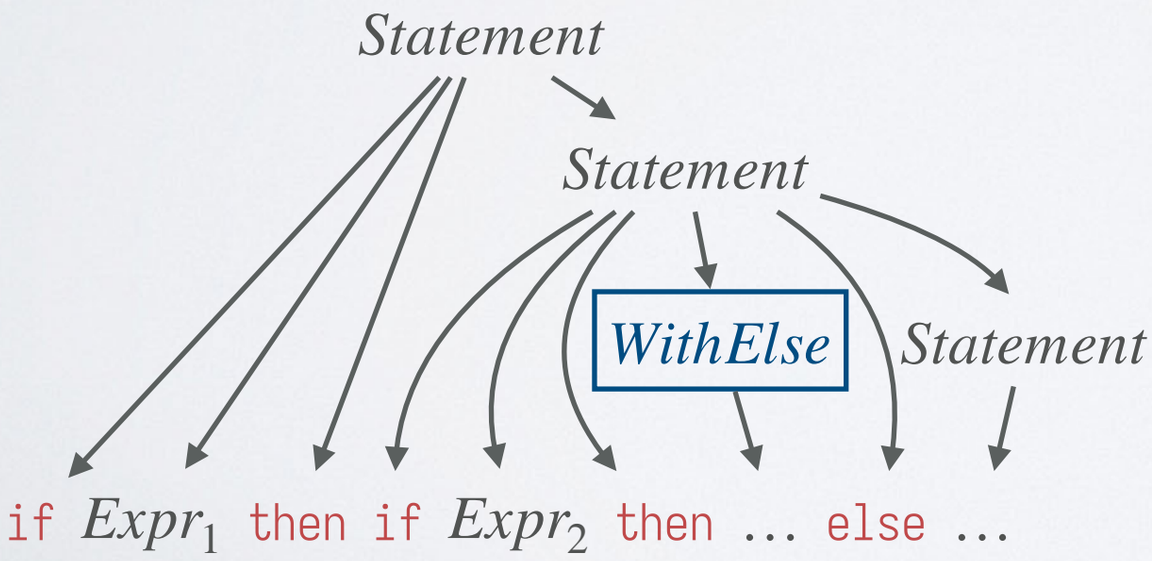
消除二义性

$Statement$ ::= `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

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

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

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

```
{
  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

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

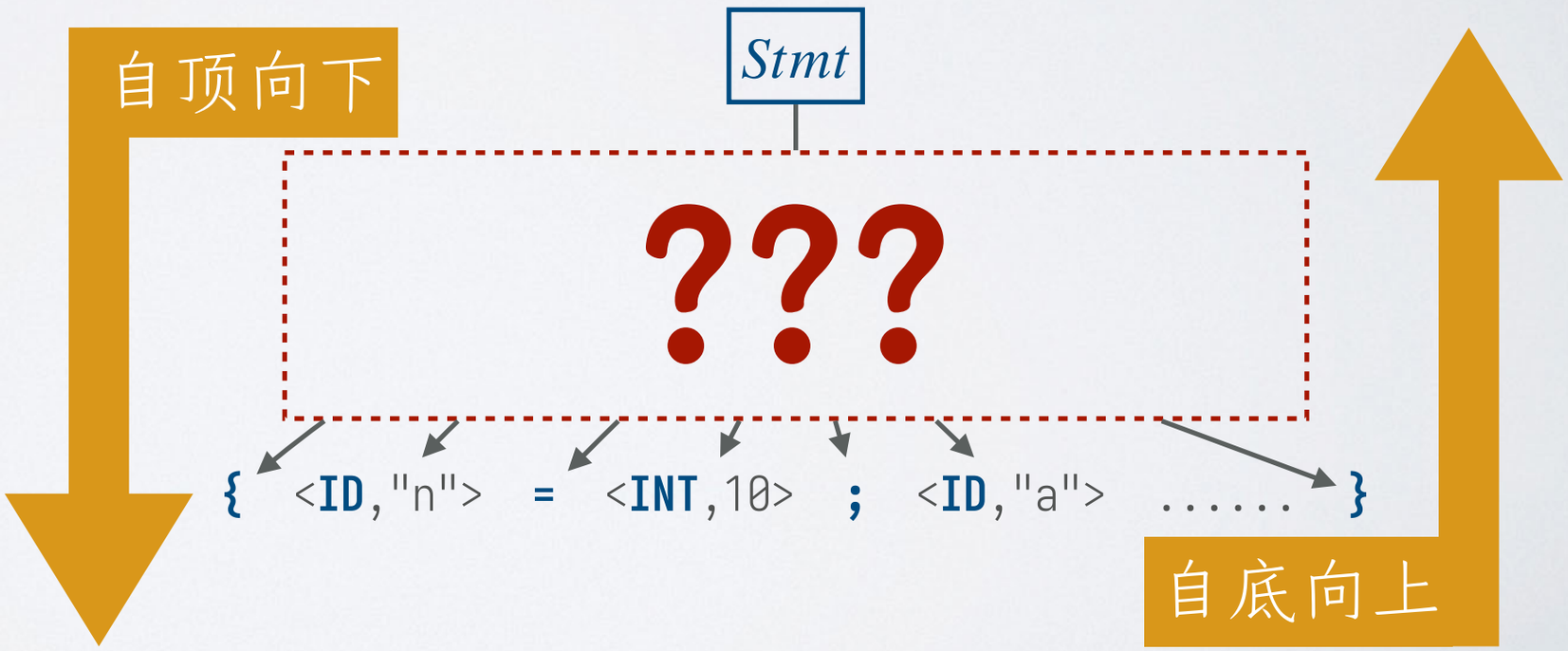
若「甲」等於一者
……
若非
……
也

主要内容

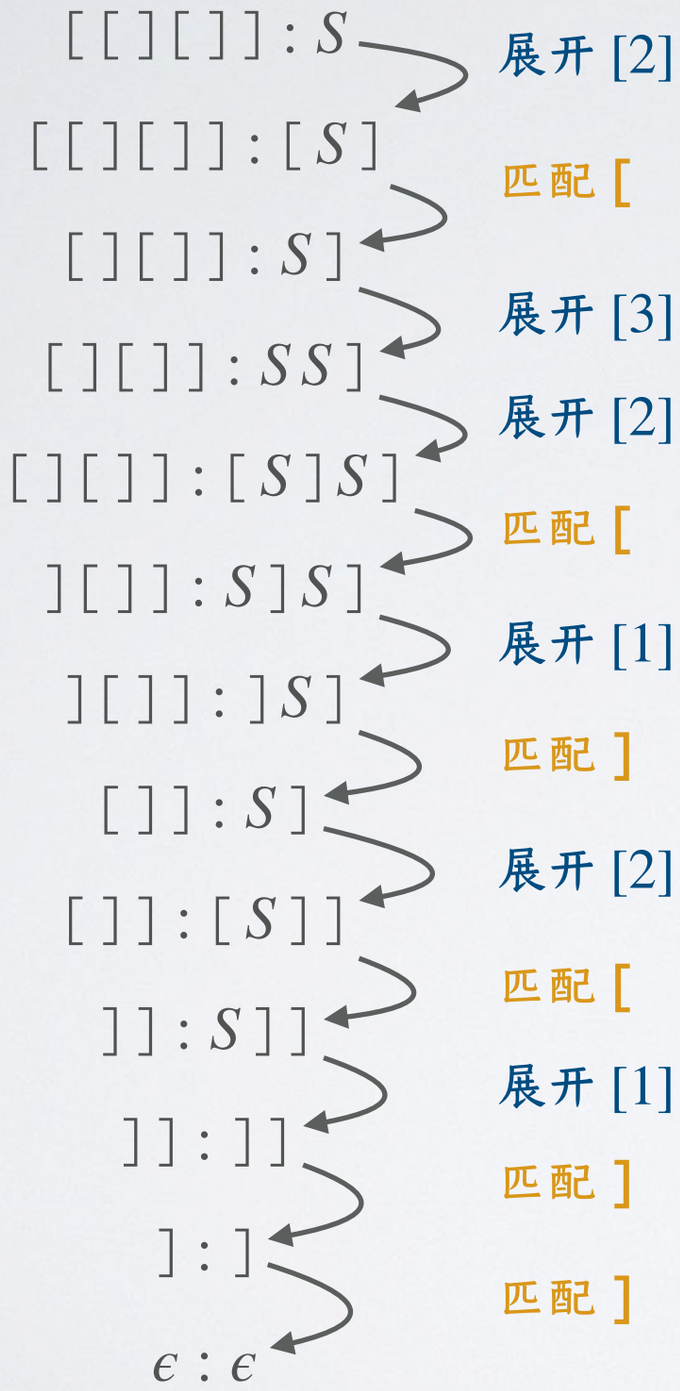
- 语法分析的作用
- 语法分析的规约
- **语法分析的手动实现**

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;
}
```



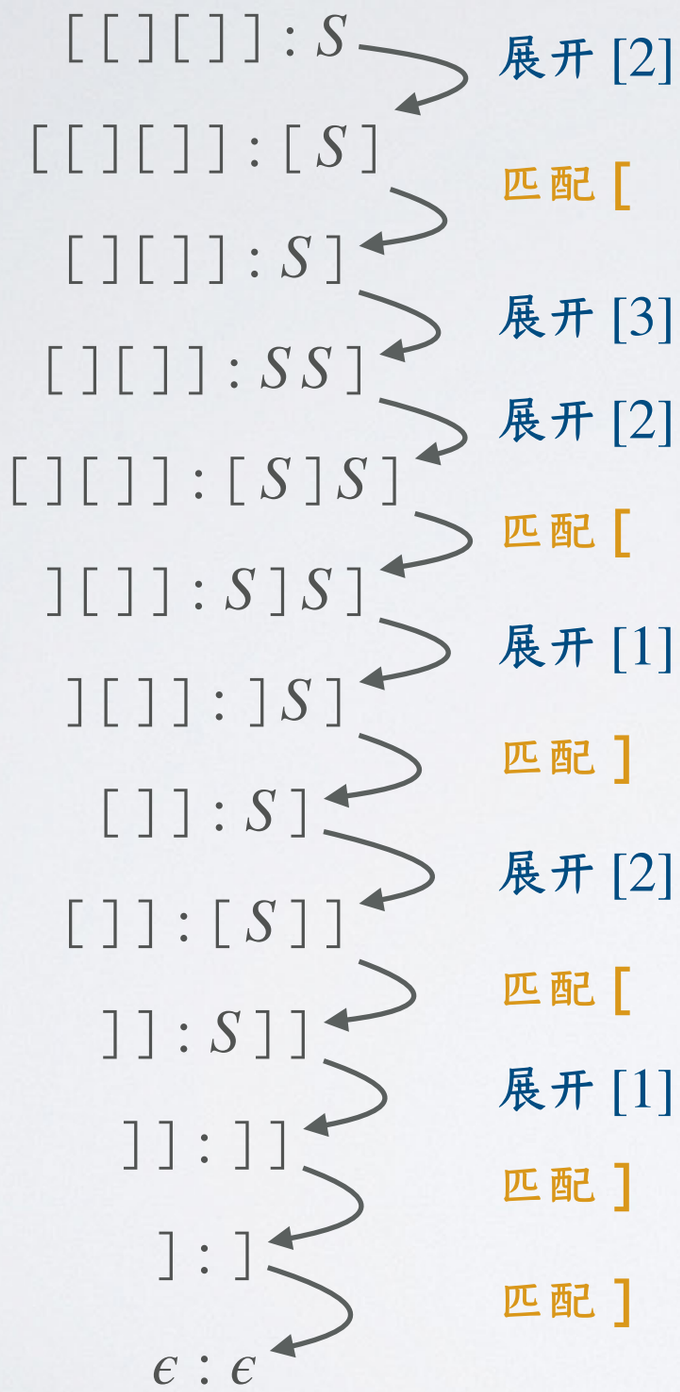
自顶向下语法分析



[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

$$
\begin{aligned}
Expr ::= & \ Expr + Term \\
| & \ Expr - Term \\
| & \ \ldots
\end{aligned}
$$

```
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：
$$
\begin{aligned}
S &::= A\,\mathsf{a} \mid \mathsf{b} \\
A &::= S\,\mathsf{d} \mid \epsilon
\end{aligned}
$$

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

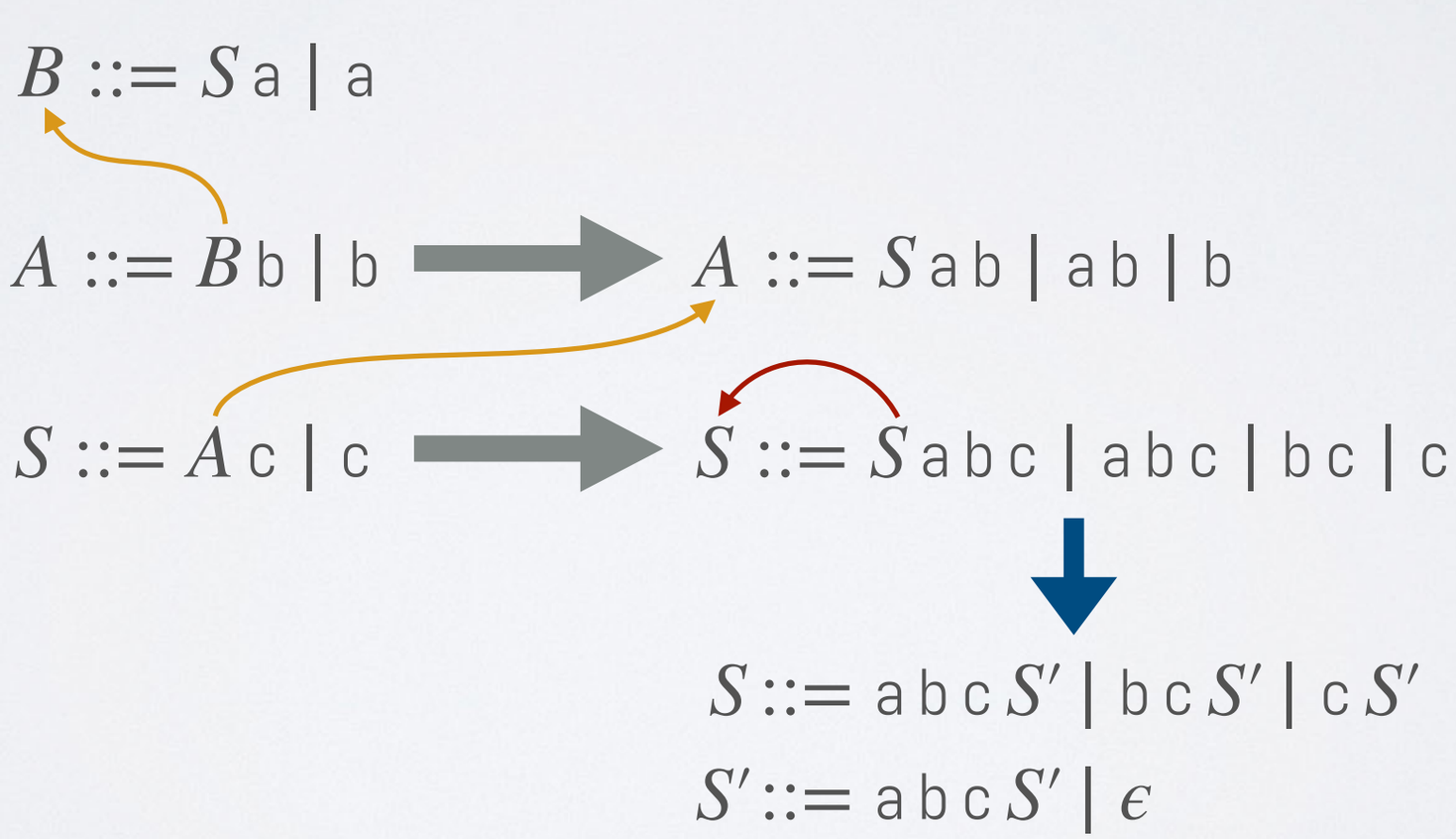
- 把文法的非终结符号整理成某种顺序 $A_1, A_2, \ldots, A_n$
- 依次遍历每个非终结符号 A_i
  - 对每个 $j < i$ 和形如 $A_i ::= A_j \gamma$ 的规则替换为 $A_i ::= \delta_1 \gamma \mid \ldots \mid \delta_k \gamma$
    - 其中 $A_j ::= \delta_1 \mid \ldots \mid \delta_k$ 是 A_j 的产生规则
  - 消除 $A_i ::= \delta_1 \gamma \mid \ldots \mid \delta_k \gamma$ 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 $\text{FIRST}(\beta)$

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

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

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

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

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

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

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

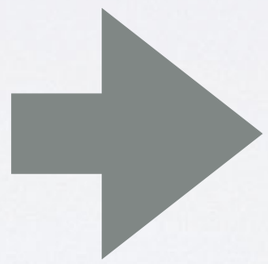
```
Expr'() {
  if (token == PLUS) {
    token = next_token();
    if (Term()) { if (Expr'()) return true; }
  } else if (token == MINUS) {
    token = next_token();
    if (Term()) { if (Expr'()) return true; }
  } else if (token == EOF || token == RPAREN) return true;
  return false;
}
```

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

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

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

- 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**

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	✗	✗

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

- **实践演示**

玩具语言

- C 语言，但只有主函数，只有 int 类型，只有直线代码

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

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

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

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

玩具语言的语法规约

$$
\begin{aligned}
Expr &\leftarrow Term\ ((\texttt{+} \mid \texttt{-})\ Term)^* \\
Term &\leftarrow Factor\ ((\texttt{*} \mid \texttt{/} \mid \texttt{\%})\ Factor)^* \\
Factor &\leftarrow \texttt{(}\ Expr\ \texttt{)} \mid \texttt{ID} \mid \texttt{NUMBER}
\end{aligned}
$$

保证四则运算的优先级以及左结合

$$
\begin{aligned}
Stmt &\leftarrow Block \\
&\mid \texttt{int ID}\ (\texttt{=}\ Expr)?\ \texttt{;} \\
&\mid \texttt{ID =}\ Expr\ \texttt{;} \\
&\mid \texttt{return}\ Expr\ \texttt{;} \\
Block &\leftarrow \texttt{\{}\ Stmt^*\ \texttt{\}}
\end{aligned}
$$

如果后续需要加入条件语句，则可以体现 PEG 的优势：

$$
\begin{aligned}
Stmt &\leftarrow \texttt{if (}\ Expr\ \texttt{)}\ Stmt\ \texttt{else}\ Stmt \\
&\mid \texttt{if (}\ Expr\ \texttt{)}\ Stmt \\
&\mid \ \ldots
\end{aligned}
$$

$$
\begin{aligned}
Func &\leftarrow \texttt{int ID ( )}\ Block \\
Prog &\leftarrow Func\ \texttt{EOF}
\end{aligned}
$$

用 EOF 显示标明程序的结尾

通过记忆化搜索来实现

从 pos 位置开始解析，若成功则记录解析结果 result 和结果后的位置 next

Term

```
bool Parser::parse_exp(int pos) {
  if (m_exp.memoized[pos]) return m_exp.result[pos] != nullptr;
  if (parse_term(pos)) {
    int next = m_term.next[pos];
    vector<ParseTree*> children{m_term.result[pos]};
    while (true) {
      if (expect_token(next, T_PLUS) || expect_token(next, T_MINUS)) {
        if (parse_term(next + 1)) {
          children.push_back(new TerminalParseTree(tokens[next]));
          children.push_back(m_term.result[next + 1]);
          next = m_term.next[next + 1];
          continue;
        }
      }
      break;
    }
    return m_exp.set(pos, new NonTerminalParseTree("Exp", std::move(children)),
                     next);
  }
  return m_exp.set_invalid(pos);
}
```

通过循环解析 $((+ \mid -)\,Term)^*$

贪心：尽可能多地解析 $(+ \mid -)\,Term$

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 文法
  - 也可以使用 PEG 来描述
- 自顶向下语法分析可以通过递归下降法进行实现
  - 最左推导，消除左递归，提取左公因子，计算 FIRST、FOLLOW 集合，LL(1) 文法
  - Packrat 分析算法
- **后续可能讲的专题：**
  - **自底向上语法分析**
  - **语法分析的自动生成**

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？
- 可以用递归下降法分析二义性文法吗？有什么问题？
- 尝试构造一个 LL(0) 文法。这种文法有意义吗？
- 有没有什么语言的语法分析的规约是不能用 CFG 表达的？它们的语法分析应该如何进行？