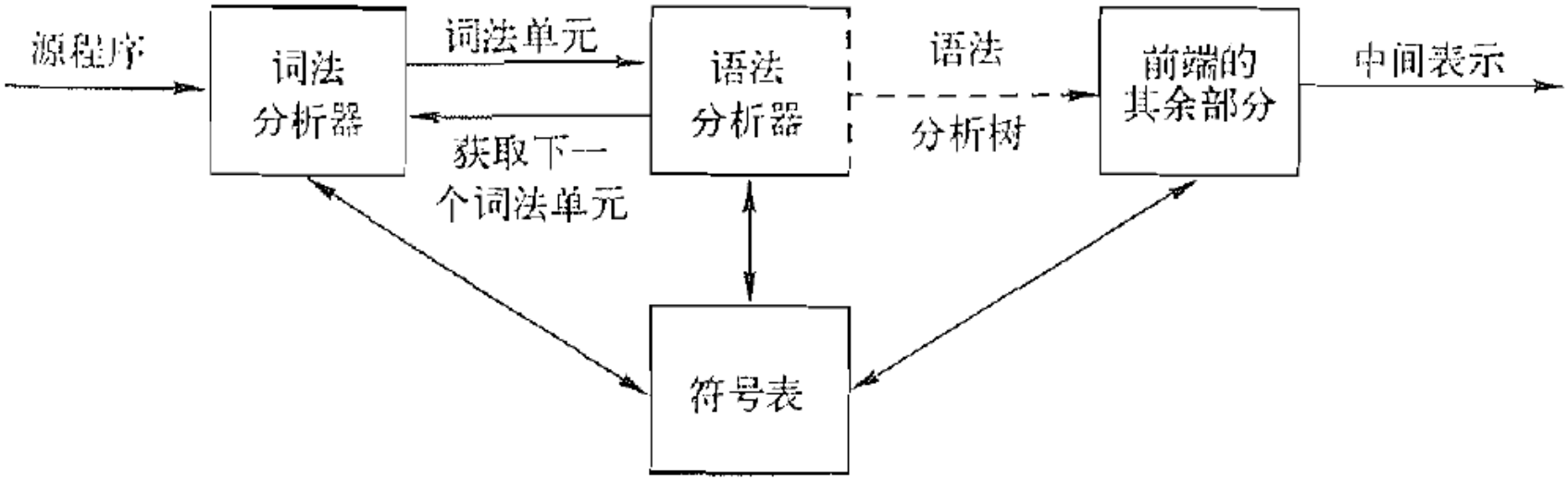
语法分析

魏恒峰

hfwei@nju.edu.cn

2020 年 11 月 25 日

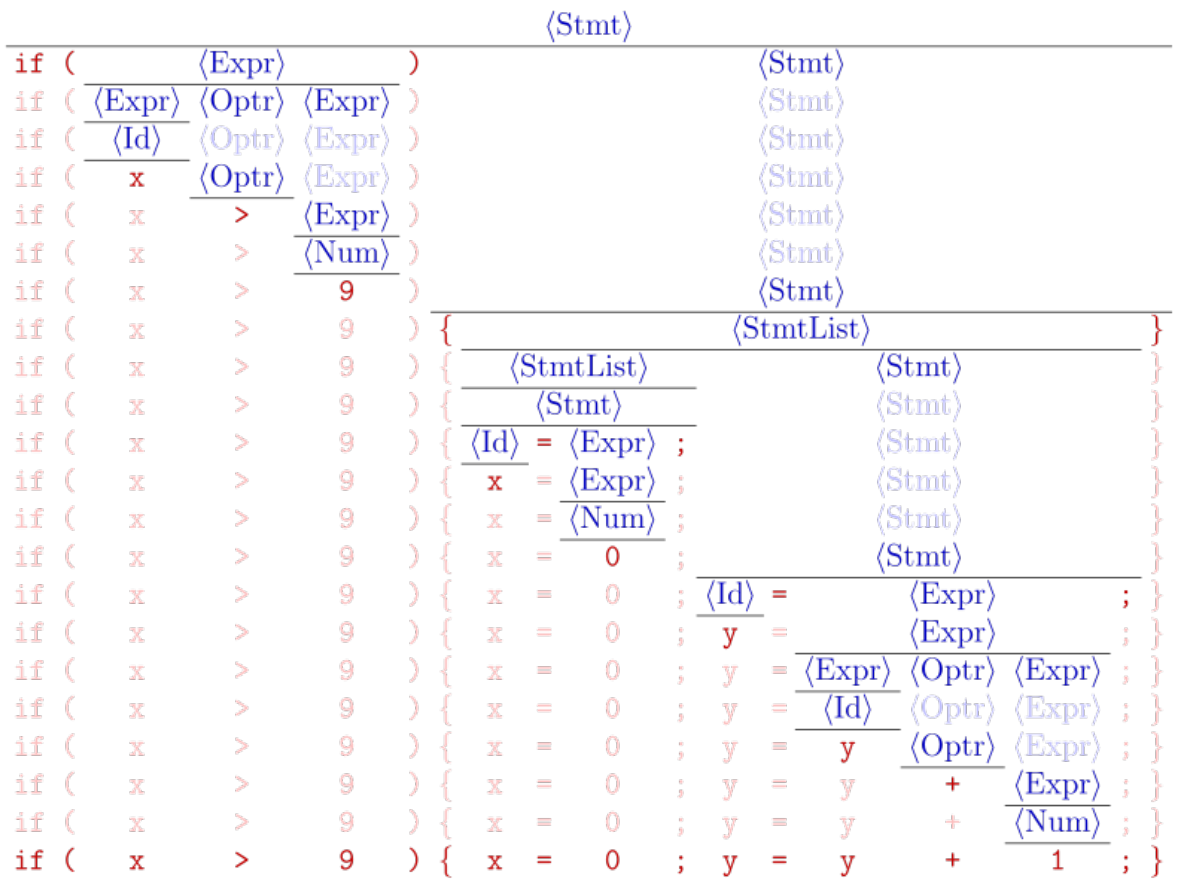
输入: 词法单元流 & **语言的语法规则**



输出: 语法分析树 (Parse Tree)

语法分析**举例**

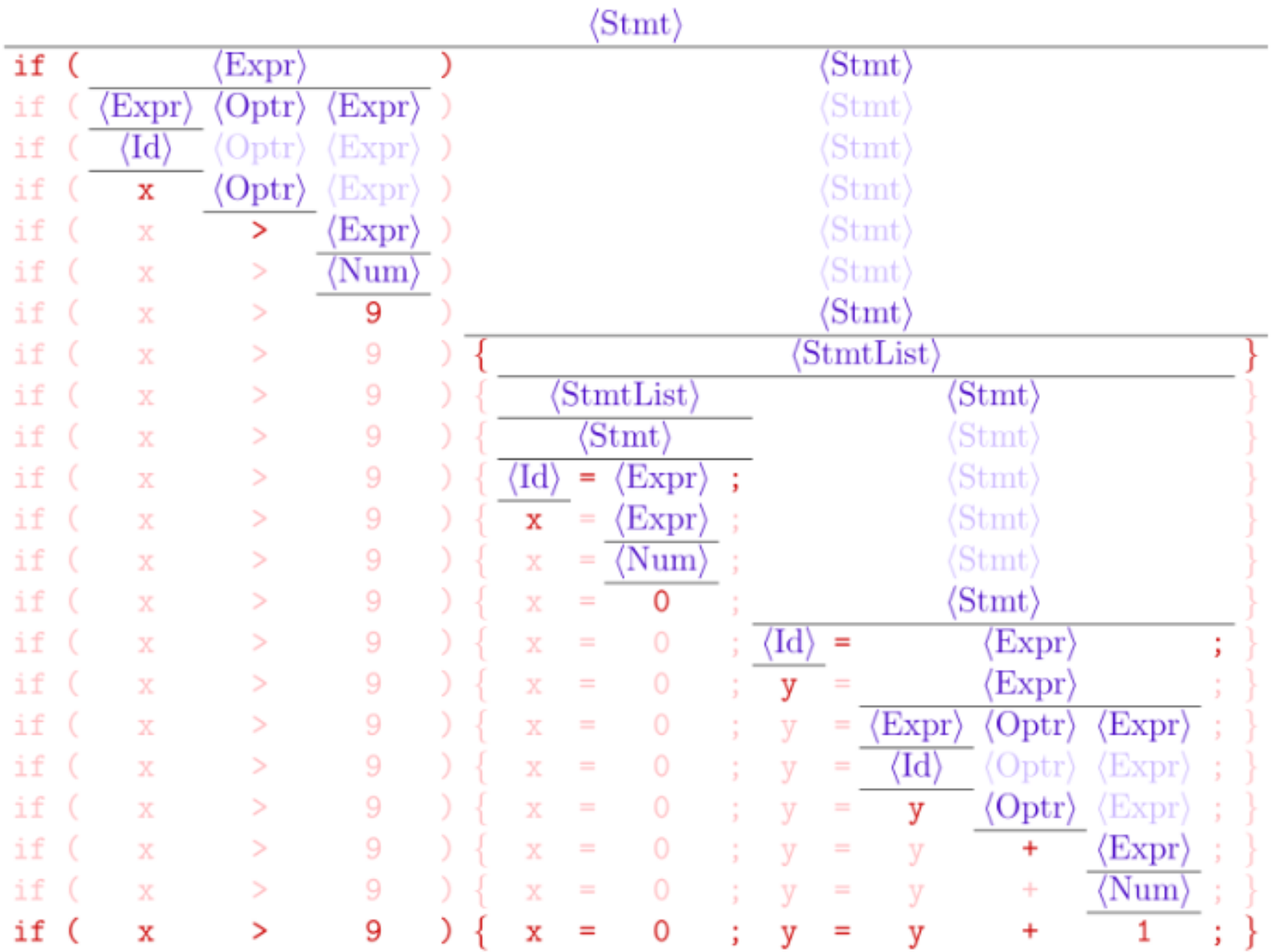
⟨Stmt⟩ → ⟨Id⟩ = ⟨Expr⟩ ;
⟨Stmt⟩ → { ⟨StmtList⟩ }
⟨Stmt⟩ → if (⟨Expr⟩) ⟨Stmt⟩
⟨StmtList⟩ → ⟨Stmt⟩
⟨StmtList⟩ → ⟨StmtList⟩ ⟨Stmt⟩
⟨Expr⟩ → ⟨Id⟩
⟨Expr⟩ → ⟨Num⟩
⟨Expr⟩ → ⟨Expr⟩ ⟨Optr⟩ ⟨Expr⟩
⟨Id⟩ → x
⟨Id⟩ → y
⟨Num⟩ → 0
⟨Num⟩ → 1
⟨Num⟩ → 9
⟨Optr⟩ → >
⟨Optr⟩ → +



语法分析阶段的主题之一：**上下文无关文法**

⟨Stmt⟩ → ⟨Id⟩ = ⟨Expr⟩ ;
⟨Stmt⟩ → { ⟨StmtList⟩ }
⟨Stmt⟩ → if (⟨Expr⟩) ⟨Stmt⟩
⟨StmtList⟩ → ⟨Stmt⟩
⟨StmtList⟩ → ⟨StmtList⟩ ⟨Stmt⟩
⟨Expr⟩ → ⟨Id⟩
⟨Expr⟩ → ⟨Num⟩
⟨Expr⟩ → ⟨Expr⟩ ⟨Optr⟩ ⟨Expr⟩
⟨Id⟩ → x
⟨Id⟩ → y
⟨Num⟩ → 0
⟨Num⟩ → 1
⟨Num⟩ → 9
⟨Optr⟩ → >
⟨Optr⟩ → +

语法分析阶段的主题之二: **构建语法分析树**

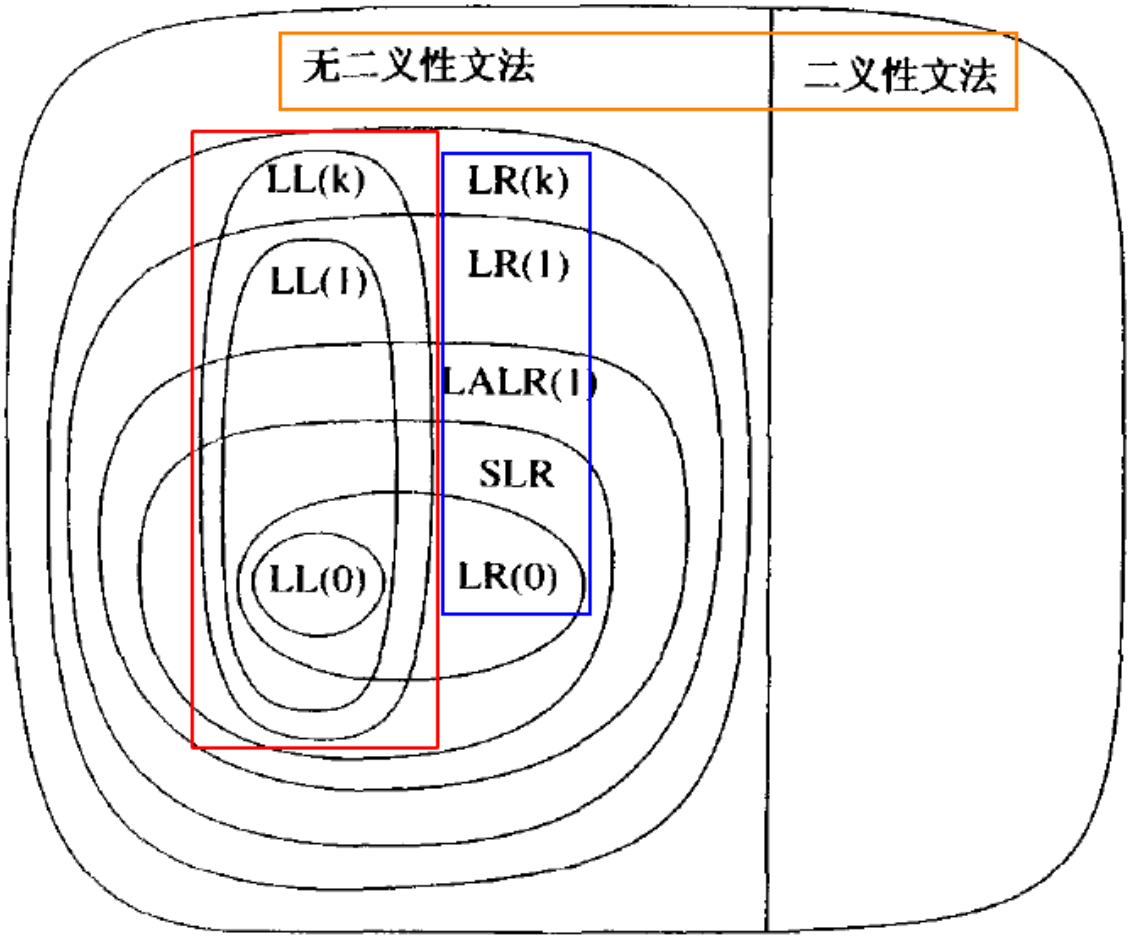


语法分析阶段的主题之三: **错误恢复**

报错、**恢复**、继续分析

只考虑**无二义性**的文法

这意味着, 每个句子对应唯一的一棵语法分析树



今日份主题: $LL(1)$ **语法分析器**

自顶向下的、

递归下降的、

预测分析的、

适用于$LL(1)$ **文法**的、

$LL(1)$ 语法分析器

自顶向下构建语法分析树

根节点是文法的起始符号 S

叶节点是词法单元流 w

仅包含终结符号与特殊的**文件结束符** \$

自顶向下构建语法分析树

根节点是文法的起始符号 S

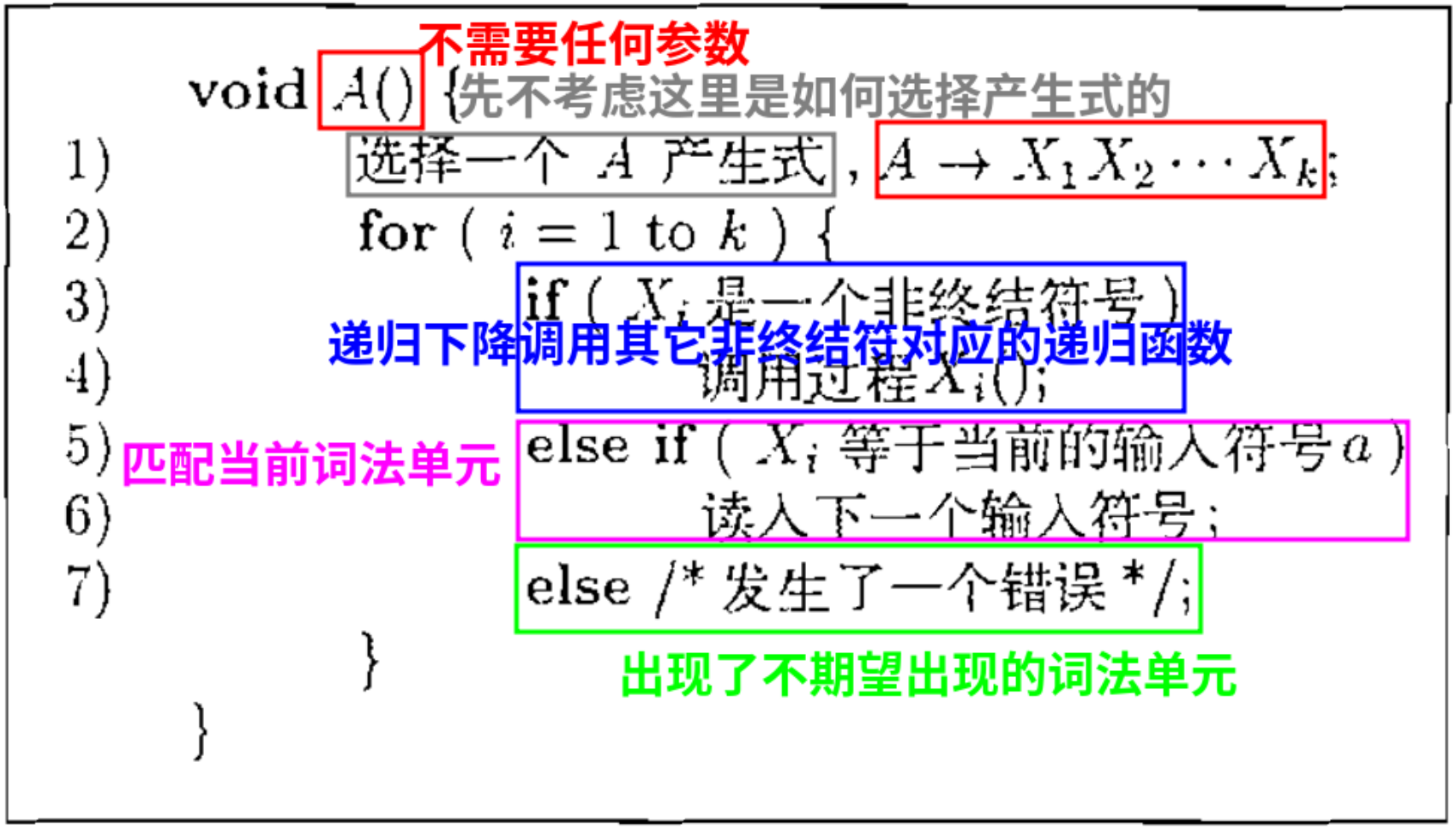
每个**中间节点**表示**对某个非终结符应用某个产生式进行推导**

(Q：选择哪个非终结符, 以及选择哪个产生式)

叶节点是词法单元流 w

仅包含终结符号与特殊的**文件结束符** \$

递归下降的实现框架



为每个**非终结符**写一个**递归函数**

内部按需调用其它非终结符对应的递归函数

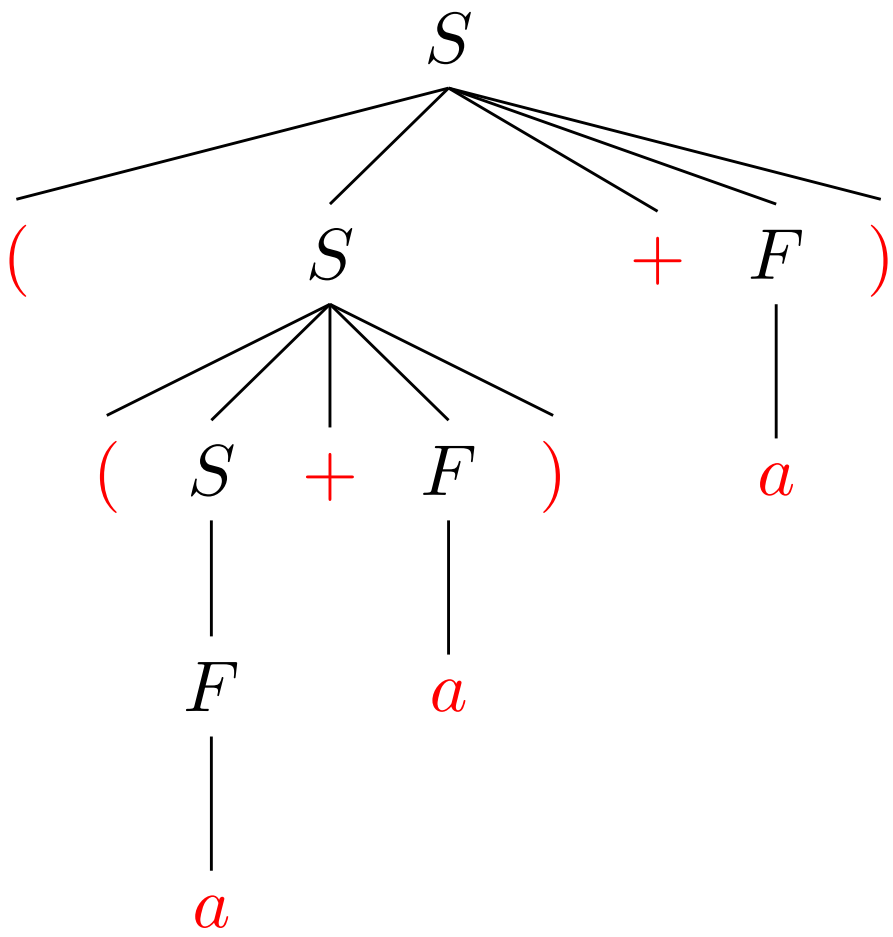
$$S \to F$$

$$S \to (S + F)$$

$$F \to a$$

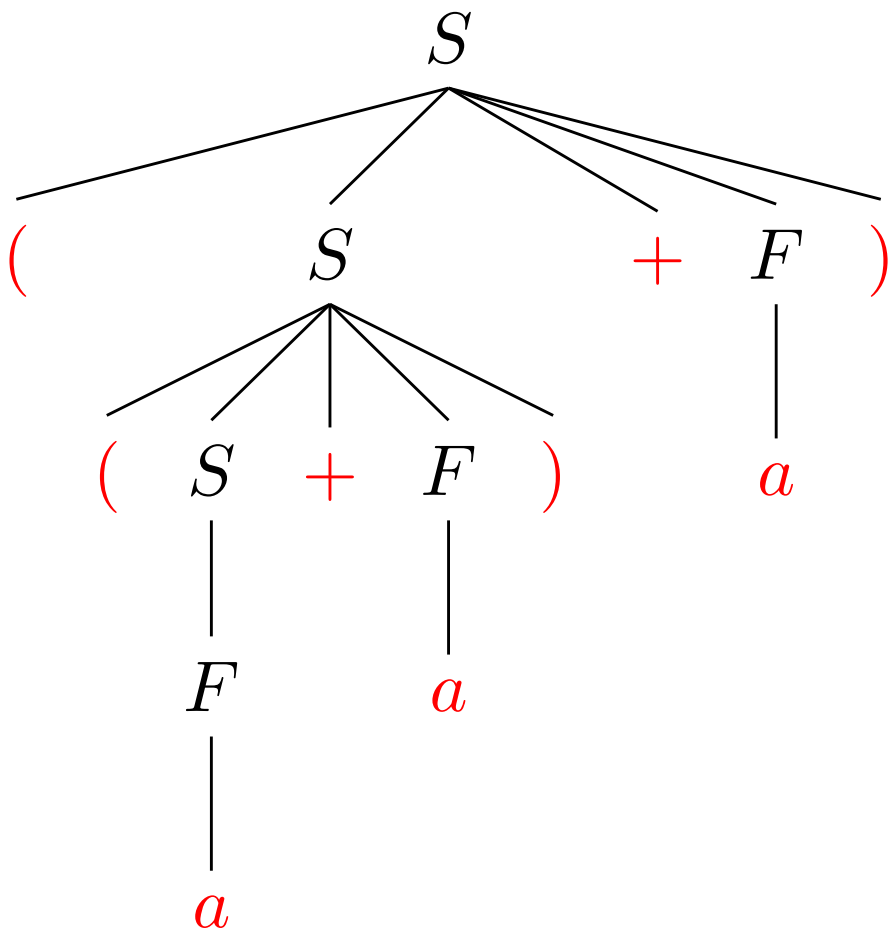
$$w = ((a + a) + a)$$

板书演示**递归下降**过程



$$w = ((a + a) + a)$$

板书演示**递归下降**过程



$$w = ((a + a) + a)$$

每次都选择语法分析树**最左边**的非终结符进行展开

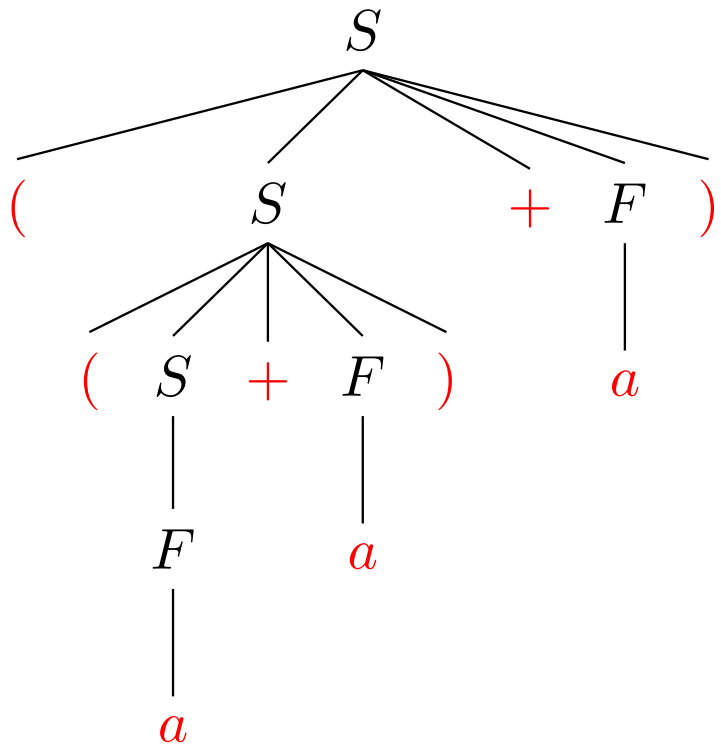
同样是展开非终结符 S,

为什么前两次选择了 $S \to (S + F)$, 而第三次选择了 $S \to F$?

$$S \to F$$

$$S \to (S + F)$$

$$F \to a$$



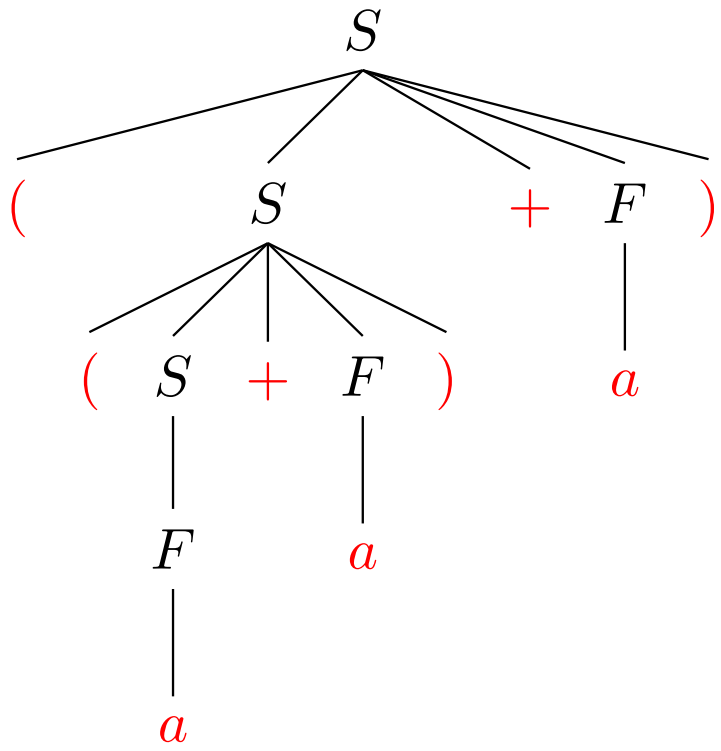
同样是展开非终结符 S,

为什么前两次选择了 $S \to (S + F)$, 而第三次选择了 $S \to F$?

$$S \to F$$

$$S \to (S + F)$$

$$F \to a$$



因为它们面对的**当前词法单元**不同

使用**预测分析表**确定产生式

$$S \to F$$

$$S \to (S + F)$$

$$F \to a$$

	(	)	a	+	\$
S	2		1		
F			3		

指明了每个**非终结符**在面对不同的**词法单元或文件结束符**时，

该选择哪个**产生式** (按编号进行索引) 或者**报错**

Definition ($LL(1)$ 文法)

如果文法 G 的**预测分析表**是**无冲突**的, 则 G 是 $LL(1)$ 文法。

无冲突: 每个单元格里只有一个生成式 (编号)

$$S \to F$$

$$S \to (S + F)$$

$$F \to a$$

	(	)	a	+	\$
S	2		1		
F			3		

对于当前选择的非终结符,

仅根据输入中当前的词法单元即可确定需要使用哪条产生式

递归下降的、预测分析实现方法

$$S \to F$$

$$S \to (S + F)$$

$$F \to a$$

	(	)	a	+	$
S	2		1		
F			3		

```
1: procedure MATCH(t)
2:     if token = t then
3:         token ← NEXT-TOKEN()
4:     else
5:         ERROR(token, t)
```

```
1: procedure S()
2:     if token = '(' then
3:         MATCH('(')
4:         S()
5:         MATCH('+')
6:         F()
7:         MATCH(')')
8:     else if token = 'a' then
9:         F()
10:    else
11:        ERROR(token, {'(', 'a'})
```

递归下降的、预测分析实现方法

$$S \to F$$

$$S \to (S + F)$$

$$F \to a$$

	(	)	a	+	$
S	2		1		
F			3		

```
1: procedure MATCH(t)
2:     if token = t then
3:         token ← NEXT-TOKEN()
4:     else
5:         ERROR(token, t)
```

```
1: procedure F()
2:     if token = 'a' then
3:         MATCH('a')
4:     else
5:         ERROR(token, {'a'})
```

如何计算给定文法 G 的预测分析表?

$\textsc{First}(\alpha)$ 是可从 α 推导得到的句型的**首终结符号**的集合

Definition ($\textsc{First}(\alpha)$ 集合)

对于任意的 (产生式的右部) $\alpha \in (N \cup T)^*$:

$$\textsc{First}(\alpha) = \left\{ t \in T \cup \{\epsilon\} \mid \alpha \overset{*}{\Rightarrow} t\beta \vee \alpha \overset{*}{\Rightarrow} \epsilon \right\}.$$

如何计算给定文法 G 的预测分析表?

$\textsc{First}(\alpha)$ 是可从 α 推导得到的句型的**首终结符号**的集合

Definition ($\textsc{First}(\alpha)$ 集合)

对于任意的 (产生式的右部) $\alpha \in (N \cup T)^*$:

$$\textsc{First}(\alpha) = \left\{ t \in T \cup \{\epsilon\} \mid \alpha \overset{*}{\Rightarrow} t\beta \vee \alpha \overset{*}{\Rightarrow} \epsilon \right\}.$$

考虑非终结符 A 的所有产生式 $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_m$,

如果它们对应的 $\textsc{First}(\alpha_i)$ 集合互不相交,

则只需查看当前输入词法单元, 即可确定选择哪个产生式 (或报错)

如何计算给定文法 G 的预测分析表?

FOLLOW(A) 是可能在某些句型中**紧跟在 A 右边的终结符**的集合

Definition (FOLLOW(A) 集合)

对于任意的 (产生式的左部) 非终结符 $A \in N$:

$$\text{FOLLOW}(A) = \{t \in T \cup \{\$\} \mid \exists w.\ S \overset{*}{\Rightarrow} w = \beta A t \gamma\}.$$

如何计算给定文法 G 的预测分析表?

$\textsc{Follow}(A)$ 是可能在某些句型中**紧跟在 A 右边的终结符**的集合

Definition ($\textsc{Follow}(A)$ 集合)

对于任意的 (产生式的左部) 非终结符 $A \in N$:

$$\textsc{Follow}(A) = \{t \in T \cup \{\$\} \mid \exists w.\ S \overset{*}{\Rightarrow} w = \beta A t \gamma\}.$$

考虑产生式 $A \to \alpha$,

如果从 α 可能推导出空串 ($\alpha \overset{*}{\Rightarrow} \epsilon$),

则只有当当前词法单元 $t \in \textsc{Follow}(A)$, 才可以选择该产生式

先计算每个符号 X 的 $\textsc{First}(X)$ 集合

```
1: procedure FIRST(X)
2:     if X ∈ T then                                  ▷ 规则 1: X 是终结符
3:         First(X) = X
4:     for X → Y_1 Y_2 ... Y_k do                     ▷ 规则 2: X 是非终结符
5:         First(X) ← First(X) ∪ First(Y_1)
6:         for i ← 2 to k do
7:             if ε ∈ L(Y_1 ... Y_{i-1}) then
8:                 First(X) ← First(X) ∪ First(Y_i)
9:         if ε ∈ L(Y_1 ... Y_k) then                 ▷ 规则 3: X 可推导出空串
10:            First(X) ← First(X) ∪ {ε}
```

不断应用上面的规则, 直到每个 $\textsc{First}(X)$ 都不再变化 (**闭包!!!**)

再计算每个符号串 α 的 $\textsc{First}(\alpha)$ 集合

$$\alpha = X\beta$$

$$\textsc{First}(\alpha) = \begin{cases} \textsc{First}(X) & \epsilon \in L(X) \\ \textsc{First}(X) \cup \textsc{First}(\beta) & \epsilon \notin L(X) \end{cases}$$

$$X \to Y$$

$$X \to a$$

$$Y \to \epsilon$$

$$Y \to c$$

$$Z \to d$$

$$Z \to XYZ$$

$$X \to Y$$

$$X \to a$$

$$Y \to \epsilon$$

$$Y \to c$$

$$Z \to d$$

$$Z \to XYZ$$

$$\textsc{First}(X) = \{a, c, \epsilon\}$$
$$\textsc{First}(Y) = \{c, \epsilon\}$$
$$\textsc{First}(Z) = \{a, c, d\}$$

$$\textsc{First}(XYZ) = \textsc{First}(X) = \{a, c\}$$

为每个非终结符 X 计算 $\textsc{Follow}(X)$ 集合

1: **procedure** $\textsc{follow}(X)$
2: **for** X 是开始符号 **do** ▷ 规则 1: X 是开始符号
3: $\textsc{Follow}(X) \leftarrow \textsc{Follow}(X) \cup \{\$\}$
4: **for** $A \to \alpha X \beta$ **do** ▷ 规则 2: X 是某产生式右部中间的一个符号
5: $\textsc{Follow}(X) \leftarrow \textsc{Follow}(X) \cup (\textsc{First}(\beta) \setminus \{\epsilon\})$
6: **if** $\epsilon \in \textsc{First}(\beta)$ **then**
7: $\textsc{Follow}(X) \leftarrow \textsc{Follow}(X) \cup \textsc{Follow}(A)$
8: **for** $A \to \alpha X$ **do** ▷ 规则 3: X 是某产生式右部的最后一个符号
9: $\textsc{Follow}(X) \leftarrow \textsc{Follow}(X) \cup \textsc{Follow}(A)$

不断应用上面的规则, 直到每个 $\textsc{Follow}(X)$ 都不再变化 (**闭包!!!**)

$$X \to Y$$

$$X \to a$$

$$Y \to \epsilon$$

$$Y \to c$$

$$Z \to d$$

$$Z \to XYZ$$

$$X \to Y$$

$$X \to a$$

$$Y \to \epsilon$$

$$Y \to c$$

$$Z \to d$$

$$Z \to XYZ$$

$$\textsc{Follow}(X) = \{c, \$\}$$
$$\textsc{Follow}(Y) = \{a, c, d, \$\}$$
$$\textsc{Follow}(Z) = \emptyset$$

如何根据First 与 Follow 集合计算给定文法 G 的预测分析表?

按照以下规则, 在表格 $[A, t]$ 中填入生成式 $A \to \alpha$ (编号):

$$t \in \textsc{First}(\alpha) \tag{1}$$

$$\alpha \stackrel{*}{\Rightarrow} \epsilon \wedge t \in \textsc{Follow}(A) \tag{2}$$

如何根据First 与 Follow 集合计算给定文法 G 的预测分析表?

按照以下规则, 在表格 $[A, t]$ 中填入生成式 $A \to \alpha$ (编号):

$$t \in \textsc{First}(\alpha) \tag{1}$$

$$\alpha \stackrel{*}{\Rightarrow} \epsilon \wedge t \in \textsc{Follow}(A) \tag{2}$$

Definition ($LL(1)$ 文法)

如果文法 G 的**预测分析表**是**无冲突**的, 则 G 是 $LL(1)$ 文法。

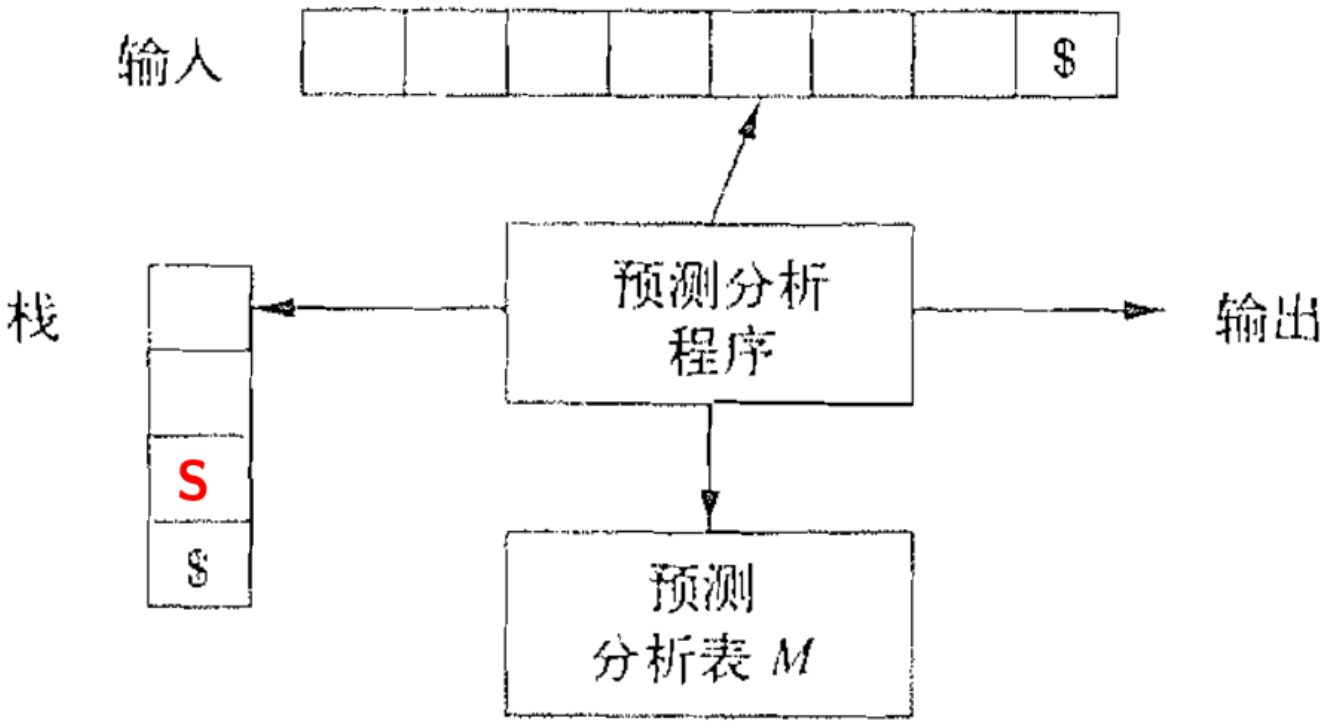
$LL(1)$ 语法分析器

L：从左向右 (left-to-right) 扫描输入

L：构建最左 (leftmost) 推导

1：只需向前看一个输入符号便可确定使用哪条产生式

非递归的预测分析算法



非递归的预测分析算法

设置 ip 使它指向 w 的第一个符号，其中 ip 是输入指针;
令 X = 栈顶符号;
while ($X \neq \$$) { /* 栈非空 */
    if (X 等于 ip 所指向的符号 a) 执行栈的弹出操作，将 ip 向前移动一个位置;
    else if (X 是一个终结符号) $error()$;
    else if ($M[X, a]$ 是一个报错条目) $error()$;
    else if ($M[X, a] = X \to Y_1 Y_2 \cdots Y_k$) {
        输出产生式 $X \to Y_1 Y_2 \cdots Y_k$;
        弹出栈顶符号;
        将 $Y_k, Y_{k-1}, \ldots, Y_1$ 压入栈中，其中 Y_1 位于栈顶。
    }
    令 X = 栈顶符号;
}

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$S \rightarrow \text{begin } S\ L$

$S \rightarrow \text{print } E$

$L \rightarrow \text{end}$

$L \rightarrow ;\ S\ L$

$E \rightarrow \text{num} = \text{num}$

顺序语句、条件语句、打印语句

$S \rightarrow$ if E then S else S

$S \rightarrow$ begin S L

$S \rightarrow$ print E

$L \rightarrow$ end

$L \rightarrow$; S L

$E \rightarrow$ num = num

每个产生式都以一个终结符开头, 且这些终结符各不相同

因此, 仅根据输入中的**当前词法单元**, 就可以确定应该使用哪条产生式

$S \rightarrow$ if E then S else S

$S \rightarrow$ begin S L

$S \rightarrow$ print E

$L \rightarrow$ end

$L \rightarrow$; S L

$E \rightarrow$ num = num

```
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};
extern enum token getToken(void);
```

getToken: 语法分析器**按需**向词法分析器索要下一个词法单元

$S \rightarrow$ if E then S else S

$S \rightarrow$ begin S L

$S \rightarrow$ print E

$L \rightarrow$ end

$L \rightarrow$; S L

$E \rightarrow$ num = num

```
enum token tok;
void advance() {tok=getToken();}
void eat(enum token t) {if (tok==t) advance(); else error();}
```

$eat(t)$: 根据当前的产生式, **预期**的词法单元应该是 t

匹配 t, 继续试图匹配下一个词法单元; 否则, 报错

$S \rightarrow$ if E then S else S

$S \rightarrow$ begin S L

$S \rightarrow$ print E

$L \rightarrow$ end

$L \rightarrow$; S L

$E \rightarrow$ num = num

```
void S(void) {switch(tok) {
        case IF:    eat(IF); E(); eat(THEN); S();
                                  eat(ELSE); S(); break;
        case BEGIN: eat(BEGIN); S(); L(); break;
        case PRINT: eat(PRINT); E(); break;
        default:    error();
      }}
void L(void) {switch(tok) {
        case END:   eat(END); break;
        case SEMI:  eat(SEMI); S(); L(); break;
        default:    error();
      }}
void E(void) {  eat(NUM); eat(EQ); eat(NUM); }
```

为每个**非终结符**写一个**递归函数**

对于每个**产生式**, 写一个 case **分支语句**

板书演示这个语法分析器的工作过程

```
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};
extern enum token getToken(void);

enum token tok;
void advance() {tok=getToken();}
void eat(enum token t) {if (tok==t) advance(); else error();}

void S(void) {switch(tok) {
        case IF:    eat(IF); E(); eat(THEN); S();
                                  eat(ELSE); S(); break;
        case BEGIN: eat(BEGIN); S(); L(); break;
        case PRINT: eat(PRINT); E(); break;
        default:    error();
       }}
void L(void) {switch(tok) {
        case END:   eat(END); break;
        case SEMI:  eat(SEMI); S(); L(); break;
        default:    error();
       }}
void E(void) {  eat(NUM); eat(EQ); eat(NUM); }
```

$$E \to E + T \mid E - T \mid T$$

$$T \to T * F \mid T/F \mid F$$

$$F \to (E) \mid \textbf{id} \mid \textbf{num}$$

```
void E(void) {switch (tok) {
              case ?: E(); eat(PLUS); T(); break;
              case ?: E(); eat(MINUS); T(); break;
              case ?: T(); break;
              default: error();
             }}
```

E 在**不消耗任何词法单元**的情况下, 直接递归调用 E, 造成**死循环**

$$E \to E + T \mid E - T \mid T$$

$$T \to T * F \mid T/F \mid F$$

$$F \to (E) \mid \mathbf{id} \mid \mathbf{num}$$

```
void E(void) {switch (tok) {
              case ?: E(); eat(PLUS); T(); break;
              case ?: E(); eat(MINUS); T(); break;
              case ?: T(); break;
              default: error();
             }}
```

E 在**不消耗任何词法单元**的情况下, 直接递归调用 E, 造成**死循环**

$$E \to E + T \mid E - T \mid T$$

$$T \to T * F \mid T/F \mid F$$

$$F \to (E) \mid \mathbf{id} \mid \mathbf{num}$$

```
void E(void) {switch (tok) {
              case ?: E(); eat(PLUS); T(); break;
              case ?: E(); eat(MINUS); T(); break;
              case ?: T(); break;
              default: error();
             }}
```

更重要的是, E 与 T 的产生式所能生成的句子可能**以相同的终结符开头**

因此, **无法仅根据输入中当前的词法单元确定要使用的生成式**

消除左递归

$$E \to E + T \mid T$$

消除左递归

$$E \to E + T \mid T$$

$$E \to TE'$$

$$E' \to +TE' \mid \epsilon$$

将左递归转为**右递归**

$$E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$$

$$E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$$

$$\begin{aligned}
E &\to TE' \\
E' &\to +\, TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *\, FT' \mid \epsilon \\
F &\to (E) \mid \mathbf{id} \mid \mathbf{num}
\end{aligned}$$

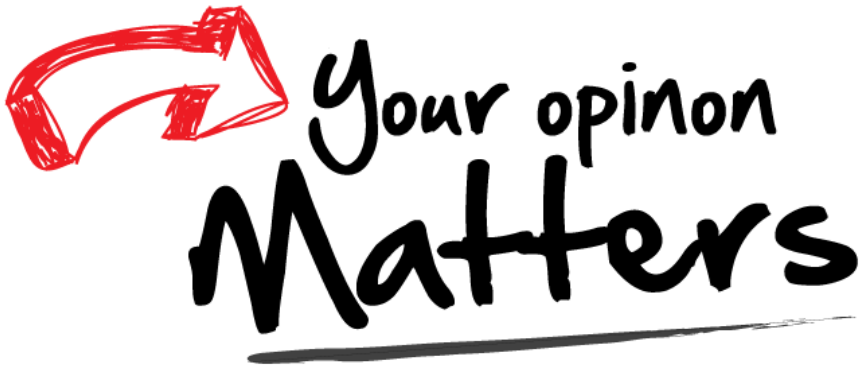
$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \beta_n$$

其中, β_i 都不以 A 开头

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$


Thank
You!



Office 926

hfwei@nju.edu.cn