

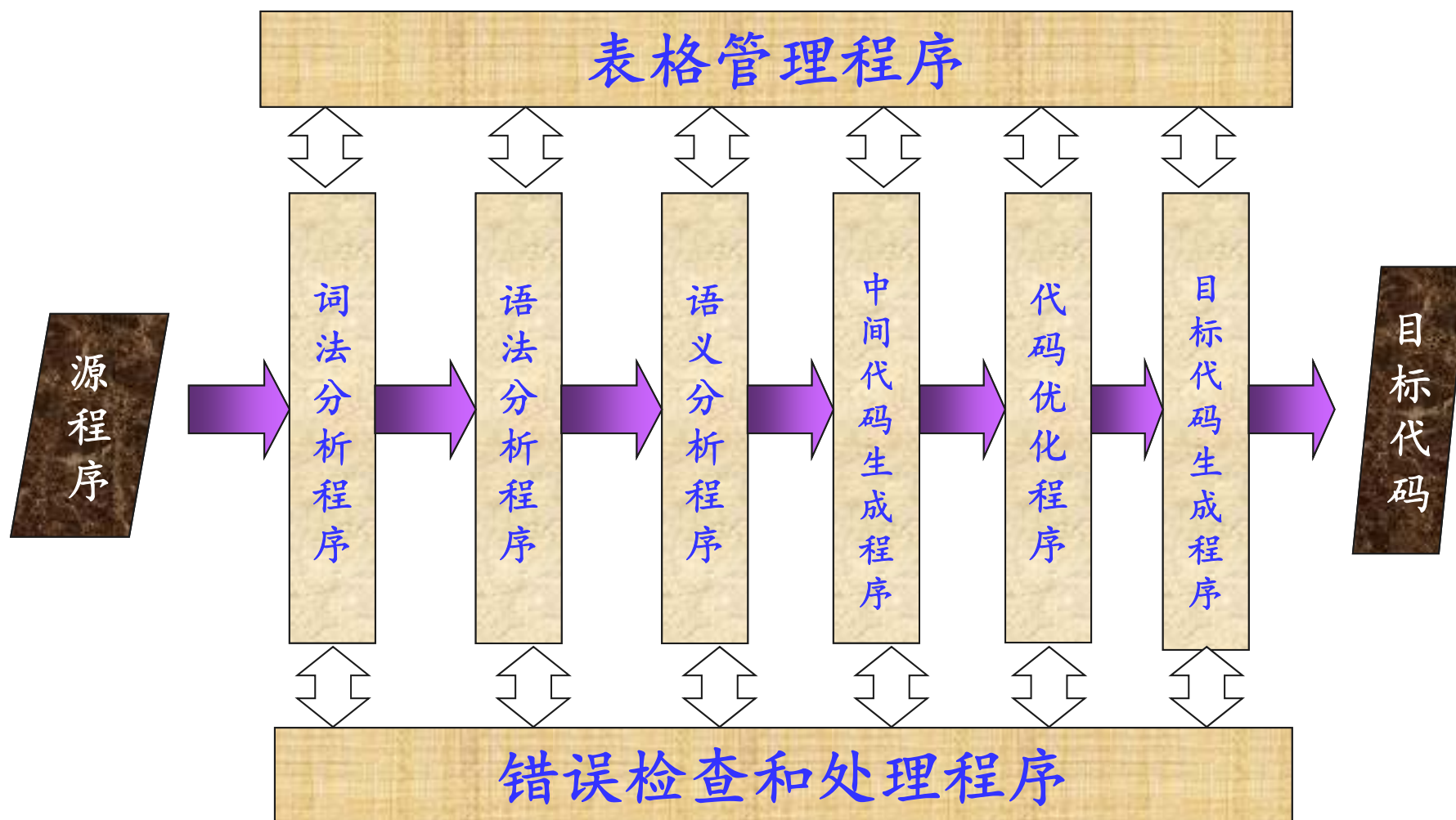


# 编译原理

---

武汉大学计算机学院  
编译原理课程组

# 编译程序的结构



属性字: <类别号, 自身值>  
class value

# 符号表

```
position := initial + rate * 60
```

# Scanner

[illegible]

$\langle id, 1 \rangle \langle := \rangle \langle id, 2 \rangle \langle + \rangle \langle id, 3 \rangle \langle * \rangle \langle number, 4 \rangle$

# 编译过程——语法分析

例: **position := initial + rate \* 60**

显示程序语法

内部结点: 结构名

叶子结点: 单词

语法规则:

$\langle \text{赋值语句} \rangle ::= \langle \text{标识符} \rangle " := " \langle \text{表达式} \rangle$

$\langle \text{表达式} \rangle ::= \langle \text{表达式} \rangle "+" \langle \text{表达式} \rangle$

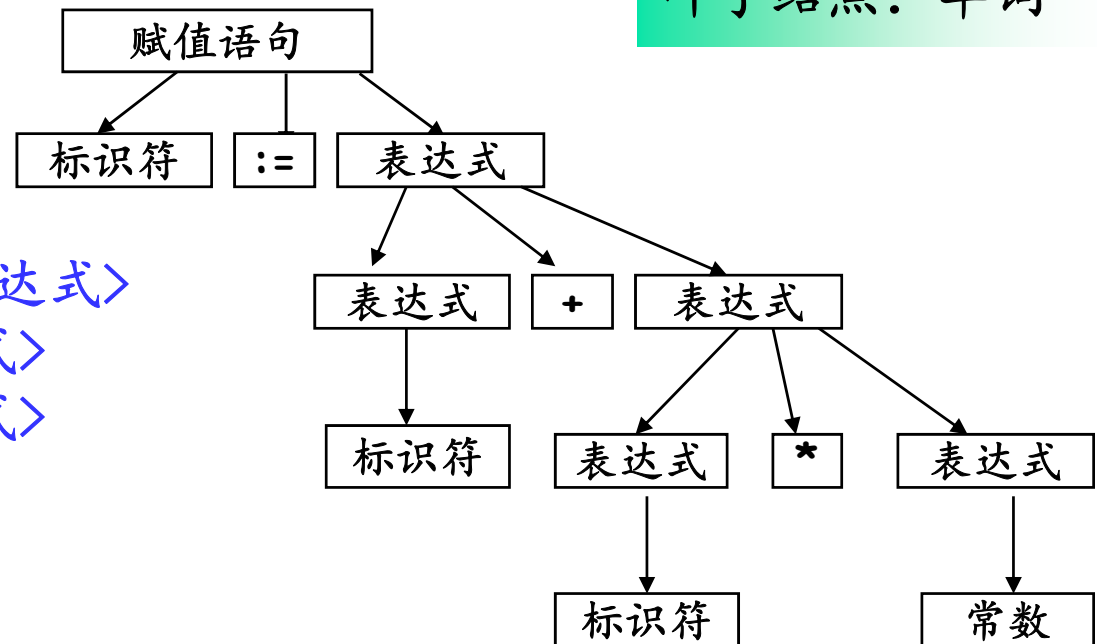
$\langle \text{表达式} \rangle ::= \langle \text{表达式} \rangle "*" \langle \text{表达式} \rangle$

$\langle \text{表达式} \rangle ::= "(" \langle \text{表达式} \rangle ")"$

$\langle \text{表达式} \rangle ::= \langle \text{标识符} \rangle$

$\langle \text{表达式} \rangle ::= \langle \text{整数} \rangle$

$\langle \text{表达式} \rangle ::= \langle \text{实数} \rangle$

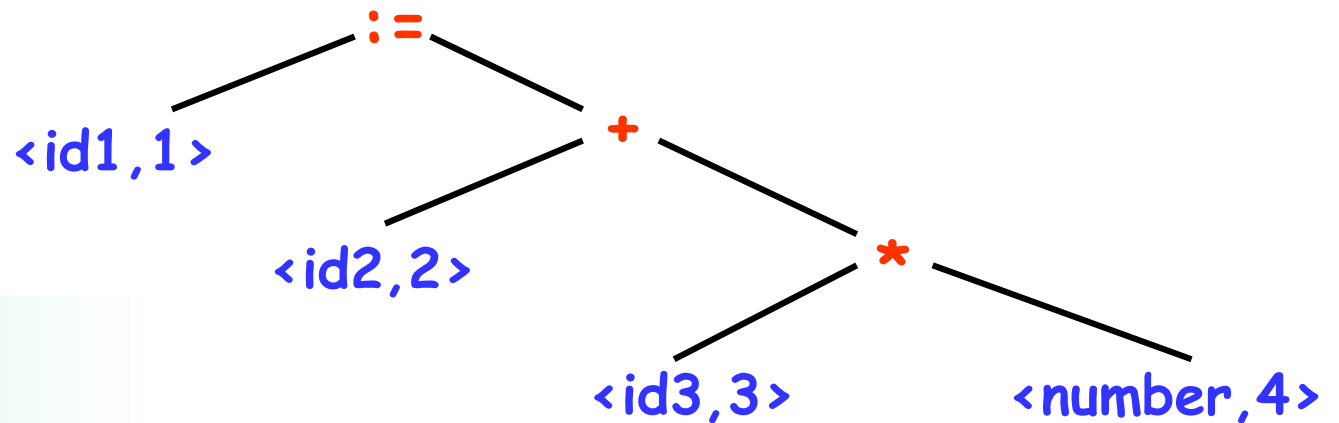


分析树 (Parse tree)

# 语法分析

<id,1> <:=> <id,2> <+> <id,3> <\*> <number,4>

Parser



层次结构分析

内部结点：运算

其子结点：该运算的分量

抽象语法树 (Abstract Syntax Tree)



## 第5章 自上而下语法分析

---

- |   |         |
|---|---------|
| 1 | 基本思想    |
| 2 | 存在的问题   |
| 3 | 解决方法    |
| 4 | LL(1)方法 |
| 5 | 递归子程序法  |





## 5.0 语法分析的功能及基本思想

---

### 1. 语法分析的功能

依据语法规则，逐一分析词法分析时得到的单词串，把单词串分解成各类语法单位，即确定它们是怎样组成说明和语句，以及说明和语句又是怎样组成程序的。分析时如发现有不合语法规则的地方，便将出错的位置及出错性质打印报告给程序员；如无语法错误，则用另一种中间形式给出正确的语法结构，供下一阶段分析使用。



## 5.0 语法分析的功能及基本思想

### 2. 自上而下语法分析的基本思想

从**推导**的角度看，从识别符号出发，不断建立直接推导，试图构造一个**最左推导**序列，最终**推导出与输入符号串相同的符号串**。

从**语法树**的角度看，以识别符号为根结点，试图向下构造一棵**语法树**，其**末端结点符号串**正好与输入符号串相同。

相应于高级语言的编译过程，自上而下语法分析就是从该高级语言文法的开始符号——**<程序>**出发，试图推导得到该文法的**句子**——**源程序或与其等价的单词串**。





## 5.0 语法分析的功能及基本思想

### 3. 自上而下语法分析遇到的问题

【例】  $G_{16}[S]: S \rightarrow aAbc|aB, A \rightarrow ba, B \rightarrow beB|d$

待检查串  $abed$

- 带回溯的自上而下分析方法（不确定的自上而下分析方法）

- 确定的自上而下分析方法

限制文法：若某非终结符有多个候选式，当候选式首符号是终结符时，应保证它们互不相同。

【例】  $G_{17}[S]: S \rightarrow aBc|bCd, B \rightarrow eB|f, C \rightarrow dC|c$

待检查串  $ae fc$



## 5.0 语法分析的功能及基本思想

### 3. 自上而下语法分析遇到的问题

如果文法中存在如下形式的产生式

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

那么在自上而下的语法分析过程中，当要对A展开时，应按哪一个候选式展开呢？即如何确定替换A的 $\alpha_i$ 。如果选择错误，将导致回溯。

在分析的过程中，匹配失败后，必须退回到出错点，选择其它可能的产生式重新推导，这个过程称为回溯。



## 5.0 语法分析的功能及基本思想

### 3. 自上而下语法分析遇到的问题

当文法中出现左递归时（存在非终结符号U，对于它有  $U \rightarrow U\dots$  或  $U \xRightarrow{+} U\dots$ ），会使分析过程陷入无限循环。

例如对文法G[S]:

$S \rightarrow AB$

$A \rightarrow Aa \mid bB$

$B \rightarrow Sb \mid a$

分析符号串baaa



## 5.0 语法分析的功能及基本思想

---

### 4. 自上而下语法分析中问题的解决方法

- 避免回溯

- 消除左递归

## 5.1 消除左递归的方法

### 1. 直接左递归的消除

- 采用EBNF表示

$[x]$ —— $x$ 可以出现零次或一次

$\{x\}$ —— $x$ 可以出现零次到多次

$x(y|z)$ ——等价于  $xy$  或  $xz$

消除直接左递归：

设有产生式	$U \rightarrow Ux_1   Ux_2   \dots   Ux_m   y_1   y_2   \dots   y_n$
变换为	$U \rightarrow (y_1   y_2   \dots   y_n) \{ x_1   x_2   \dots   x_m \}$



## 5.1 消除左递归的方法

### 1. 直接左递归的消除

- 直接改写法

引进新的非终结符号，将左递归改写为右递归。

设有产生式  $U \rightarrow Ux_1 | Ux_2 | \dots | Ux_m | y_1 | y_2 | \dots | y_n$

其中  $y_i (i=1, 2, \dots, n)$  均不以符号  $U$  为首， $x_1, \dots, x_m$  均不为  $\varepsilon$ ，

增加新非终结符号  $U'$ ，将上述产生式变换为

$$U \rightarrow y_1 U' | y_2 U' | \dots | y_n U'$$

$$U' \rightarrow x_1 U' | x_2 U' | \dots | x_m U' | \varepsilon$$

## 5.1 消除左递归的方法

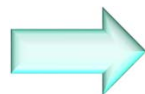
直接左递归的消除

文法**G**[**E**]:

**E** → **E** + **T** | **T**

**T** → **T** \* **F** | **F**

**F** → (**E**) | **i**



文法**G'** [**E**]:

**E** → **T****E'**

**E'** → +**T****E'** | ε

**T** → **F****T'**

**T'** → \***F****T'** | ε

**F** → (**E**) | **i**

## 5.1 消除左递归的方法

### 2. 间接左递归的消除 $U \xRightarrow{+} U \dots$

文法G[S]:

$S \rightarrow Qc \mid c$

$Q \rightarrow Rb \mid b$

$R \rightarrow Sa \mid a$

间接左递归



直接左递归



## 5.1 消除左递归的方法

2. 间接左递归的消除  $U \xRightarrow{+} U \dots$

(1) 将所有非终结符排序:  $U_1, U_2, \dots, U_n$

(2) 执行循环语句:

FOR  $i:=1$  TO  $n$  DO

BEGIN

① FOR  $j:=1$  TO  $i-1$  DO

将产生式规则  $U_i \rightarrow U_j \gamma$  改写

{改写方法: 若  $U_j \rightarrow \delta_1 | \delta_2 | \dots | \delta_k$ ,

则  $U_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \dots | \delta_k \gamma$ };

② 消除  $U_i$  产生式规则中的直接左递归

END

(3) 化简由(2)所得文法, 消去多余产生式。



## 5.1 消除左递归的方法

---

### 2. 间接左递归的消除

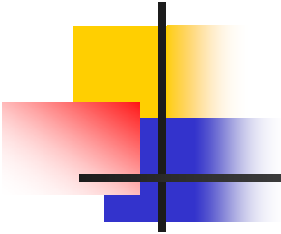
例：设有文法G[S]：

$$S \rightarrow Qc \mid c$$

$$Q \rightarrow Rb \mid b$$

$$R \rightarrow Sa \mid a$$

试消除其左递归。

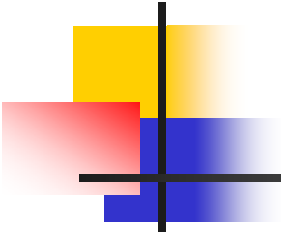


## 5.2 LL(1) 文法

---

### 1. 回溯现象的危害

处理回溯是一个复杂的过程，其中包括恢复指针、删去已匹配的子树及一些语义处理等工作，并且难以确定出错的确切位置，导致效率很低，代价极高。



## 5.2 LL(1) 文法

### 2. 避免回溯的条件

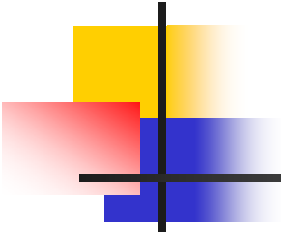
在进行自上而下语法分析时，对于如下形式的产生式

$$U \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

看当前输入符号是什么，**哪一个** $\alpha_i$ **推导出的首终结符号和当前输入符号相同**，就选**那个** $\alpha_i$ 来替换U进行推导。

对 $\forall U \in V_N$ :  $\forall i \neq j, i, j = 1, 2, \dots, n$

$$\text{SELECT}(U \rightarrow \alpha_i) \cap \text{SELECT}(U \rightarrow \alpha_j) = \emptyset$$



## 5.2 LL(1) 文法

要求文法中任一非终结符号U的产生式右部 $\alpha_1|\alpha_2|\dots|\alpha_n$  满足:

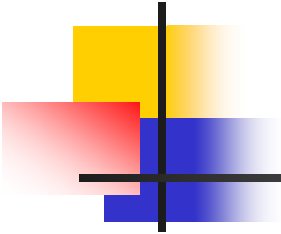
(1)相对于 $\alpha_1, \alpha_2, \dots, \alpha_n$ 的各符号串的首终结符号集合总是两两互不相交的。

$$\text{FIRST}(\alpha_i) = \{a \mid \alpha_i \xRightarrow{*} a\dots, a \in V_T\}$$

特别地, 如果  $\alpha_i \xRightarrow{*} \varepsilon$ , 则令  $\varepsilon \in \text{FIRST}(\alpha_i)$ 。

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset \quad (i \neq j)$$

提取公共左因子可以修改文法, 使文法满足上述条件。



## 5.2 LL(1) 文法

要求文法中任一非终结符号U的产生式右部 $\alpha_1|\alpha_2|\dots|\alpha_n$  满足:

(1)  $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset \quad (i \neq j)$

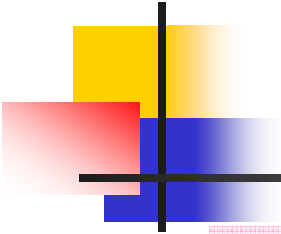
(2) 如果 $\alpha_j \xRightarrow{*} \varepsilon$ 时, 仅有上述条件(1)的限制还不够

后继终结符号集合 $\text{FOLLOW}(U)$ 定义为

$$\text{FOLLOW}(U) = \{ a \mid S \xRightarrow{*} \dots Ua \dots, a \in V_T \}$$

特别地, 当 $S \xRightarrow{*} \dots U$ 时, 规定输入结束符 $\# \in \text{FOLLOW}(U)$ 。

$$\alpha_j \xRightarrow{*} \varepsilon \text{ 时, } \text{FIRST}(\alpha_i) \cap \text{FOLLOW}(U) = \emptyset$$



## 5.2 LL(1) 文法

### 3. LL(1)文法

文法中任一非终结符号U的产生式右部 $\alpha_1|\alpha_2|\dots|\alpha_n$ 满足:

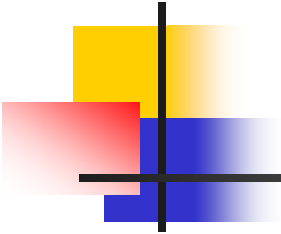
① $\alpha_1, \alpha_2, \dots, \alpha_n$ 的终结首符号集两两互不相交, 即

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset \quad (i \neq j)$$

②如果 $\alpha_j \xRightarrow{*} \varepsilon$ 时, 文法还同时满足

$$\text{FIRST}(\alpha_i) \cap \text{FOLLOW}(U) = \emptyset$$

则称该文法为LL(1)文法。



## 5.2 LL(1) 文法

### 3. LL(1)文法

文法中任一非终结符号U的产生式右部 $\alpha_1|\alpha_2|\dots|\alpha_n$ 满足:

对U的所有产生式的可选集两两互不相交, 即

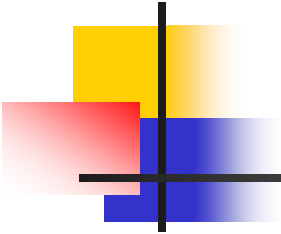
$$\text{SELECT}(U \rightarrow \alpha_i) \cap \text{SELECT}(U \rightarrow \alpha_j) = \emptyset \quad (i \neq j)$$

则称该文法为LL(1)文法。

$$\text{SELECT}(U \rightarrow \alpha) = \begin{cases} \text{FIRST}(\alpha), & \alpha \text{不可空} \\ (\text{FIRST}(\alpha) \setminus \{\epsilon\}) \cup \text{FOLLOW}(U), & \alpha \text{可空} \end{cases}$$

问题: 如何判定一个文法是否为LL(1)文法?





## 5.2 LL(1) 文法

### 4. FIRST集、FOLLOW集的求法

- FIRST的求法——推导

**$G[E]: E \rightarrow TE'$**

**$E' \rightarrow +TE' | \epsilon$**

**$T \rightarrow FT'$**

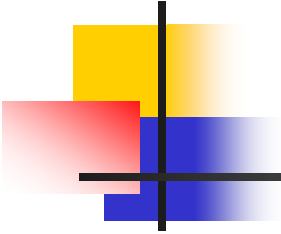
**$T' \rightarrow *FT' | \epsilon$**

**$F \rightarrow (E) | i$**

**求FIRST(E') = ?**

**FIRST(FT') = ?**

**FIRST(T'+T) = ?**



## 5.2 LL(1) 文法

---

### 4. FIRST集、FOLLOW集的求法

- FOLLOW(U)的求法

(1) 对于文法的识别符号S, 令 $\# \in \text{FOLLOW}(S)$ ;

(2)  $A \rightarrow \alpha U \beta$ ,

则 $\text{FIRST}(\beta)$ 中的非 $\epsilon$ 的元素属于 $\text{FOLLOW}(U)$ ;

(3)  $A \rightarrow \alpha U$ , 或 $A \rightarrow \alpha U \beta$ 而 $\text{FIRST}(\beta)$ 含有 $\epsilon$ ,

则 $\text{FOLLOW}(A)$ 的元素属于 $\text{FOLLOW}(U)$ 。

## 5.2 LL(1) 文法

- (1) 对于文法的识别符号S, 令 $\# \in \text{FOLLOW}(S)$ ;
- (2)  $A \rightarrow \alpha U \beta$ , 则 $\text{FIRST}(\beta)$ 中的非 $\epsilon$ 的元素属于 $\text{FOLLOW}(U)$ ;
- (3)  $A \rightarrow \alpha U$ , 或 $A \rightarrow \alpha U \beta$ 而 $\text{FIRST}(\beta)$ 含有 $\epsilon$ ,  
则 $\text{FOLLOW}(A)$ 的元素属于 $\text{FOLLOW}(U)$ 。

$G[E]: E \rightarrow TE'$

$E' \rightarrow +TE' | \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \epsilon$

$F \rightarrow (E) | i$

(1) 求  $\text{FOLLOW}(T) = ?$

(2)  $G[E]$ 是否为LL(1)文法?

## 5.2 LL(1) 文法



LL(1) 文法/分析法 的条件:

压缩的、无左递归、无回溯

WHY?

**FOLLOW(U)的求法**

(2)  $A \rightarrow \alpha U \beta$ , 不能是多余产生式,  $A$  必须是可达的!

则 **FIRST( $\beta$ )** 中的非  $\epsilon$  的元素属于 **FOLLOW(U)**;

(3)  $A \rightarrow \alpha U$ , 或  $A \rightarrow \alpha U \beta$  而 **FIRST( $\beta$ )** 含有  $\epsilon$ ,

则 **FOLLOW(A)** 的元素属于 **FOLLOW(U)**。



## 5.3 确定的LL(1)分析器

### 1. 基本思想

从 $\text{左}$ 到右扫描输入符号串，从开始符号出发生成句子的最 $\text{左}$ 推导。

对于形如  $U \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$

的产生式，只要向输入符号串中查看一个输入符号，便能惟一确定当前应选择的产生式，由此而得名 $\text{LL}(1)$ 分析法。

当需要向输入串中查看 $k$ 个输入符号，才能惟一确定当前应选择的产生式时，称为 $\text{LL}(k)$ 分析法。

## 5.3 确定的LL(1)分析器

### LL(1)文法

并非所有的文法都能通过消除左递归和反复提取公因子，改造为LL(1)文法的。

G[S]:

$S \rightarrow Ac \mid Bd$

$A \rightarrow aAc \mid b$

$B \rightarrow aBd \mid b$

G[S]:

$S \rightarrow aAcc \mid bc \mid aBdd \mid bd$

$A \rightarrow aAc \mid b$

$B \rightarrow aBd \mid b$

G[S]:

$S \rightarrow aX \mid bY$

$X \rightarrow Acc \mid Bdd$

$Y \rightarrow c \mid d$

$A \rightarrow aAc \mid b$

$B \rightarrow aBd \mid b$

无论重复多少次改写，都无法把G[S]改造成LL(1)文法。



# LL(1) 文法与LL(1) 语言的性质

结论1: 任何LL(1)文法都是**无**二义性的。

结论2: 左递归文法必然**不**是LL(1)文法。

结论3: 存在一种算法, 它能判定任一文法是否为LL(1)文法。

结论4: 存在一种算法, 它能判定任意两个LL(1)文法是否产生相同的语言。

结论5: **非**LL(1)语言是存在的。

结论6: **不**存在这样的算法, 它能判定任一上下文无关语言是否能由LL(1)文法产生。

# 非LL(1) 文法举例

任何**LL(1)**文法都是**无**二义性的。

**二义性文法**的冲突消解

**G[S]:**

$$S \rightarrow \text{if } E \text{ then } S \text{ } S' \mid a$$
$$S' \rightarrow \text{else } S \mid \epsilon$$
$$E \rightarrow b$$

分析串：  
if b then if b then a **else** a

$\text{SELECT}(S' \rightarrow \text{else } S) \cap \text{SELECT}(S' \rightarrow \epsilon) \neq \Phi$

	a	b	else	if	then	#
S	$S \rightarrow a$			$S \rightarrow \text{if } E \text{ then } S \text{ } S'$		
S'			$S' \rightarrow \text{else } S$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				





## 5.3 确定的LL(1)分析器

---

### 2. 程序结构

一张分析表 $M$ 和一个符号栈 $S$ 。

分析表的元素 $M[U, a]$ 为一条关于该非终结符号 $U$ 的产生式，指出当该非终结符号 $U$ 面临输入符号 $a$ 时应选择的产生式，分析表的元素也可能是一个出错标志，指出非终结符号 $U$ 不能面临终结符号 $a$ 。



## 5.3 确定的LL(1)分析器

文法G[E]:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid i$

	i	+	*	(	)	#
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow i$			$F \rightarrow (E)$		

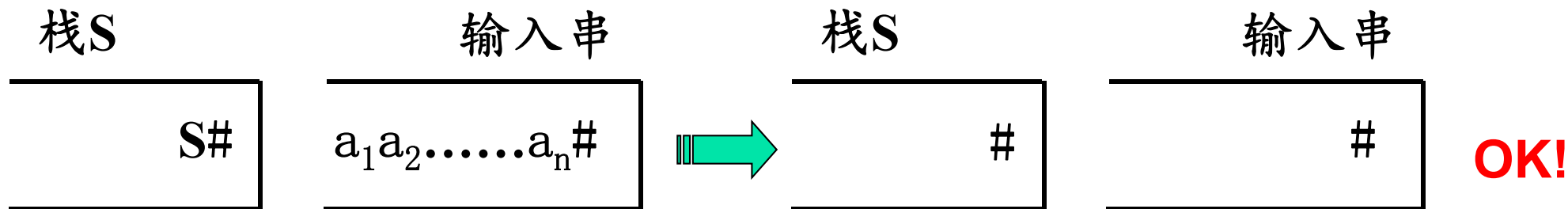
问题：如何判定一个文法是否为LL(1)文法？

## 5.3 确定的LL(1)分析器

### 2. 程序结构

一张分析表M和一个符号栈S。

符号栈S用于存放文法的符号，当文法和待分析的符号串确定后，符号栈的内容随分析过程而不断变化。开始时，栈底放“#”  
栈顶为开始符号S，假定输入符号串以“#”结束。





## 5.3 确定的LL(1)分析器

### 3. 分析算法:

由符号栈S栈顶元素 $S[k]$ 和输入串R中当前元素 $R[j]$ 查阅分析表 $M[S[k], R[j]]$ 以获得下一步骤应有信息。对于栈顶符号 $X$ 和当前输入符号 $a$ ，分析程序每次都执行下述三种可能的动作之一：

- (1) 终结符号 $X=a=\#$ ，则分析成功结束。
- (2) 终结符号 $X=a\neq\#$ ，则 $X$ 退栈，指针指向下一个输入符号。
- (3) 终结符号 $X\neq a$ ，则报语法错，不能接受该输入符号串。

(4)  $X$ 为非终结符号，则查分析表 $M$ 。 $X$ 退栈，并将查到的产生式右部符号按反序一一推进栈中（若产生式右部为 $\epsilon$ ，则不推进任何符号）；若查到出错标记，则调用出错程序。



## 5.3 确定的LL(1)分析器

### 4. 分析表的构造

分析表M的构造算法为：

- (1) 对**FIRST (x)** 中的每一终结符号a, 置 **$M[U, a] = "U \rightarrow x"$** ;
- (2) 如果 **$\epsilon \in \text{FIRST}(x)$**  , 则对于属于**FOLLOW (U)** 的每一个终结符号b或#, 分别置 **$M[U, b] = "U \rightarrow x"$** 和 **$M[U, \#] = "U \rightarrow x"$** ;
- (3) 将M中所有不能按规则(1)与(2)构造的元素置出错标志**ERROR (空白)**。



## 5.3 确定的LL(1)分析器

### 分析表的构造举例

文法G[E]:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid i$

	i	+	*	(	)	#
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow i$			$F \rightarrow (E)$		

试用LL(1)分析方法分析串 ‘i\*i’ 和 ‘i\*’。



## 5.4 递归子程序法

---

### 递归下降分析方法（递归子程序法）

#### 1. 基本思想

对每一个语法成分（用非终结符号代表），构造相应的分析子程序，该分析子程序分析相应于该语法成分（非终结符号）的符号串。

由于语法成分之间不可避免会含有递归，所以分析子程序之间也会有递归调用，故而又称为递归子程序法。



## 5.4 递归子程序法

---

### 2. 分析过程

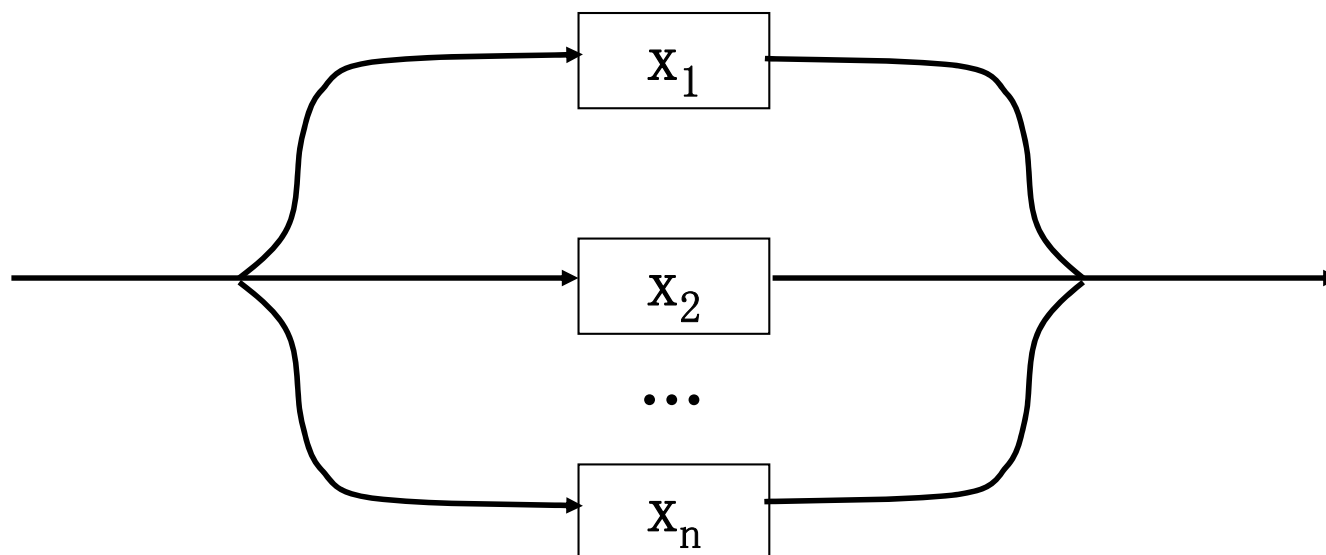
从开始符号出发，在语法规则支配下，逐个扫描输入符号串中的符号，根据文法和当前的输入符号预测到下一个语法成分是U时，便确定U为目标，并调用U的分析子程序 $P(U)$ 工作。在 $P(U)$ 工作的过程中，又有可能确定U或其它非终结符号为子目标，并调用相应的分析子程序。如此继续下去，直到得到结果。



## 5.4 递归子程序法

### 3. 分析子程序构造方法

- i. 对于每个非终结符号 $U$ ，编写一个相应的子程序 $P(U)$ ；
- ii. 对于产生式 $U \rightarrow x_1 | x_2 | \dots | x_n$ ，有一个关于 $U$ 的子程序 $P(U)$ 。





## 5.4 递归子程序法

---

### 3. 分析子程序构造方法

- i. 对于每个非终结符号 $U$ ，编写一个相应的子程序 $P(U)$ ;
- ii. 对于产生式 $U \rightarrow x_1|x_2|\dots|x_n$ ，有一个关于 $U$ 的子程序 $P(U)$ 。

```
IF CH IN FIRST( $x_1$ ) THEN P( $x_1$ )
  ELSE IF CH IN FIRST( $x_2$ ) THEN P( $x_2$ )
    ELSE ...
      ...
    IF CH IN FIRST( $x_n$ ) THEN P( $x_n$ )
      ELSE ERROR
```



## 5.4 递归子程序法

### 3. 分析子程序构造方法

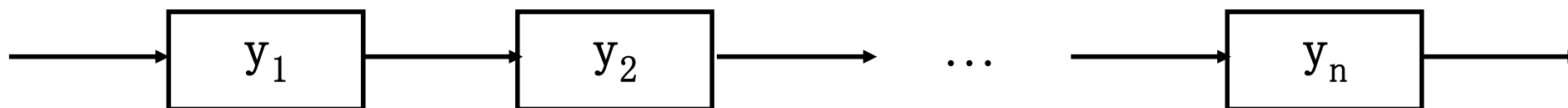
- i. 对于每个非终结符号 $U$ ，编写一个相应的子程序 $P(U)$ ;
- ii. 对于产生式 $U \rightarrow x_1|x_2|\dots|x_n$ ，有一个关于 $U$ 的子程序 $P(U)$ 。 $U$ 可空

```
IF CH IN FIRST( $x_1$ ) THEN P( $x_1$ )
  ELSE IF CH IN FIRST( $x_2$ ) THEN P( $x_2$ )
    ELSE ...
      ...
    IF CH IN FIRST( $x_n$ ) THEN P( $x_n$ )
      ELSE IF not(CH IN FOLLOW( $U$ )) THEN ERROR
```

## 5.4 递归子程序法

### 3. 分析子程序构造方法

- i. 对于每个非终结符号 $U$ ，编写一个相应的子程序 $P(U)$ ;
- ii. 对于产生式 $U \rightarrow x_1 | x_2 | \dots | x_n$ ，有一个关于 $U$ 的子程序 $P(U)$ 。
- iii. 对于 $x = y_1 y_2 \dots y_n$ ;     **BEGIN  $P(y_1)$ ;  $P(y_2)$ ; ...;  $P(y_n)$  END**





## 5.4 递归子程序法

### 3. 分析子程序构造方法

- i. 对于每个非终结符号 $U$ ，编写一个相应的子程序 $P(U)$ ;
- ii. 对于产生式 $U \rightarrow x_1|x_2|\dots|x_n$ ，有一个关于 $U$ 的子程序 $P(U)$ 。
- iii. 对于 $x=y_1y_2\dots y_n$ ;     **BEGIN  $P(y_1)$ ;  $P(y_2)$ ; ...;  $P(y_n)$  END**

如果① $y_i \in V_N$ ，则 $P(y_i)$ 就代表调用处理 $y_i$ 的子程序;

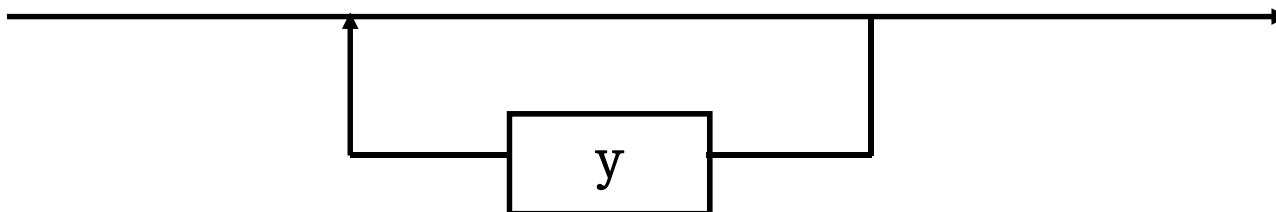
② $y_i \in V_T$ ，则 $P(y_i)$ 为形如下述语句的一段程序

**IF  $CH=y_i$  THEN READ(CH) ELSE ERROR**

## 5.4 递归子程序法

### 3. 分析子程序构造方法

- i. 对于每个非终结符号 $U$ ，编写一个相应的子程序 $P(U)$ ;
- ii. 对于产生式 $U \rightarrow x_1 | x_2 | \dots | x_n$ ，有一个关于 $U$ 的子程序 $P(U)$ 。
- iii. 对于 $x = y_1 y_2 \dots y_n$ ;     **BEGIN**  $P(y_1)$ ;  $P(y_2)$ ; ...;  $P(y_n)$  **END**
- iv. 如果 $x = \{y\}$ ，在程序中就是一个循环。





## 5.4 递归子程序法

---

约定：

每进入一个分析子程序前，已读到该子程序相应的非终结符号能推导出的第一个终结符号。

例如，当读到IF语句的第一个单词IF时，便知道将要进行IF语句的识别，于是调用对应于<IF条件语句>的分析子程序进行分析。



## 5.4 递归子程序法

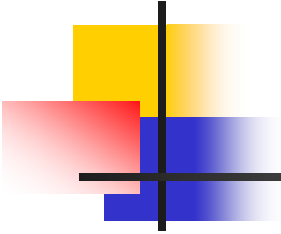
---

### 4. 递归子程序的构造举例

文法G[E]:

$$E \rightarrow eBA$$
$$A \rightarrow a \mid bAc$$
$$B \rightarrow Ed \mid aC$$
$$C \rightarrow e \mid dC \mid \epsilon$$





## 5.5 自上而下分析法

---

### □ 带回溯的自上而下分析法（非确定的）

- 对cfg具有通用性，几乎无限制
- 速度慢，效率低

### □ 不带回溯的自上而下分析法（确定的）

（LL(k)分析法，递归子程序法）

- 要求文法是压缩、无左递归、无回溯的
- 对文法有限制，不适用于所有cfg



## 5.5 不带回溯的自上而下分析法

---

### □ LL(k)分析法

- 表驱动的非递归预测分析
- 显式地维护了一个栈

### □ 递归子程序法

- 递归下降分析方法
- 通过递归调用，隐式地维护了一个栈



# 实 习

---

实习题：构造一个小语言的语法分析程序。

要求：输入属性字文件，输出源程序是否符合语法要求的结果：

正确——该程序符合语法要求。

错误——指出错误位置。

如：输入 `i:=1+`;

输出 表达式错误。

输入 `program ex1; begin i:=1 end.`

输出 该程序是正确的。



## 第5章 内容小结

---

- 基本思想
- 左递归的消除方法
- FIRST集、FOLLOW集的求法
- LL(1)文法
- LL(1)分析表的构造
- 递归子程序的构造



# 下章内容简介 —— 第6章

---

- 基本思想
- 存在的问题及解决方法
- 短语和句柄
- 简单优先分析方法
- 算符优先分析方法