




编译原理

第四章 语法分析

--自上而下分析

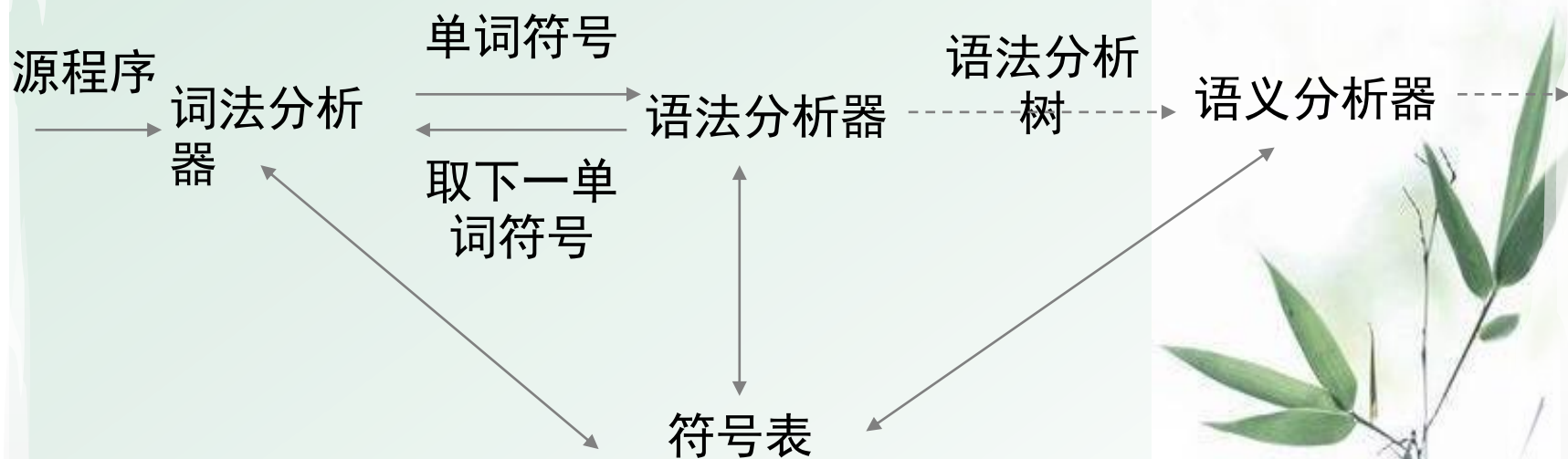


➤ 语法分析器的功能

- 自顶向下语法分析 (Top Down Parsing)
分析方法：递归子程序法、预测分析法
 - 自下而上分析法 (Bottom up Parsing)
算符优先分析法、LR分析法
- 

4.1 语法分析器的功能

- 按文法的产生式, 识别输入符号串是否为一个句子;
;
- 建立一棵与输入串相匹配的语法分析树



分析器的输出

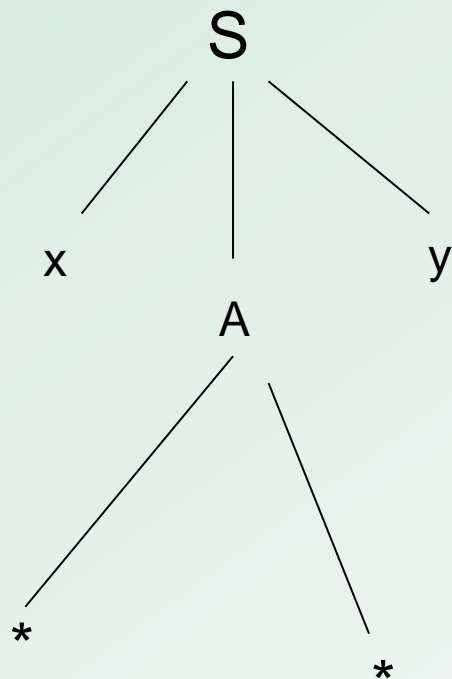
- 分析树
 - 格式化的程序
 - 合法的表达式、语句、函数
- 出错处理要求
 - 尽快发现错误，准确定位，
 - 可能时进行恢复处理，继续语法分析
- **top down parsing**: 推导
- **bottom up parsing**: 归约



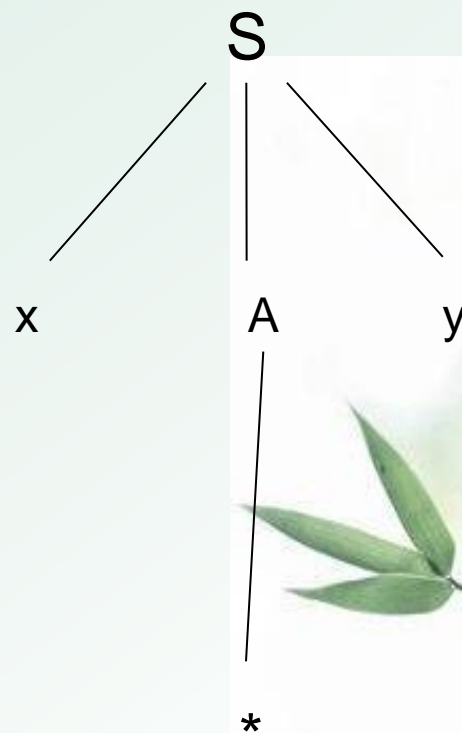
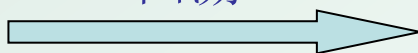
4.2 自上而下语法分析

- 基本思想:
- 从文法的开始符号出发, 向下推导出句子;
- 试图用一切可能的办法, 从文法开始符号出发, 自上而下地为输入串建立一棵语法树; (寻找一个最左推导)
- 此分析过程本质上是一种试探过程。
- 非确定的自上而下分析法实际上采用的是一种穷尽一切可能的试探法。

文法: (1) $S \rightarrow xAy$
(2) $A \rightarrow ** \mid *$
输入串 $x*y$ 记为 α



回溯



遇到的主要问题:

- (1) 无限循环
 - 当文法中出现左递归时，分析过程将陷入无限循环
- (2) 回溯
 - 如果对同一个非终结符号，存在若干个候选，如 $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_m$ 推导过程中，如果候选选择错误，将导致回溯
- (3) 虚假匹配
- (4) 难以定位出错位置
- (5) 效率低下



1. 文法左递归的消除

- 左递归:

- 对于某些 A , 存在推导 $A \Rightarrow^+ A \alpha$

- 直接左递归的消除方法①:

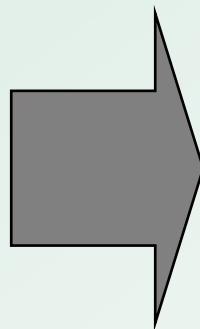
- ◆ 引入新的非终结符 A'

- ◆ 将 $A \rightarrow A \alpha \mid \beta$ 替换为

- ◆ $A \rightarrow \beta A'$ 和 $A' \rightarrow \alpha A' \mid \varepsilon$

ε

表达式文法直接左递归的消除

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

$$E \rightarrow T E'$$
$$E' \rightarrow + T E' \mid \varepsilon$$
$$T \rightarrow F T'$$
$$T' \rightarrow * F T' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

把直接左递归改为直接右递归：

将规则 $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

改写为：

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$$

Note：这并不一定能消除整个文法的左递归性

消除左递归算法:

- (1) 把文法G的所有非终结符按一种顺序排列;
- (2) FOR $i:=1$ TO n DO

BEGIN

FOR $j:=1$ TO $i-1$ DO

把形如 $A \rightarrow B\gamma$ 的规则改写

{方法是, 如果 $B \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$,

则 $A \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$ }

消除A规则中的直接左递归

END

- (3) 化简由(2)所得的文法, 即消去多余规则。

例：间接左递归的消除

$S \rightarrow Ac \mid c$

$A \rightarrow Bb \mid b$

$B \rightarrow Sa \mid a$

将 B 代入

$A \rightarrow \underline{S}ab \mid \underline{a}b \mid b$

将 A 代入

$S \rightarrow \underline{S}abc \mid \underline{a}bc \mid \underline{b}c \mid c$

消除直接左递归:

$S \rightarrow abcS' \mid bcS' \mid cS'$

$S' \rightarrow abcS' \mid \varepsilon$

删除多余规则

$A \rightarrow Sab \mid ab \mid b$

$B \rightarrow Sa \mid a$

直接左递归的消除方法：

采用扩充的BNF表示法改写含直接左递归的规则。

- 使用 $\{\beta\}$ 表示 β^* ;
- 使用 $[\beta]$ 表示 β 的出现可有可无;
- 使用 $()$ 在规则中提取因子。

例文法为：

$$E \rightarrow T | E + T$$

$$T \rightarrow F | T * F$$

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

$$E \rightarrow T \{ + T \}$$

$$T \rightarrow F \{ * F \}$$

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

2、消除回溯

- 引起回溯的原因：

①有规则 $A \rightarrow b\alpha_1 \mid b\alpha_2 \mid \dots \mid \alpha_m$

②有规则 $A \rightarrow \alpha_i \mid \text{且} \alpha_i \quad \varepsilon$

- 消除回溯：对非终结符 A 匹配时，根据所面临的输入符号，准确地指派唯一的一个候选

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_m$$



定义 FIRST、FOLLOW、SELECT 集

① $\text{FIRST}(\alpha) = \{a \mid \alpha \xRightarrow{*} a\dots, a \in V_T\}$

特别,如果 $a \xRightarrow{*} \varepsilon$, 则令 $\varepsilon \in \text{FIRST}(\alpha)$

② 非终结符A的后继符号集

$$\text{FOLLOW}(A) = \{a \mid S \xRightarrow{*} \dots Aa\dots, a \in V_T \text{ 或为 } \$\}$$

③ 规则的选择集合

$$\text{SELECT}(A \rightarrow \alpha) = \begin{cases} \text{FIRST}(\alpha) & (\text{若 } \alpha \not\xRightarrow{*} \varepsilon) \\ \text{FIRST}(\alpha) \setminus \{\varepsilon\} \cup \text{FOLLOW}(A) & (\text{若 } \alpha \xRightarrow{*} \varepsilon) \end{cases}$$

FIRST(X) 的计算法

对所有语法符号 X, 重复进行以下计算

- 1) 若 $X \in V_T$, 则 $\text{FIRST}(X) = \{X\}$ 。
- 2) 若 $X \in V_N$, 有产生式 $X \rightarrow a\cdots$, 则将 a 加入 $\text{FIRST}(X)$;
有产生式 $X \rightarrow \varepsilon$, 则将 ε 加入 $\text{FIRST}(X)$ 。
- 3) 若有产生式 $X \rightarrow Y\cdots$, 且 $Y \in V_N$,
则 $\text{FIRST}(Y)$ 的非 ε 元素 $\in \text{FIRST}(X)$;

若有产生式 $X \rightarrow Y_1 \cdots Y_k$, 并对于某个 i, 使得 $Y_1 \dots Y_{i-1} \xRightarrow{*} \varepsilon$,
则将 $\text{FIRST}(Y_i)$ 的所有非 ε 元素加入到 $\text{FIRST}(X)$ 中;

若 $Y_1 \dots Y_k \xRightarrow{*} \varepsilon$, 则将 ε 加入到 $\text{FIRST}(X)$ 。

FIRST(α) 的计算法?

例：表达式文法 $E \rightarrow T E' \quad E' \rightarrow + T E' \mid \varepsilon$
 $T \rightarrow F T' \quad T' \rightarrow * F T' \mid \varepsilon$
 $F \rightarrow (E) \mid \text{id}$ 的 FIRST 集

$\text{FIRST}(F) = \{ (, \text{id} \}$

$\text{FIRST}(T) = \text{FIRST}(F) = \{ (, \text{id} \}$

$\text{FIRST}(E) = \text{FIRST}(T) = \{ (, \text{id} \}$

$\text{FIRST}(E') = \{ +, \varepsilon \}$

$\text{FIRST}(T') = \{ *, \varepsilon \}$

FOLLOW(A) 的计算法

- 1) 将 \$ 加入到 FOLLOW(S)
 - 句子的结束符

对于所有非终结符, 重复进行以下计算

- 2) 若 $A \rightarrow \alpha B \beta$,
 - 则将 $\text{FIRST}(\beta)$ 的非 ϵ 元素加入 FOLLOW(B)
- 3) 如果 $A \rightarrow \alpha B$ 或 $A \rightarrow \alpha B \beta$, 且 $\beta \xRightarrow{*} \epsilon$,
 $A \neq B$,
 - 则将 FOLLOW(A) 的所有元素加入 FOLLOW(B)

例：表达式文法 $E \rightarrow T E'$ $E' \rightarrow + T E' \mid \varepsilon$
 $T \rightarrow F T'$ $T' \rightarrow * F T' \mid \varepsilon$
 $F \rightarrow (E) \mid \text{id}$ 的 FOLLOW 集

$\text{FOLLOW}(E) = \{), \# \}$

$\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{), \# \}$

$\text{FOLLOW}(T) = \{ +,), \# \}$

$\text{FOLLOW}(T') = \text{FOLLOW}(T) = \{ +,), \# \}$

$\text{FOLLOW}(F) = \{ *, +,), \# \}$

表达式文法是 **LL(1)** 文法

- $E \rightarrow T E'$
- $E' \rightarrow + T E' \mid \varepsilon$
- $T \rightarrow F T'$
- $T' \rightarrow * F T' \mid \varepsilon$
- $F \rightarrow (E) \mid id$

考察：文法消除了左递归；

- $E' : FIRST(E') = \{+, \varepsilon\}, FOLLOW(E') = \{), \#\}$
- $T' : FIRST(T') = \{*, \varepsilon\}, FOLLOW(T') = \{+,), \#\}$
- $F : ($ 和 id 不同

LL (1) 文法输入串的匹配

非终结符号 $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_m$

面临的输入符号 a

- 1) 若 $a \in \text{SELECT} (A \rightarrow \alpha_i)$, 则指派 α_i
- 2) 否则, a 的出现是一种语法错

此即有效的无回溯的自上而下分析



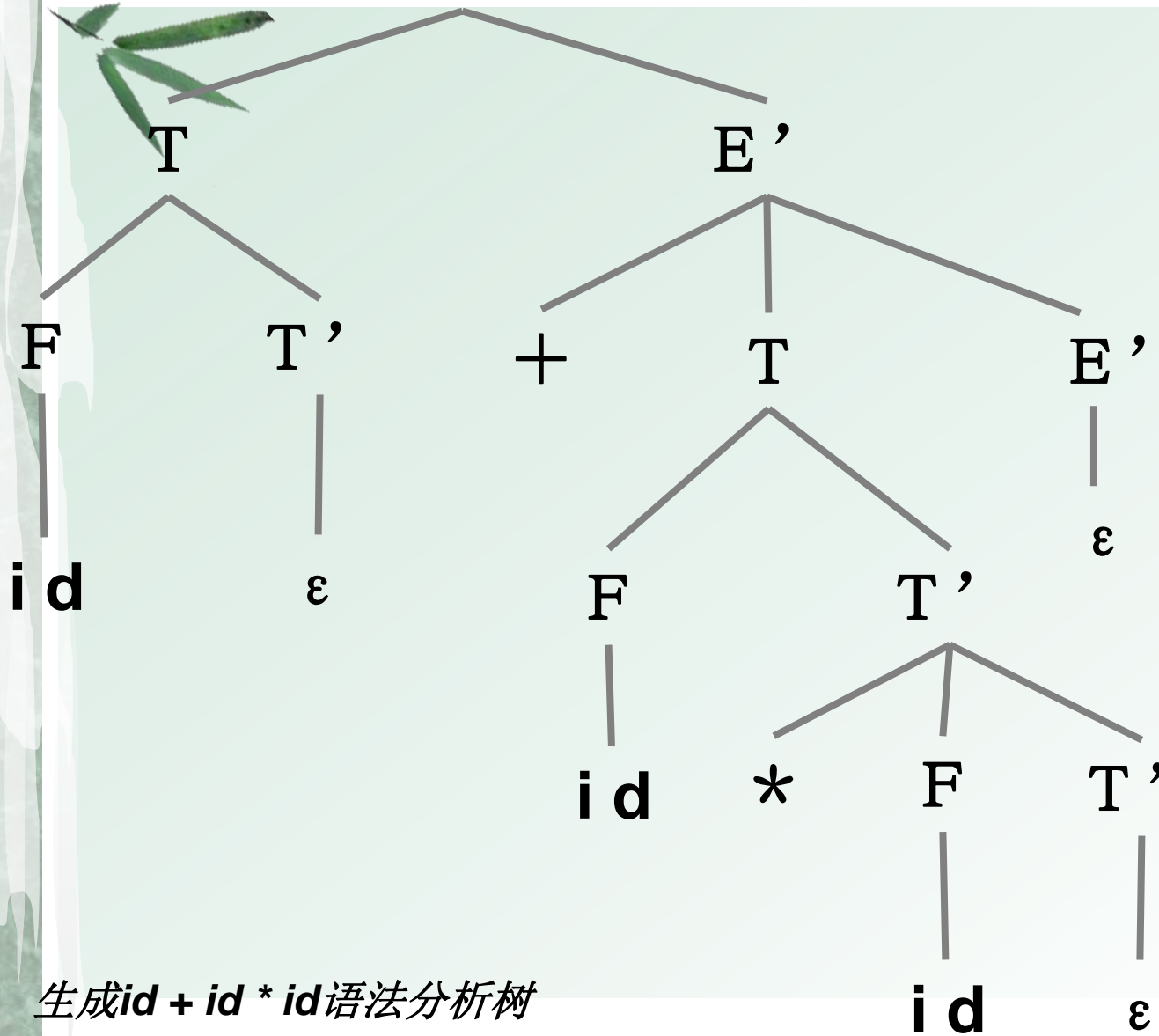
例：分析符号串 $\text{id} + \text{id} * \text{id}$ 符合表达式文法：

- $E \rightarrow T E'$
- $E' \rightarrow + T E' \mid \varepsilon$
- $T \rightarrow F T'$
- $T' \rightarrow * F T' \mid \varepsilon$
- $F \rightarrow (E) \mid \text{id}$

按照最左推导过程，构造分析树



推导过程



$E \rightarrow T E'$

$E' \rightarrow + T E' \mid \varepsilon$

$T \rightarrow F T'$

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

$F \rightarrow (E) \mid id$

生成 $id + id * id$ 语法分析树

某些非LL(1)文法改写为LL(1)文法

- ◆ 消除左递归;
- ◆ 反复提取公共左因子

例：存在左因子的文法

- **stmt \rightarrow if expr stmt**
- **| if expr stmt ; else stmt**
- **| other**

存在左因子 **if expr stmt**

左因子提取方法

- 对于所有形如

$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \dots \mid \alpha \beta_n \mid \gamma$ 的规则

- 改写为

$A \rightarrow \alpha A' \mid \gamma$ 和 $A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

- 文法改写结果:

$\text{stmt} \rightarrow \text{if expr stmt } S' \mid \text{other}$

$S' \rightarrow \varepsilon \mid ; \text{ else stmt}$

Note: 并非一切非LL(1)的文法都能改写成LL(1)的。

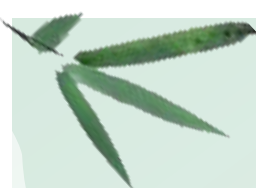
例如文法:

$S \rightarrow Ae \mid Bd$


$A \rightarrow aAe \mid b$

$B \rightarrow aBd \mid b$





例：给出语言 $L = \{1^n a 0^n 1^m a 0^m \mid n > 0, m \geq 0\}$ 的 LL(1) 文法 $G[S]$ ，并说明 $G[S]$ 为 LL(1) 文法的理由。



4.4 递归下降分析法

递归下降分析程序的构造:

- 对文法的每一个非终结符编写一个子程序，识别该非终结符所表示的语法成分。
- 这些子程序相互递归调用。



递归子程序法：

- 1) 编写文法、消除二义性
- 2) 消除左递归和提取左因子
- 3) 求各规则的SELECT集
- 4) 检查是不是 LL(1) 文法

若不是 LL(1), 说明文法的复杂性超过自顶向下方法的分析能力

5) 按照文法规则，编写程序算法

◆为每个非终结符设置一个子程序，按照候选规则编写控制结构；

◆按照 SELECT 集识别终结符，调用非终结符的子程序。

例：表达式文法的分析器

- E 的子程序 // $E \rightarrow T\{+T\}$

E ()

{

 T () ; // T 的调用

 while (sym== '+')

 { //当前符号等于+时

 match('+'); //处理终结符+

 T () ; // T 的过程调用

 }

} // sym: 当前符号

T 的子程序 $T \rightarrow F\{*F\}$

```
T ( ) ;
```

```
{
```

```
    F ( ) ;
```

// F 的调用

```
    while (sym== '*' )
```

```
    {
```

//当前符号等于*时

```
        match(' *' );
```

//处理终结符*

```
        F ( ) ;
```

// F 的递归调用

```
    }
```

```
}
```

F 的子程序

$F \rightarrow id|(E)$

```
F ( ) ;  
{  
    if (sym== '(' )  
        {  
            match(' ( ');    //当前符号等于 (   
            E ( ) ;           // 处理终结符 (   
                               // E 的递归调用   
            match(' ) ');    // 处理终结符)   
        }  
    else if (sym==id)  
        match(id)            // 处理终结符id   
    else error()             // 出错处理   
}
```

主程序

```
main ( )
{
    sym=nexttoken;    //调词法分析程序
    E ( ) ;           // E 的调用
    if (sym== '$' )    printf( "success!" );
        else printf( "fail!" );
}
match( t:token );
{
    if (sym==t)        sym=nexttoken;
    else error();      // 出错处理程序
}
```


递归子程序法优缺点分析

- 优点:

- 1) 直观、简单、可读性好
- 2) 便于扩充

- 缺点:

- 1) 递归算法的实现效率低
- 2) 处理能力相对有限
- 3) 通用性差，难以自动生成



4.5 预测分析程序法

- 使用一张分析表和一个栈来实现LL(1)分析
- 预测分析表 $M[A,a]$ 是一个矩阵，行代表非终结符，列代表终结符或‘\$’；矩阵元素代表对应的产生式或出错标志
- 栈STACK中存放文法符号,栈底先放一个结束符‘\$’

预测分析器模型

输入缓冲区

..... a\$

×

⋮

⌘

栈

总控程序

输出

预测分析表M

二维数组[非终结符A, 终结符a]
表项用于指示产生式或出错标志

预测分析表构造算法:

1、计算文法的每个非终结符的FIRST集和FOLLOW集
以及每条规则的SELECT集。

- 2、1) 对于每一产生式 $A \rightarrow \alpha$, 执行 2)
2) 对于SELECT ($A \rightarrow \alpha$)中的每一终结符a(或\$),
将 $A \rightarrow \alpha$ 填入 $M[A,a]$
3) 将所有无定义的 $M[A,b]$ 标上出错标识。

例 表达式文法的LL(1)预测分析表

非终结符	输入符号					
	id	+	*	()	\$
E	$\rightarrow TE'$			$\rightarrow TE'$		
E'		$\rightarrow +TE$			$\rightarrow \epsilon$	$\rightarrow \epsilon$
T	$\rightarrow FT'$			$\rightarrow FT'$		
T'		$\rightarrow \epsilon$	$\rightarrow *FT'$		$\rightarrow \epsilon$	$\rightarrow \epsilon$
F	$\rightarrow id$			$\rightarrow (E)$		

执行实例：分析 $id+id*id$

栈	输入缓冲区	所用产生式
\$E	id+id*id\$	
\$E'T	id+id*id\$	$E \rightarrow TE'$
\$E'T'F	id+id*id\$	$T \rightarrow FT'$
\$E'T'id	id+id*id\$	$F \rightarrow id$
\$E'T'	+id*id\$	
\$E'	+id*id\$	$T' \rightarrow \varepsilon$
\$E'T+	+id*id\$	$E' \rightarrow +TE'$
\$E'T	id*id\$	

栈	输入缓冲区	所用产生式
\$E'T	id*id\$	
\$E'T'F	id*id\$	$T \rightarrow FT'$
\$E'T'id	id*id\$	$F \rightarrow id$
\$E'T'	*id\$	
\$E'T'F*	*id\$	$T' \rightarrow *FT'$
\$E'T'F	id\$	
\$E'T'id	id\$	$F \rightarrow id$
\$E'T'	\$	
\$E'	\$	$T' \rightarrow \varepsilon$
\$	\$	$E' \rightarrow \varepsilon$

分析过程

输出的产生式序列形成了按最左推导生成的分析树

预测分析法（状态矩阵法）

- 1) 编写文法，消除二义性；
- 2) 消除左递归、提取左因子；(改写文法)
- 3) 求 **FIRST** 集和 **FOLLOW** 集
- 4) 检查是不是 **LL(1)** 文法
 - 若不是 **LL(1)**,说明文法的复杂性超过自顶向下方法的分析能力
- 5) 按照 **LL(1)** 文法构造预测分析表
- 6) 实现预测分析器

- 自顶向下分析的问题:
- 对于某些语言现象, 难以用 **LL(1)** 文法来描述
- 消除左递归和提取左因子影响文法的可读性, 造成语义处理的困难

