

编译原理

--自顶向下的语法分析

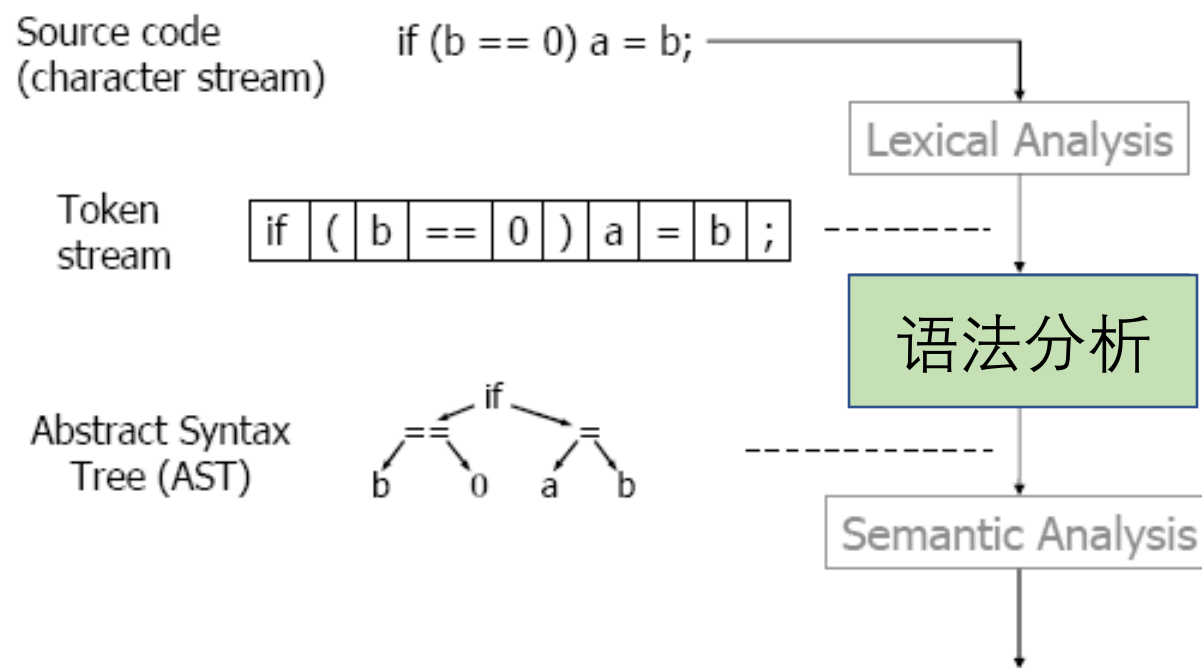
—— 刘爽 ——

中国人民大学信息学院

■ Outline

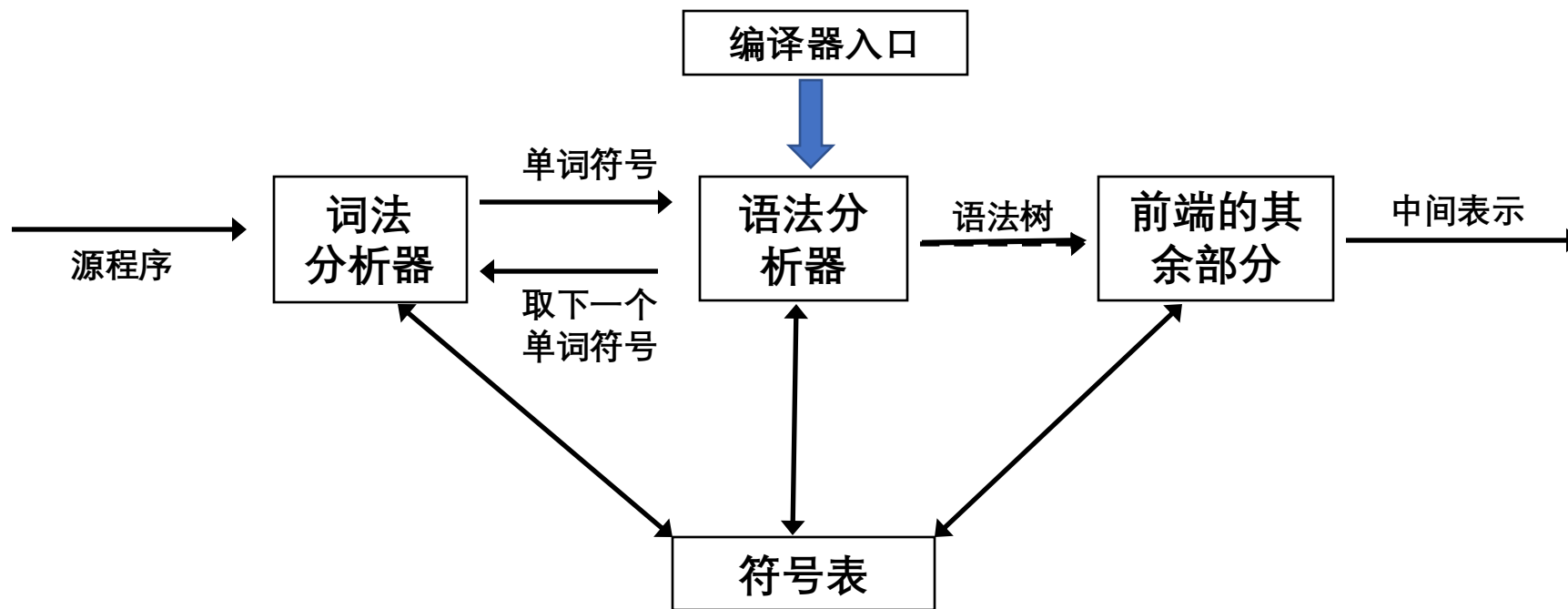
- 语法分析器的功能
- 自上而下分析面临的问题
- LL(1)分析法
- 递归下降分析程序构造
- 预测分析程序
- LL(1)分析中的错误处理

■ 语法分析器所处的位置



■ 语法分析器的作用

- 接收词法分析器提供的记号串
- 检查记号串是否能由源程序语言的文法产生
- 用易于理解的方式提示语法错误信息，并能从常见的错误中恢复过来

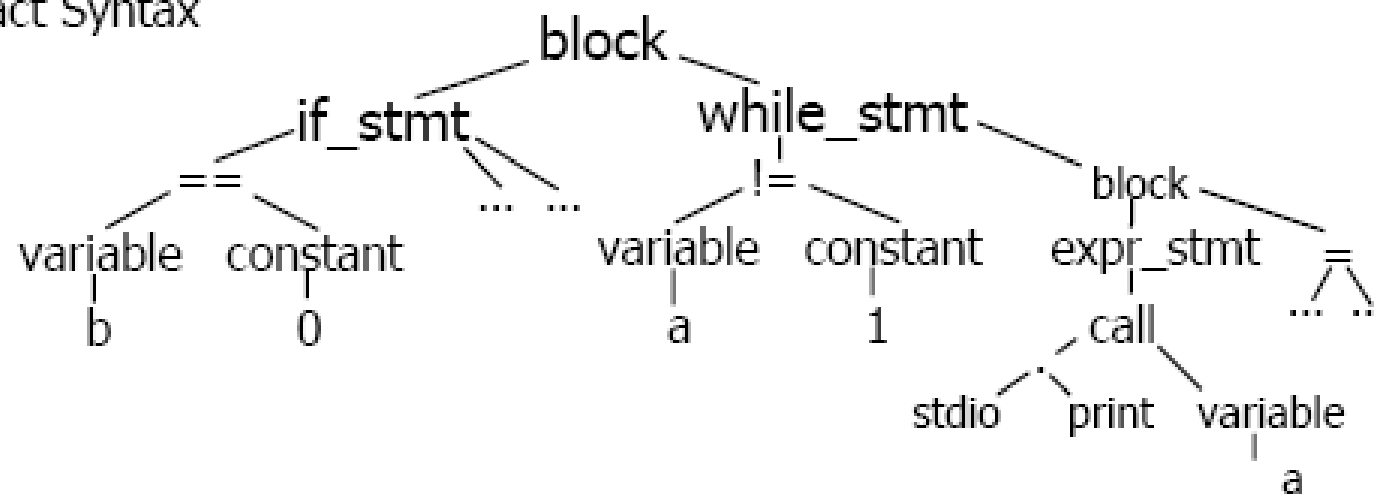


■ 语法分析的例子

源代码
(token stream)

```
if (b==0)
  a=b;
while (a!=1){
  stdio.print(a);
  a=a-1;
}
```

Abstract Syntax
Tree



■ 语法分析器工作原理

- 语言的结构是用上下文无关文法描述的，因此，语法分析器的工作本质上就是按照文法的产生式，识别输入符号串是否为一个句子。
- 语法分析器是从左向右的扫描输入字符串，每次读入一个符号，并判断，看是否从文法的开始符号出发推导出这个输入串。或者，从概念上讲，就是要建立一棵与输入串匹配的语法分析树。
- 语法分析器分类
 - 通用的语法分析方法，用来分析任何文法，生成编译器时效率太低
 - 编译器使用的语法分析方法—处理文法的一些子类
 - 自顶向下（建立分析树，即从句型到句子）—LL文法，其分析器常用手工实现
 - 自底向上（建立分析树，即从句子到句型）—LR文法，其分析器常利用自动生成工具构造

■ Outline

- 语法分析器的功能
- 自上而下分析面临的问题
- LL(1)分析法
- 递归下降分析程序构造
- 预测分析程序
- LL(1)分析中的错误处理

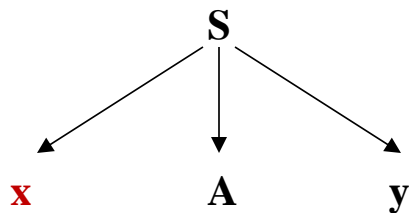
■ 自顶向下分析面临的问题 – 回溯

- 假定文法 $G[S]: S \rightarrow xAy \quad A \rightarrow **|*$
以及输入串 $x*y$ (记为 α)
- 初始化:

S

x*y
↑
P

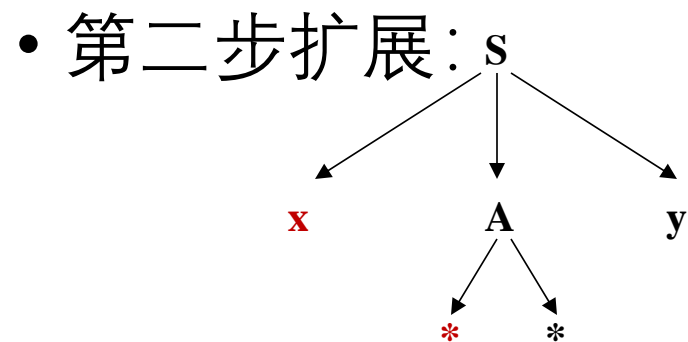
- 第一步扩展



x*y
↑
P

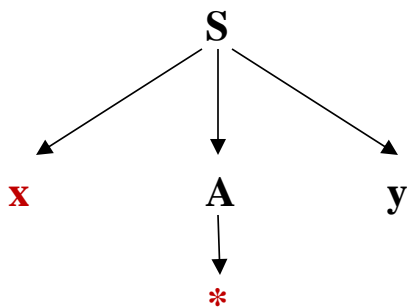
■ 自顶向下分析面临的问题-回溯

- 假定文法 $G[S]$: $S \rightarrow xAy$ $A \rightarrow **|*$
以及输入串 $x*y$ (记为 α)



$x*y$
P

- 回溯



$x*y$
P

注意: 不替换已经使用的产生式, 如果替换, 就是回退 (回溯) .

所谓确定与非确定的分析, 取决于是否发生回溯.

■ 左递归

给定文法G[E]:

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid i$

该文法是左递归的，导致分析过程陷入无限循环

$E \Rightarrow E+T \Rightarrow E+T+T \Rightarrow E+T+T+T \Rightarrow \dots$

对于要分析的字符串 $i+i*i$ ，如果不向前探索几步，不知道什么时候体制使用该产生式

■ 自顶向下分析面临的问题

- 自顶向下分析的主旨是，对任何输入串，**穷尽**一切可能的办法，从文法的**开始符号**（根结）出发，**自顶向下**的为输入串建立一棵**语法树**。这种分析过程本质上是一种**试探过程**，是反复使用不同产生式谋求**匹配输入串**的过程。因而代价极高。
- 自顶向下分析的一般方法是带“**回溯**”的。耗时且需要处理复杂情况，需要找到克服“回溯”的方法。
- **左递归**问题导致分析过程陷入无限循环，因为**必须消除文法的左递归性**。
一个文法是左递归的，如果存在非终结符 P ， $P^{\pm} > P\alpha$
- **虚假匹配**问题:即某些“成功”匹配是暂时性的。

■ Outline

- 语法分析器的功能
- 自上而下分析面临的问题
- LL(1)分析法
- 递归下降分析程序构造
- 预测分析程序
- LL(1)分析中的错误处理

■ LL(1)分析法

- 这里LL(1)中的第一个L表示从左到右扫描输入串，第二个L表示最左推导，1表示分析时每一步只需向前查看一个符号。
- 消除左递归
- 消除回溯

■ 消除直接左递归 I

$$P \rightarrow P\alpha \mid \beta \quad \xrightarrow{\beta \text{ 不以 } P \text{ 开头}} \quad \begin{array}{l} P \rightarrow \beta P' \\ P' \rightarrow \alpha P' \mid \varepsilon \end{array}$$

得到的非直接左递归形式的文法与原文法等价

例子

$$E \rightarrow E+T \mid T \quad \longrightarrow \quad \begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \varepsilon \end{array}$$

$$T \rightarrow T*F \mid F \quad \longrightarrow \quad \begin{array}{l} T \rightarrow FT' \\ T' \rightarrow *FT' \mid \varepsilon \end{array}$$

$$F \rightarrow (E) \mid i \quad \longrightarrow \quad F \rightarrow (E) \mid i$$

■ 消除直接左递归 II

- 一般而言,假定关于**A**的全部产生式是:

$$\mathbf{A} \rightarrow \mathbf{A}\alpha_1 | \mathbf{A}\alpha_2 | \dots | \mathbf{A}\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$$

- 其中 $\beta_i (i=1,2,\dots,n)$ 不以**A**打头;

$$\alpha_j (j=1,2,\dots,m) \text{ 不等于 } \varepsilon,$$

- 那么可以把上述产生式改写为:

$$\mathbf{A} \rightarrow \beta_1 \mathbf{A}' | \beta_2 \mathbf{A}' | \dots | \beta_n \mathbf{A}'$$

$$\mathbf{A}' \rightarrow \alpha_1 \mathbf{A}' | \alpha_2 \mathbf{A}' | \dots | \alpha_m \mathbf{A}' | \varepsilon$$

使用该方法,可以把直接左递归都去掉(改为直接右递归)。但这并不意味着已经消除了整个文法的左递归性。

例如文法:

$$S \rightarrow Aa | b$$

$$A \rightarrow Sd | \varepsilon$$

非终结符号S是左递归的,
因为有 $S \Rightarrow Aa \Rightarrow Sda$,
但它不是直接左递归的

■ 消除左递归的一般算法

- 如果一个文法**不含回路**（形如 $P \Rightarrow^+ P$ 的推导），也**不含以 ε 为右部的产生式**，那么执行下述算法将保证**消除左递归**（但改写后的文法可能含有 ε 为右部的产生式）。

消除左递归算法

1. 排序

(1) 把文法G的所有非终结符按任意一种顺序排列成 P_1, P_2, \dots, P_n ; 按此顺序执行
(2) FOR **$i:=1$** TO **n** DO

2.1 代入

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

把形如 $P_i \rightarrow P_j \gamma$ 的规则改写成

$P_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ 其中 $P_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ 是关于 P_j 的所有规则;

2.2 消除直接左递归

消除关于 P_i 规则的直接左递归

3. 化简

(3) 化简由(2)得到的文法，即去除那些由开始符号出发永远无法到达的非终结符的产生规则。

■ 消除左递归的例子

1. 排序

$R \rightarrow Sa \mid a$
 $Q \rightarrow Rb \mid b$
 $S \rightarrow Qc \mid c$

2.1 代入

$R \rightarrow Sa \mid a$
 $Q \rightarrow Sab \mid ab \mid b$
 $S \rightarrow Sabc \mid abc \mid bc \mid c$

2.2 消除直接左递归

$R \rightarrow Sa \mid a$
 $Q \rightarrow Sab \mid ab \mid b$
 $S \rightarrow abcS' \mid bcS' \mid cS'$
 $S' \rightarrow abcS' \mid \varepsilon$

若初始排序
为S,Q R?

3. 化简

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

■ 练习

1. 排序



$S \rightarrow Qc \mid c$
 $Q \rightarrow Rb \mid b$
 $R \rightarrow Sa \mid a$

$S \rightarrow Qc \mid c$
 $Q \rightarrow Rb \mid b$
 $R \rightarrow bcaR' \mid caR' \mid aR'$
 $R' \rightarrow bcaR' \mid \varepsilon$

■ 消除回溯-提取左因子 I

- 语句 \rightarrow if 条件 then 语句 else 语句 α_i
| if 条件 then 语句 α_j

非终结符“语句”的两个候选的终结首符集都是{if}
即 $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) \neq \Phi$

要消除回溯，就要保证对文法G每个非终结符A的所有产生式 $\alpha_i, \alpha_j, i, j \in [1, n]$ 且 $i \neq j$, $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \Phi$

如何把一个文法改造成任何非终结符的所有候选首符集两两不相交?
 \rightarrow 提取公共左因子

■ 消除回溯-提取左因子 II

- 提取左因子的方法
 - 假定 A 的规则是：
$$A \rightarrow \delta\beta_1 \mid \delta\beta_2 \mid \cdots \mid \delta\beta_n \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_m$$
（其中，每个 γ 不以 δ 开头）
 - 那么这些规则可以改写为：
$$A \rightarrow \delta A' \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$
- 经过反复提取左因子，就能够把每个非终结符（包括新引进者）的所有候选首符集变成两两不相交。我们为此要付出的代价是大量引进新的非终结符和 ϵ 产生式。

FIRST集合

- 令 G 是一个不含左递归的文法，对 G 的所有非终结符的每个候选 α 定义它的终结首符集 $FIRST(\alpha)$ 为：

- $FIRST(\alpha) = \{a \mid \alpha \xRightarrow{*} a\cdots, a \in V_T\}$
- 若 $\alpha \xRightarrow{*} \varepsilon$ ，则规定 $\varepsilon \in FIRST(\alpha)$

$FIRST(\alpha)$ 是 α 的所有可能推导的开头终结符或可能的 ε

- 如果非终结符 A 的所有候选首符集两两不相交，即 A 的任何两个不同候选 α_i 和 α_j ：

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$$

那么当要求 A 匹配输入串时， A 就能根据它所面临的第一个输入符号 a ，准确的指派某一个候选前去执行任务。这个候选就是那个终结首符集含 a 的 α 。

A 不再试探性的选派候选去完成任务，而是根据面临的输入符号 a 确定性的指派唯一的候选，被指派候选的工作成败完全代表了 A ，即消除回溯。

■ LL(1)分析条件—示例

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid i$

$\text{FIRST}(TE') = \{ (, i \}$

$\text{FIRST}(+TE') = \{ + \}$

$\text{FIRST}(FT') = \{ (, i \}$

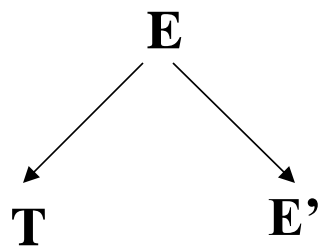
$\text{FIRST}(*FT') = \{ * \}$

$\text{FIRST}((E)) = \{ (\}$

$\text{FIRST}(i) = \{ i \}$

对输入串 $i+i$ 进行自顶向下分析

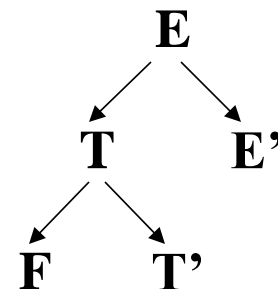
1



$i \in \text{FIRST}(TE')$

$i + i$
↑
IP

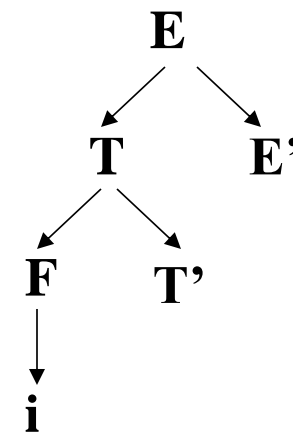
2



$i \in \text{FIRST}(FT')$

$i + i$
↑
IP

3



$i \in \text{FIRST}(i)$

$i + i$
↑
IP

■ LL(1)分析条件—示例

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid i$

$\text{FIRST}(TE') = \{ (, i \}$

$\text{FIRST}(+TE') = \{ + \}$

$\text{FIRST}(FT') = \{ (, i \}$

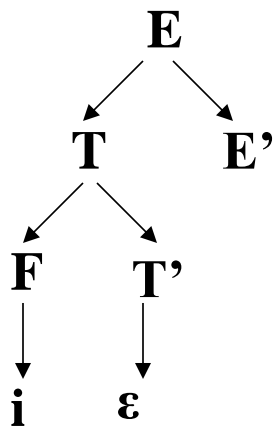
$\text{FIRST}(*FT') = \{ * \}$

$\text{FIRST}((E)) = \{ (\}$

$\text{FIRST}(i) = \{ i \}$

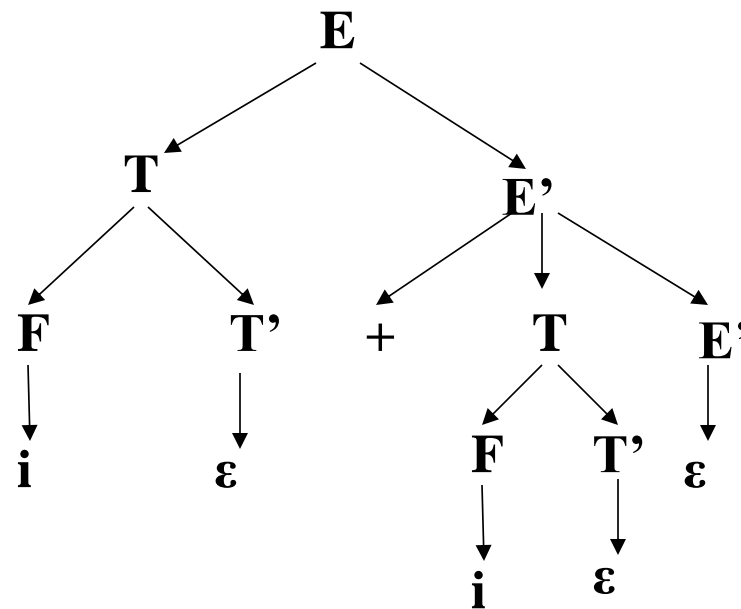
对输入串 $i+i$ 进行自顶向下分析

4



$i + i$
↑
IP

+不属于 T' 的任一候选式的首符集



是不是当非终结符 A 面临输入符号 a ，且 a 不属于 A 的任意候选首符集但 A 的某个候选首符集包含 ε 时就一定可以使 A 自动匹配？

只有当 a 是允许在文法的某个句型中跟在 A 后面的终结符时，才能允许 A 自动匹配，否则， a 在此出现就是一种语法错误

FOLLOW集合

- 如果A的某个候选首符集中包含 ϵ 怎么办？
- 假定S是文法G的开始符号，对于G的任何非终结符A，我们定义(后随符号集 FOLLOW)
 - $\text{FOLLOW}(A) = \{a \mid S \xRightarrow{*} \dots Aa \dots, a \in V_T\}$
 - 若 $S \xRightarrow{*} \dots A$, 则规定 $\# \in \text{FOLLOW}(A)$

即 $\text{FOLLOW}(A)$ 是所有句型中出现在紧接A之后的终结符或“#”。

- 开始符号的FOLLOW集初始化时加入“#”。
- 当非终结符A面临输入符号a，且a不属于A的任意候选首符集但A的某个候选首符集包含 ϵ 时，只有当 $a \in \text{FOLLOW}(A)$ 时，才可能允许A自动匹配 ϵ 。

■ LL(1)分析条件

- 满足构造不带回溯的自顶向下分析的文法条件。
 - 文法不含左递归
 - 对于文法中每一个非终结符A的各个产生式的候选首符集两两不相交。
即, 若 $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$, 则 $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \Phi$ ($i \neq j$)
 - 对文法中的每个非终结符A, 若它存在某个候选首符集包含 ϵ , 则,
 $\text{FIRST}(A) \cap \text{FOLLOW}(A) = \Phi$

如果一个文法G满足以上条件, 则称该文法G为LL(1)文法。

这里LL(1)中的第一个L表示从左到右扫描输入串, 第二个L表示最左推导, 1表示分析时每一步只需向前查看一个符号。

■ LL(1)分析条件

- 对于一个LL(1)文法，可以对其输入串进行有效的无回溯的自顶向下分析。
- 假设要用非终结符A进行匹配，面临的输入符号为a，A的所有产生式为 $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$
 - 若 $a \in \text{FIRST}(\alpha_i)$ ，则指派 α_i 去执行匹配任务。
 - 若a不属于任何一个候选首字符集，则：
 - 若 ϵ 属于某个 $\text{FIRST}(\alpha_i)$ ，且 $a \in \text{FOLLOW}(A)$ ，则让A与 ϵ 自动匹配；
 - 否则，a的出现是一种语法错误。

根据LL(1)文法的条件，每一步这样的工作都是确信无疑的

■ Outline

- 语法分析器的功能
- 自上而下分析面临的问题
- LL(1)分析法
- 递归下降分析程序构造
- 预测分析程序
- LL(1)分析中的错误处理

■ 扩充的巴科斯范式

- 前面的巴科斯范式只用到了两个元符号“ \rightarrow ”和“ $|$ ”
- 扩充的巴科斯范式加入几个元语言符号：
 - 用花括号 $\{\alpha\}$ 表示闭包运算 α^* 。
 - 用 $\{\alpha\}_n^0$ 表示 α 可任意重复0次至n次。 $\{\alpha\}_0^0 = \{\alpha\}^0 = \epsilon$ 。
 - 用方括号 $[\alpha]$ 表示 $\{\alpha\}_1^0$ ，即表示 α 的出现可有可无（等价于 $\alpha | \epsilon$ ）。
- 例如，通常的“实数”可定义为：
decimal \rightarrow [sign]integer.{digit}[exponent]
exponent \rightarrow E[sign]integer
integer \rightarrow digit{digit}
sign \rightarrow + | -

■ 扩充的巴科斯范式—例子

巴科斯范式

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid i$

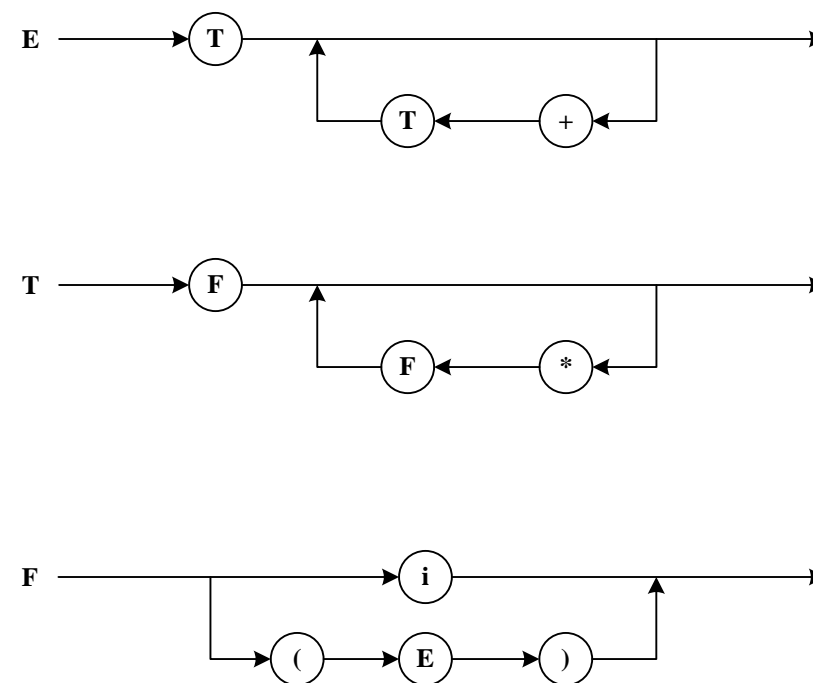
扩充的巴科斯范式

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

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

$F \rightarrow (E) \mid i$

语法图



■ 递归下降分析程序构造

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

```
PROCEDURE E;  
BEGIN  
  T;  
  WHILE SYM = '+' DO  
    BEGIN  
      ADVANCE; T  
    END  
  END  
END
```

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

```
PROCEDURE T;  
BEGIN  
  F;  
  WHILE SYM = '*' DO  
    BEGIN  
      ADVANCE; F  
    END  
  END  
END
```

$F \rightarrow (E) \mid i$

```
PROCEDURE F;  
  IF SYM = 'i' THEN ADVANCE;  
  ELSE  
    IF SYM = '(' THEN  
      BEGIN  
        ADVANCE  
        E;  
        IF SYM = ')' THEN ADVANCE;  
        ELSE ERROR;  
      END  
    ELSE ERROR;
```

■ 递归下降分析程序的构造

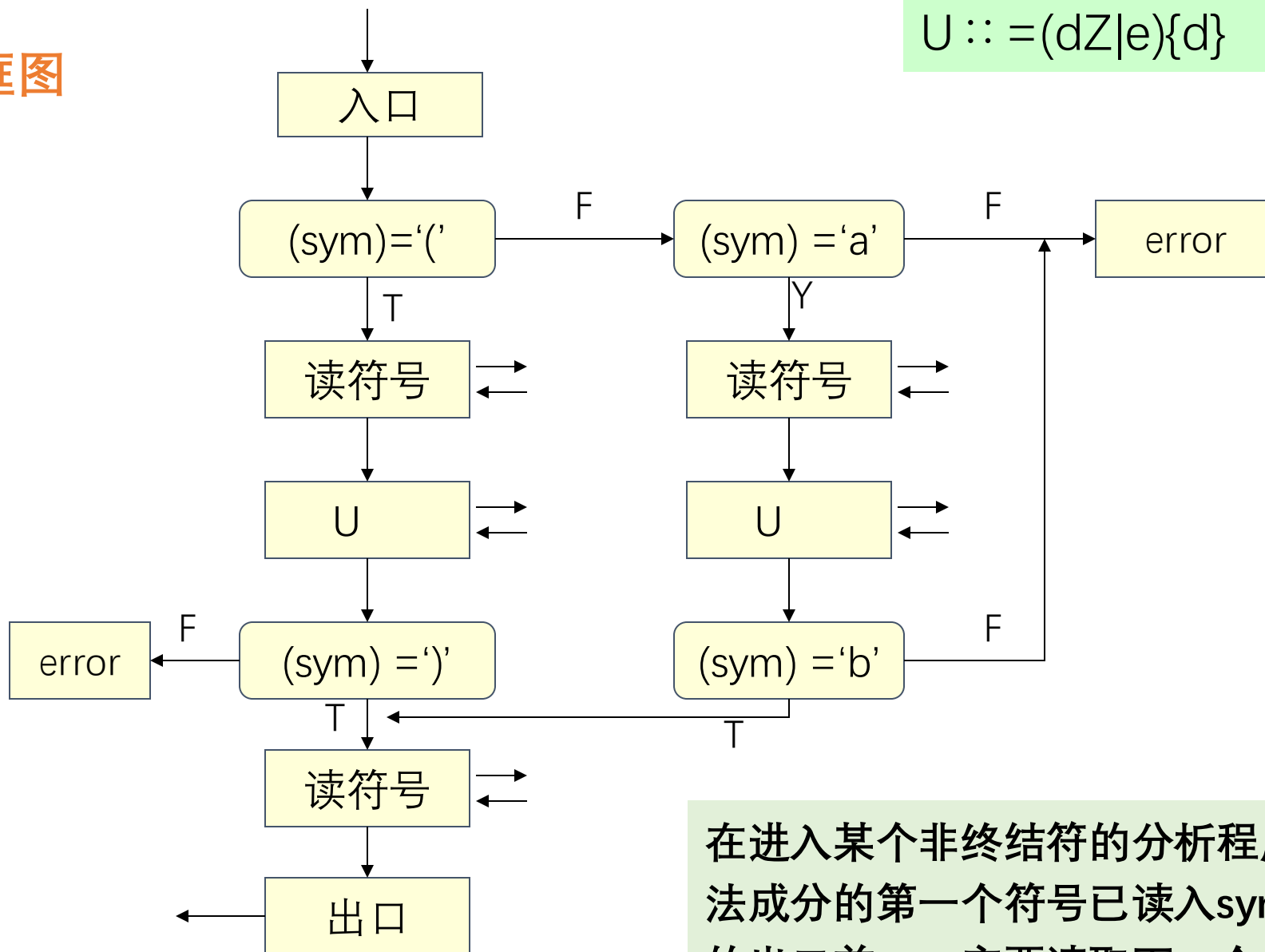
- 当文法满足**LL(1)** 条件时，我们就可以为它构造一个**不带回溯**的**自顶向下**分析程序，这个分析程序是由一组递归过程组成的。每个过程对应文法的一个**非终结符**。这样的分析程序称为**递归下降分析器**。

具体做法：对语法的每一个非终结符都编一个分析程序，当根据文法和当时的输入符号预测到要用某个非终结符去匹配输入串时，就调用该非终结符的分析程序。

算法框图

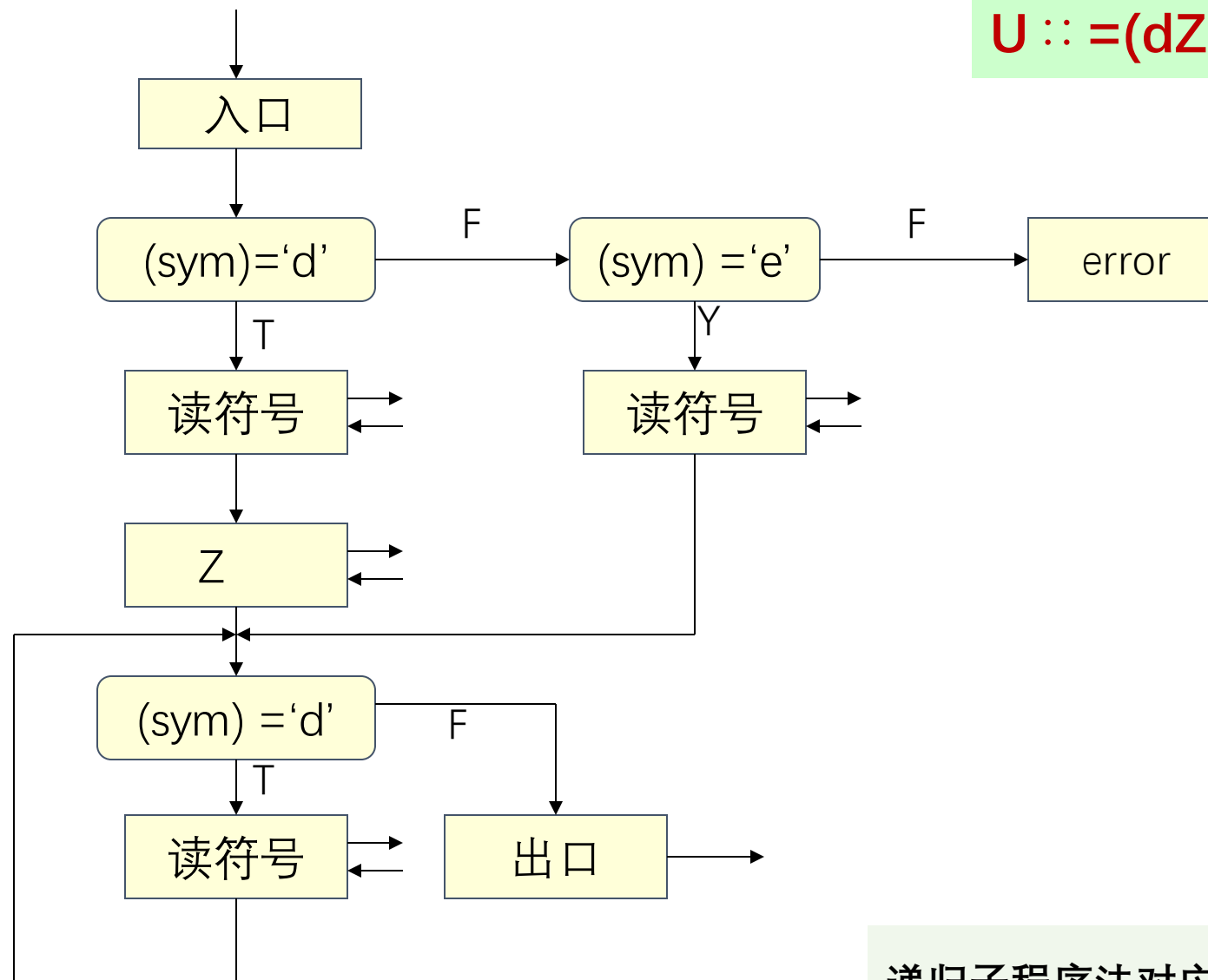
非终结符号的分析子程序的功能是：用规则右部符号串去匹配输入串。

$Z ::= ('U')|aUb$
 $U ::= (dZ|e)\{d\}$



在进入某个非终结符的分析程序时其所要分析的语法成分的第一个符号已读入sym中。在分析子程序的出口前，一定要读取下一个符号到sym中。

$Z ::= '('U')'|aUb$
 $U ::= (dZ|e)\{d\}$



递归子程序法对应的是最左推导过程

■ 递归下降分析法总结

优点: 1) 从适当的文法出发写一个递归下降分析器很容易.
2) 分析器和文法很相似,故分析器 的可靠性很高.

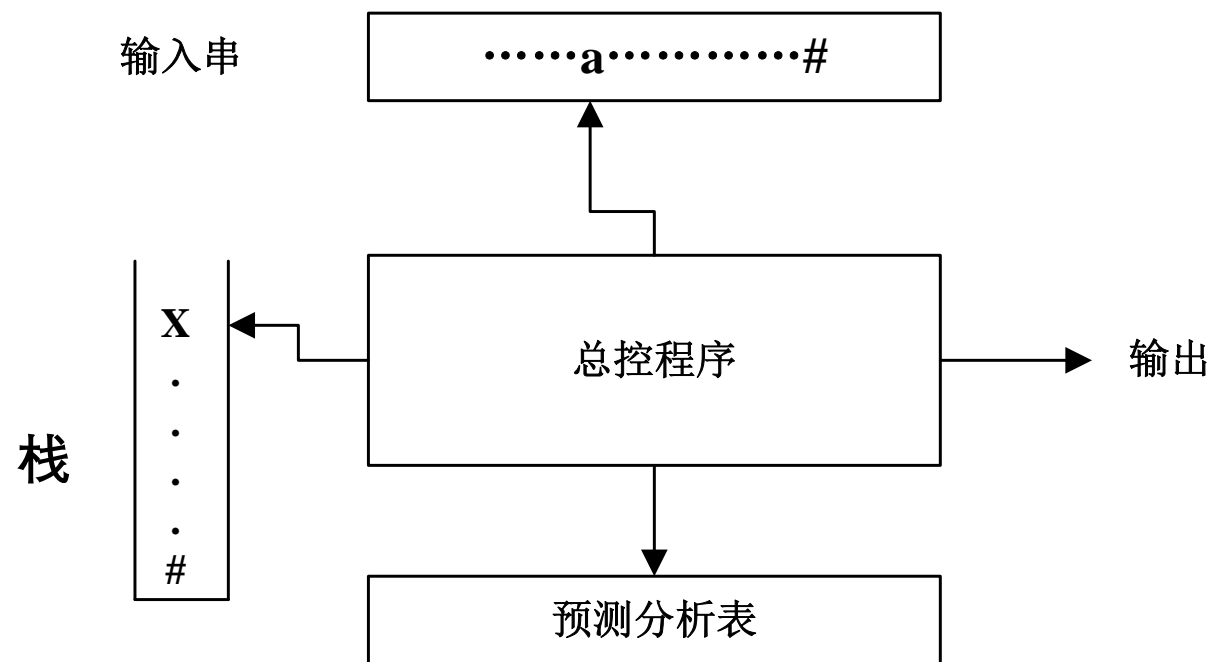
缺点: 1)因不断的递归调用,分析速度慢.
2)分析器本身比较大.

■ Outline

- 语法分析器的功能
- 自上而下分析面临的问题
- LL(1)分析法
- 递归下降分析程序构造
- 预测分析程序
- LL(1)分析中的错误处理

■ 预测分析程序工作过程

- 实现LL(1)分析的一种有效方法是使用一张分析表和一个栈进行联合控制。下面要介绍的预测分析程序就是属于这种类型的LL(1)分析器。



■ 预测分析表

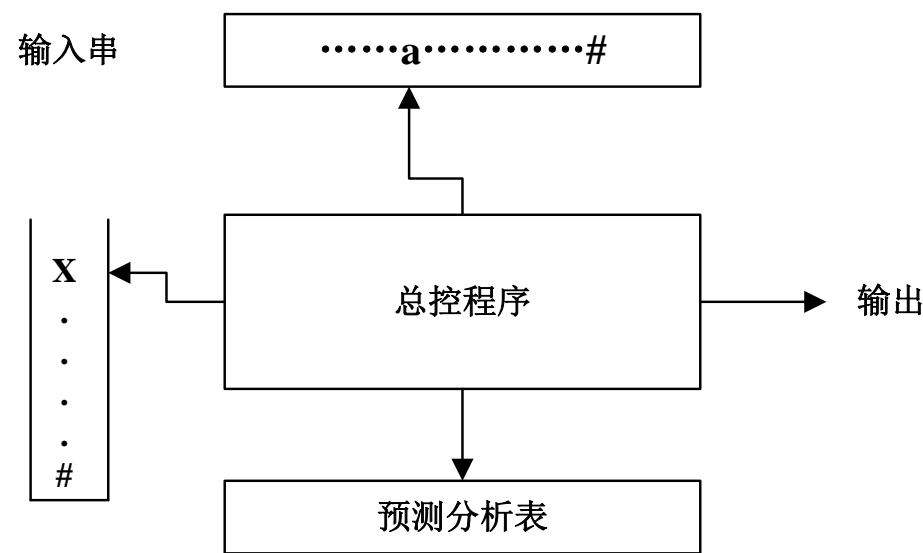
- **预测分析表**是一个 $M[A,a]$ 形式的矩阵。其中A为非终结符，a是终结符或‘#’。
- 矩阵元素 $M[A, a]$ 中存放着一条关于A的产生式，指出当A面临输入符号a时所应采用的候选。
- $M[A, a]$ 中也可能存放一个“出错标志”，指出A根本不该面临输入符号a。

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

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

■ 预测分析工作过程概述

- 预测分析程序的总控程序在任何时候都是按STACK栈顶符号X和当前的输入符号a行事的。如下图所示，对于任何(X, a)，总控程序每次都执行下述三种可能的动作之一：
 - 若 $X = a = \text{'\#'}$ ，则宣布分析成功，停止分析过程。
 - 若 $X = a \neq \text{'\#'}$ ，则把X从STACK栈顶弹出，让a指向下一个输入符号。
 - 若X是一个非终结符，则查看分析表M。
 - 若 $M[X, a]$ 中存放着关于X的一个产生式，那么，先把X弹出STACK栈顶，然后把产生式的右部符号串按**反序**——推进STACK栈（若右部符号为 ϵ ，则意味着不推任何符号进栈）。在把产生式的右部符号推进栈的同时应该做这个产生式对应的语义动作（目前暂且不管）。若 $M[X, a]$ 中存放着“出错标志”，则调用出错诊断程序ERROR。



■ 预测分析算法 I

- 输入:串 w ,文法 G 的分析表 M .
- 输出:如果 $w \in L(G)$,则产生 w 的最左推导,否则输出错误信息.
- 方法: 初态时,分析栈为 $\#S$,栈顶 S 是文法的开始符号;缓冲区为 $w\#$.
分析器按下列操作进行 语法分析:

预测分析算法二

```
(1) push #S; 指针ip指向串w#的第一个符号a;
(2) repeat
(3)     令X为栈顶符号,a为ip所指的符号;
(4)     if X为终结符或# then
(5)         if X=a=# then 接受w
(6)         else if X=a ≠ # then 弹出X,并使ip前进
(7)         else error
(8)     else /* X为非终结符*/
(9)         if M[X,a]=X→Y1Y2...Yk then
(10)             begin
(11)                 弹出X;
(12)                 push Yk,...,Y2,Y1 /* Y1在栈顶*/
(13)             end
(14)         else error
(15) until X=#; /* 栈为空 */
```


■ 预测分析工作过程举例

输入串 **i*i+i**

预测分析表

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

预测分析步骤

步骤	符号栈	输入串	所用产生式
0	#E	i*i+i#	
1	#E'T	i*i+i#	$E \rightarrow TE'$
2	#E'T'F	i*i+i#	$T \rightarrow FT'$
3	#E'T'i	i*i+i#	$F \rightarrow i$
4	#E'T'	*i+i#	
5	#E'T'F*	*i+i#	$T' \rightarrow *FT'$
6	#E'T'F	i+i#	
7	#E'T'i	i+i#	$F \rightarrow i$
8	#E'T'	+i#	
9	#E'	+i#	$T' \rightarrow \epsilon$
10	#E'T+	+i#	$E' \rightarrow +TE'$
11	#E'T	i#	
12	#E'T'F	i#	$T \rightarrow FT'$
13	#E'T'i	i#	$F \rightarrow i$
14	#E'T'	#	
15	#E'	#	$T' \rightarrow \epsilon$
16	#	#	$E' \rightarrow \epsilon$

■ Review

- 令 G 是一个不含左递归的文法，对 G 的所有非终结符的每个候选 α 定义它的终结首符集 $FIRST(\alpha)$ 为：
 - $FIRST(\alpha) = \{a \mid \alpha \xRightarrow{*} a\cdots, a \in V_T\}$
 - 若 $\alpha \xRightarrow{*} \varepsilon$ ，则规定 $\varepsilon \in FIRST(\alpha)$
- 假定 S 是文法 G 的开始符号，对于 G 的任何非终结符 A ，我们定义它的后随符号集 $FOLLOW(A)$ 为：
 - $FOLLOW(A) = \{a \mid S \xRightarrow{*} \cdots Aa\cdots, a \in V_T\}$
 - 若 $S \xRightarrow{*} \cdots A$ ，则规定 $\# \in FOLLOW(A)$

■ Review: LL(1)分析条件

- 满足构造**不带回溯**的**自顶向下**分析的文法条件。
 - 文法**不含左递归**
 - 对于文法中每一个非终结符A的各个产生式的**候选首符集**两两不相交。
即, 若 $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$, 则 $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \Phi \quad (i \neq j)$
 - 对文法中的每个非终结符A, 若它存在某个候选首符集包含 ϵ , 则,
 $\text{FIRST}(A) \cap \text{FOLLOW}(A) = \Phi$

如果一个文法G满足以上条件, 则称该文法G为**LL(1)**文法。

这里LL(1)中的第一个L表示从左到右扫描输入串, 第二个L表示最左推导, 1表示分析时每一步只需向前查看一个符号。

■ 构造FIRST集合的算法 I

对每一个文法符号 $X \in V_T \cup V_N$ 构造 $\text{FIRST}(X)$: 应用下列规则,直到每个集合 FIRST 不再增大为止.

(1)如果 $X \in V_T$,则 $\text{FIRST}(X)=\{X\}$.

(2)如果 $X \in V_N$,且有产生式 $X \rightarrow a...$,则把 a 加入到 $\text{FIRST}(X)$ 中;若 $X \rightarrow \varepsilon$ 也是一个产生式,则把 ε 加入到 $\text{FIRST}(X)$ 中.

(3)如果 $X \rightarrow Y...$ 是一个产生式且 $Y \in V_N$,则把 $\text{FIRST}(Y) \setminus \{\varepsilon\}$ 加到 $\text{FIRST}(X)$ 中;

如果 $X \rightarrow Y_1 Y_2 \dots Y_k$ 是一个产生式, $Y_1, \dots, Y_{i-1} \in V_N$,而且对任何 $j, j \in [1, i-1]$, $\varepsilon \in \text{FIRST}(Y_j)$, (即 $Y_1 Y_2 \dots Y_{i-1} \xRightarrow{*} \varepsilon$), 则把 $\text{FIRST}(Y_i) \setminus \{\varepsilon\}$ 加到 $\text{FIRST}(X)$ 中; 特别是, 若所有的 $\text{FIRST}(Y_j)$ 均含有 $\varepsilon, j=1, 2, \dots, k$, 则把 ε 加到 $\text{FIRST}(X)$ 中.

■ 构造FIRST集合的算法II

对文法G的任何符号串 $\alpha = X_1 X_2 \dots X_n$ 构造集合**FIRST(α)**

(1) 首先置FIRST(α) = FIRST(X_1) \setminus \{\epsilon\}.

(2) 如果对任何j, $j \in [1, i-1]$, $\epsilon \in \text{FIRST}(X_j)$, 则把FIRST(X_i) \setminus \{\epsilon\}加入到**FIRST(α)**中. 特别是, 若所有的**FIRST(X_j)**均含有 ϵ , $j=1, 2, \dots, n$, 则把 ϵ 加到**FIRST(α)**中.

■ FIRST集合构造的例子

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid i$

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

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

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

$\text{FIRST}(T) = \{(, i\}$

$\text{FIRST}(E) = \{(, i\}$

$\text{FIRST}(TE') = \{(, i\}$

$\text{FIRST}(+TE') = \{+\}$

$\text{FIRST}(FT') = \{(, i\}$

$\text{FIRST}(*FT') = \{*\}$

$\text{FIRST}((E)) = \{(\}$

$\text{FIRST}(i) = \{i\}$

■ 构造结合FOLLOW的算法

对文法G的每个非终结符A构造FOLLOW(A)的办法是：连续应用下列规则,直到每个后随符号集FOLLOW不再增大为止.

- 1) 对于文法的开始符号S, 置#于FOLLOW(S)中;
- 2) 若 $A \rightarrow \alpha B \beta$ 是一个产生式, 则把 $\text{FIRST}(\beta) \setminus \{\varepsilon\}$ 加至FOLLOW(B)中;
- 3) 若 $A \rightarrow \alpha B$ 是一个产生式, 或 $A \rightarrow \alpha B \beta$ 是一个产生式而 $\beta \Rightarrow \varepsilon$ (即 $\varepsilon \in \text{FIRST}(\beta)$), 则把FOLLOW(A)加至FOLLOW(B)中.

■ FOLLOW集合构造的例子

$E \rightarrow TE'$	$\text{FIRST}(E') = \{+, \epsilon\}$
$E' \rightarrow +TE' \mid \epsilon$	$\text{FIRST}(T') = \{*, \epsilon\}$
$T \rightarrow FT'$	$\text{FIRST}(F) = \{(, i\}$
$T' \rightarrow *FT' \mid \epsilon$	$\text{FIRST}(T) = \{(, i\}$
$F \rightarrow (E) \mid i$	$\text{FIRST}(E) = \{(, i\}$

- $\text{FOLLOW}(E) = \{), \# \}$
- $\text{FOLLOW}(E') = \{), \# \}$
- $\text{FOLLOW}(T) = \{ +,), \# \}$
- $\text{FOLLOW}(T') = \{ +,), \# \}$
- $\text{FOLLOW}(F) = \{ *, +,), \# \}$

■ 分析表的构造算法

• 构造分析表M的算法如下：

- (1) 对文法G的每个产生式 $A \rightarrow \alpha$, 执行第(2)和(3)步;
- (2) 对每个终结符 $a \in \text{FIRST}(\alpha)$, 把 $A \rightarrow \alpha$ 加入 $M[A, a]$ 中;
- (3) 若 $\epsilon \in \text{FIRST}(\alpha)$, 则对任何 $b \in \text{FOLLOW}(A)$, 把 $A \rightarrow \alpha$ 加入 $M[A, b]$ 中;
- (4) 把所有无定义的 $M[A, a]$ 标上“出错标志”。

用上述算法可以对任何文法G构造它的分析表M。但对于某些文法，有些 $M[A, a]$ 可能会有若干个产生式，或者说有些 $M[A, a]$ 是多重定义的。

如果G是左递归或二义的，那么M至少含有一个多重定义入口。因此消除左递归和提取左公因子有助于获得无多重定义的分析表

一个文法G的预测分析表M不含多重入口，当且仅当该文法为LL(1)的

■ 分析表的构造例子

$E \rightarrow TE'$	$FIRST(E) = \{ (, i \}$	$FIRST(TE') = \{ (, i \}$	$FOLLOW(E) = \{ \}, \# \}$
$E' \rightarrow +TE' \mid \epsilon$	$FIRST(E') = \{ +, \epsilon \}$	$FIRST(+TE') = \{ + \}$	$FOLLOW(E') = \{ \}, \# \}$
$T \rightarrow FT'$	$FIRST(T) = \{ (, i \}$	$FIRST(FT') = \{ (, i \}$	$FOLLOW(T) = \{ +, \}, \# \}$
$T' \rightarrow *FT' \mid \epsilon$	$FIRST(T') = \{ *, \epsilon \}$	$FIRST(*FT') = \{ * \}$	$FOLLOW(T') = \{ +, \}, \# \}$
$F \rightarrow (E) \mid i$	$FIRST(F) = \{ (, i \}$	$FIRST((E)) = \{ (\}$	$FOLLOW(F) = \{ *, +, \}, \# \}$
		$FIRST(i) = \{ i \}$	

- 构造分析表M的算法如下：
- (1) 对文法G的每个产生式 $A \rightarrow \alpha$, 执行第(2)和(3)步；
 - (2) 对每个终结符 $a \in FIRST(\alpha)$, 把 $A \rightarrow \alpha$ 加入 $M[A, a]$ 中；
 - (3) 若 $\epsilon \in FIRST(\alpha)$, 则对任何 $b \in FOLLOW(A)$, 把 $A \rightarrow \epsilon$ 加入 $M[A, b]$ 中；
 - (4) 把所有无定义的 $M[A, a]$ 标上“出错标志”。

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

■ Outline

- 语法分析器的功能
- 自上而下分析面临的问题
- LL(1)分析法
- 递归下降分析程序构造
- 预测分析程序
- LL(1)分析中的错误处理



错误处理概述

1. 必备功能之一

正确的源程序：通过编译生成目标代码

错误的源程序：通过编译发现并指出错误

2. 错误处理能力

- (1) 诊察错误的能力
- (2) 报错及时准确
- (3) 一次编译找出错误的多少
- (4) 错误的改正能力
- (5) 遏止重复的错误信息的能力


错误分类

从编译角度，将错误分为两类：**语法错误**和**语义错误**

语法错误：源程序在语法上不合乎文法

如：

A[I, J := B +* C



语义错误主要包括：**程序不符合语义规则**或
超越具体计算机系统的限制

语义规则：

- 标识符先说明后引用
- 标识符引用要符合作用域规定
- 过程调用时实参与形参要一致
- 参与运算的操作数类型一致
- 下标变量下标不能越界

超越系统限制：

- 数据溢出错误
- 符号表、静态存储分配数据区溢出
- 动态存储分配数据区溢出

■ 错误的诊察和报告

错误诊察：

1. 违反语法和语义规则以及超过编译系统限制的错误。

编译程序: 语法和语义分析时
(语义分析要借助符号表)

2. 下标越界，计算结果溢出以及动态存储数据区溢出。

目标程序: 目标程序运行时
编译程序要生成相应的目标程序
作检查和进行处理

错误报告：

1. 出错位置：即源程序中出现错误的位置

实现: 行号计数器 line_no
 单词序号计数器 char_no

一旦诊察出错误，当时的计数器内容就是出错位置

2. 出错性质：

可直接显示文字信息

可给出错误编码

■ 报告错误的两种方式

(1) 分析完以后再报告(显示或者打印)

编译程序可设一个保存错误信息的数据区(可用记录型数组), 将语法语义分析所诊察到的错误送数据区保存, 待源程序分析完以后, 显示或打印错误信息。

例: A[x, y := B+*C



源程序行号	错误序号	错误性质
x x	6	缺少 "]"
x x	10	表达式语法错误

(2) 边分析边报告

可以在分析一行源程序时若发现有错, 立即输出该行源程序, 并在其下输出错误信息。

Line - no A[x, y := B+ *C

缺 "]" or **n** 表达式语法错 **m**

错误编号

有时候报错不一定
十分准确 (位置和性质),
需进一步分析

```
begin
.....
i := 1 step 1 until n do
.....
end
```

错误处理技术

发现错误后，在报告错误的同时还要对错误进行处理，以方便编译能进行下去。目前有两种处理办法：

1. 错误改正：指编译诊察出错误以后，根据文法进行错误改正。

如：A[i, j] := B + *C

要正确地改正错误
是很困难的

但不是总能做到,如A:=B-C*D+E)

2. 错误局部化处理：指当编译程序发现错误后，尽可能把错误的影响限制在一个局部的范围，避免错误扩散和影响程序其他部分的分析。

(1) 一般原则：当诊察到错误以后，就暂停对后面符号的分析，跳过错误所在的语法成分然后继续往下分析。

词法分析：发现不合法字符，显示错误，并跳过该标识符(单词)继续往下分析。

语法规义分析：跳过所在的语法成分(短语或语句)，一般是跳到语句右界符，然后从新语句继续往下分析。

- (2) 错误局部化处理的实现（递归下降分析法）

CX：全局变量，存放错误信息。

•用递归下降分析时，如果发现错误，便将有关错误信息（字符串或者编号）送CX，然后转出错误处理程序；

•出错程序先打印或显示出错位置以及出错信息，然后跳出一段源程序，直到跳到语句的右界符（如：end）或正在分析的语法成分的合法后继符号为止，然后再往下分析。

目标程序运行时错误检测与处理

下标变量下标值越界
计算结果溢出
动态存储分配数据区溢出

- 在编译时生成检测该类错误的代码。

对于这类错误,要正确的报告出错误位置很难,因为
目标程序与源程序之间难以建立位置上的对应关系

一般处理办法:

当目标程序运行检测到这类错误时,就调用异常处理程序,打印错误信息和运行现场(寄存器和存储器中的值)等,然后停止程序运行。

■ LL(1)分析中的错误处理

- 错误情况
 - 栈顶的终结符与当前的输入符号不匹配。
 - 非终结符A处于栈顶，面临的输入符号为a，但分析表M中M[A, a]为空。
- 基本做法：
 - 就是跳过输入串中的一些符号直至遇到“同步符号”为止。这种做法的效果有赖于同步符号集的选择。

■ 同步符号集的选择

1. 把FOLLOW(A)中的所有符号放入非终结符A的同步符号集。如果我们跳读一些输入符号直至出现FOLLOW(A)中的同步符号，把A从栈中弹出来，这样就可能使分析继续下去。
2. 对于非终结符A来说，只用FOLLOW(A)作为它的同步符号集是不够的。例如，如果分号作为语句的终结符，那么作为语句开头的关键字就可能不在产生表达式的非终结符的FOLLOW集合中。这样，在一个赋值语句后少一个分号就可能导致作为下一个语句开头的关键字被跳过；
3. 如果把FIRST(A)中的符号加入非终结符A的同步符号集，那么当FIRST(A)中的一个符号在输入中出现时，可以根据A恢复语法分析；
4. 如果一个非终结符产生空串，那么推导 ϵ 的产生式可以作为缺省的情况，这样做可以推迟某些错误检查，但不能导致放弃一个错误。这种方法减少在错误恢复期间必须考虑的非终结符数；
5. 如果不能匹配栈顶的终结符号，一种简单的想法是弹出栈顶的这个终结符号，并发出一条信息，说明已经插入这个终结符，继续进行语法分析。结果，这种方法使一个单词符号的同步符号集包含所有其他单词符号。

■ 错误处理例子

对于改后的分析表，如果遇到 $M[A, a]$ 是空，则跳过输入符号 a ，若该项为“同步”且 A 不是初始状态，则弹出栈顶的非终结符；如果 A 是初始状态，则需要继续读入下一个输入符号，直至该项不为空或“同步”；若栈顶的终结符号不匹配输入符号，则弹出栈顶的终结符。

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

输入串： $)i*+i$

步骤	符号栈	输入串	附注
0	#E)i*+i#	错，跳过)
1	#E	i*+i#	i属于FIRST(E)
2	#E'T	i*+i#	$E \rightarrow TE'$
3	#E'T'F	i*+i#	$T \rightarrow FT'$
4	#E'T'i	i*+i#	$T \rightarrow i$
5	#E'T'	*+i#	
6	#E'T'F*	*+i#	$T' \rightarrow *FT'$
7	#E'T'F	+i#	错，同步，弹出F
8	#E'T'	+i#	
9	#E'	+i#	$T' \rightarrow \epsilon$
10	#E'T+	+i#	$E' \rightarrow +TE'$
11	#E'T	i#	
12	#E'T'F	i#	$T \rightarrow FT'$
13	#E'T'i	i#	$F \rightarrow i$
14	#E'T'	#	
15	#E'	#	$T' \rightarrow \epsilon$
16	#	#	$E' \rightarrow \epsilon$

■ 练习

考虑文法G1[E]:

$$E \rightarrow TE'$$
$$E' \rightarrow +E \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow T \mid \varepsilon$$
$$F \rightarrow PF'$$
$$F' \rightarrow *F' \mid \varepsilon$$
$$P \rightarrow (E) \mid a \mid b \mid \&$$

- (1) 计算该文法每个非终结符的FIRST和FOLLOW
- (2) 证明这个文法是LL(1)的
- (3) 构造该文法的预测分析表

■ 练习

考虑文法G1[E]:

$$E \rightarrow TE'$$
$$E' \rightarrow +E \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow T \mid \varepsilon$$
$$F \rightarrow PF'$$
$$F' \rightarrow *F' \mid \varepsilon$$
$$P \rightarrow (E) \mid a \mid b \mid \&$$

- (1) 计算该文法每个非终结符的FIRST和FOLLOW
- (2) 证明这个文法是LL(1)的
- (3) 构造该文法的预测分析表

$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \text{FIRST}(P) = \{ (, a, b, \& \}.$

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

$\text{FIRST}(T') = \{ (, a, b, \&, \varepsilon \}$

$\text{FIRST}(F') = \{ *, \varepsilon \}$

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

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

$\text{FOLLOW}(F) = \text{FOLLOW}(F') = \{ (, a, b, \&, +,), \# \}$

$\text{FOLLOW}(P) = \{ *, (, a, b, \&, +,), \# \}$

■ 练习

$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \text{FIRST}(P) = \{ (, a, b, \& \}$
 $\text{FIRST}(E') = \{ +, \epsilon \}$
 $\text{FIRST}(T') = \{ (, a, b, \&, \epsilon \}$
 $\text{FIRST}(F') = \{ *, \epsilon \}$
 $\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{), \# \}$
 $\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{ +,), \# \}$
 $\text{FOLLOW}(F) = \text{FOLLOW}(F') = \{ (, a, b, \&, +,), \# \}$
 $\text{FOLLOW}(P) = \{ *, (, a, b, \&, +,), \# \}$

$\text{FIRST}(TE') = \text{FIRST}(T) = \{ (, a, b, \& \}$
 $\text{FIRST}(+E) = \{ + \}$
 $\text{FIRST}(FT') = \text{FIRST}(F) = \{ (, a, b, \& \}$
 $\text{FIRST}(PF') = \text{FIRST}(P) = \{ (, a, b, \& \}$
 $\text{FIRST}(*F') = \{ * \}$ $\text{FIRST}((E)) = \{ (\}$

LL(1) 文法要满足以下三个条件:

- 文法不含左递归 满足
- 对文法的每一个非终结符A的所有产生式, 其候选首符集两两不相交: 只需考察E', F', F', P 的所有产生式的候选首符集合 满足
- 对文法的每一个非终结符A的某个产生式, 若存在某个候选首符集包含 ϵ , 则
 $\text{FIRST}(A) \cap \text{FOLLOW}(A) = \phi$, 考察E', T', F' 满足
 故而该文法是LL(1)的

	+	*	()	a	b	&	#
E			$E \rightarrow TE'$		$E \rightarrow TE'$	$E \rightarrow TE'$	$E \rightarrow TE'$	
E'	$E' \rightarrow +E$			$E' \rightarrow \epsilon$				$E' \rightarrow \epsilon$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	$T \rightarrow FT'$	$T \rightarrow FT'$	
T'	$T' \rightarrow \epsilon$		$T' \rightarrow T$	$T' \rightarrow \epsilon$	$T' \rightarrow T$	$T' \rightarrow T$	$T' \rightarrow T$	$T' \rightarrow \epsilon$
F			$F \rightarrow PF'$		$F \rightarrow PF'$	$F \rightarrow PF'$	$F \rightarrow PF'$	
F'	$F' \rightarrow \epsilon$	$F' \rightarrow *F'$	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$
P			$P \rightarrow (E)$		$P \rightarrow a$	$P \rightarrow b$	$P \rightarrow \&$	

■ 自顶向下的语法分析总结

- 自顶向下的分析存在的问题
 - 左递归问题→消除左递归的方法
 - 回溯问题→消除回溯
- 自顶向下的分析方法
 - LL(1)分析法
 - LL(1)分析器的逻辑结构及工作过程
 - LL(1)文法的条件
 - 递归下降和预测分析
 - 分析表的构造方法 (FIRST, FOLLOW)

阅读材料：《程序设计语言编译原理（第3版）》，陈火旺等编著，国防工业出版社，2004年，第四章
《编译原理与技术》张莉等编著，第四章