School of Computer Science & Technology
Harbin Institute of Technology

第四章 自顶向下的语法分析

重点： 自顶向下分析的基本思想，预测分析器总体结构，预测分析表的构造，递归下降分析法基本思想，简单算术表达式的递归下降分析器。

难点： FIRST 和 FOLLOW 集的求法，对它们的理解以及在构造LL(1)分析表时的使用。递归子程序法中如何体现分析的结果。

回顾：推导或归约

- 定义2.17 设$G=(V，T，P，S)$是一个文法，如果$\alpha\rightarrow\beta\in P$，$\gamma$，$\delta\in(V\cup T)^*$，则称$\gamma\alpha\delta$在$G$中**直接推导**出$\gamma\beta\delta$，记作：$\gamma\alpha\delta \underset{G}{\Rightarrow} \gamma\beta\delta$

- 与之相对应，也可以称$\gamma\beta\delta$在文法G**中直接归约**成$\gamma\alpha\delta$。

回顾：最左推导与最右推导

- 最左推导(Left-most Derivation)
  - 每次推导都施加在句型的最左边的语法变量上。——与最右归约对应

$$E \Rightarrow E+E$$
$$\Rightarrow \mathrm{id}+E$$
$$\Rightarrow \mathrm{id}+E*E$$
$$\Rightarrow \mathrm{id}+\mathrm{id}*E$$
$$\Rightarrow \mathrm{id}+\mathrm{id}*\mathrm{id}$$

回顾：最左推导与最右推导

- 最右推导(Right-most Derivation)
  - 每次推导都施加在句型的最右边的语法变量上。——与最左归约对应
  - 对应的规范(Canonical)句型

$$E \Rightarrow E+E$$
$$\Rightarrow E+E*E$$
$$\Rightarrow E+E*id$$
$$\Rightarrow E+id*id$$
$$\Rightarrow id+id*id$$

第4章 自顶向下的语法分析

4.1 语法分析概述

4.2 自顶向下的语法分析面临的问题与文法的改造

4.3 预测分析法

4.4 递归下降分析法

4.5 本章小结

语法分析的功能和位置

- **语法分析(syntax analysis)是编译程序的核心部分，其任务是检查词法分析器输出的单词序列是否是源语言中的句子，亦即是否符合源语言的语法规则。**

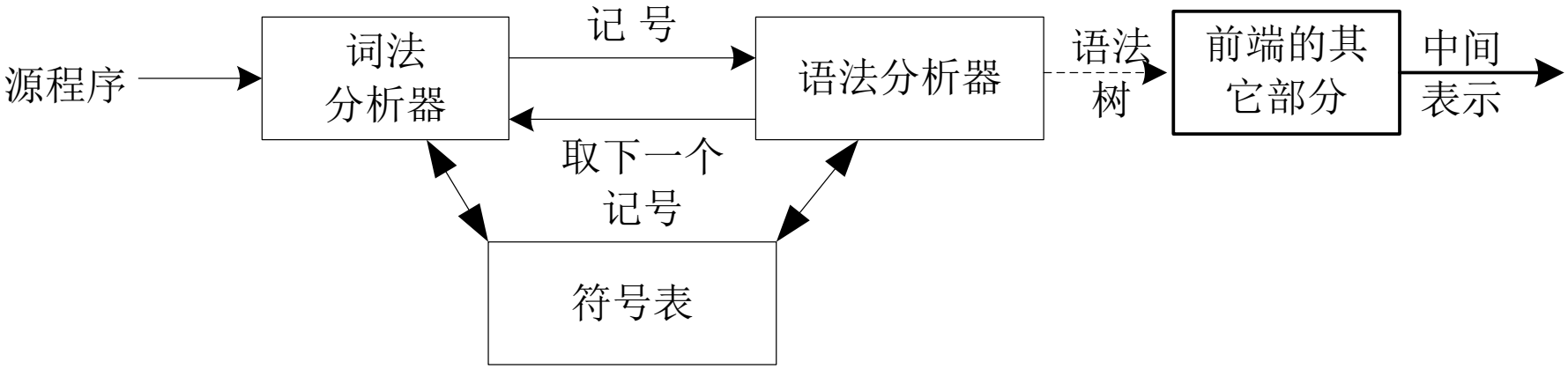


图4.1　语法分析器在编译器中的位置

语法分析的功能和位置

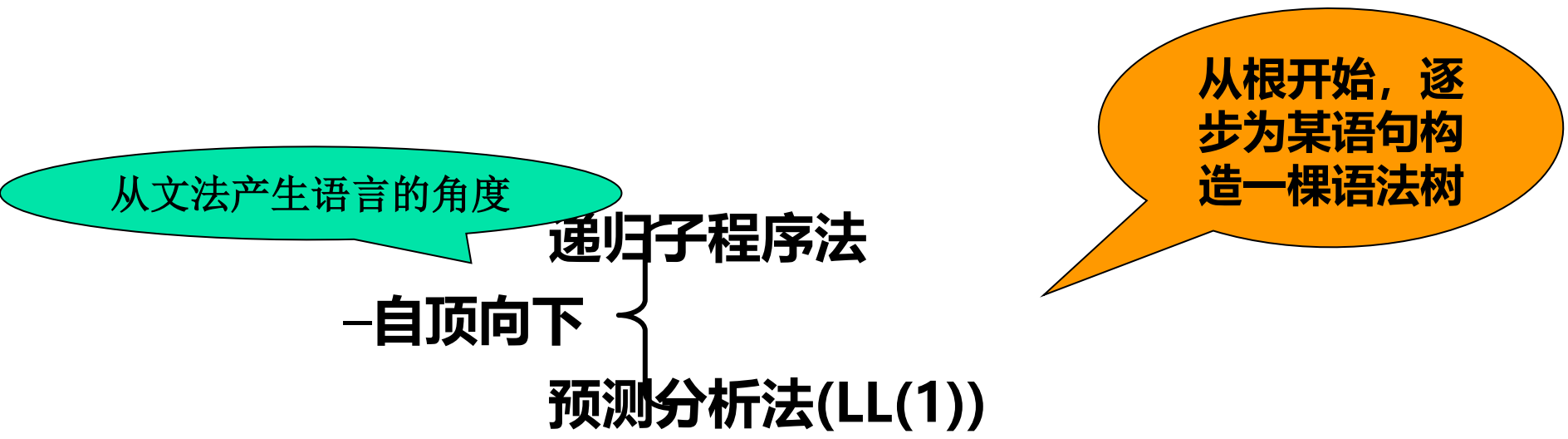
- 语言是满足一定**组成规则**的句子集合，句子是满足一定**组成规则**的单词序列，单词则是满足一定**组成规则**的字符串

- 这些组成规则就是文法中的产生式

- 语法分析的主要工作是根据源语言的文法，判别**某个单词序列**是否是源语言的一个句子

4.1 语法分析概述

- 判断某个单词序列是否是源语言的句子，主要有两种方式：
  - **产生句子**的方式：从文法的开始符号开始，逐步**推导**出这个单词序列，也称为自顶向下的语法分析



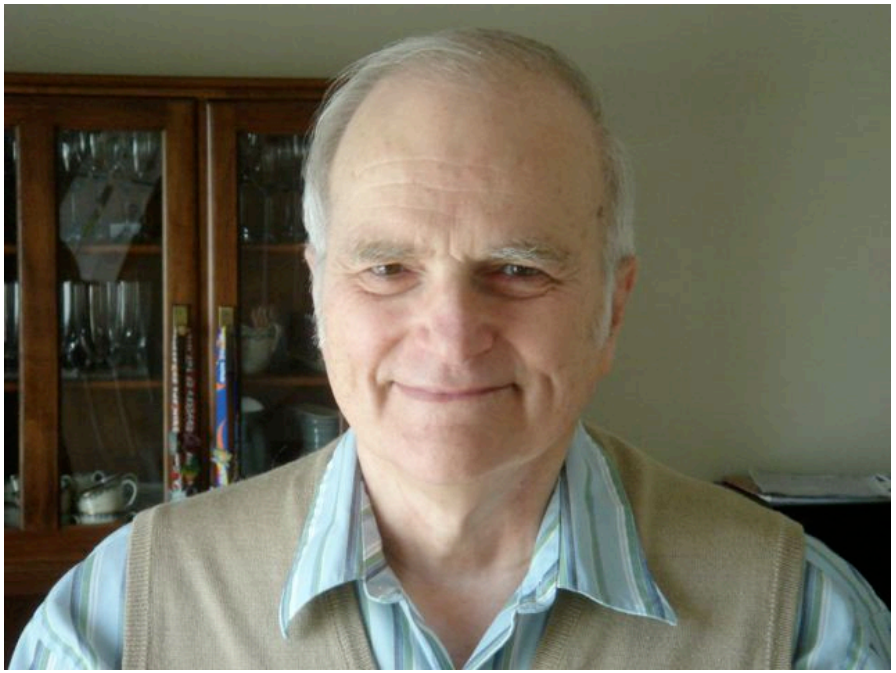
4.1 语法分析概述

- 判断某个单词序列是否是源语言的句子，主要有两种方式：
  - 识别句子的方式：逐步将构成程序的单词序列归约为文法的开始符号，也称为自底向上的语法分析

介绍的材料

自顶向下推导(**LL**文法)：Philip M. Lewis and Richard Stearns

P. M. Lewis II, R. E. Stearns: Syntax-Directed Transduction, *Journal of the ACM*, **15**(3): 465-488 (1968)

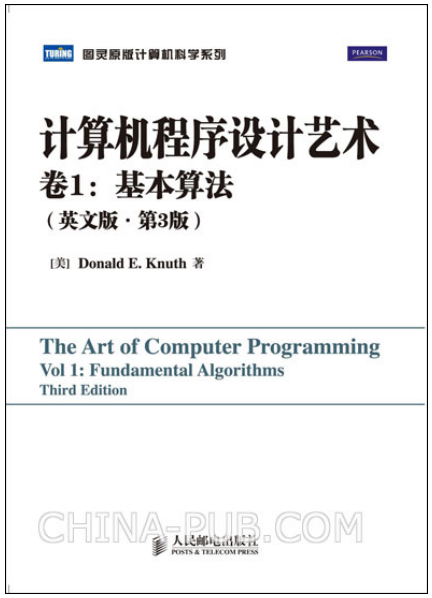


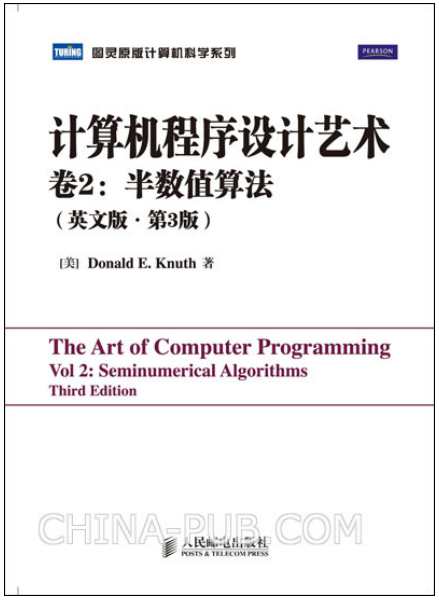
介绍的材料

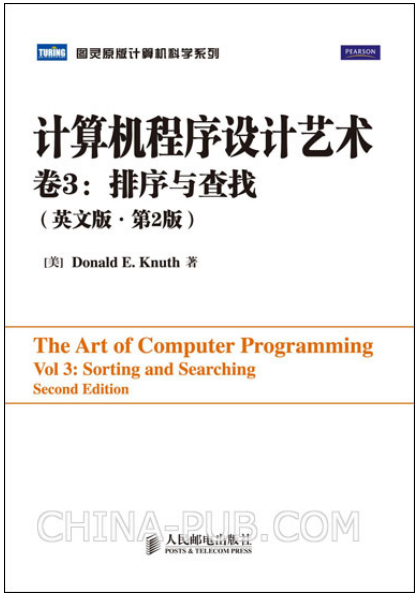
自底向上归约**(LR**文法**)**：Donald Knuth

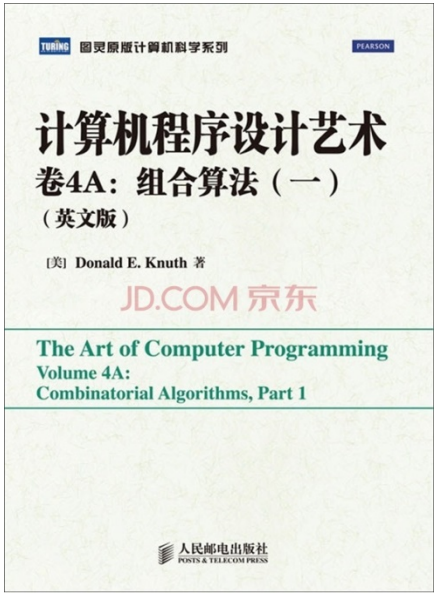
D.E.Knuth,On the translation of languages from left to right, *Information and Control* , 8, 607-639 (1965)

Donald Knuth(高德纳)

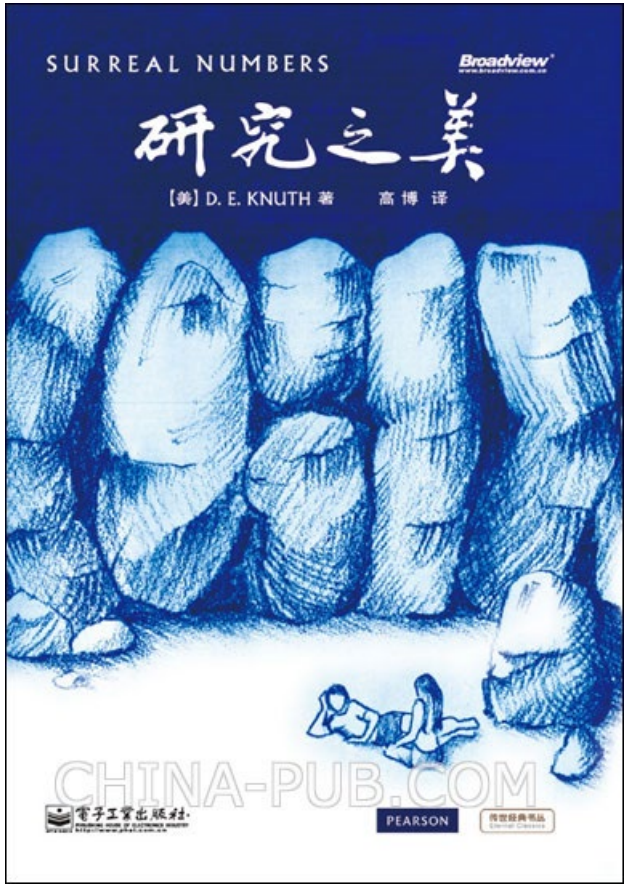


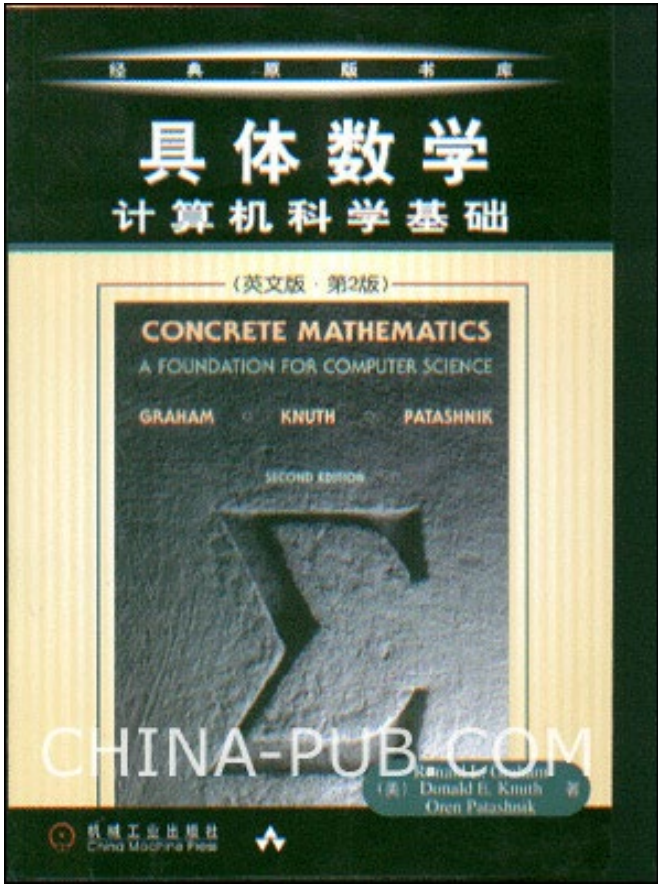






Donald Knuth(高德纳)





4.1 语法分析概述

- 无论是自顶向下还是自底向上，语法分析器都是自左到右地扫描输入单词序列，每次读入一个单词，针对输入单词序列建立一颗语法分析树。
- 不同的分析方法对应着不同的构建树的方式
- 手工构建（**LL**文法）或自动生成（**LR**文法）

4.1 语法分析概述

语法分析处理时的恢复策略

1. 紧急方式恢复策略：丢弃输入记号，直到发现某个指定的同步记号为止。同步记号通常是定界符（分号或**end**标记），标志着一条新语句的开始。
2. 短语级恢复策略：对剩余输入做局部纠正

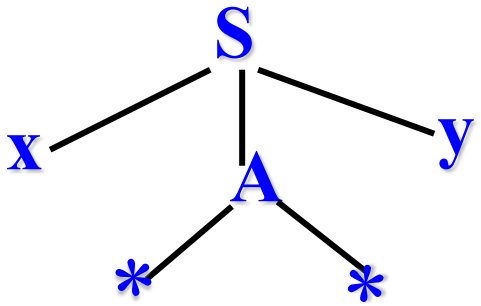
4.2 自顶向下的语法分析面临的问题与文法的改造

- **自顶向下语法分析的基本思想**
  - **从文法的开始符号出发，试图为输入符号串的一个最左推导。**
  - **即从树根S开始，构造出能够得到输入串的语法树**
- **例:设有G：S→xAy, A→**|*，输入串：x**y**

语法分析器是自左向右地扫描输入单词序列的

$S \Rightarrow xAy$

$\Rightarrow x**y$



4.2.1 自顶向下分析面临的问题

- 自顶向下分析实际上是一种**试探性**的过程，可能导致分析效率极低甚至失败
- 下面依次看自顶向下分析面临的三种问题

4.2.1 自顶向下分析面临的问题

1.二义性问题

- 对于文法G，如果$L(G)$中存在一个具有两棵或两棵以上分析树的句子，则称G是二义性的。
- 也可以等价地说：如果$L(G)$中存在一个具有两个或两个以上最左(或最右)推导的句子，则G是二义性文法。
- 如果一个文法G是二义性的，假设$w \in L(G)$且w**存在两个最左推导**，则在对w进行自顶向下的语法分析时，语法分析程序将**无法确定**采用w的哪个最左推导。

4.2.1 自顶向下分析面临的问题

1.二义性问题

- G_{12}: E → id | c | E + E | E – E | E * E | E / E | E **E | (E)
- 句子id_1 + c * id_2
- 对应两棵不同的语法树

4.2.1 自顶向下分析面临的问题

1.二义性问题

- G_{12}: E → id | c | E + E | E – E | E * E | E / E | E **E | (E)
- 解决办法1：改造文法，引入新的文法变量
- G_{exp}:

$$E \rightarrow E+T \mid E-T \mid T$$
$$T \rightarrow T*F \mid T/F \mid F$$
$$F \rightarrow F \uparrow P \mid P$$
$$P \rightarrow c \mid \text{id} \mid (E)$$

4.2.1 自顶向下分析面临的问题

1.二义性问题

- G_{12}: E → id | c | E + E | E – E | E * E | E / E | E **E | (E)
- 解决办法2：根据优先级关系，保证高优先级运算符优先的原则
- 乘法优先原则：句子id_1 + c * id_2

4.2.1 自顶向下分析面临的问题

2.回溯问题

- 文法中每个语法变量A的产生式右部称为A的**候选式**
- 如果A有多个候选式存在**公共前缀**，则自顶向下的语法分析程序将无法根据当前输入符号准确地选择用于推导的产生式，只能试探。
- 当试探不成功时就需要退回到上一步推导，看A是否还有其它的候选式，这就是**回溯**(backtracking)。

4.2.1 自顶向下分析面临的问题

2.回溯问题

- G_e：
  E→T
  E→E+T
  E→E-T
  T→F
  T→T*F
  T→T/F
  F→(E)
  F→id

- 例如：考虑为输入串id+id*id建立最左推导

4.2.1 自顶向下分析面临的问题

2.回溯问题

我们将采用提取左因子的方法来改造文法，以便减少推导过程中回溯现象的发生

当然，单纯通过提取左因子无法彻底避免回溯现象的发生。

4.2.1 自顶向下分析面临的问题

3.左递归引起的无穷推导问题

- **如果存在推导$A \overset{+}{\Rightarrow} \alpha A\beta$，则称文法$G$是递归的，当$\alpha=\varepsilon$时称之为左递归；**
- **如果$A \overset{+}{\Rightarrow} \alpha A\beta$至少需要两步推导，则称文法$G$是间接递归的，当$\alpha=\varepsilon$时称之为间接左递归；**
- **如果文法G中存在形如$A\rightarrow\alpha A\beta$的产生式，则称文法$G$是直接递归的，当$\alpha=\varepsilon$时称之为直接左递归。**

4.2.1 自顶向下分析面临的问题

3.左递归引起的无穷推导问题

- G_{er}: $E \rightarrow E+T$

  $E \rightarrow E-T$

  $E \rightarrow T$

  $T \rightarrow F$

  $T \rightarrow T*F$

  $T \rightarrow T/F$

  $F \rightarrow (E)$

  $F \rightarrow id$

- 考虑为输入串id+id*id建立一个最左推导

4.2.2 对上下文无关文法的改造

1.二义性问题

2.左递归引起的无穷推导问题

3.回溯问题

4.2.2 对上下文无关文法的改造

1.消除二义性

- 改造的方法就是通过引入新的语法变量等，使文法含有更多的信息。
- 许多二义性文法是由于概念不清，即语法变量的定义不明确导致的，此时通过引入新的语法变量即可消除文法的二义性。

4.2.2 对上下文无关文法的改造

1.消除二义性

- G_{12}: E → id | c | E + E | E – E | E * E | E / E | E **E | (E)

- 对G_{exp}:

$$E \rightarrow E+T \mid E-T \mid T$$
$$T \rightarrow T*F \mid T/F \mid F$$
$$F \rightarrow F\uparrow P \mid P$$
$$P \rightarrow c \mid \text{id} \mid (E)$$

G_{exp}比G_{12}提供了更多的信息

4.2.2 对上下文无关文法的改造

1.消除二义性

- <*stmt*>→ if <*expr*> then <*stmt*>
  | if <*expr*> then <*stmt*> else <*stmt*>
  | other

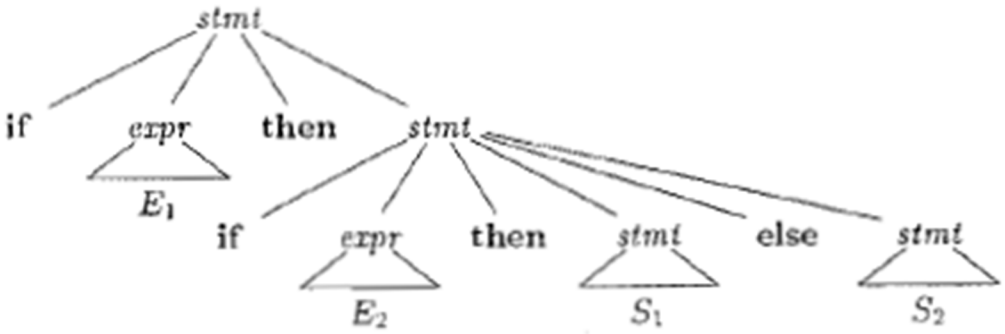
- 根据if语句中else与then配对情况将其分为配对的语句和不配对的语句两类。上述if语句的文法没有对这两个不同的概念加以区分，只是简单地将它们都定义为<stmt>，从而导致该文法是二义性的。

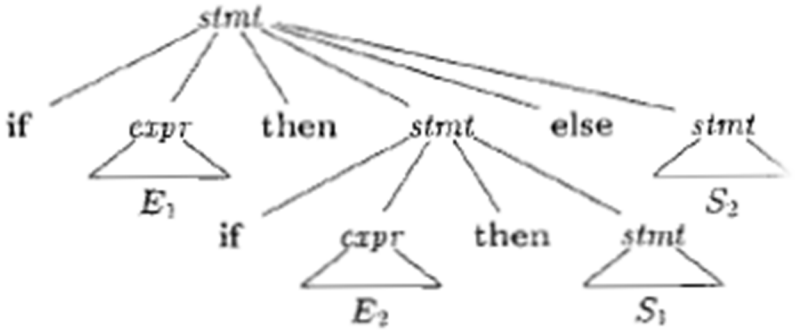
- if e_1 then if e_2 then s_1 else s_2

4.2.2 对上下文无关文法的改造

1.消除二义性

- <*stmt*>→ if <*expr*> then <*stmt*>
  | if <*expr*> then <*stmt*> else <*stmt*>
  | other
- if e_1 then if e_2 then s_1 else s_2





4.2.2 对上下文无关文法的改造

引入语法变量<*unmatched_stmt*>来表示不配对语句，<*matched_stmt*>表示配对语句

- <*stmt*>→ <*matched_stmt*> | <*unmatched_stmt*>

<*matched_stmt*>→ if <*expr*> then <*matched_stmt*> else <*matched_stmt*> | other

<*unmatched_stmt*>→ if <*expr*> then <*matched_stmt*>

| if <*expr*> then <*matched_stmt*> else <*unmatched_stmt*>

4.2.2 对上下文无关文法的改造

2.消除左递归

$A \overset{+}{\Rightarrow} \alpha A\beta$，当$\beta = \varepsilon$时称之为右递归

- 直接左递归的消除(转换为右递归)
- 引入新的变量A'，将左递归产生式$A \to A\alpha | \beta$替换为$A \to \beta A'$　　$A' \to \alpha A' | \varepsilon$

4.2.2 对上下文无关文法的改造

2.消除左递归

- $E \rightarrow E+T|T \quad T \rightarrow T*F|F \quad F \rightarrow (E)|\text{id}$替换为：
- $E \rightarrow TE' \quad E' \rightarrow +TE'|\varepsilon \quad T \rightarrow FT' \quad T' \rightarrow *FT'|\varepsilon$

  $F \rightarrow (E)|\text{id}$

4.2.2 对上下文无关文法的改造

2.消除左递归

- 一般地，假设文法G中的语法变量A的所有产生式如下：$A\rightarrow A\alpha_1|A\alpha_2|\ldots|A\alpha_n|\beta_1|\beta_2|\ldots|\beta_m$

  其中，$\beta_i(i=1,2,\ldots,m)$不以A打头。则用如下的产生式代替A的所有产生式即可消除其直接左递归：

  $A\rightarrow\beta_1A'|\beta_2A'|\ldots|\beta_mA'$　　$A'\rightarrow\alpha_1A'|\alpha_2A'|\ldots|\alpha_nA'|\varepsilon$

4.2.2 对上下文无关文法的改造

2.消除左递归

- 上述方法只能消除直接左递归，无法消除间接左递归
- S → Ac | c
  A → Bb | b
  B → Sa | a

4.2.2 对上下文无关文法的改造

2.消除左递归

- 上述方法只能消除直接左递归，无法消除间接左递归
- 消除间接左递归的基本思想：为语法变量编号，再采用带入法将间接左递归变为直接左递归，然后采用上述方法来消除直接左递归

4.2.2 对上下文无关文法的改造

算法4.1 消除左递归。

输入：不含循环推导和ε-产生式的文法G;

输出：与G等价的无左递归文法;

步骤：

循环推导即$A \overset{+}{\Rightarrow} A$，

ε-产生式即$A \overset{+}{\Rightarrow} \varepsilon$

1．将G的所有语法变量排序(编号)，假设排序后的语法变量记为 A_1，A_2，…，A_n;

2．for i←1 to n {

3．　for j←1 to i-1 {

4．　对每个形如$A_i \rightarrow A_j\beta$的产生式，其中，$A_j \rightarrow \alpha_1|\alpha_2|\ldots|\alpha_k$是所有当前$A_j$产生式，用产生式$A_i \rightarrow \alpha_1\beta|\alpha_2\beta|\ldots|\alpha_k\beta$替换

5．　}

6．　消除A_i产生式中的所有直接左递归

7．　}

4.2.2 对上下文无关文法的改造

算法4.1 消除左递归。

输入：不含循环推导和ε-产生式的文法G。

输出：与G等价的无

步骤：

1．**将G的所有语法变** **为**
   $A_1, A_2, \ldots, A$

2．for $i \leftarrow 1$ to n {

3．　for $j \leftarrow 1$ to $i-1$ {

4．　**对每个形如$A_i \rightarrow A_j\alpha$的产生式，$A_j$是所有当前$A_j$产生式，用产生式$A_i \rightarrow \alpha_1\beta|\alpha_2\beta|\ldots|\alpha_k\beta$替换**

5．　}

6．　**消除A_i产生式中的所有直接左递归**

7．　}

i=1时，3-5行的循环体不执行，此时执行第6行的消除左递归操作，消除了所有A_1变量的直接左递归

此时，A_1的所有的具有$A_1 \rightarrow A_b\alpha$形式的产生式，必有b>1

4.2.2 对上下文无关文法的改造

算法4.1 消除左

输入：不含循

输出：与*G*等价

步骤：

1．将*G*的所有

A_1，A_2，

2．for *i*←1 to

3．　for *j*←1 t

4．　对每个形如$A_i \rightarrow A_j\beta$的产生式，其中，$A_j \rightarrow \alpha_1|\alpha_2|\ldots|\alpha_k$是所有当前$A_j$产生式，用产生式$A_i \rightarrow \alpha_1\beta|\alpha_2\beta|\ldots|\alpha_k\beta$替换

5．　}

6．　消除A_i产生式中的所有直接左递归

7．　}

再看i=2，A_2的右部最左元素是变量的产生式形式可能是$A_2 \rightarrow A_1\alpha \mid A_2\beta \mid A_3\gamma|\ldots\ldots$

已知，当前的A_1的右部最左元素是变量的产生式形式是$A_1 \rightarrow A_2\alpha' \mid A_3\beta'|\ldots\ldots$

经过第4行的替换操作， A_2的所有的具有$A_2 \rightarrow A_b\alpha$形式的产生式，必有b>=2

执行第6行的消除直接左递归后，A_2的所有具有$A_2 \rightarrow A_b\alpha$形式的产生式，必有b>2

4.2.2 对上下文无关文法的改造

算法4.1 消除左递归。

输入：不含循环推

输出：与G等价的

步骤：

1. **将G的所有语**
   A_1，A_2，·
2. **for** i←1 **to** n
3. **for** j←1 t
4. **对每个**
   有当前A_j产
5. }
6. **消除A_i产生式中的所有直接左递归**
7. }

i=3，A_3的右部最左元素是变量的产生式形式可能是$A_3 \rightarrow A_1\alpha \mid A_2\beta \mid A_3\gamma \mid \ldots\ldots$

已知，当前的A_1的右部最左元素是变量的产生式形式是$A_1 \rightarrow A_2\alpha' \mid A_3\beta' \mid \ldots\ldots$，

当前的A_2的右部最左元素是变量的产生式形式是$A_2 \rightarrow A_3\alpha' \mid A_4\beta' \mid \ldots\ldots$

经过第4行的替换操作，A_3的所有的具有$A_3 \rightarrow A_b\alpha$形式的产生式，必有b>=3

执行第6行的消除直接左递归后，A_3的所有具有$A_3 \rightarrow A_b\alpha$形式的产生式，必有b>3

4.2.2 对上下文无关文法的改造

算法4.1 消除左递归。

输入：不含循环推导和ε-产生式的文法G;

输出：与G等价的无左递归文法。

步骤：

1. 将G的所有语
   A_1，A_2，…
2. for i←1 to n {
3. for j←1 to
4. 对每个形
   有当前A_j产生式，用产生式$A_i \to \alpha_1\beta|\alpha_2\beta|\ldots|\alpha_k\beta$替换
5. }
6. 消除A_i产生式中的所有直接左递归
7. }

以此类推，i=n，经过第4行的替换操作，A_n的所有的具有$A_n \to A_b\alpha$形式的产生式，必有b>=n

执行第6行的消除直接左递归后，A_n的所有具有$A_n \to A_b\alpha$形式的产生式，必有b>n

换句话说，已经消除了左递归

4.2.2 对上下文无关文法的改造

2.消除左递归

- S → Ac | c，A → Bb | b, B → Sa | a
- 语法变量排序：B、A、S
- i = 1时，B → Sa | a 无直接左递归
- i = 2时，A →Bb | b，替换变量B后得到，
  A→Sab|ab|b
- i = 3时，S→Ac|c，替换变量A后得到
  S→Sabc|abc|bc|c，执行消除直接左递归操作

4.2.2 对上下文无关文法的改造

- 3.提取左因子
  - 对每个语法变量A，找出它的两个或更多候选式的最长公共前缀α。如果$\alpha \neq \varepsilon$，则用下面的产生式替换所有的A产生式$A \to \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma_1|\gamma_2|\ldots|\gamma_n$，其中$\gamma 1$，$\gamma 2$，…，$\gamma n$表示所有不以$\alpha$开头的候选式：

$$A \to \alpha A'|\gamma_1|\gamma_2|\ldots|\gamma_n$$

$$A' \to \beta_1|\beta_2|\ldots|\beta_n$$

  - 其中，A'是新引入的语法变量。反复应用上述变换，直到任意语法变量都没有两个候选式具有公共前缀为止。

4.2.2 对上下文无关文法的改造

- **3.提取左因子**
  - <*stmt*>→ if <*expr*> then <*stmt*>
    | if <*expr*> then <*stmt*> else <*stmt*>
  - <*stmt*>→ if <*expr*> then <*stmt*><*stmt_new*>
    <*stmt_new*> → ε | *else* <*stmt*>
  - **提取左因子并不能完全消除回溯**

4.2.2 对上下文无关文法的改造

- **希望彻底消除回溯，实现确定的自顶向下分析的文法要求**
  - **1. 无二义性**
  - **2. 无左递归**
  - **3. 任意一个语法变量A的各个候选式所能推导出的第一个终结符必须各不相同**

LL(1)文法就是可以彻底消除回溯实现确定自顶向下分析的文法

4.2.3 LL(1)文法

- 不确定的自顶向下分析
  - 分析需要回溯，导致分析存在不确定性
  - 代价高、效率低，实际中几乎不被采用
- 确定的自顶向下分析
  - 不能处理所有文法，这里讨论什么样的文法可以进行确定的自顶向下分析

4.2.3 LL(1)文法

问题：什么样的文法对其句子才能进行确定的自顶向下分析?

- 确定的自顶向下分析首先从文法的开始符号出发，每一步推导都根据当前句型的最左语法变量A和当前输入符号a，选择A的某个候选式$\alpha(alpha)$来替换A，并使得从$\alpha(alpha)$推导出的第一个终结符恰好是a。
- 当A有多个候选式时，当前选中的候选式必须是唯一的。
- 第一个终结符是指符号串的第一个符号，并且是终结符号，可以称为首符号。在自顶向下的分析中，它对选取候选式具有重要的作用。为此引入首符号集(FIRST)的概念。

4.2.3 LL(1)文法

1. 假设α是文法$G=(V, T, P, S)$的符号串，即$\alpha\in(V\cup T)^*$，从α推导出的串的首符号集FIRST(α)记作：

$FIRST(\alpha)=\{a|\alpha \overset{*}{\Rightarrow} a\beta, a\in T, \beta\in(V\cup T)^*\}$。

2. 如果$\alpha \overset{*}{\Rightarrow} \varepsilon$，则$\varepsilon\in FIRST(\alpha)$。

3. 如果文法G中的所有A产生式为$A\rightarrow\alpha_1|\alpha_2|\ldots|$

且$\varepsilon\notin FIRST(\alpha_1)\cup FIRST(\alpha_2)\cup\ldots\cup FIRST(\alpha_m)$

且对$\forall i, j, 1\le i, j\le m; i\neq j$，均有

$FIRST(\alpha_i)\cap FIRST(\alpha_j)=\varnothing$成立，则可以对$G$的句子进行确定的自顶向下分析

引入后随符号集的概念

4.2.3 LL(1)文法

- 如果存在$\alpha_i \overset{*}{\Rightarrow} \varepsilon$形式的产生式A→ α_i ，给定当前终结符是a
  - 如果A的所有候选式的首符号集都不含有a，那么可能选择$A \rightarrow \varepsilon$来推导，因为a可能是当前句型中紧跟A后出现
  - 如果某个候选式α_j的首符号集包括a，并且a可能是当前句型中紧跟A后出现，还是无法确定到底选择α_i 还是α_j 来进行推导

4.2.3 LL(1)文法

如果存在$A \to \varepsilon$这样的产生式，则需定义FOLLOW(A)

$\forall A \in V$定义A的**后续符号集为**：

1. $\text{FOLLOW}(A)=\{a \mid S \overset{*}{\Rightarrow} \alpha A a \beta,\ a \in T,\ \alpha,\beta \in (V \cup T)^*\}$
2. 如果A是某个句型的最右符号，则将结束符#添加到FOLLOW(A)中
3. 给定$\alpha_j \overset{*}{\Rightarrow} \varepsilon$和当前终结符a $\in$FOLLOW(A)，则如果对$\forall i(1 \le i \le m;\ i \ne j)$，$\text{FIRST}(\alpha_i) \cap \text{FOLLOW}(A)=\varnothing$均成立，且A的所有候选式的首符号集都不含有a，可以确定当前终结符a是由α_j后面的变量生成的

4.2.3 LL(1)文法

综上所述，对文法G的句子进行确定的自顶向下语法分析的充分必要条件是：如果G的任意两个具有相同左部的产生式$A\rightarrow\alpha|\beta$满足下列条件：

1. 如果α、β均不能推导出ε，则$\mathrm{FIRST}(\alpha)\cap\mathrm{FIRST}(\beta)=\varnothing$；
2. α和β至多有一个能推导出ε；
3. 如果$\beta \overset{*}{\Rightarrow}\varepsilon$，则$\mathrm{FIRST}(\alpha)\cap\mathrm{FOLLOW}(A)=\varnothing$,则称$G$为$LL(1)$文法。

第一个L代表从左向右扫描输入符号串，第二个L代表产生最左推导，1代表在分析过程中执行每步推导都要向前查看一个输入符号

4.2.3 LL(1)文法

LL(1)文法要求：

$\varepsilon \in \text{FIRST}(\alpha_1), \varepsilon \in \text{FIRST}(\alpha_2), \ldots, \varepsilon \in \text{FIRST}(\alpha_n)$

中至多有一个成立。

如果有多个成立呢？

算法4.2 计算FIRST(X)。

输入：文法$G=(V, T, P, S)$，$X\in(V\cup T)$;

1. FIRST(X)= $\varnothing$;
2. if ($X\in T$) then FIRST(X):= $\{X\}$;
3. if $X\in V$ then
   begin
4. if ($X\to\varepsilon\notin P$) then
   FIRST(X):= FIRST(X)$\cup\{a|X\to a\ldots\in P\}$;
5. if ($X\to\varepsilon\in P$) then
   FIRST(X):= FIRST(X)$\cup\{a|X\to a\ldots\in P\}\cup\{\varepsilon\}$
   end
6. 对$\forall X\in V$，重复过程7-10，直到所有FIRST集不变为止。
7. if ($X\to Y\ldots\in P$ and $Y\in V$) then
   FIRST(X):= FIRST(X)$\cup$(FIRST(Y)-$\{\varepsilon\}$);
8. if ($X\to Y_1\ldots Y_n\in P$ and $Y_1\ldots Y_{i-1}\overset{*}{\Rightarrow}\varepsilon$) then
9. for k=2 to i do FIRST(X):= FIRST(X)$\cup$(FIRST(Y_k)-$\{\varepsilon\}$);
10. if $Y_1\ldots Y_n\overset{*}{\Rightarrow}\varepsilon$ then FIRST(X):=FIRST(X)$\cup\{\varepsilon\}$;

算法4.3 计算FIRST(α)。

输入：文法$G=(V, T, P, S)$，$\alpha\in(V\cup T)^*$，$\alpha= X_1\ldots X_n$;

输出：FIRST(α);

步骤：

1．计算FIRST(X_1);

2．FIRST(α):= FIRST(X_1)-{ε};

3．k:=1;

4．while (ε∈FIRST(X_k) and $k<n$) do begin

5．　FIRST(α):= FIRST(α)∪(FIRST(X_{k+1})-{ε});

6．　k:=k+1 end

7．if ($k=n$ and ε∈FIRST(X_k)) then

FIRST(α):=FIRST(α)∪{ε};

例　表达式文法的语法符号的FIRST 集

FIRST(F)={（, id}

FIRST(T)=FIRST(F)={（, id}

FIRST(E)=FIRST(T)={（, id}

FIRST(E')={+，ε}

FIRST(T')={*,ε}

FIRST(+)={+}

FIRST(*)={*}

FIRST(（)={（}

FIRST()）)={)｝

FIRST(id)={id}

E→TE'

E'→+TE’|ε

T→FT'

T'→*FT’|ε

F→(E)|id

计算某变量FIRST集时，关注该变量出现在左部的产生式

算法4.4 计算FOLLOW集。

输入：文法$G=(V，T，P，S)$

输出：FOLLOW(A);

步骤：

1．对$\forall X\in V$，FOLLOW(X)

2．FOLLOW(S) := {#}，#为句子的结束符;

3．对$\forall X\in V$，

FOLLOW

4．若$A\rightarrow\alpha B\beta\in$

FOLLOW( );

5．若$A\rightarrow\alpha B$或$A\rightarrow\alpha B\beta\in P$且$\beta\overset{*}{\Rightarrow}\varepsilon$，$A\neq B$，则

FOLLOW(B):=FOLLOW(B)∪FOLLOW(A);

如果A是某个句型的最右符号，则将结束符'#'添加到FOLLOW(A)

第5行表示，任何一个终结符，如果可以紧跟着A出现，那么该终结符也可以紧跟着B出现，反之不然

例 表达式文法的语法变量的 FOLLOW 集

E→TE'　E'→+TE'|ε

T→FT'　T'→*FT'|ε

F→(E)|id

计算某变量FOLLOW集时关注其出现在右部的产生式

FIRST(F)={（,id}

FIRST(T)=FIRST(F)={（,id}

FIRST(E)=FIRST(T)={（,id}

FIRST(E')={+，ε}

FIRST(T')={*,ε}

FOLLOW(E) = { #,) }

FOLLOW(E')= FOLLOW(E) = { #,) }

FOLLOW(T) = {FIRST(E')-{ε}}∪FOLLOW(E)∪FOLLOW(E')= {+,),#}

FOLLOW(T')= FOLLOW(T)= {+,),#}

FOLLOW(F) = {FIRST(T')-{ε}} ∪FOLLOW(T)∪FOLLOW(T')

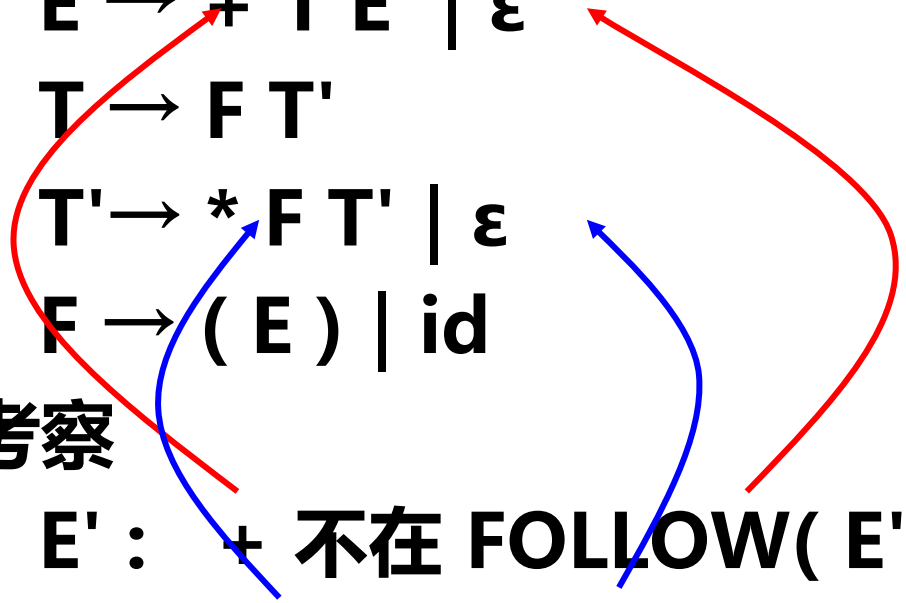
={*,+,),#}

表达式文法是 LL(1) 文法

- E → T E'
- E'→ + T E' | ε
- T → F T'
- T'→ * F T' | ε
- F → (E) | id

考察

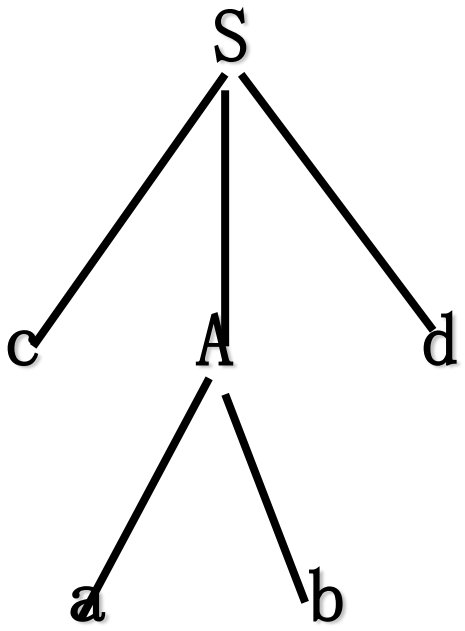
- E' : + 不在 FOLLOW(E') = {), # }
- T' : * 不在 FOLLOW(T') = { +,), # }
- F: (和 id 不同



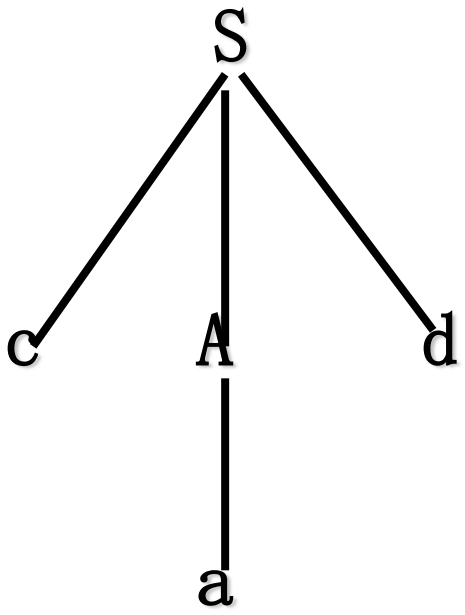
例 对文法

- S→cAd
- A→ab|a

输入 cad 的分析



非 LL(1)文法的不确定性



不确定性的解决方法

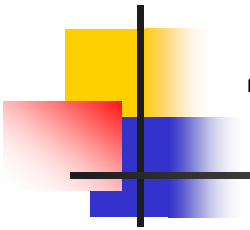
1) 采用回溯算法

- 过于复杂，效率低下

2）改写文法

- 将非LL(1)文法改写为等价的LL(1)文法

- 无法改写时：
  - 增加其它的判别因素
  - 文法过于复杂，无法用自顶向下方法处理



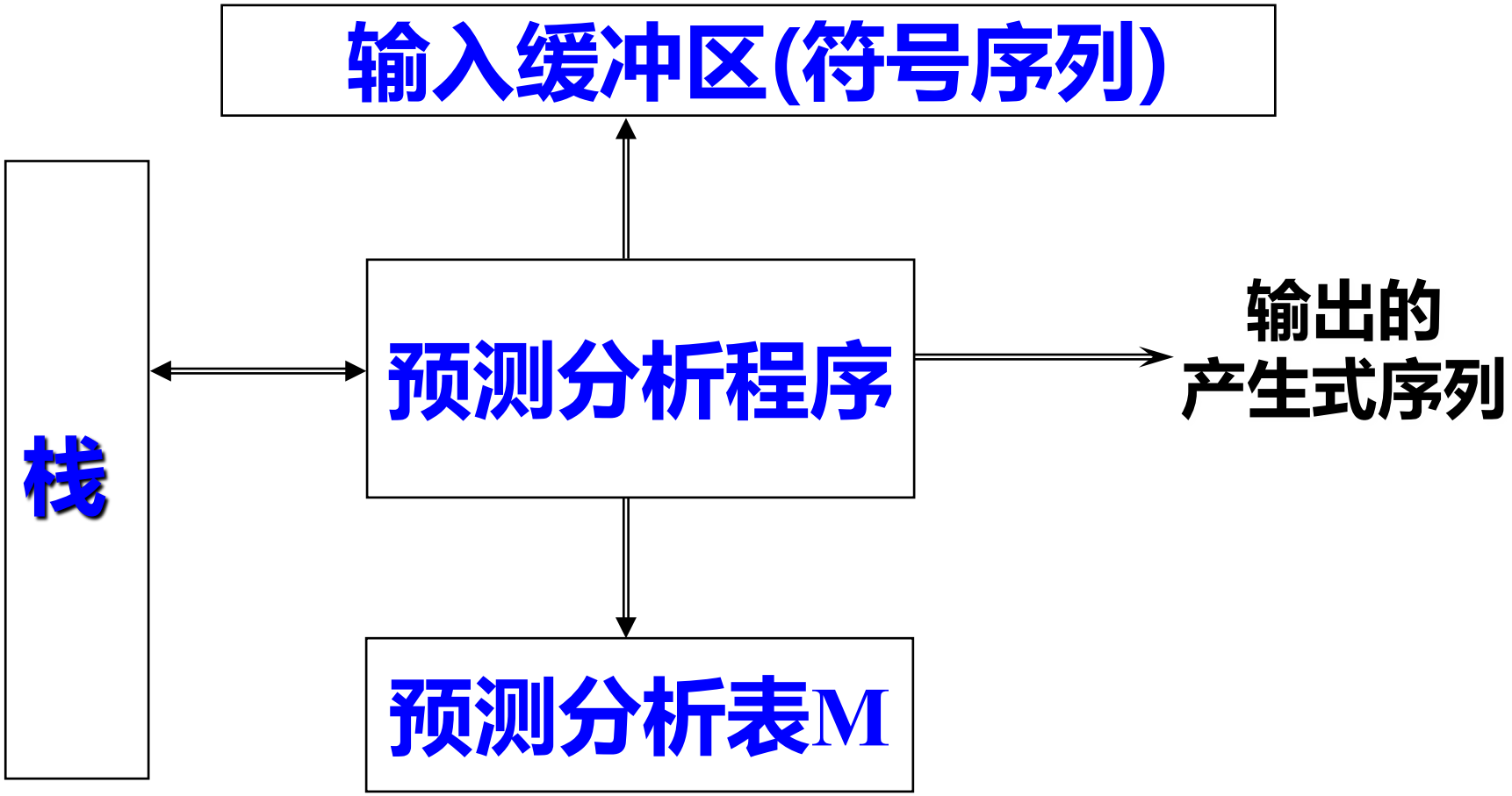
4.3 预测分析法

- 一种高效的自顶向下分析法
- 能够对LL(1)文法实现确定的自顶向下分析

4.3 预测分析法

- 通用控制算法，采用表驱动方式实现
  - 分析表M[A,a]，即LL(1)分析表，存储执行LL(1)分析的信息，其中A是语法变量，a是输入符号。
  - 分析栈，存放文法符号序列，#为栈底符号，初始时栈顶是开始符号
  - 输入缓冲区，包括待分析的串和结束符
- 系统维持一个分析表和一个分析栈，根据当前输入缓冲区中扫描到的符号，选择当前语法变量（处于栈顶）的候选式进行推导——希望找到相应输入符号串的最左推导。

4.3.1 预测分析器的构成



系统的执行与特点

- **在系统启动时，输入指针指向输入串的第一个字符，分析栈中存放着栈底符号#和文法的开始符号。**
- **根据栈顶符号A和读入的符号a，查看分析表M,以决定相应的动作。**
  - **如果A=a=#，分析成功并停机**
  - **如果A=a≠#，**弹出栈顶符号A，并将输入指针指向下一个符号
  - 如果A是语法变量，程序访问分析表M的M[A,a]表项，该表项或者是一个A产生式，或者是出错信息。
  - 如果M[A,a]={A→UVW}，则用WVU(栈顶)替换原栈顶符号A，输出该产生式

系统的执行与特点

- 优点：
  1）效率高
  2）便于维护、自动生成

- 关键——分析表M的构造

预测分析表的构造算法

算法4.6 预测分析表(*LL*(1)分析表)的构造算法。

输入：文法*G*;

输出：分析表*M*;

步骤：

1．对*G*中的任意一个产生式*A*→α*(alpha)*, 执行第2和第3步;

2． for ∀*a*∈FIRST(α), 将*A*→α*(alpha)*填入*M*[*A*, *a*];

3． if ε∈FIRST(α) then ∀*a*∈FOLLOW(*A*),将*A*→α*(alpha)*填入*M*[*A*, *a*];

4．将所有无定义的*M*[*A*, *b*]标上出错标志。

例4.10 考虑简单算术表达式文法的实现

FOLLOW(E')={), # }

FOLLOW(T')={ +,), # }

FIRST(TE')={(,id}　　FIRST(+TE')={+}

FIRST(FT')={(,id}　　FIRST(*FT')={*}

FIRST((E))={(}　　FIRST(id)={id}

E→TE'　　E'→+TE'|ε　　T→FT'

T'→*FT'|ε　　F→(E)|id

E→TE'　E'→+TE'　E'→ε　T→FT'

T'→*FT'　T'→ε　F→(E)　F → id

语法变量	输入符号					
	id	+	*	(	)	#
E	E →TE′			E → TE′		
E′		E′ → +TE′			E′ → ε	E′ → ε
T	T → FT′			T → FT′		
T′		T′ → ε	T′ → *FT′		T′ → ε	T′ → ε
F	F → id			F → (E)		

FOLLOW(E')={), # }

FOLLOW(T')={ +,), # }

FIRST(TE')={(,id}　FIRST(+TE')={+}　FIRST(FT')={(,id}

FIRST(*FT')={*}　FIRST((E))={(}　FIRST(id)={id}

简单算术表达式文法的预测分析表

非终结符	输入符号					
	id	+	*	(	)	#
E	→TE’			→TE’		
E '		→+TE’			→ε	→ε
T	→FT’			→FT’		
T '		→ε	→*FT’		→ε	→ε
F	→id			→(E)		

预测分析法的实现步骤

1. 构造文法
2. 改造文法：消除二义性、消除左递归、提取左因子
3. 求每个候选式的FIRST集和变量的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，需要附加新的“信息”
5. 构造预测分析表
6. 实现预测分析器

算法4.5 预测分析程序的总控程序。

输入：输入串w和文法$G=(V, T, P, S)$的分析表M;

输出：如果w属于$L(G)$，则输出w的最左推导，否则报告错误;

步骤:

1. **将栈底符号#和文法开始符号S压入栈中;**
2. **repeat**
3. X:=**当前栈顶符号;**
4. a:=**当前输入符号;**
5. **if** $X \in T \cup \{\#\}$ **then**
6. **if** $X=a$ **then**
7. {**if** $X \neq \#$ **then begin**
8. **将X弹出栈;**
9. **前移输入指针**
10. **end}**

预测分析程序的总控程序

11. else error// 即当前栈顶终结符和当前
// 输入符号不一致
12. else //也就是X属于一个语法变量
13. if $M[X, a]=Y_1Y_2\ldots Y_k$ then begin
14. 将X弹出栈;
15. 依次将Y_k，…，Y_2，Y_1压入栈;
16. 输出产生式$X\rightarrow Y_1Y_2\ldots Y_k$
17. end
18. else error
19. until X=#

简单算术表达式文法的预测分析表

非终结符	输入符号					
	id	+	*	(	)	#
E	→TE’			→TE’		
E '		→+TE’			→ε	→ε
T	→FT’			→FT’		
T '		→ε	→*FT’		→ε	→ε
F	→id			→(E)		

对输入串id+id*id进行分析的过程

栈	输入缓冲区	输出
#E	id+id*id#	
#E'T	id+id*id#	E→TE'
#E'T'F	id+id*id#	T→FT'
#E'T'id	id+id*id#	F→id
#E'T'	+id*id#	
#E'	+id*id#	T'→ε
#E'T+	+id*id#	E'→+TE'
#E'T	id*id#	

#E'T	id*id#	(接上表)
#E'T'F	id*id#	T→FT'
#E'T'id	id*id#	F→id
#E'T'	*id#	
#E'T'F*	*id#	T'→*FT'
#E'T'F	id#	
#E'T'id	id#	F→id
#E'T'	#	
#E'	#	T'→ε
#	#	E'→ε

输出的产生式序列形成了最左推导对应的分析树

4.3.3 预测分析中错误的处理

- 错误处理
  - 1. 栈顶终结符号和下一个输入符号不匹配
  - 2. 栈顶是语法变量A，a是下一个输入符号，M[A, a]是空白表项

4.3.3 预测分析中错误的处理

- 紧急方式错误恢复策略
- 发现错误时跳过一些输入符号，直到下一个语法成分包含的第一个符号为止（同步记号）

4.3.3 预测分析中错误的处理

- 同步记号的一般选择策略：对语法变量A，如果M[A,a]无定义，并且a属于FOLLOW(A)，则增加M[A,a]为“同步点”(synch)。当程序到达这个同步点时，放弃对A的识别，而转入分析A后面的符号。

4.3.3 预测分析中错误的处理

- **同步记号的启发式选择方法如下：**
  - **把FOLLOW(*A*)的所有符号放入语法变量*A*的同步记号集合。**

跳过当前输入符号直到FOLLOW(A)的某个元素出现，那么分析栈弹出A，继续执行下面的分析

4.3.3 预测分析中错误的处理

- 同步记号的启发式选择方法如下：
  - 把FOLLOW(*A*)的所有符号放入语法变量*A*的同步记号集合。
  - 把高层结构的开始符号加到低层结构的同步记号集合中。

语言通常具有层次结构，例如，表达式包括在语句中，而语句包括在块中，等等。如果表达式的识别出现问题，可以直接从下一个语句继续识别，或者如果当前语句出现问题，可以直接从下一个块继续识别。

4.3.3 预测分析中错误的处理

- 同步记号的[illegible]
  - 把FOLL[illegible]同步记号集
  - 把高层结[illegible]记号集合中。
  - 把FIRST(*A*)的符号加入*A*的同步记号集合。

很可能，对于变量A的识别错误是因为误输入某几个字符，因此如果当前输入字符串出现FIRST(A)的字符，那么可能对A重新开始识别

4.3.3 预测分析中错误的处理

- 同步记号的启发式选择方法如下：
  - 把FOLLOW(*A*)的所有符号放入语法变量*A*的同步记号集合
  - 把高层结构
  - 把FIRST(*A*)
  - 如果语法变量可以产生空串，若出错时栈顶是这样的语法变量，则可以使用产生空串的产生式。
  - 如果符号在栈顶而不能匹配，则弹出此符号。

一个可用的错误处理选项，可以减少在错误处理阶段需要考虑的变量的数量

4.3.3 预测分析中错误的处理

语法变量	输入符号					
	id	+	*	(	)	#
E	E → TE′			E → TE′	synch	synch
E′		E′ → +TE′			E′ → ε	E′ → ε
T	T → FT′	synch		T → FT′	synch	synch
T′		T′ → ε	T′ → *FT′		T′ → ε	T′ → ε
F	E → id	synch	synch	F → (E)	synch	synch

FOLLOW(E)={#,)}

FOLLOW(E′)={#,)}

FOLLOW(T)={+,), #}

FOLLOW(T′)={+,), #}

FOLLOW(F) = {*, + ,), #}

4.3.3 预测分析中错误的处理

语法变量	输入符号					
	id	+	*	(	)	#
E	E → TE′			E → TE′	synch	synch
E′		E′ → +TE′			E′ → ε	E′ → ε
T	T → FT′	synch		T → FT′	synch	synch
T′		T′ → ε	T′ → *FT′		T′ → ε	T′ → ε
F	E → id	synch	synch	F → (E)	synch	synch

使用方法：

1. 如果表项M[A, a]为空，则跳过输入符号a；
2. 如果表项是synch，则弹出栈顶的语法变量并试图恢复分析；
3. 如果栈顶的记号与输入符号不匹配，则从栈顶弹出该记号。

4.3.3 预测分析中错误的处理

语法变量	输入符号					
	id	+	*	(	)	#
E	E → TE′			E → TE′	synch	synch
E′		E′ → +TE′			E′ → ε	E′ → ε
T	T → FT′	synch		T → FT′	synch	synch
T′		T′ → ε	T′ → *FT′		T′ → ε	T′ → ε
F	E → id	synch	synch	F → (E)	synch	synch

识别+id * + id的动作

识别+id * + id的动作

步骤	栈	输入	备注
1	#E	+id*+id#	跳过+
2	#E	id*+id#	正常识别
3	#E′T	id*+id#	
4	#E′T′F	id*+id#	
5	#E′T′id	id*+id#	
6	#E′T′	*+id#	
7	#E′T′F*	*+id#	
8	#E′T′F	+id#	Synch

语法变量	输入符号					
	id	+	*	(	)	#
E	E → TE′			E → TE′	**synch**	**synch**
E′		E′ → +TE′			E′ → ε	E′ → ε
T	T → FT′	**synch**		T → FT′	**synch**	**synch**
T′		T′ → ε	T′ → *FT′		T′ → ε	T′ → ε
F	E → id	**synch**	**synch**	F → (E)	**synch**	**synch**

识别
+ id

语法变量	输入符号					
	id	+	*	(	)	#
E	E → TE′			E → TE′	synch	synch
E′		E′ → +TE′			E′ → ε	E′ → ε
T	T → FT′	synch		T → FT′	synch	synch
T′		T′ → ε	T′ → *FT′		T′ → ε	T′ → ε
F	E → id	synch	synch	F → (E)	synch	synch

9	#E′T′	+id#	
10	#E′	+id#	
11	#E′T+	+id#	
12	#E′T	id#	
13	#E′T′F	id#	
14	#E′T′id	id#	
15	#E′T′	#	
16	#E′	#	
17	#	#	

4.4 递归下降分析法

所谓递归下降分析法，是指根据各个候选式的结构，为文法的每个语法变量编写一个处理程序，用来识别该语法变量所代表的语法成分

4.4 递归下降分析法

1. 对应每个变量设置一个处理子程序，例如对于语法变量A，有产生式：

$$A \rightarrow X_1 X_2 \cdots X_k \cdots X_n$$

(1) 当遇到X_k是终结符号时直接进行匹配;

(2) 当遇到X_k是语法变量时就调用X_k对应的处理子程序.

2. 要求处理子程序是可以**递归调用**的

4.4.1 递归下降分析法的基本思想

例4.14 对于产生式$E' \rightarrow +TE'$，与E'对应的子程序可以按如下方式来编写：

```
procedure E'
  begin
     match('+');
     T;                    /*调用识别T的过程*/
     E'                    /*调用识别E'的过程*/
  end;
```

4.4.1 递归下降分析法的基本思想

其中，服务子程序*match*用来匹配当前的输入记号，其代码为：

```
procedure match(t:token);
  begin
     if lookhead=t  then // lookhead用于保存当前输入记号
        lookhead:=nexttoken; // nexttoken读取下一个符号
     else error                    /*调用出错处理程序*/
  end;
```

4.4.2 语法图和递归子程序法

- 状态转换图（语法图）是非常有用的设计工具，既可用于词法分析，也可以用来帮助建立语法分析程序

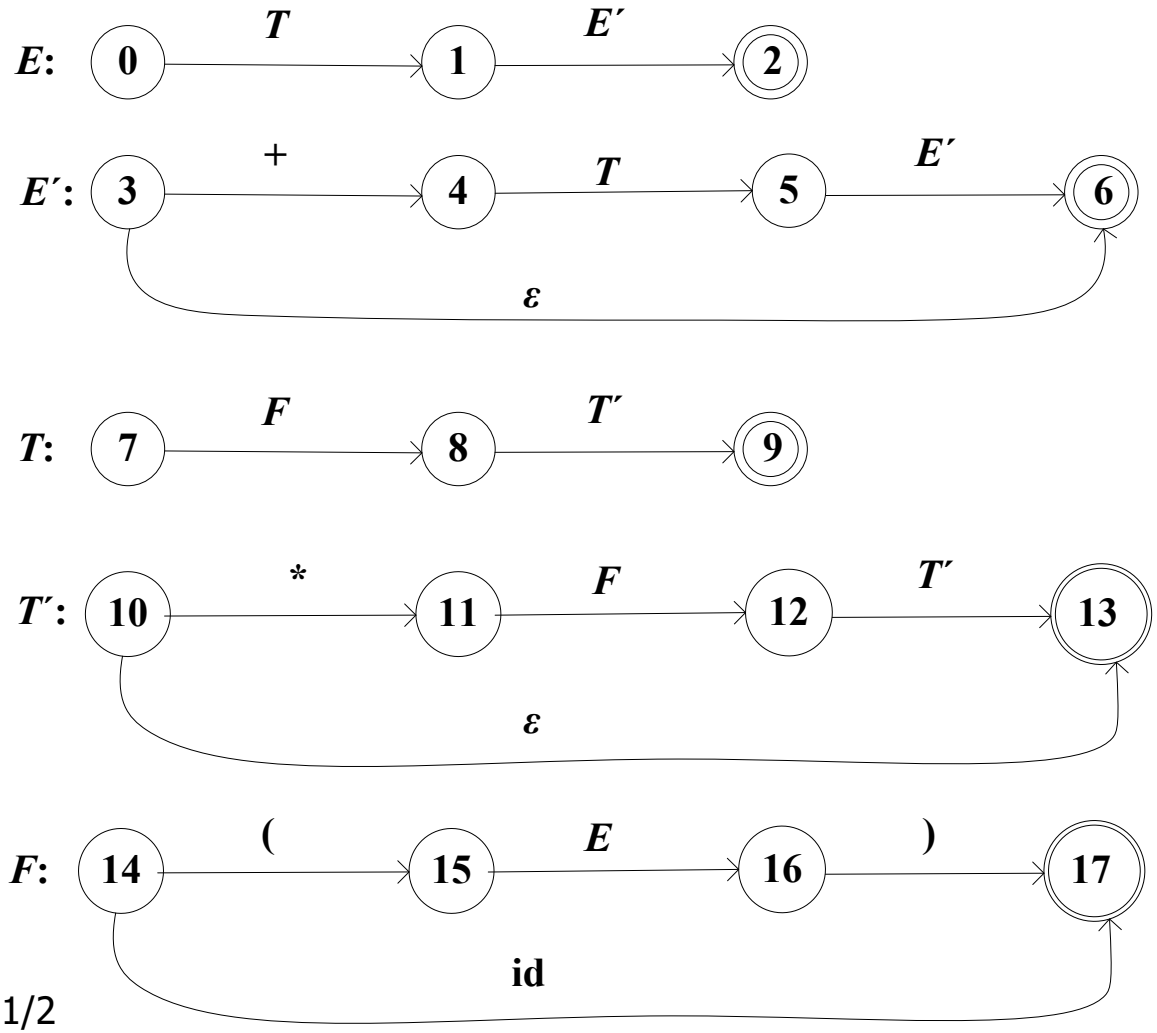
4.4.2 语法图和递归子程序法

- 语法分析器和词法分析器的状态转换图不同
  - 每个非终结符对应一个状态转换图，边上的标记是记号和非终结符
  - 记号上的转换意味着如果该记号是下一个输入符号，就应进行转换
  - 非终结符A上的转换是对与A对应的过程的调用

4.4.2 语法图和递归子程序法

- 从文法构造状态转换图，对每个非终结符A执行如下操作
  - 创建一个开始状态和一个终止状态（返回状态）
  - 对每个产生式$A \rightarrow X_1 X_2 \ \ldots X_n$，创建一条从开始状态到终止状态的路径，边上的标记分别为X_1，X_2，…，X_n
  - 如上的状态转换图表达的是语法成分的构成，简称**语法图**

例4.15 简单表达式文法的语法图



E→TE'

E'→+TE'|ε

T→FT'

T'→*FT'|ε

F→(E)|id

4.4.3基于语法图的语法分析器工作方式

- 初始时，分析器进入状态图的开始状态，输入指针指向输入符号串的第一个符号。
- 如果经过一些动作后，它进入状态s，且从状态s到状态t的边上标记了终结符a，此时下一个输入符又正好是a，则分析器将输入指针向右移动一位，并进入状态t。

4.4.3基于语法图的语法分析器工作方式

- 另一方面，如果边上标记的是非终结符A，则分析器进入A的初始状态，但不移动输入指针。一旦到达A的终态，则立刻进入状态t，事实上，分析器从状态s转移到状态t时，它已经从输入符号串“读”了A （**调用A对应的过程**）。
- 最后，如果从s到t有一条标记为ε的边，那么分析器从状态s直接进入状态t而不移动输入指针。

4.4.4 语法图的化简与实现

(1) 左因子提取

将形如$A \rightarrow YX|YZ$的产生式替换为$A \rightarrow Y(X|Z)$;

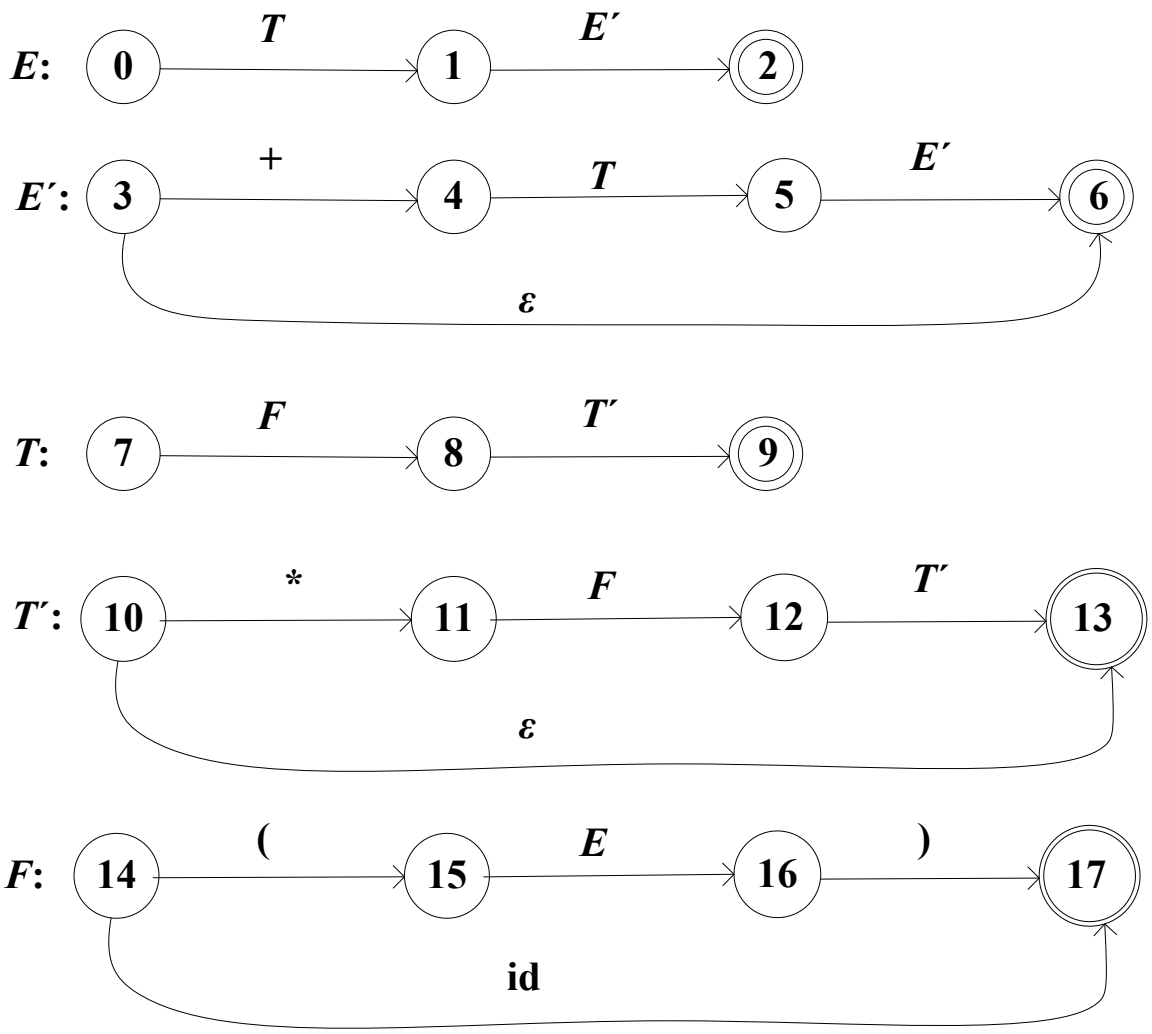
(2) 右因子提取

将形如$A \rightarrow YX|ZX$的产生式替换为$A \rightarrow (Y|Z)X$;

(3) 尾递归消除

将形如$X \rightarrow YX|Z$的产生式替换为$X \rightarrow Y^*Z$。

4.4.4 语法图的化简与实现



E→TE’

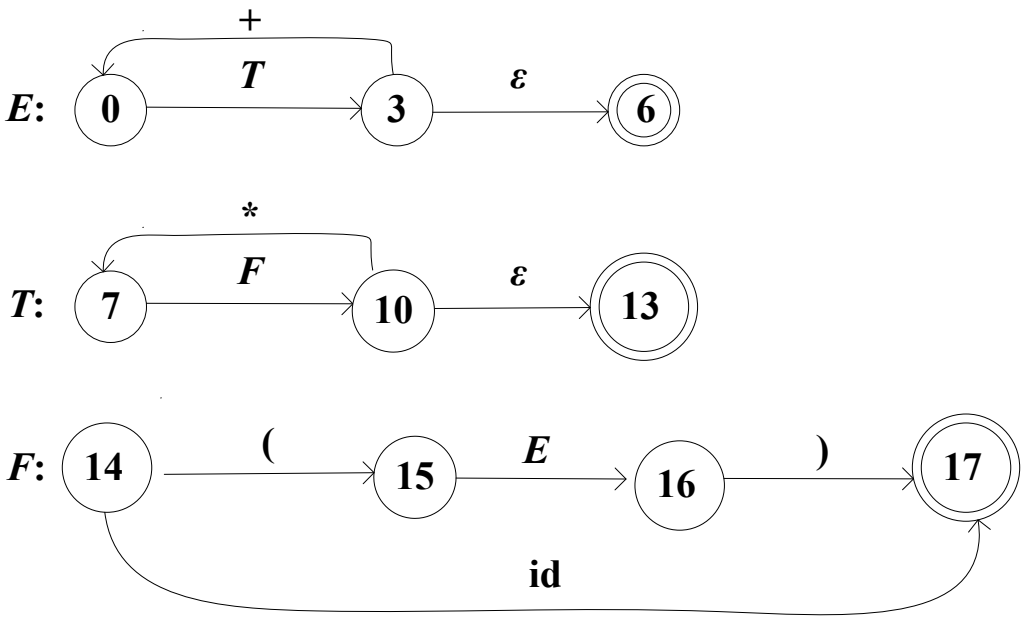
E'→+TE'|ε

T→FT'

T'→*FT'|ε

F→(E)|id

4.4.4 语法图的化简与实现



E → T(+T)*

T → F(*F)*

F → (E) | id

图4.6算术表达式的简化语法图

例4.16 简单算术表达式的语法分析器

- E 的子程序(E→T(+T)*)

```
procedure E;
  begin
    T;                          T的过程调用
    while lookhead='+' do
      begin                     当前符号等于+时
        match(‘+’);             处理终结符+
        T                       T的过程调用
      end
  end;                          lookhead：当前符号
```

T 的子程序(T→F(*F)*)

```
procedure T;
  begin
     F;                          F的过程调用
     while lookhead='*' then
       begin                                  当前符号等于*时
          match(‘*’);          处理终结符*
          F                  F的递归调用
       end
  end;
```

F 的子程序(F→(E)|id)

```
procedure F;
  begin
     if lookhead='(' then
        begin                  当前符号等于（
           match('(');         处理终结符（
           E;                          E 的递归调用
           match(')');         处理终结符)
        end
     else if lookhead=id then
                    match(id)          处理终结符id
     else error                出错处理
  end
```

主程序

```
begin
    lookhead:=nexttoken;        调词法分析程序
    E                           E 的过程调用
end
```

服务子程序

```
procedure match(t:token);
 begin
    if lookhead=t  then
         lookhead:=nexttoken
    else error        出错处理程序
 end;
```

4.4.5 递归子程序法的实现步骤

1) 构造文法；
2) 改造文法：消除二义性、消除左递归、提取左因子；
3) 求每个候选式的FIRST集和语法变量的FOLLOW集；
4) 检查G是不是 $LL(1)$ 文法，若G不是 $LL(1)$文法，说明文法G的复杂性超过了自顶向下方法的分析能力，需要附加新的“信息”；
5) 按照$LL(1)$文法画语法图；
6) 化简语法图；
7) 按照语法图为每个语法变量设置一个子程序。

递归子程序法的优缺点分析

- 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

本章小结

1.自顶向下分析法和自底向上分析法分别寻找输入串的最左推导和最左归约

2.自顶向下分析会遇到二义性问题、回溯问题、左递归引起的无穷推导问题，需对文法进行改造：消除二义性、消除左递归、提取公共左因子

3. *LL*(1)文法是一类可以进行确定分析的文法，利用FIRST集和FOLLOW集可以判定某个上下文无关文法是否为*LL*(1)文法

本章小结

4. $LL(1)$文法可以用$LL(1)$分析法进行分析。

5. 递归下降分析法根据各个候选式的结构为每个非终结符编写一个子程序。

6. 使用语法图可以方便地进行递归子程序的设计。

随堂测试(教材P146)19题

设有如下文法G：

E→TE’

E’ → +E | ε

T → FT’

T’ → T | ε

F → PF’

F’ → *F | ε

P → (E) | a | ∧

(1) 求该文法各语法变量的FIRST集和FOLLOW集

(2) 该文法是否为LL(1)文法？若是，构造其LL(1)分析表