第三讲　自顶向下语法分析

2025-09

1. 基本思想

对于给定语言的文法和一个单词符号串（终结符串），一般的自顶向下分析过程是：从文法开始符号进行推导，每一步推导都获得文法的一个句型，直到产生一个句子，恰好是所期望的单词符号串。每一步推导是对当前句型中剩余的某个非终结符进行展开，即使用以该非终结符为左部的某个产生式的右部替换该非终结符。如果不存在这样的推导，则表明该单词符号串存在语法错误。

例如，给定如下文法 $G[S]$：

$$
\begin{aligned}
&S \to AB \\
&A \to aA \mid \varepsilon \\
&B \to b \mid bB
\end{aligned}
$$

针对单词符号串 $aaab$ 的一个自顶向下分析过程为：

$$
\begin{aligned}
&S && \text{// 使用产生式 } S \to AB \\
\Rightarrow\ &AB && \text{// 使用产生式 } A \to aA \\
\Rightarrow\ &aAB && \text{// 使用产生式 } B \to b \\
\Rightarrow\ &aAb && \text{// 使用产生式 } A \to aA \\
\Rightarrow\ &aaAb && \text{// 使用产生式 } A \to aA \\
\Rightarrow\ &aaaAb && \text{// 使用产生式 } A \to \varepsilon \\
\Rightarrow\ &aaab
\end{aligned}
$$

2. 带回溯的自顶向下分析

一般的自顶向下分析过程存在两类非确定性。其一是在每一步推导中，选择对哪一个非终结符进行展开。其二是如果选定的非终结符是多个产生式的左部，那么应该选择使用哪一个产生式。这种非确定性导致推导过程需要不断地进行试探和回溯。例如，对于文法 $G[S]$ 单词符号串 $aaab$，以下是一个试探的推导过程：

$$
\begin{aligned}
&S && \text{// 使用产生式 } S \to AB \\
\Rightarrow\ &AB && \text{// 使用产生式 } A \to aA \\
\Rightarrow\ &aAB && \text{// 使用产生式 } B \to bB \\
\Rightarrow\ &aAbB && \text{// 使用产生式 } A \to aA \\
\Rightarrow\ &aaAbB && \text{// 使用产生式 } A \to aA \\
\Rightarrow\ &aaaAbB && \text{// 使用产生式 } A \to \varepsilon \\
\Rightarrow\ &aaabB && \text{// 必须回溯}
\end{aligned}
$$

回溯会带来很高的复杂度，这在编译程序的设计中是不现实的。试想一个实际中需要编

译的程序单位可能会包含多少个单词符号，就会得出这样的结论。那么在自顶向下分析过程中，如何避免回溯呢？解决办法就是消除上述两类非确定性。

很容易想到，如果我们只允许最左推导或最右推导，那么就可以避开上述的第一类非确定性。由于通常都是从左到右读入单词符号串，所以我们可以规定在自顶向下分析过程只使用最左推导。例如，对于文法 $G[S]$ 单词符号串 $aaab$，以下试探推导过程的每一步总是对最左边的非终结符进行展开：

S // 使用产生式 $S \to AB$
$\Rightarrow AB$ // 使用产生式 $A \to aA$
$\Rightarrow aAB$ // 使用产生式 $A \to aA$
$\Rightarrow aaAB$ // 使用产生式 $A \to \varepsilon$
$\Rightarrow aaB$ // 必须回溯

这个试探推导过程虽然出现了必须回溯的情形，但我们能够确信这是由于第二类非确定性造成的。也就是说，在前面的推导步骤中选错了产生式。

3. 自顶向下预测分析

如果在自顶向下分析过程中，我们能够同时消除两类非确定性，即能够保证非终结符选择和产生式选择都是确定的，那么这样的分析过程就是**确定的自顶向下分析**。

为了解决消除第二类非确定性的问题，通常采取的策略是向前查看确定数目的单词符号，然后确定应该选择哪一个产生式进行最左推导。这种方法称为**自顶向下预测分析**。成功分析的结果是一个唯一的最左推导。

我们先来看一个可以进行自顶向下预测分析的例子。

对于本讲前面所讨论的文法 $G[S]$，以及任意的单词符号序列 $a^n b^m$（$n \geq 0, m > 0$），总存在如下的最左推导：

S // 使用产生式 $S \to AB$
$\Rightarrow AB$ // 使用产生式 $A \to aA$
$\Rightarrow aAB$ // 使用产生式 $A \to aA$
……
$\Rightarrow a^n AB$ // 使用产生式 $A \to \varepsilon$
$\Rightarrow a^n B$ // 使用产生式 $B \to bB$
$\Rightarrow a^n bB$ // 使用产生式 $B \to bB$
……
$\Rightarrow a^n b^{m-1} B$ // 使用产生式 $B \to b$
$\Rightarrow a^n b^m$

不难看出，对于给定的 $a^n b^m$ 来说，这是唯一的最左推导。然而，问题是当最左的非终结符为 A 时，我们如何在产生式 $A \to aA$ 和 $A \to \varepsilon$ 之间做出选择；同样，当最左的非终结符为 B 时，我们如何在产生式 $B \to bB$ 和 $B \to b$ 之间做出选择。

我们来分析一下向前查看确定数目的单词符号进行预测分析的可能性。首先，在需要展开的最左非终结符为 A 时，我们通过向前查看 1 个单词符号，就可以在产生式 $A \to aA$ 和 $A \to \varepsilon$ 之间作出选择。当查看到下一个单词符号为 a 时，选择 $A \to aA$；而当查看到下一

个单词符号为 b 时，则选择 $A \to \varepsilon$。然而，在需要展开的最左非终结符为 B 时，当查看到下一个单词符号为 b 时，并不足以在 $B \to bB$ 和 $B \to b$ 之间做出选择。我们将向前查看单词符号的数目变为 2。通过分析，当后两个单词符号为 bb 时，我们应该选择 $B \to bB$；而当后两个单词符号中只有一个 b 时（后跟一个单词符号序列的结束符号），应该选择 $B \to b$。

由于 $L(G[S]) = \{ a^n b^m \mid n \geq 0, m > 0 \}$，所以我们可以得出结论：只要向前查看 2 个单词符号，就可预测分析 $L(G[S])$ 中的所有句子。

我们先来看一个不能进行自顶向下预测分析的例子。设有如下文法 $G'[S]$：

$S \to Sa$
$S \to b$

对于 $L(G'[S])$ 中 的任意的单词符号序列 ba^n （$n \geq 0$），总存在如下的最左推导：

S // 使用产生式 $S \to Sa$
$\Rightarrow Sa$ // 使用产生式 $S \to Sa$
$\Rightarrow Saa$ // 使用产生式 $S \to Sa$
……
$\Rightarrow Sa^n$ // 使用产生式 $S \to b$
$\Rightarrow ba^n$

然而，在第 1 步推导时，在向前查看到第 2 个单词符号为 a 时，我们才会选择产生式 $S \to Sa$；在第 2 步推导时，在向前查看到第 3 个单词符号为 a 时，我们才会选择产生式 $S \to Sa$；这样，在第 n 步推导时，我们就需要向前查看 n+1 个单词。这样，无论向前查看单词符号的数确定为多少，都无法满足对 $L(G)$ 中所有句子进行预测分析的需求。

可见，不是所有文法都是可以成功实施自顶向下预测分析的。在实践中，通常我们可以对文法的设计进行适当限制，以满足预测分析的要求。比如，我们将在下一节讨论的 LL(1) 文法是一种满足这种要求的文法，并且足以满足多数程序设计语言的文法描述需求。

4. LL(1) 分析

LL(1) 分析是应用较普遍的一种自顶向下预测分析方法。

LL(1) 中的第一个 “L” 代表从左（Left）向右扫描单词符号，第二个 “L” 代表产生的是最左（Leftmost）推导，“1” 代表向前查看（lookahead）一个单词符号。

LL(1) 分析方法仅适用于 LL(1) 文法。为给出 LL(1) 文法的定义，我们首先介绍两个重要概念：First 集合和 Follow 集合。这些概念在介绍自底向上分析方法时也要用到。

4.1 First 集合和 Follow 集合

我们先来看 **First 集合**的定义：

设上下文无关文法 $G = (V_N, V_T, P, S)$。对 $\alpha \in (V_N \cup V_T)^*$，

First（α）= { a | $\alpha \Rightarrow^* a\beta$, a $\in V_T$, $\beta \in (V_N \cup V_T)^*$，或者 $\alpha \Rightarrow^* \varepsilon$ 时 a = ε}

直观来理解，一个句型 α 若可以推导出另一个以终结符 a 开头的句型，那么 a 属于 First（α）；若 α 可以推导出 ε，那么 ε 属于 First（α）。

由于任何推导都可以对应一个最左推导（可以参考2.2.5的结果直接得出该结论），所以 First 集合的定义也可以基于最左推导给出：

$$\text{First}(\alpha) = \{ a \mid \alpha \Rightarrow_{lm}^* a\beta, a \in V_T, \beta \in (V_N \cup V_T)^* \text{，或者 } \alpha \Rightarrow_{lm}^* \varepsilon \text{ 时 } a = \varepsilon\}$$

在实际应用中，对于给定文法 $G = (V_N, V_T, P, S)$，我们通常会关心下列集合 X_G 中符号串的 First 集合：

$$X_G = V_N \cup V_T \cup \{\varepsilon\} \cup \{v \mid A \to u \in P, \text{且 } v \text{ 是 } u \text{ 的后缀}\}$$

对所有 $x \in X_G$，可以通过如下过程计算 First（x）：

- 初始时，对 $x \in V_T \cup \{\varepsilon\}$，置 First $(x) = \{x\}$；对其它 x，置 First $(x) = \Phi$；
- 重复如下步骤，直到所有 First 集合没有变化为止：

  (1) 对于 $y_1y_2\ldots y_k \in \{v \mid A \to u \in P$，且 v 是 u 的后缀$\}$，其中 $k \geq 1$，$y_j \in V_N \cup V_T$（$1 \leq j \leq k$），若 $\forall j{:}1 \leq j \leq i\text{-}1\ (\varepsilon \in \text{First}(y_j)) \wedge \varepsilon \notin \text{First}(y_i)$，其中 $1 \leq i \leq k$，则令

  $$\text{First}(y_1y_2\ldots y_k) = \text{First}(y_1) \cup \text{First}(y_2) \cup \ldots \cup \text{First}(y_i) - \{\varepsilon\}$$

  否则，若 $\forall j{:}1 \leq j \leq k\ (\varepsilon \in \text{First}(y_j))$，则令

  $$\text{First}(y_1y_2\ldots y_k) = \text{First}(y_1) \cup \text{First}(y_2) \cup \ldots \cup \text{First}(y_k)$$

  (2) 若有 $A \to y_1y_2\ldots y_k \in P$，则置 First (A) = First$(A) \cup$ First $(y_1y_2\ldots y_k)$。

例 1 设上下文无关文法 $G = (V_N, V_T, P, S)$，其中 P 为

$$S \to AB$$
$$A \to Da \mid \varepsilon$$
$$B \to cC$$
$$C \to aADC \mid \varepsilon$$
$$D \to b \mid \varepsilon$$

试计算下列集合 X_G 中句型的 First 集合：

$$X_G = V_N \cup V_T \cup \{\varepsilon\} \cup \{v \mid A \to u \in P, \text{且 } v \text{ 是 } u \text{ 的后缀}\}$$

解 集合 X_G 为：

$$X_G = \{\varepsilon, S, A, B, C, D, a, b, c, AB, Da, cC, aADC, ADC, DC\}$$

初始时，置 First $(a) = \{a\}$，First $(b) = \{b\}$，First $(c) = \{c\}$，First $(\varepsilon) = \{\varepsilon\}$，而对其它的 $x \in X_G$，置 First $(x) = \Phi$。

第一轮应用以上三个步骤之后，X_G 中句型的 First 集合更新情况为：

$$\text{First}(cC) = \{c\}$$
$$\text{First}(aADC) = \{a\}$$
$$\text{First}(A) = \{\varepsilon\}$$
$$\text{First}(B) = \{c\}$$

First (*C*) = {ε, *a*}
First (*D*) = {ε, *b*}

第二轮应用以上三个步骤之后，X_G 中句型的 First 集合更新情况为：

First (*AB*) = {*c*}
First (*Da*) = {*b*, *a*}
First (*ADC*) = {ε, *b*, *a*}
First (*DC*) = {ε, *b*, *a*}
First (*S*) = {*c*}
First (*A*) = {ε , *b*, *a*}

第三轮应用以上三个步骤之后，X_G 中句型的 First 集合更新情况为：

First (*AB*) = {*b*, *a*, *c*}
First (*S*) = {*b*, *a*, *c*}

第四轮应用以上三个步骤之后，X_G 中句型的 First 集合无变化。我们将最后结果整理如下：

First (ε) = {ε}
First (*S*) = {*b*, *a*, *c*}
First (*A*) = {ε , *b*, *a*}
First (*B*) = {*c*}
First (*C*) = {ε, *a*}
First (*D*) = {ε, *b*}
First (*a*) = {*a*}
First (*b*) = {*b*}
First (*c*) = {*c*}
First (*AB*) = {*b*, *a*, *c*}
First (*Da*) = {*b*, *a*}
First (*cC*) = {*c*}
First (*aADC*) = {*a*}
First (*ADC*) = {ε, *b*, *a*}
First (*DC*) = {ε, *b*, *a*}

我们再来看 **Follow 集合**的定义：

设上下文无关文法 $G = (V_N , V_T , P, S)$。对每个 $A \in V_N$ ，

$$\text{Follow}(A) = \{ a \mid S\# \Rightarrow^* \alpha A \beta \# \quad \text{且 } a \in \text{First}(\beta\#), \alpha, \beta \in (V_N \cup V_T)^* \}$$

其中，# 代表输入单词符号序列的结束符。

显然，一定有 # ∈ Follow(*S*)。

直观来理解，若 *G* 中存在一个包含子串 *X*a 的句型，那么 a 属于 Follow(*X*)；*G* 中存在一个以 *X* 结尾的句型，那么 # 属于 Follow(*X*)。

对所有 $A \in V_N$ ，我们可以通过如下过程计算 Follow(*A*)：

- 初始时，置 Follow(S) = {#}；对其它 $A \in V_N$ ，置 Follow(A) = Φ 。
- 重复如下步骤，直到所有 Follow 集合不再变化为止：

  若有 $A \to \alpha B \beta \in P$，其中 $\alpha, \beta \in (V_N \cup V_T)^*$，则

  (1) 令 Follow(B) = Follow(B) ∪ (First(β) – {ε})；

  (2) 若 ε ∈ First(β)， 则令 Follow(B) = Follow(B) ∪ Follow (A)。

注：根据 Follow 集合的定义，上述计算 Follow 集合的过程，我们默认所涉及到的文法符号都是可达的。

例 2 设上下文无关文法 $G = (V_N , V_T , P, S)$，其中 P 为

$S \to AB$
$A \to Da \mid \varepsilon$
$B \to cC$
$C \to aADC \mid \varepsilon$
$D \to b \mid \varepsilon$

试计算 V_N 中所有非终结符的 Follow 集合。

解 初始时，置 Follow(S) = {#}；对其它 $X \in V_N$ ，置 Follow(X) = Φ 。

在例 1 结果的基础上，我们进行每一轮的迭代计算。

依次考虑每一个右端含有非终结符的产生式，第一轮计算为：

由于 $S \to AB \in P$，所以置 Follow(A) = Follow(A) ∪ (First (B)– {ε}) = {c}；

由于 $S \to AB \in P$，ε ∈ First (ε)，所以置 Follow(B) = Follow(B) ∪ Follow(S) = {#}；

由于 $A \to Da \in P$，所以置 Follow(D) = Follow(D) ∪ (First (a)– {ε}) = {a}；

由于 $B \to cC \in P$，ε ∈ First (ε)，所以置 Follow(C) = Follow(C) ∪ Follow(B) = {#}；

由于 $C \to aADC$，所以置 Follow(A) = Follow(A) ∪ (First (DC)– {ε}) = {c, b, a }；

由于 $C \to aADC$，ε ∈ First (DC)，所以置 Follow(A) = Follow(A) ∪ Follow(C) = {c, b, a, #}；

由于 $C \to aADC$，所以置 Follow(D) = Follow(D) ∪ (First (C)– {ε}) = {a}；

由于 $C \to aADC$，ε ∈ First (C)，所以置 Follow(D) = Follow(D) ∪ Follow(C) = {a, #}；

再依次考虑这些产生式，进行第二轮计算，结果无变化。我们将最后结果整理如下：

Follow(S) = {#}
Follow(A) = {c, b, a, #}
Follow(B) = {#}
Follow(C) = {#}
Follow(D) = {a, #}

4.2 LL（1）文法

下面我们来定义 **LL(1) 文法**。为方便，我们先引入预测集合的概念。

设上下文无关文法 $G = (V_N, V_T, P, S)$。对任何产生式 $A \to \alpha \in P$，其预测集合 $PS(A \to \alpha)$ 定义如下：

- 如果 $\varepsilon \notin \text{First}(\alpha)$，那么 $PS(A \to \alpha) = \text{First}(\alpha)$ ；
- 如果 $\varepsilon \in \text{First}(\alpha)$，那么 $PS(A \to \alpha) = (\text{First}(\alpha) - \{\varepsilon\}) \cup \text{Follow}(A)$

文法 G 是 LL(1) 的，当且仅当对于 G 中任何两个有相同左部的不同产生式 $A \to \alpha$ 和 $A \to \beta$，都满足：$PS(A \to \alpha) \cap PS(A \to \beta) = \Phi$。

例 3 设上下文无关文法 $G = (V_N, V_T, P, S)$，其中 P 为

$$
\begin{aligned}
&S \to AB \\
&A \to Da \mid \varepsilon \\
&B \to cC \\
&C \to aADC \mid \varepsilon \\
&D \to b \mid \varepsilon
\end{aligned}
$$

试指出该文法是否 LL(1) 文法。

解 根据LL(1) 文法的定义，我们只需要考虑产生式 $A \to Da$ 和 $A \to \varepsilon, C \to aADC$ 和 $C \to \varepsilon$，以及 $D \to b$ 和 $D \to \varepsilon$ 的预测集合之间是否相交。

先分别计算出这些产生式的预测集合：

$$
\begin{aligned}
&PS(A \to Da) = \{b, a\} \\
&PS(A \to \varepsilon) = \{c, b, a, \#\} \\
&PS(C \to aADC) = \{a\} \\
&PS(C \to \varepsilon) = \{\#\} \\
&PS(D \to b) = \{b\} \\
&PS(D \to \varepsilon) = \{a, \#\}
\end{aligned}
$$

可见：$PS(A \to Da) \cap PS(A \to \varepsilon) \neq \Phi$。因此，该文法不是 LL(1) 文法。

4.3 LL（1）分析的实现

在这一小节里，我们介绍常用的两种 LL（1）分析的实现方法。

4.3.1 递归下降 LL（1）分析程序

在递归下降 LL（1）分析程序的设计中，每个非终结符都对应一个分析子程序，分析程序从调用文法开始符号所对应的分析子程序开始执行。非终结符对应的分析子程序根据下一个单词符号可确定自顶向下分析过程中应该使用的产生式，根据所选定的产生式，分析子程序的行为依据产生式右端依次出现的符号来设计：

- 每遇到一个终结符，则判断当前读入的单词符号是否与该终结符相匹配，若匹配，则继续读取下一个单词；若不匹配，则进行错误处理。
- 每遇到一个非终结符，则调用相应的分析子程序。

例如，设有如下产生式：

<function> → FUNC ID (<parameter_list>) <statement>

其中，<function>，<parameter_list>，和 <statement> 是非终结符，而 FUNC 和 ID 是终结符。若它是左部为 <function> 的唯一产生式，那么非终结符 <function> 对应的分析子程序 ParseFunction() 的设计可描述为：

```
void ParseFunction()
{
    MatchToken(T_FUNC);          // T_FUNC 为匹配终结符 FUNC 的单词符号
    MatchToken(T_ID);            // T_ID 为匹配终结符 ID 的单词符号
    MatchToken(T_LPAREN);        //T_LPAREN 为匹配终结符 '(' 的单词符号
    ParseParameterList();
    MatchToken(T_RPAREN);        // T_RPAREN 为匹配终结符 ')' 的单词符号
    ParseStatement();
}
```

其中，ParseParameterList() 和 ParseStatement() 分别为非终结符 <parameter_list> 和 <statement> 对应的分析子程序，而函数 MatchToken() 则是用于匹配当前终结符和正在扫描的单词符号（若匹配，则调用词法分析程序取下一个单词符号；否则报告词法错误信息）。函数 MatchToken() 的一种简单的设计为：

```
void MatchToken(int expected)
{
    if (lookahead != expected)   //判别当前扫描的符号是否匹配所期待的终结符
    {
        printf("syntax error \n");   //若不匹配，则报告出错信息，跳出
        exit(0);
    }
   else                                  //若匹配，消费掉当前终结符
       lookahead = getToken();     //并向词法分析程序申请读入下一个终结符
}
```

其中，lookahead 为全局量，存放当前扫描的终结符。

为方便，我们在随后的讨论以及例子中，将继续使用这个 MatchToken() 函数。

一般情况下，设 LL（1）文法中某一非终结符 A 对应的所有产生式的集合为：

$$A \rightarrow u_1 \mid u_2 \mid \ldots \mid u_n$$

那么相对于非终结符 A 的分析子程序 ParseA() 可以具有如下形式的一般结构：

```
void ParseA()
{
    switch (lookahead)  {
        case PS(A→u1):
                ……     /*根据 u1 设计的分析过程*/
            break;
```

```
            case PS(A→u2):
                    ……    /*根据 u2 设计的分析过程*/
                    break;
            ...
            case PS(A→ un):
                    ……    /*根据 un 设计的分析过程*/
                    break;
            default:
                    printf("syntax error \n");
                    exit(0);
        }
    }
```

值得注意的是：由于是 LL（1）文法，所以产生式 $A \to u_1$，$A \to u_2$，...，以及 $A \to u_n$ 的预测集合是两两互不相交的，故上述选择语句中的各个选择之间是互斥的。

例 4 设文法 $G[S]$ 为

$$S \to AaS \mid BbS \mid d$$
$$A \to a$$
$$B \to \varepsilon \mid c$$

试验证该文法是 LL(1) 文法，并设计一个基于该文法的递归下降分析程序。

解 先验证该文法是 LL(1) 的。为此，我们先计算出所需要的 First 集合和 Follow 集合：

First（AaS）= {a}
First（BbS）= {c,b}
First（d）= {d}
First（a）= {a}
First（ε）= {ε}
First（c）= {c}
Follow（S）= {#}
Follow（A）= {a}
Follow（B）= {b}

再计算出各产生式的预测集合：

$PS(S \to AaS)$ = {a}
$PS(S \to BbS)$ = {c,b}
$PS(S \to d)$ = {d}
$PS(A \to a)$ = {a}
$PS(B \to \varepsilon)$ = {b}
$PS(B \to c)$ = {c}

因为 $PS(S \to AaS)$，$PS(S \to BbS)$ 以及 $PS(S \to d)$ 互不相交， $PS(B \to \varepsilon)$ 和 $PS(B \to c)$ 不相交，所以，$G[S]$ 是 LL(1) 文法。

这样，开始符号 S 对应的分析子程序可以设计为：

```
void ParseS( )
{
    switch (lookahead)  {
        case a:
            ParseA( );
            MatchToken(a);
            ParseS( );
            break;
        case b,c:
            ParseB( );
            MatchToken(b);
            ParseS( );
            break;
        case d:
            MatchToken(d);
            break;
        default:
            printf("syntax error \n")
            exit(0);
     }
}
```

这里，我们假设了下一个单词符号与该文法的终结符号具有相同的表示形式，下同。

终结符 *A* 对应的分析子程序可以设计为：

```
void ParseA( )
{
    if (lookahead==a)  {
        MatchToken(a);
    }
    else {
        printf("syntax error \n");
        exit(0);
    }
}
```

终结符 *B* 对应的分析子程序可以设计为：

```
void ParseB( )
{
    if (lookahead==c)  {
        MatchToken(c);
    }
    else if (lookahead==b) {
    }
    else {
```

```
            printf("syntax error \n");
            exit(0);
        }
    }
```

在实践中，这种递归下降分析程序的设计思想也常用于其它形式的文法描述。比如，在 PL/0 语法分析程序时采用了 EBNF 形式的语法描述。对于这种语法描述，我们将会得到更加简洁和高效的分析子程序，有利于语法分析程序的手工构造。

EBNF 形式的语法描述中，每一条规则右部的语法成分之间除了可以有连接算符之外，还包含其它一些算符，主要有选择、重复、任选以及优先括号等。这种更加丰富的表达方式有利于精简分析子程序的设计，从而提高递归下降分析程序的效率。在递归下降分析子程序的设计中，针对不同算符，可选择不同的处理语句：

- $X_1 | X_2 | \ldots | X_m$　表示多个成分之间的选择，可对应到选择语句；
- { X }　表示成分 X 的重复（0 到多次），可对应到循环语句；
- [X]　表示成分 X 的任选（0 或 1 次），可对应到 if-then 语句；
- (X)　表示成分 X 的优先处理，可对应到复合语句。

在处理多个成分之间的选择时，也需要保证各成分的预测集合之间互不相交。计算预测集合可以采取类似于 4.1 节叙述的方法，先计算出必要的 First 集合和 Follow 集合。由于本课不涵盖有关 EBNF 形式定义的内容，所以这里不讨论这些集合的严格定义的算法，而是在随后介绍的实例中进行必要的讨论。为帮助理解这些实例，我们列出关于 First 集合的一些性质：

- First（$X_1 | X_2 | \ldots | X_m$）= First（X_1）∪ … ∪ First（X_m）
- First（{ X }）= First（X）∪ {ε}
- First（[X]）= First（X）∪ {ε}
- First（(X)）= First（X）

4.3.2 表驱动 LL（1）分析程序

递归下降分析程序比较直观，容易设计。但不足之处就是递归调用可能带来的效率问题。这一小节里，我们介绍另外一种 LL（1）分析程序的实现方法，称为表驱动的方法。这一方法用到了两个重要的数据结构（对象）：一个二维表和一个栈，称前者为预测分析表，后者为下推栈。

我们先介绍预测分析表的结构和构造算法。

设上下文无关文法 $G = (V_N, V_T, P, S)$。G 的**预测分析表**是一个二维表 M，表的每一行 $A \in V_N$ 对应一个非终结符，表的每一列 $a \in V_T \cup \{\#\}$ 对应某个终结符或输入结束符 # 。表中的项 $M(A,a) \subseteq P$ 是一个产生式集合，它由如下过程得到：

- 对文法 G 的每个产生式 $A \to \alpha \in P$，若它的预测集合 $PS(A \to \alpha)$ 中包含 $a \in V_T \cup \{\#\}$，那么将 $A \to \alpha$ 加入 $M[A,a]$。

在预测分析表中，当 $M(A,a)$ 不含产生式时，就对应一个出错位置。

可以证明: 一个文法 G 的预测分析表中每个表项最多只包含一个产生式，当且仅当 G 是 LL(1) 文法。对于 LL（1）文法，预测分析表也称为 LL（1）分析表。

例 5 设文法 $G[S]$ 为

$$S \to AaS \mid BbS \mid d$$
$$A \to a$$
$$B \to \varepsilon \mid c$$

试给出该文法的预测分析表。

解 根据例 4 的计算结果，各产生式的预测集合为：

$$PS(S \to AaS) = \{a\}$$
$$PS(S \to BbS) = \{c,b\}$$
$$PS(S \to d) = \{d\}$$
$$PS(A \to a) = \{a\}$$
$$PS(B \to \varepsilon) = \{b\}$$
$$PS(B \to c) = \{c\}$$

这样，我们得到如图 1 的预测分析表。可以看出，表中每个表项最多只包含一个产生式。因此，文法 $G[S]$ 为 LL(1) 文法，即图 1 是一个 LL（1）分析表。

	a	b	c	d	#
S	$S \to AaS$	$S \to BbS$	$S \to BbS$	$S \to d$	
A	$A \to a$				
B		$B \to \varepsilon$	$B \to c$		

图 1　预测分析表

表驱动 LL（1）分析程序的工作原理为：初始时，下推栈只包含 #；首先将文法开始符号入栈；然后执行如下步骤：

（1）若栈顶为终结符，则判断当前读入的终结符是否与该终结符相匹配，若匹配，再读取下一终结符继续分析；若不匹配，则进行错误处理。

（2）若栈顶为非终结符，则根据这一非终结符和当前输入终结符查询 $LL(1)$分析表，若相应表项中是（唯一的）产生式，则将此非终结符出栈，并把产生式右部符号从右至左入栈；若表项为空，则进行错误处理。

（3）重复（1）和（2），直到栈顶为 # 同时输入也遇到结束符 # 时，分析结束。

图 2 给出了该分析过程的一个更具体的描述。

```
初始时#入栈，然后文法开始符号入栈；首个输入符号读进a；
  flag =TRUE;
  while (flag)  do  {
     栈顶符号出栈并放在X中;
     if (XÎVT) {
         if  (X==a)
             把下一个输入符号读进a;
         else ERROR;
      }
      else if  (X=='#') {
         if  (a=='#')  flag = FALSE;
         else ERROR;     /* 退出，转出错处理 */
      }
      else if (M[X,a] == {X®X1X2…Xk})  Xk,XK-1,…,X1依次进栈;
      else  ERROR;     /* 退出，转出错处理 */
  }
  /*分析成功，过程完毕*／
```

-

图 2　表驱动 LL（1）分析过程

例如，基于图 1 中的 LL（1）分析表，对于输入符号串 *aabd* 的分析过程如图 3 所示。初始时，下推栈内容为 *#S* ，*S* 为栈顶；余留符号串内容为 *aabd#*，其中，*#*为输入结束符，最左边的 *a* 为下一个输入符号。由于当前栈顶为非终结符，所以通过查图 1 中的分析表得到下一步骤需要用到的产生式为 $S \rightarrow AaS$（*S* 所在行，*a* 所在列）。下一步动作是将栈顶的 *S* 出栈，将 *AaS* 中的符号从右至左依次压栈。对栈顶为非终结符的情形依此类推。当栈顶为终结符（含*#*）时，如步骤 3），检查到该终结符 *a* 与下一个输入符号是匹配的，然后将栈顶的 *a* 出栈，下推栈内容变为 *#Sa* ，同时余留符号串内容变为 *abd#*。假若在某一步发现栈顶的终结符与下一个输入符号不匹配，那末就要转到错误处理过程。重复类似的过程，直到步骤 10），下推栈栈顶和余留符号串内容都是 *#*，分析成功并返回。

步骤	下推栈	余留符号串	下一步动作
1）	*#S*	*a a b d #*	**应用产生式** *S*®*AaS*
2）	*#S a A*	*a a b d #*	**应用产生式** *A*®*a*
3）	*#S a a*	*a a b d #*	匹配栈顶和当前输入符号
4）	*#S a*	*a b d #*	匹配栈顶和当前输入符号
5）	*#S*	*b d #*	**应用产生式** *S*®*BbS*
6）	*#S b B*	*b d #*	**应用产生式** *B*®e
7）	*#S b*	*b d #*	匹配栈顶和当前输入符号
8）	*#S*	*d #*	**应用产生式***S*®*d*
9）	*#d*	*d #*	匹配栈顶和当前输入符号
10）	*#*	*#*	返回：分析成功

图 3　例：表驱动 LL（1）分析过程

5. 文法变换：消除左递归、提取左公因子

如果所设计的文法不是 LL(1) 的，那么就不方便采用 LL(1) 分析方法。有时对文法进行适当的变换，有利于构造出所期望的文法类型。比如，消除左递归和提取左公因子。

当的变换，有利于构造出所期望的文法类型。

5.1 消除左递归

若文法中含有形如 $P \to P_1\alpha_0$，$P_1 \to P_2\alpha_1$，…，$P_n \to P\alpha_n$ 的一组产生式，其中 $n \geq 0$，那么称该文法是**左递归的**。若 $n=0$，则文法中含有形如 $P \to P\alpha$ 的产生式，称为**直接左递归**。相应地，若 $n>0$，我们称之为**间接左递归**。

在实际设计的文法中，包含左递归和左公因子的文法一般不是 LL(1) 的，大家可以讨论一下其中缘由。所以如果想要获得 LL(1) 文法，多数情况下需要消除文法中的左递归和左公因子。

下面我们介绍消除左递归的一般方法。先考虑直接左递归的情形。

对于下列构成直接左递归的产生式组合：

$$P \to P\alpha \mid \beta$$

其中，$\alpha \neq \varepsilon$，以及 β 不以 P 打头，我们可将它们替换为：

$$P \to \beta Q$$
$$Q \to \alpha Q \mid \varepsilon$$

其中 Q 为新增加的非终结符。对于如下具有更一般形式直接左递归的一组产生式：

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

其中 α_i（$1 \leq i \leq m$）不为 ε，β_j（$1 \leq j \leq n$）不以 P 打头，我们可将这一组产生式替换为：

$$P \to \beta_1 Q \mid \beta_2 Q \mid \ldots \mid \beta_n Q$$
$$Q \to \alpha_1 Q \mid \alpha_2 Q \mid \ldots \mid \alpha_m Q \mid \varepsilon$$

其中 Q 为新增加的非终结符。

例 6 设文法 $G[E]$ 为

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid a$$

试变换该文法，得到一个等价的不含左递归的文法。

解 消除直接左递归后得到等价的文法 $G'[E]$：

$$E \to TE'$$
$$E' \to +TE'$$
$$E' \to \varepsilon$$

$T \to FT'$

$T' \to *F\ T'$

$T' \to \varepsilon$

$F \to (E)$

$F \to a$

对于无回路（不存在满足 $A \Rightarrow^{+} A$ 的非终结符 A）、无 ε-产生式（不存在形如 $A \to \varepsilon$ 的产生式）的文法，通过下列步骤可消除**一般左递归**（包括直接和间接左递归）：

（1）以某种顺序将文法非终结符排列为 A_1，A_2，…，A_n。

（2）for $i = 1, n$ do {

for $j = 1, i$-1 do {

用 $A_i \to \alpha_1 r \mid \alpha_2 r \ldots \mid \alpha_k r$ 反复替代形如 $A_i \to A_j$ r 的产生式，

其中 $A_j \to \alpha_1 \mid \alpha_2 \ldots \mid \alpha_k$ 是关于 A_j 的全部产生式;

}

消除关于 A_i 的直接左递归;

}

（3）化简由（2）得到的文法。

例 7 设文法 $G[S]$ 为

$S \to PQ \mid a$

$P \to QS \mid b$

$Q \to SP \mid c$

试变换该文法，得到一个等价的不含左递归的文法。

解 这是一个含有间接左递归的文法，并且无回路和 ε-产生式。我们按照上述消除一般左递归的方法，首先将非终结符排序为 S，P 和 Q，然后进行如下变换：

先考虑关于 S 的产生式。不含直接左递归，所以不发生变化。

再考虑关于 P 的产生式。因为两个产生式都没有 S 打头，所以执行内层 for 循环后不发生变化。同样，关于 P 的产生式不含直接左递归，所以仍不发生变化。

最后考虑关于 Q 的产生式。执行内层 for 循环的过程可描述为

$Q \to SP \mid c$

替换为 $Q \to PQP \mid aP \mid c$

再替换为 $Q \to QSQP \mid bQP \mid aP \mid c$

然后再消去关于 Q 的直接左递归，得到

$Q \to bQPR \mid aPR \mid cR$

$R \to SQPR \mid \varepsilon$

我们列出最终结果：

$$S \to PQ \mid a$$
$$P \to QS \mid b$$
$$Q \to bQPR \mid aPR \mid cR$$
$$R \to SQPR \mid \varepsilon$$

这里应该指出，如果对非终结符的不同排序方式，依上述步骤消除一般左递归后所得到的文法从形式上看可能会有区别，但它们之间是相互等价的。如，我们将非终结符排序为 Q，P 和 S，则变换后所得到的文法为：

$$S \to cSQR \mid bQR \mid aR$$
$$P \to SPS \mid cS \mid b$$
$$Q \to SP \mid c$$
$$R \to PSQR \mid \varepsilon$$

5.2 提取左公因子

对形如

$$P \to \alpha\beta \mid \alpha\gamma$$

的一对产生式，可用如下三个产生式替换：

$$P \to \alpha Q$$
$$Q \to \beta \mid \gamma$$

其中 Q 为新增加的未出现过的非终结符

这种文法变换称为提取左公因子。

在实际设计中，一个含有左公因子的文法通常不是 LL(1) 的（大家可以想一下其中的原因）。所以为了设计 LL(1) 文法时，一般需要提取文法中的所有左公因子。下面我们给出提取左公因子的一般变换规则。

设有如下形式的一组产生式

$$P \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_m \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$$

其中，α 称为 $\alpha\beta_1$，$\alpha\beta_2$，… 和 $\alpha\beta_m$ 之间的**左公因子**，而每个 γ 不以 α 开头。**提取左公因子** α 后，这一组产生式改写为：

$$P \to \alpha Q \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$$
$$Q \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

其中 Q 为新增加的非终结符。

例 8 设文法 $G[S]$ 为

$$S \to \underline{if}\ C\ \underline{then}\ S \mid \underline{if}\ C\ \underline{then}\ S\ \underline{else}\ S \mid a$$
$$C \to \underline{true} \mid \underline{false}$$

其中，*if*，*then*，*else*，*true* 和 *false* 为终结符。试变换该文法，得到一个等价的不含左

公因子的文法。

解 提取 $S \to \underline{if}\ C\ \underline{then}\ S \mid \underline{if}\ C\ \underline{then}\ S\ \underline{else}\ S$ 之间的左公因子 $\underline{if}\ C\ \underline{then}\ S$ 后，得到如下等价于 $G[S]$ 的文法 $G'[S]$：

$$
\begin{aligned}
&S \to \underline{if}\ C\ \underline{then}\ S\ A \mid a\\
&A \to \underline{else}\ S \mid \varepsilon\\
&C \to \underline{true} \mid \underline{false}
\end{aligned}
$$

其中 A 为新增加的非终结符。

6. LL(1) 分析中的出错处理

在编译程序设计中，错误处理主要包含两个方面的任务：一是报错，发现错误时应尽可能准确指出错误位置和错误属性；二是错误恢复，尽可能进行校正，使编译工作可以继续下去，提高程序调试的效率。

关于如何设计错误处理程序，目前并没有特别值得关注的理论成果。在这一小节里，我们只简单讨论 LL（1）分析过程中实现错误处理的最基本方法。

对于表驱动 LL（1）分析，在以下两种情况下需要报错：

- 栈顶的终结符与当前输入符号不匹配；
- 非终结符 A 位于栈顶，面临的输入符号为 a，但分析表 M 的表项 $M[A, a]$ 为空。

一种简单的错误恢复措施是**应急恢复**（panic-mode error recovery）。例如，在表驱动 LL（1）分析中，我们可以专为在表项 $M[A, a]$ 为空的情形指定一些所谓的**同步符号**。在分析过程中遇到这种情形时，就跳过输入符号串中的一些符号直至遇到同步符号为止。一种简便的做法是将 First(A) 或 Follow(A) 中的所有符号当作 A 的同步符号，相应的处理过程可设计为：

- 跳过输入符号串中的一些符号直至遇到 Follow(A) 中的符号，然后把 A 从栈中弹出，便可以使分析继续下去；
- 跳过输入符号串中的一些符号直至遇到 First(A) 中的符号时，可根据 A 恢复分析。

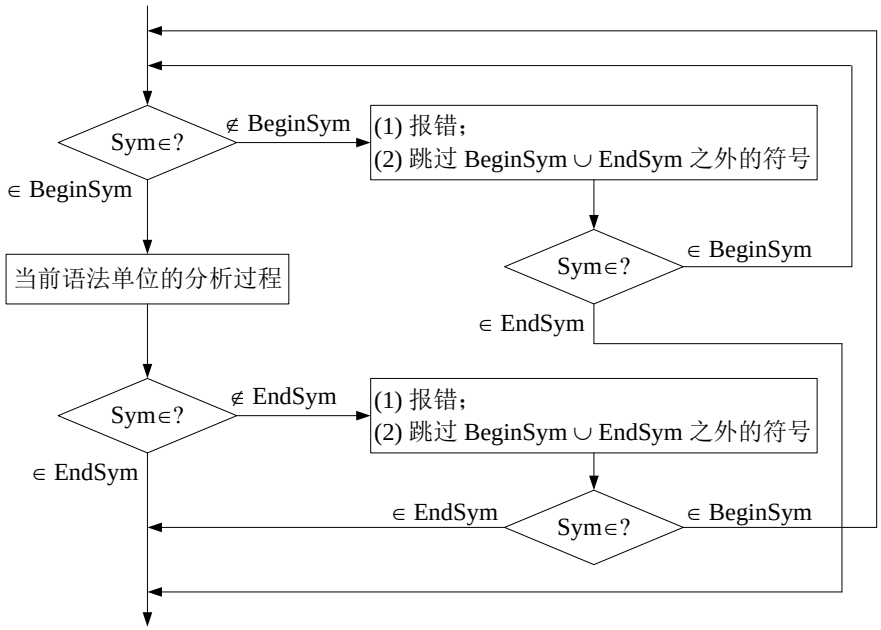

图 4　短语层恢复可采取的流程

另外一种称为**短语层恢复**（phrase-level error recovery）的措施比上述方法精确一些。原因在于，短语是与上下文相关的一个概念，而上述只考虑非终结符的同步符号的方法是与上下文无关的。有关短语的概念我们将在介绍自底向上语法分析时给出。

图 4 描述了短语层错误恢复可采取的一种流程：

- 在进入某个语法单位的分析时，检查当前符号是否属于进入该语法单位需要的符号集合 BeginSym。若不属于，则报错，并滤去补救的符号集合 S =BeginSym ∪ EndSym 外的所有符号。
- 在该语法单位分析结束时，检查当前符号是否属于离开该语法单位时需要的符号集合 EndSym。若不属于，则报错，并滤去补救的符号集合 S =BeginSym ∪ EndSym 外的所有符号。
- 无论是上述哪种情况，若遇 BeginSym 中的符号，则重新分析该语法单位；若遇 EndSym 中的符号，则退出该语法单位的分析。

我们来看在递归下降分析程序中采用短语层错误恢复的一个简单例子。设有如下文法产生式：

$$B \to [\,A\,] \mid (\,A\,)$$
$$A \to a$$

相应于非终结符 B，分析子程序可设计为：

```
procedure ParseB ( EndSym )
{
    if (   lookahead ∉ {'[','('}   )   {
                报错;   跳过 S 之外的单词;        / * S = { '[', '(' }  ∪  EndSym */
```

```
    }
    while ( lookahead ∈ {'[','('} )   {
        if ( lookahead == '[' ) {
                MatchToken( '[' );
                ParseA ( EndSym∪{']'} );
                MatchToken(']');
        }
        else {
                MatchToken( '(' );
                ParseA ( EndSym∪{')'} );
                MatchToken(')');
        }
        if ( lookahead ∉ EndSym )   {
                报错;  跳过 S 之外的单词;        /* S = { '[', '(' } ∪EndSym */
       }
   }
}
```

该子程序对应于图 4 所描述的流程。BeginSym={'[','('}，实际上是取 First（B）。EndSym 作为参数，由上一级子程序传入。值得注意的是，调用下一级分析子程序 ParseA 时，根据 A 不同上下文的后跟符号不同而使用了不同的参数。在方括号上下文中，使用的参数是 EndSym∪{']'}；而在圆括号上下文中，使用的参数是 EndSym∪{')'}。这可以体现"短语层"恢复的含义。试比较，若不区分方括号还是圆括号上下文，那么我们有可能使用 EndSym∪{']', ')'} 作为参数。后果如何？同学们可以思考一下。

这里，我们需要对前面（4.3.1 节）的函数 MatchToken()进行一个小的修改：

```
void MatchToken(int expected)
{
    if (lookahead != expected)     //判别当前扫描的单词符号是否与期望的终结符匹配
    {
        printf("syntax error \n");   //若不匹配，则报告出错信息，跳出
    }
    else                                //若匹配，消费掉当前单词符号
        lookahead = getToken();     //并向词法分析程序申请并读入下一个单词符号
}
```

其中，lookahead 为全局量，存放当前扫描的单词符号。

注意，与前面相比，修改后的 MatchToken 函数在报错后不退出系统。下同。

相应于非终结符 A，分析子程序可设计为：

```
procedure ParseA ( EndSym )
{
    if ( lookahead ∉ { 'a' } )   {
            报错;  跳过 S 之外的单词;        /* S = { 'a' } ∪EndSym */
```

```
        }
        while ( lookahead ∈ { 'a' } ) {
            MatchToken ( 'a' );
            if ( lookahead ∉ EndSym )  {
                    报错;  跳过 S 之外的单词;        /* S = { 'a' } ∪ EndSym */
            }
        }
    }
```

7. LL(k) 文法（选讲）

LL(1) 分析的自顶向下预测分析策略是向前查看 1 个单词符号，然后确定应该选择哪一个产生式进行最左推导。这种方法要求文法必须是 LL(1) 文法。

如果允许向前查看任意 k（$k > 0$）个单词符号，则进行自顶向下预测分析时对文法的要求可能会宽松一些。一般地，可以将 LL(1) 文法扩展至 LL(k) 文法。

直观来看，对于任何 CFG $G=(V, T, P, S)$, 如果向前查看 k（$k > 0$）个终结符，就可预测分析 $L(G[S])$ 中的所有句子，即在最左推导的每一步均能够确定应该选择哪一个产生式进行推导，这样的 CFG G 就是 $LL(k)$ 文法。

以下是关于 LL(k) 文法的几个重要结论：

- 对于整数 k>0，一个上下文无关文法是否为 LL(k) 文法是可判定的。
- 对于一个上下文无关文法，是否存在一个整数 k>0 使得该文法是 LL(k) 文法，是不可判定的。
- 对于一个上下文无关文法，是否存在一个与之等价的 LL(k)文法（$k > 0$），是不可判定的。
- 两个 LL(k) 文法的语言是否相等是可判定的。
- LL(k) 文法是无二义文法。
- LL(k) 文法中不存在左递归的非终结符。
- 对于整数 k>0，任何一个 LL(k) 文法 G 的语言 $L(G)$，存在一个不含ε产生式的 LL(k+1) 文法可以产生语言 $L(G)$-{ ε}。
- 对于整数 k>0，给定任何不含ε产生式的 LL(k+1) 文法，存在一个含ε产生式的 LL(k) 文法可产生同样的语言。
- 对于整数 k>0，不含ε产生式的 LL(k) 文法的语言类真包含于不含ε产生式的 LL(k+1) 文法的语言类。
- 对于整数 k>0，LL(k) 文法的语言类真包含于 LL(k+1) 文法的语言类。

这些结论在本课程的课堂讲稿（ppt）中有列举，目的是引起大家关注，但只是作为选讲，不属于课程的主体内容，也不会进行考察。

为满足一些同学的兴趣，以下对其中个别结论的由来及其与本讲前面部分较为相关的内

容进行一些介绍，对于其余结论仅指出可进一步阅读的参考材料。

首先，需要有LL(k)文法的严格数学定义。这里，我们引用 Aho 和 Ullman 教材[1] 中的 $LL(k)$ 文法定义。其他不同的定义方式，本质上也是一致的，如Stearn等人的定义[2，3]。

先引入 **First$_k$ 集合**的概念。

设有CFG $G=(V, T, P, S)$，$\alpha\in(V\cup T)^*$，以及某个固定的 $k>0$。定义

$\text{First}_k(\alpha) = \{ w \mid w\in T^*$，当$|w|< k$时有$\alpha \Rightarrow_{lm}^* w$，或者当$|w|=k$时有$\alpha \Rightarrow^* wx$，其中$x\in(V\cup T)^*\}$。

CFG $G=(V, T, P, S)$ 是 ***LL(k)* 文法**，如果对任意满足条件 $\text{First}_k(x) = \text{First}_k(y)$ 的两个最左推导

（1）$S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\beta\alpha \Rightarrow^* wx$ 和

（2）$S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\gamma\alpha \Rightarrow^* wy$

都满足 $\beta = \gamma$。

该定义可直观解释为，G 是 $LL(k)$ 文法，如果对给定左句型 $wA\alpha$，以及输入串中 w 之后终结符串的前 k 个符号构成的串 z，满足 $z\in\text{First}_k(A\alpha)$，那么下一步最左推导所使用的产生式 $A\to\sigma$ 是唯一确定的。

若存在某个 $k>0$，文法 G 是 $LL(k)$ 文法，则称 G 是一个 ***LL* 文法**。

基于下列定理（Aho 和 Ullman 教材[1]中的定理5.2），可以帮助更好地理解 $LL(k)$ 文法。

定理: 设 CFG $G=(V, T, P, S)$, G 是 $LL(k)$ 文法，当且仅当下列条件成立：如果 $A\to\beta$ 和 $A\to\gamma$ 是 P 中不同的产生式，那么，对任何满足 $S \Rightarrow_{lm}^* wA\alpha$ 的左句型 $wA\alpha$，都有 $\text{First}_k(\beta\alpha)\cap\text{First}_k(\gamma\alpha) = \varnothing$。

证明

（if）假设 G 不是 $LL(k)$ 文法，则存在左句型 $wA\alpha$ 以及两个不同的产生式 $A\to\beta$ 和 $A\to\gamma$ 是，使得

（1）$S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\beta\alpha \Rightarrow^* wx$ 和

（2）$S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\gamma\alpha \Rightarrow^* wy$

同时满足 $\text{First}_k(x) = \text{First}_k(y)$。这与 $\text{First}_k(\beta\alpha)\cap\text{First}_k(\gamma\alpha) = \varnothing$ 矛盾。

（only if）假设 $A\to\beta$ 和 $A\to\gamma$ 是 P 中不同的产生式，存在满足 $S \Rightarrow_{lm}^* wA\alpha$ 的左句型 $wA\alpha$，使得 $\text{First}_k(\beta\alpha)\cap\text{First}_k(\gamma\alpha)\neq\varnothing$。则存在 $x\in\text{First}_k(\beta\alpha)\cap\text{First}_k(\gamma\alpha)$，使得

（1）$S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\beta\alpha \Rightarrow^* wxy$ 和

（2）$S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\gamma\alpha \Rightarrow^* wxz$

因为 G 是 $LL(k)$ 文法，因此满足 $\beta = \gamma$，这与 $A\to\beta$ 和 $A\to\gamma$ 是 P 中不同的产生式矛盾。

证毕。

接第 3 节的分析，不难验证，设 k=2，下列文法 $G[S]$满足上述定理：

$$S \to AB$$
$$A \to aA \mid \varepsilon$$
$$B \to b \mid bB$$

因此，$G[S]$是 $LL(2)$ 文法。

以下是一个含有左递归的文法：

$$S \to Sa$$
$$S \to b$$

对于该文法产生的任何终结符序列 ba^n （$n \geq 0$），最左推导形如：

$$S \Rightarrow Sa \Rightarrow Saa \Rightarrow \ldots \Rightarrow Sa^n \Rightarrow ba^n$$

对任何$k>0$，取$n \geq k-1$，则存在左句型 Sa^{k-1}，即 $S \Rightarrow^*_{lm} Sa^{k-1}$。在上述定理中，取$w$为空串，$\alpha$为$a^{k-1}$，$\beta$为$Sa$，$\gamma$为$b$，则满足 $\mathrm{First}_k(\beta\alpha) \cap \mathrm{First}_k(\gamma\alpha) \neq \varnothing$。因此，该文法不是任何 $LL(k)$ 文法（$k>0$）。

这一结论可推广到任何含有左递归的文法（假定不含无用符号），无论是直接左递归还是间接左递归，即上面所列的结论之一“LL(k) 文法中不存在左递归的非终结符”。这一结论的证明也可参考 Rosenkrantz 和 Stearn 论文[2]中的引理5。

再看一个有关左公因子的例子。以下是一个含有左公因子的文法：

$$S \to abA \mid abB$$
$$A \to a$$
$$B \to b$$

根据上述定理，不难验证，该文法不是 $LL(1)$和 $LL(2)$文法，但它是 $LL(k)$（$k \geq 3$）文法。

另外，从上述定理可以直接得出上面所列的结论之一。

推论：任何 $LL(k)$ 文法一定是无二义文法。

设 $G=(V, T, P, S)$ 是 $LL(k)$ 文法，这一推论需要证明任何 $w \in T^*$均有唯一的最左推导，若不然，则一定存在如下两个不同的最左推导：

$$S \Rightarrow^*_{lm} xA\alpha \Rightarrow_{lm} x\beta\alpha \Rightarrow^*_{lm} xyz = w \quad 和 \quad S \Rightarrow^*_{lm} xA\alpha \Rightarrow_{lm} x\gamma\alpha \Rightarrow^*_{lm} xyz = w$$

这里，$A \to \beta$ 和 $A \to \gamma$ 是 P 中不同的产生式，根据上述定理，应满足$\mathrm{First}_k(\beta\alpha) \cap \mathrm{First}_k(\gamma\alpha) = \varnothing$。但从上面的两个推导，一定有$\mathrm{First}_k(yz) \in \mathrm{First}_k(\beta\alpha)$ 以及 $\mathrm{First}_k(yz) \in \mathrm{First}_k(\gamma\alpha)$。矛盾。

对于上面所列的前 3 个结论，可参考 Aho 和 Ullman 教材[1]中第 5.1.6 节的讨论，其中涵盖了关于第 1 个结论中关于 LL(k) 文法的一个判定算法。关于第 2、3 个结论的不可判定问题，该教材作为练习给出，参见其中的习题 5.1.11 和 5.1.12。这两个不可判定问题的证明也可参考 Rosenkrantz 和 Stearn 论文[2]中的定理 11 和定理 13，同时其定理 1 对应的是第 1 个结论。

关于“两个 LL(k) 文法的语言是否相等”的可判定性证明，可参考 Aho 和 Ullman 教材[1]中的定理 8.6，或者 Rosenkrantz 和 Stearn 论文[2]中的定理 8。

对于倒数第四个结论，其证明可分别参考 Rosenkrantz 和 Stearn 论文[2]中的定理 3，或者 Aho 和 Ullman 教材[1]中的定理 8.4。

对于倒数第三个结论，其证明可参考 Rosenkrantz 和 Stearn 论文[2]中的定理 5，或者 Aho 和 Ullman 教材[1]中的定理 8.7。

对于倒数第二个结论，其证明可参考 Rosenkrantz 和 Stearn 论文[2]中的定理 7。证明中用到一个特殊的例子，对于整数 k>0，语言 $\{ a^n(b^kd + b + cc)^n \mid n\geq 1 \}$不能被任何不含ε产生式的 LL($k$) 文法接受，但可以被如下包含ε产生式的 LL($k$) 文法 $G[S]$接受：

$S \rightarrow aDA$
$D \rightarrow aDA \mid \varepsilon$
$A \rightarrow cc \mid bB$
$B \rightarrow \varepsilon \mid b^{k-1}d$

根据倒数第四个结论，存在一个不含ε产生式的 LL(k+1) 文法接受该语言。

与此类似，在 Aho 和 Ullman 教材[1]中，用了另一个不同的语言 $L_k = \{ a^nw \mid n\geq 1, w \in \{ b, c, b^kd \}^n\}$，并说明语言 L_k 不能被任何不含ε产生式的 LL(k) 文法接受。

从倒数第四个结论和倒数第二个结论，可以推论出最后一个结论。最后一个结论也可参考 Aho 和 Ullman 教材[1]中的定理 8.8。

实际上，可以认为 LL(0)语言类是 LL(1)语言类的真子集，因此最后一个结论对于整数 $k\geq 0$ 都成立，体现了 LL(k)语言类真包含于 LL(k+1)语言类的层次关系。

可将 4.3.2 节所介绍的表驱动 LL(1)分析算法推广至 k-预测分析算法（推广 LL(1)分析表至 LL(k)分析表）。参考 Aho 和 Ullman 教材[1]中的定理 5.1，如果在输入串后面加一个结束符，k-预测分析过程可以对应于一个 DPDA。我们以例 5 的下列 LL(1)文法 $G[S]$为例：

$S \rightarrow AaS \mid BbS \mid d$
$A \rightarrow a$
$B \rightarrow \varepsilon \mid c$

下面将针对 LL(1)文法 $G[S]$定义一个空栈接受的 DPDA E，使得其语言 $N(E) = \{ w\# \mid w \in L(G[S])$，以及 # 是输入结束符 $\}$。显然，$N(E)$ 满足前缀性质，后者是空栈接受方式 *DPDA* 语言的必要条件。

令 $G[S] = (V, T, P, S)$, 其中 $V = \{S, A, B\}$, $T = \{a, b, c, d\}$, 以及 P 由上面的 6 个产生式构成。我们定义 $E = (Q, T, V \cup T, \delta, q0, \#)$, 其中

$Q = \{q0\} \cup \{<t, \varepsilon> \mid t \in T \cup \{\varepsilon, \#\}\} \cup \{<t, p> \mid t \in T \cup \{\#\}, p \in P,$ 同时 $t \in PS(p)\}$

其中 $PS(p)$ 为产生式 p 的预测集合，其定义参见图 1。

δ 定义如下:

$\delta(q0, \varepsilon, \#) = \{ (<\varepsilon, \varepsilon>, S\#) \}$, $\delta(<\varepsilon, \varepsilon>, a, S) = \{ (<a, S \rightarrow AaS>, S) \}$, $\delta(<\varepsilon, \varepsilon>, b, S) = \{ (<b, S \rightarrow BbS>, S) \}$,

$\delta(<\varepsilon, \varepsilon>, c, S) = \{ (<c, S\rightarrow BbS>, S) \}$, $\delta(<\varepsilon, \varepsilon>, d, S) = \{ (<d, S\rightarrow d>, S) \}$,

$\delta(<\varepsilon, \varepsilon>, a, A) = \{ (<a, A\rightarrow a>, A) \}$, $\delta(<\varepsilon, \varepsilon>, b, B) = \{ (<b, B\rightarrow \varepsilon>, B) \}$, $\delta(<\varepsilon, \varepsilon>, c, B) = \{ (<c, B\rightarrow c>, B) \}$,

$\delta(<a, S\rightarrow AaS>, \varepsilon, S) = \{ (<a, A\rightarrow a>, AaS) \}$, $\delta(<b, S\rightarrow BbS>, \varepsilon, S) = \{ (<b, B\rightarrow \epsilon>, BbS) \}$,

$\delta(<c, S\rightarrow BbS>, \varepsilon, S) = \{ (<c, B\rightarrow c>, BbS) \}$, $\delta(<d, S\rightarrow d>, \varepsilon, S) = \{ (<d, \varepsilon>, d) \}$,

$\delta(<a, A\rightarrow a>, \varepsilon, A) = \{ (<a, \varepsilon>, a) \}$, $\delta(<b, B\rightarrow \varepsilon>, \varepsilon, B) = \{ (<b, \varepsilon>, \varepsilon) \}$, $\delta(<c, B\rightarrow c>, \varepsilon, B) = \{ (<c, \varepsilon>, c) \}$,

$\delta(<a, \varepsilon>, \varepsilon, a) = \delta(<b, \varepsilon>, \varepsilon, b) = \delta(<c, \varepsilon>, \varepsilon, c) = \delta(<d, \varepsilon>, \varepsilon, d) = \delta(<\#, \varepsilon>, \varepsilon, \#) = \{ (<\varepsilon, \varepsilon>, \varepsilon) \}$

$\delta(<\varepsilon, \varepsilon>, a, a) = \delta(<\varepsilon, \varepsilon>, b, b) = \delta(<\varepsilon, \varepsilon>, c, c) = \delta(<\varepsilon, \varepsilon>, d, d) = \delta(<\varepsilon, \varepsilon>, \#, \#) = \{ (<\varepsilon, \varepsilon>, \epsilon) \}$

DPDA E 接受输入串 $aabd\#$ 的步骤为

$(q0, aabd\#, \#) \vdash (<\varepsilon, \varepsilon>, aabd\#, S\#) \vdash (<a, S\rightarrow AaS>, abd\#, S\#)$

$\vdash (<a, A\rightarrow a>, abd\#, AaS\#) \vdash (<a, \varepsilon>, abd\#, aaS\#) \vdash (<\varepsilon, \varepsilon>, abd\#, aS\#)$

$\vdash (<\varepsilon, \varepsilon>, bd\#, S\#) \vdash (<b, S\rightarrow BbS>, bd\#, S\#) \vdash (<b, B\rightarrow \varepsilon>, d\#, BbS\#)$

$\vdash (<b, \varepsilon>, d\#, bS\#) \vdash (<\varepsilon, \varepsilon>, d\#, S\#) \vdash (<d, S\rightarrow d>, \#, S\#) \vdash (<d, \varepsilon>, \#, d\#)$

$\vdash (<\varepsilon, \varepsilon>, \#, \#) \vdash (<\varepsilon, \varepsilon>, \varepsilon, \varepsilon)$

可以看出，这一步骤相当于图 3 所示的针对输入串 *aabd* 的自顶向下表驱动 LL（1）分析过程。

参考文献

[1] Alfred V. Aho, Jefferey D. Ullman. The Theory of Parsing, Translation, and Compiling, Volume 1 & Volume 2, Prentice-Hall Series in Automatic Computation，1972.

[2] D. J. Rosenkrantz and R. E. Stearn, Properties of Deterministic Top-Down Grammars, Information and Control 17, 226-256, 1970.

[3] P. M. II, Lewis, and R. E. Stearn , Syntax-directed transductions, J. ACM 15, 465-488, 1968.

课后作业

1. 设有文法 $G[S]$：

$$S \rightarrow aSb \mid aab$$

若针对该文法设计一个自顶向下预测分析过程，则需要向前查看多少个输入符号？

2. 设有文法 $G[E]$：

$$S \to \mathrm{i}\, S\, A$$
$$A \to \mathrm{e}\, S \mid \varepsilon$$
$$S \to \mathrm{a}$$

针对文法G'[S]，试给出各产生式右部文法符号串的 First 集合，各产生式左部非终结符的 Follow 集合，以及各产生式的预测集合PS，即完成下表：

G 中的规则 r	$First\ (rhs(r))$	$Follow\ (lhs(r))$	$PS\ (r)$
$S \to \mathrm{i}\, S\, A$			
$A \to \mathrm{e}\, S$			
$A \to \varepsilon$		此处不填	
$S \to \mathrm{a}$		此处不填	

表中的 $rhs(r)$ 表示产生式 r 右部的文法符号串，$lhs(r)$ 表示产生式 r 左部的非终结符。

3. 设有文法 $G[E]$：

$$E \to TA$$
$$A \to \vee TA \mid \varepsilon$$
$$T \to FB$$
$$B \to \wedge FB \mid \varepsilon$$
$$F \to (E) \mid i$$

试针对该文法填写下表

G中的规则 r	$first\ (rhs(r))$	$follow\ (lhs(r))$	$PS\ (r)$
$E \to TA$			
$A \to \vee TA$			
$A \to \varepsilon$		此处不填	
$T \to FB$			
$B \to \wedge FB$			
$B \to \varepsilon$		此处不填	
$F \to (E)$			
$\mathrm{F} \to \mathrm{i}$		此处不填	

其中, $rhs(r)$ 表示产生式 r 的右部，$lhs(r)$ 表示产生式 r 的左部

4. 试验证下列文法 G[S]是 LL(1) 文法:

$$S \to P \mid \varepsilon$$

$$P \to (\ P\ )\ P \mid a$$

其中 （，），以及 a 为终结符

5. 计算下列文法中每个非终结符的 First 集和 Follow 集，以及每个产生式的预测集合，并判断该文法是否 LL(1)文法（说明原因）：

（1）文法 $G_1[S]$：

$$S \to (S) \mid aT$$
$$T \to +ST \mid \varepsilon$$

（2）文法 $G_2[S]$：

$$S \to TP$$
$$T \to +PT \mid \varepsilon$$
$$P \to (S) \mid \mathrm{a}$$

（3）文法 $G_3[S]$：

$$S \to A1B$$
$$A \to 0A \mid \varepsilon$$
$$B \to 0B \mid 1B \mid \varepsilon$$

6. 验证如下文法是 LL(1)文法，并基于该文法构造递归下降分析程序：

（1）文法 $G[S]$：

$$S \to AB$$
$$A \to aA$$
$$A \to \varepsilon$$
$$B \to bB$$
$$B \to \varepsilon$$

（2）文法 $G[E]$：

$$E \to [\ F\ ]\ A$$
$$A \to E \mid \varepsilon$$
$$F \to aB$$
$$B \to aB \mid \varepsilon$$

其中［，］，以及 a 为终结符

7. 给出习题 5 中所有文法的预测分析表，并根据分析表指出相应文法是否 LL(1)的，同时验证习题 5 的结果。

8. 给出习题 6 中所有文法的 LL(1)分析表。

9. 基于图 1 中的 LL(1)分析表，根据图 2 描述的方法，给出输入符号串 *baacbd* 的表驱动

LL(1)分析过程。

10. 通过变换求出与下列文法 $G[S]$ 等价的一个文法，使其不含直接左递归：

$$S \to AbB$$
$$A \to Aa \mid a$$
$$B \to Ba \mid Bb \mid b$$

11. 按照本讲介绍的消除一般左递归算法消除下面文法 $G[S]$中的左递归（要求依非终结符的排序 S、Q、P 执行该算法）：

$$S \to PQ \mid a$$
$$P \to QS \mid b$$
$$Q \to SP \mid c$$

12. 按照本讲介绍的消除一般左递归算法消除下面文法 $G[P]$中的左递归（要求依非终结符的不同排序分别执行该算法）：

$$P \to Qa \mid a$$
$$Q \to Pb \mid b$$

13. 变换下列文法， 求出与其等价的一个文法，使变换后的文法不含左递归和左公因子：

（1）文法 $G[P]$：

$$P \to Pa \mid Pb \mid c$$

（2）文法 $G'[S]$：

$$S \to a\,S\,A\,c \mid a$$
$$A \to A\,b \mid d$$

14. 给定某类表达式文法 $G[E]$：

$$E \to +\,E\,R \mid -\,E\,R \mid \underline{positive}\;R$$
$$R \to *\,E\,R \mid \varepsilon$$

其中，+ 和 – 分别代表一元正和一元负运算，* 代表普通的二元乘法运算，$\underline{positive}$ 为代表正整数（非0）的单词。

(1) 针对文法 $G[E]$，下表给出各产生式右部文法符号串的 *First* 集合，各产生式左部非终结符的 *Follow* 集合，以及各产生式的预测集合*PS*。试填充其中空白表项（共3处）的内容：

$G[E]$的规则 r	*First* ($rhs(r)$)	*Follow* ($lhs(r)$)	*PS* (r)
$E \to +\,E\,R$	+		+
$E \to -\,E\,R$	-	此处不填	-
$E \to \underline{positive}\;R$	$\underline{positive}$	此处不填	$\underline{positive}$
$R \to *\,E\,R$	*		*
$R \to \varepsilon$	ε	此处不填	

表中，*rhs(r)* 为产生式 r 右部的文法符号串，*lhs(r)* 为产生式 r 左部的非终结符。

(2) G[E] 不是 LL(1) 文法，试解释为什么?

(3) 虽然G[E] 不是 LL(1) 文法，但可以采用一种强制措施，使得常规的 LL(1) 分析算法仍然可用。针对含5个单词的输入串 + - 20 * 18 ，以下基于这一措施以及上述各产生式的预测集合（或预测分析表）的一个表驱动 LL(1) 分析过程：

步骤	下推栈	余留符号串	下一步动作
1	# *E*	+ - *20* * *18* #	应用产生式 $E \rightarrow + E R$
2	# *R E* +	+ - *20* * *18* #	匹配栈顶和当前输入符号
3	# *R E*	- *20* * *18* #	应用产生式 $E \rightarrow - E R$
4	# *R R E* -	- *20* * *18* #	匹配栈顶和当前输入符号
5	# *R R E*	*20* * *18* #	应用产生式 $E \rightarrow$ *positive* R
6	# *R R R positive*	*20* * *18* #	匹配栈顶和当前输入符号
7	# *R R R*	* *18* #	
8			
9			
10			
11			
12	# *R R R*	#	应用产生式 $R \rightarrow \varepsilon$
13	# *R R*	#	应用产生式 $R \rightarrow \varepsilon$
14	# *R*	#	应用产生式 $R \rightarrow \varepsilon$
15	#	#	结束

试填写上述分析过程中第7步时使用的产生式，以及第 8~11 步的分析过程，共计13处空白；并指出采用了什么样的强制措施。