第 3 章　程序设计语言的语法描述

在第 2 章中，用正规式描述单词符号，并且讨论了如何利用正规式、NFA 和 DFA 自动构造词法分析器。本章在单词符号（单词种别）的基础上，讨论程序设计语言语法结构的形式描述，用上下文无关文法来描述程序设计语言的语法结构。在第 4 章和第 5 章中，将讨论由这种文法所形成的语法分析问题，以及语法分析器自动构造。

3.1　文法的引入

什么是文法（语法）？简单地说，文法是语言结构的定义和描述。先讨论自然语言的文法，为方便，以一个英文句子为例：

the big elephent ate a banana

这是否是一个英语句子？回答是肯定的。根据英语语法，它属于一种主谓宾结构。

3.1.1　语法树

根据英语的语法，上述句子的语法结构可用图 3.1 表示。

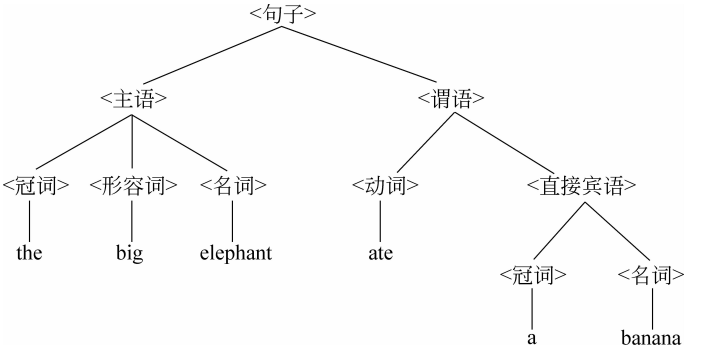

图　3.1

这种句子结构的图形表示法称为“语法树”，任何一个语法上正确的英文句子都可以根据英语语法画出相应的语法树来。借助语法树，将一个英文句子分解为多个组成部分，并以此来描述句子的语法结构。从语法树可以看到，＜句子＞由＜主语＞和＜谓语＞组成，＜主语＞由＜冠词＞、＜形容词＞和＜名词＞组成，＜谓语＞由＜动词＞和＜直接宾语＞组成，＜直接宾语＞由＜冠词＞和＜名词＞组成。

语法树是识别句子的重要工具。在语法树中，带尖括号的非叶结点称为语法单位，在形式语言中称为非终结符，其中处于根结点位置的非终结符又称为开始符号。不带尖括号的

叶结点称为单词符号,在形式语言中称为终结符。

3.1.2 语法规则和句子推导

也可以通过建立一组规则,来描述上述句子的语法结构。上面的英文句子可用下述规则来刻画。

1 <句子>→<主语><谓语>
2 <主语>→<冠词><形容词><名词>
3 <冠词>→the
4 <形容词>→big
5 <名词>→elephant
6 <谓语>→<动词><直接宾语>
7 <直接宾语>→<冠词><名词>
8 <动词>→ate
9 <冠词>→a
10 <名词>→banana

其中"→"读作"定义为",在有些书中用"::="表示。规则在形式语言中称为产生式,在"→"左部的符号串称为产生式左部,在"→"右部的符号串称为产生式右部。

规则:

3 <冠词>→the
9 <冠词>→a

可改写为:

<冠词>→the|a

"|"读作"或"。同样规则:

5 <名词>→elephant
10 <名词>→banana

可改写为:

<名词>→elephant|banana

有了规则,就可以推导和产生句子,可使用上述 10 条规则推出"大象吃香蕉"的句子。从开始符号<句子>开始推导,因从<句子>出发只有一个选择,故按规则 1 进行第 1 步推导:

<句子>⇒<主语><谓语>

表示从<句子>可推出<主语><谓语>,其中"⇒"读作"直接推出"。下一步推导从<主语><谓语>开始,这里存在两个语法单位。当存在两个或两个以上的语法单位时,可任选其中一个先进行推导。例如从<主语>开始推导,据此找到产生式左部是"<主语>"的规则 2,用规则 2 的右部"<冠词><形容词><名词>"取代"<主语>",就产生了下面的第 2 步推导:

<句子>⇒<主语><谓语>⇒<冠词><形容词><名词><谓语>

再选择规则 3、4 和 5,分别对<冠词>、<形容词>和<名词>进行推导。如此重复,直到

推出句子的全部。全部推导过程如下所示：

＜句子＞⇒＜主语＞＜谓语＞
⇒＜冠词＞＜形容词＞＜名词＞＜谓语＞
⇒the＜形容词＞＜名词＞＜谓语＞
⇒the big＜名词＞＜谓语＞
⇒the big elephant＜谓语＞
⇒the big elephant＜动词＞＜直接宾语＞
⇒the big elephant ate ＜直接宾语＞
⇒the big elephant ate ＜冠词＞＜名词＞
⇒the big elephant ate a ＜名词＞
⇒the big elephant ate a banana

为了书写和表示方便，可以使用记号“$\overset{+}{\Rightarrow}$”来表示上述推导序列。上述推导过程可缩写为：

＜句子＞$\overset{+}{\Rightarrow}$the big elephant ate a banana

若在第 3 步推导中，＜冠词＞用 a 取代，而不是用 the 取代；在第 5 步推导中，＜名词＞用 banana 取代，而不是用 elephant 取代；在第 9 步推导中，＜冠词＞用 the 取代，而不是用 a 取代；在第 10 步推导中，＜名词＞用 elephant 取代，而不是用 banana 取代，则可获得如下推导：

＜句子＞$\overset{+}{\Rightarrow}$a big banana ate the elephant

句子“大香蕉吃象”是通过上述规则推导出来的，显然它也是一个符合英语语法的句子，但是它表达的意义是荒谬的，也就是说句子的语义是错误的。一个语法正确的句子不能保证该句子的语义是正确的，判断一个句子是否正确，需要进行语法和语义两方面的检查。语法分析仅仅是检查句子语法是否正确，并不关心它的语义，在语义分析阶段才进行语义正确性检查和语义翻译。

为了说明一组规则可以推出不同的句子这一事实，再举一个例子。设有如下规则：

1　＜句子＞→＜主语＞＜谓语＞
2　＜主语＞→we|you|they
3　＜谓语＞→run|sit|eat|sleep

根据上述规则可产生下面 12 个句子，这 12 个句子称为相应于规则的语言。

{we run, we sit, we eat, we sleep, you run, you sit, you eat, you sleep, they run, they sit, they eat, they sleep}

3.1.3　递归规则和递归文法

递归是编译技术中的一个重要概念。所谓递归定义，就是定义某事物，又用到该事物本身。在规则中，递归定义就表现为在规则的左部和右部有相同的非终结符。例如：

U→xUy

设上式中U为非终结符，x 和 y 为终结符。因在产生式的左部和右部都含有非终结符U，故U→xUy 是递归规则。定义非终结符U，又用到U本身。若 x＝ε，U→Uy 称为左递归规

则;若 y=ε,则U→xU称为右递归规则。文法的递归性,除由递归规则引起外,还可能在推导过程中由规则间接产生。例如:

1 V→Uy|z

2 U→xV

上述规则虽然都不是递归规则,但是由于存在推导 $V \Rightarrow Uy \Rightarrow xVy$,即 $V \overset{+}{\Rightarrow} xVy$,称文法含有间接递归。含有递归规则或间接递归的文法,称为递归文法。

利用递归文法,可以用有穷的规则来描述无穷的语言。这不但解决了语言的定义问题,而且可使对语言的语法检查成为可能。例如定义无符号整数,若不采用递归规则,描述无符号整数全体就要使用无穷多条的规则。

1 <无符号整数>→<数字>|<数字><数字>|<数字><数字><数字>|…

2 <数字>→0|1|2|3|4|5|6|7|8|9

若采用递归规则,使用 12 条规则就能描述无符号整数全体。

1 <无符号整数>→<无符号整数><数字>|<数字>

2 <数字>→0|1|2|3|4|5|6|7|8|9

例 3.1 无符号整数 1

<无符号整数>⇒<数字>⇒1

例 3.2 无符号整数 23

<无符号整数>⇒<无符号整数><数字>⇒<数字><数字>⇒2<数字>⇒23

例 3.3 无符号整数 456

<无符号整数>⇒<无符号整数><数字>⇒<无符号整数><数字><数字>⇒<数字><数字><数字>⇒4<数字><数字>⇒45<数字>⇒456

3.2 上下文无关文法

文法是描述语言结构的形式规则(语法规则),这些规则必须是准确的,易于理解的,且有较强的描述能力,足以描述各种不同的结构。由这种规则所形成的程序设计语言应易于分析和翻译,并且能通过这些规则自动产生有效的语法分析程序。

形式语言的奠基人乔姆斯基(Chomsky)将文法分为 4 种类型,它们是:短语文法(0 型文法)、上下文有关文法(1 型文法)、上下文无关文法(2 型文法)和正规文法(3 型文法),这 4 种文法在形式语言中都有严格的定义。对于程序设计语言来说,上下文无关文法已经够用,上下文无关文法有足够的能力描述大多数现今使用的程序设计语言的语法结构。通俗地讲,上下文无关文法是这样一种文法,它所定义的语法单位和该语法单位可能出现的环境无关。例如当碰到算术表达式时,可以对它就事论事地进行处理,而不必考虑它所处的上下文。但是在自然语言中,一个句子乃至一个字,它们的意义和它们所处的上下文往往有密切的关系,因此上下文无关文法不适宜描述自然语言。

本节将讨论什么是上下文无关文法。以后,“文法”一词若无特别说明,则均指“上下文无关文法”。

3.2.1　文法和语言

一个文法 G 是一个四元式(V_T,V_N,S,P),其中:

- V_T是一个终结符的非空有限集,终结符通常用小写字母表示;
- V_N是一个非终结符的非空有限集,非终结符通常用大写字母表示;
- S 是一个特殊的非终结符(S∈V_N),称为开始符号;
- P 是一个产生式(规则)的有限集合,每个产生式的形式是"A→α",其中 A∈V_N,α∈$(V_T \cup V_N)^*$。

终结符是语言的基本符号,是指在源程序中可以看到的程序设计语言的单词,是语言不可分割的最小单位。在编译程序内部,经词法分析后单词用二元式编码(code,val)表示。在语法分析中,仅仅使用单词的种别 code。语法分析所关心的是,当前处理的单词是标识符还是常数,而不考虑标识符指的是哪个变量,常数的值是多少。所以在讨论语法分析时,终结符用单词的种别表示。根据前面约定,"i"表示标识符(标识符可用于定义变量,有时也称"i"表示变量),"x"表示无符号整数,而"y"表示无符号实数。单字符单词的种别和单词本身相同,例如单词"+"的种别用"+"表示。基本字由多个字符构成,为了直观,有时仍借用原单词形式,例如单词"if"的种别用"if"或"f"表示。

非终结符用来表示抽象的语法单位,如"算术表达式"、"布尔表达式"、"赋值语句"、"说明语句"和"程序"等。非终结符通常用大写字母表示,也可以用带尖括号的汉字表示。

开始符号是一个特殊的非终结符,它代表最感兴趣的语法单位,是定义文法的出发点。例如讨论算术表达式,那么描述算术表达式文法的开始符号就是<算术表达式>。在程序设计语言中,最感兴趣的语法单位是<程序>。

产生式是定义语法单位的一种书写规则。上下文无关文法产生式的左部必定是一个非终结符,该非终结符称为产生式的左部符号,简称左部符号。产生式的右部是终结符和非终结符经有限次连接构成的文法符号串,可以是空字 ε。4 种文法的区别从产生式来看,主要是对产生式的左部和右部的限制不同。

为了书写方便,若干个左部符号相同的产生式,如:

1　$A \rightarrow \alpha_1$

2　$A \rightarrow \alpha_2$

…　…

n　$A \rightarrow \alpha_n$

可合并为一个,缩写成:

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$$

其中 α_i($1 \leqslant i \leqslant n$)称为 A 的候选式。

例 3.4　描述算术表达式文法 G=(V_T,V_N,S,P),其中:

V_T={+、−、*、/、i、x、y、(、)}

V_N={<算术表达式>、<项>、<因子>}

S=<算术表达式>

P={

```
    <算术表达式>→<算术表达式>+<项>、
    <算术表达式>→<算术表达式>-<项>、
    <算术表达式>→<项>、
    <项>→<项>*<因子>、
    <项>→<项>/<因子>、
    <项>→<因子>、
    <因子>→(<算术表达式>)、
    <因子>→i、                          //标识符
    <因子>→x、                          //无符号整数
    <因子>→y                            //无符号实数
}
```

因已约定非终结符和终结符的书写方式,非终结符和终结符在产生式中一目了然,故终结符集 V_T 和非终结符集 V_N 无须再显式列出。若规定左部符号为开始符号的产生式,写在所定义文法的第 1 行,上述文法 G 可简单表示为如下形式:

```
1  <算术表达式>→<算术表达式>+<项>
2  <算术表达式>→<算术表达式>-<项>
3  <算术表达式>→<项>
4  <项>→<项>*<因子>
5  <项>→<项>/<因子>
6  <项>→<因子>
7  <因子>→(<算术表达式>)
8  <因子>→i
9  <因子>→x
10 <因子>→y
```

若用 E 表示<算术表达式>、T 表示<项>、F 表示<因子>,借助符号"|",算术表达式文法 G 可表示成如下最简形式:

```
1  E→E+T|E-T|T
2  T→T*F|T/F|F
3  F→(E)|i|x|y
```

一个上下文无关文法如何定义一个语言呢?其中心思想是:从文法的开始符号出发,反复使用产生式,对非终结符施行替换和展开。例如,考虑下面的文法 G:

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

其中,唯一的非终结符 E 代表仅含加乘运算的简单算术表达式。可以从 E 出发,进行一系列推导,推出各种不同的算术表达式来。例如根据规则 E→(E),可以说从 E 可直接(一步地)推出 (E)。如果用"⇒"表示"直接推出",那么这句话可表示为:

$$E\Rightarrow(E)$$

有时也称,(E)可直接归约为 E。若对(E)中的 E 使用规则 E→E+E,就有:

$$(E)\Rightarrow(E+E)$$

即从(E)可直接推出(E+E),或称(E+E)可直接归约为(E)。把上述两步合并起来,就有:

$$E\Rightarrow(E)\Rightarrow(E+E)$$

再对(E+E)中的 E 相继两次使用规则 E→i 之后，就有：

$$E\Rightarrow(E)\Rightarrow(E+E)\Rightarrow(i+E)\Rightarrow(i+i)$$

称这样的一串替换序列是从 E 推出(i+i)的一个推导。这个推导提供了一个证明，证明(i+i)是文法 G 定义的一个算术表达式。注意，推导每前进一步总是引用一条规则，而符号“⇒”仅指推导一步的意思。严格地说，称 αAβ 直接推出 αγβ，即：

$$\alpha A\beta\Rightarrow\alpha\gamma\beta$$

仅当 A→γ 是一个产生式。

如果 $\alpha_1\Rightarrow\alpha_2\Rightarrow\cdots\Rightarrow\alpha_n$，称这个序列是从 α_1 至 α_n 的一个推导，也可称直接归约序列 α_n、α_{n-1}、…、α_1 为 α_n 到 α_1 的一个归约。若存在一个从 α_1 至 α_n 的推导，则称 α_1 可推导出 α_n。用 $\alpha_1\overset{+}{\Rightarrow}\alpha_n$ 表示：从 α_1 出发，经一步或若干步可推导出 α_n；用 $\alpha_1\overset{*}{\Rightarrow}\alpha_n$ 表示：从 α_1 出发，经 0 步或若干步可推导出 α_n。换言之，$\alpha\overset{*}{\Rightarrow}\beta$ 意味着或者 $\alpha=\beta$、或者 $\alpha\overset{+}{\Rightarrow}\beta$。

假定 G 是一个文法，S 是它的开始符号。如果 $S\overset{*}{\Rightarrow}\alpha$，则称 α 是 G 的一个句型，仅含终结符的句型是一个句子。文法 G 所产生的句子的全体称为文法的语言，记作 L(G)。

$$L(G)=\{\alpha\mid S\overset{+}{\Rightarrow}\alpha\ ,\alpha\in V_T^{+}\}$$

从 E 至(i*i+i)的一个推导为：

$$E\Rightarrow(E)\Rightarrow(E+E)\Rightarrow(E*E+E)\Rightarrow(i*E+E)\Rightarrow(i*i+E)\Rightarrow(i*i+i)$$

推导过程中的文法符号串 E、(E)、(E+E)、(E*E+E)、(i*E+E)、(i*i+E)、(i*i+i)都是这个文法的句型，而(i*i+i)是这个文法的句子。

设 G_1 和 G_2 是两个不同的文法，若 $L(G_1)=L(G_2)$，则称 G_1 和 G_2 是等价文法。等价文法的存在，使我们能够在不改变文法所规定语言的前提下，为了某种目的修改文法。

由于在一个句型中可能存在多个非终结符，故从一个句型到下一个句型的推导过程往往不是唯一的。为了对句子进行确定性分析，只考虑最左推导和最右推导。所谓最左推导是指：任何一步 α⇒β，都是对 α 中最左非终结符进行替换，并且用 $\alpha\underset{L}{\Rightarrow}\beta$ 表示。同样，可定义最右推导，最右推导用 $\alpha\underset{R}{\Rightarrow}\beta$ 表示。例如，句子(i*i+i)的最左和最右推导如下所示：

$$E\underset{L}{\Rightarrow}(E)\underset{L}{\Rightarrow}(E+E)\underset{L}{\Rightarrow}(E*E+E)\underset{L}{\Rightarrow}(i*E+E)\underset{L}{\Rightarrow}(i*i+E)\underset{L}{\Rightarrow}(i*i+i)$$

$$E\underset{R}{\Rightarrow}(E)\underset{R}{\Rightarrow}(E+E)\underset{R}{\Rightarrow}(E+i)\underset{R}{\Rightarrow}(E*E+i)\underset{R}{\Rightarrow}(E*i+i)\underset{R}{\Rightarrow}(i*i+i)$$

最后，作为描述程序设计语言的上下文无关文法，对它做两点限制：

(1) 文法中不存在形如 P→P 这样的产生式。

(2) 每个非终结符 P 必须都有用处。这一方面意味着，必须存在包含 P 的句型。也就是说，从开始符号 S 出发，存在推导 $S\overset{*}{\Rightarrow}\alpha P\beta$；另一方面意味着，必须存在终结符串 $\gamma\in V_T^{*}$，使得 $P\overset{+}{\Rightarrow}\gamma$。也就是说，对于 P 不存在永不终结的回路。

以后本书中讨论的文法，均假定满足上述两个条件，这种文法称为简化了的文法。

3.2.2　文法的二义性

可以用一张图表示一个句型的推导，这种表示称为语法树，语法树有助于理解一个句子

语法结构的层次。语法树通常表示成一棵倒立的树,语法树的根结点由开始符号标记。随着推导的展开,当某个非终结符被它的某个候选式所替换时,这个非终结符的相应结点就产生下一代新结点,候选式中自左至右的每个符号对应一个新结点,并用这些符号标记相应的新结点,每个新结点和其父结点都有一条连线。在一棵树的生长过程中的任何时刻,所有没有后代的端末结点自左至右排列就是句型。

语法树是句型不同推导过程的共性抽象。如果坚持用最左(最右)推导,那么一棵语法树就完全等价于一个最左(最右)推导,这样树的步步生成和句型推导的步步展开是完全一致的。但是,一个句型是否只对应唯一的一棵语法树呢?也就是说,它只有唯一的一个最右(最左)推导呢?答案是否。例如,文法 G:

$$E \rightarrow E+E \mid E*E \mid (E) \mid i$$

关于句型 i*i+i 就存在两棵语法树。

先形成+后形成*的语法树,如图 3.2 所示。

先形成*后形成+的语法树,如图 3.3 所示。

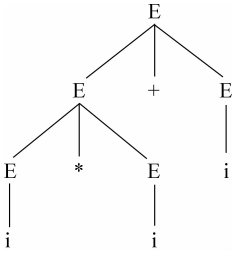

图 3.2

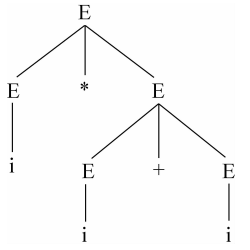

图 3.3

若采用推导的方法,对于句子 i+i*i,可获得两个最左推导序列,如下所示:

$$E \underset{L}{\Rightarrow} E+E \underset{L}{\Rightarrow} i+E \underset{L}{\Rightarrow} i+E*E \underset{L}{\Rightarrow} i+i*E \underset{L}{\Rightarrow} i+i*i$$

$$E \underset{L}{\Rightarrow} E*E \underset{L}{\Rightarrow} E+E*E \underset{L}{\Rightarrow} i+E*E \underset{L}{\Rightarrow} i+i*E \underset{L}{\Rightarrow} i+i*i$$

如果对句子 i+i*i 进行最右推导,也同样可获得两个最右推导序列,如下所示:

$$E \underset{R}{\Rightarrow} E+E \underset{R}{\Rightarrow} E+E*E \underset{R}{\Rightarrow} E+E*i \underset{R}{\Rightarrow} E+i*i \underset{R}{\Rightarrow} i+i*i$$

$$E \underset{R}{\Rightarrow} E*E \underset{R}{\Rightarrow} E*i \underset{R}{\Rightarrow} E+E*i \underset{R}{\Rightarrow} E+i*i \underset{R}{\Rightarrow} i+i*i$$

如果一个文法所产生的语言中,存在一个句子,该句子对应两棵不同的语法树,则称这个文法是二义的。也就是说,若一个文法中存在某个句子,它有两个不同的最左(最右)推导,则称这个文法是二义文法。上例中的文法 G 就是一个二义文法。

文法的二义性和文法的语言是两个不同的概念。可能有两个不同的文法 G 和 G',其中一个是二义的,而另一个是非二义的,但却有 L(G)=L(G')。也就是说,这两个文法产生的语言是相同的。对于一个程序设计语言来说,通常希望描述它的文法是无二义的,这样对语言中每个句子的分析是确定的。但是,只要能够控制和驾驭文法的二义性,文法二义性的存在并不一定是件坏事。

人们已经证明,二义性是不可判定的,即不存在一个算法,它能在有限步内判定一个文法是否是二义文法。所能做的是,为无二义性寻找一组充分条件。例如,可以规定运算符的

结合性和优先性来消除文法的二义性。运算符的优先性和结合性，是对于相邻两个运算符而言的。如果两个运算符之间有一个运算量，也认为两个运算符相邻。比方说，让 * 优先于＋，同级运算服从左结合，那么可构造出一个无二义等价文法 G'：

1　E→E＋T｜T

2　T→T * F｜F

3　F→(E)｜i

其中，E 代表＜算术表达式＞、T 代表＜项＞、F 代表＜因子＞。在这个文法中，i＋i * i 的语法树是唯一的，语法树如图 3.4 所示。

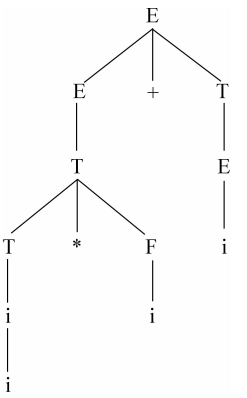

图　3.4

根据文法 G'，必须先推出＋，才可推出 * 。若先推出 * ，再推＋的话，那么必须增添括号，这样和所要求的目标句子不相符合。

3.3　文法举例

下面是一个无说明语句的简单程序设计语言文法，该文法只定义了赋值语句。赋值号的左边是标识符(变量)，赋值号的右边是算术表达式。算术表达式含有一元负和一元正以及加减乘除 6 种运算。只有一个运算对象的运算称为单目运算，有两个运算对象的运算称为双目运算。单目运算优先于双目运算；在双目运算中，乘优先于加；同级双目运算服从左结合，同级单目运算服从右结合；可以用括号改变运算顺序。运算对象可以是变量，也可以是无符号整数或无符号实数，可使用一元负和一元正运算来实现常数的正负。程序可由一个或多个赋值语句构成，需用 begin 和 end 括起，整个程序相当于一个复合语句。

1	＜程序＞→begin＜语句串＞end	P→{L}
2	＜语句串＞→＜语句串＞;＜语句＞	L→L;S
3	＜语句串＞→＜语句＞	L→S
4	＜语句＞→标识符＝＜算术表达式＞	S→i＝E
5	＜算术表达式＞→＜算术表达式＞＋＜项＞	E→E＋T
6	＜算术表达式＞→＜算术表达式＞－＜项＞	E→E－T
7	＜算术表达式＞→＜项＞	E→T
8	＜项＞→＜项＞ * ＜因子＞	T→T * F
9	＜项＞→＜项＞/＜因子＞	T→T/F
10	＜项＞→＜因子＞	T→F
11	＜因子＞→(＜算术表达式＞)	F→(E)
12	＜因子＞→－＜因子＞	F→－F
13	＜因子＞→＋＜因子＞	F→＋F
14	＜因子＞→标识符	F→i
15	＜因子＞→无符号整数	F→x
16	＜因子＞→无符号实数	F→y

请留意文法中的分号“;”,它的作用和 C 语言有所不同,分号并不是语句的组成部分,而是作为界符用于分隔语句。在语义分析(中间代码产生)时将使用分号,在本书中语句串的翻译是借助分号来实现的。

设有源程序:

```
1    Begin
2        area=2*3.14*10*(radius+height)
3    End
```

经词法分析,单词种别序列如下所示:

$$\{i=x*y*x*(i+i)\}$$

因为存在下述最左推导:

$P \underset{L}{\Rightarrow} \{L\} \underset{L}{\Rightarrow} \{S\} \underset{L}{\Rightarrow} \{i=E\} \underset{L}{\Rightarrow} \{i=T\} \underset{L}{\Rightarrow} \{i=T*F\} \underset{L}{\Rightarrow} \{i=T*F*F\} \underset{L}{\Rightarrow} \{i=T*F*F*F\} \underset{L}{\Rightarrow} \{i=F*F*F*F\} \underset{L}{\Rightarrow} \{i=x*F*F*F\} \underset{L}{\Rightarrow} \{i=x*y*F*F\} \underset{L}{\Rightarrow} \{i=x*y*x*F\} \underset{L}{\Rightarrow} \{i=x*y*x*(E)\} \underset{L}{\Rightarrow} \{i=x*y*x*(E+T)\} \underset{L}{\Rightarrow} \{i=x*y*x*(T+T)\} \underset{L}{\Rightarrow} \{i=x*y*x*(F+T)\} \underset{L}{\Rightarrow} \{i=x*y*x*(i+T)\} \underset{L}{\Rightarrow} \{i=x*y*x*(i+F)\} \underset{L}{\Rightarrow} \{i=x*y*x*(i+i)\}$

所以:

$$\{i=x*y*x*(i+i)\}$$

是文法的一个句子,也就是说源程序语法正确。

上述文法仅仅定义了赋值语句,随着讨论的深入,语句种类会不断地增加,非终结符 S 为今后语言的扩展提供了一个口子。

习　题

3-1　令+、*和^分别代表加、乘和乘幂,按如下非标准优先级和结合性质的约定,计算 1+1*2^2*1^2 的值。

(1) 优先顺序(从高到低)为+、*、^,同级运算服从左结合。

(2) 优先顺序(从高到低)为^、+、*,同级运算服从右结合。

提示:运算符的结合性和优先性是对于相邻两个运算符而言,若两个运算符之间有一个运算量,也认为两个运算符相邻。

3-2　已知文法 G 为:

1　E→T|E+T|E−T

2　T→F|T*F|T/F

3　F→(E)|i

(1) 证明 i−i/i 是文法 G 的一个句型。

(2) 画出 i−i/i 的语法树。

3-3　已知文法 G 为:

1　N→D|ND

2　D→0|1|2|3|4|5|6|7|8|9

(1) 文法 G 给出的语言 L(G)是什么?

(2) 给出句子 34、568 和 0127 的最左推导和最右推导。

3-4　已知程序段(用 C 语言表示)

```
1    a=2;
2    if(x) if(y)  a=4;  else  a=6;
```

(1) 假设 else 和最近的 if 结合,即 if(x){if(y)a=4;else a=6;}。当 x 和 y 为下列值时,求出相应 a 的值,如表 3.1 所示。

(2) 假设 else 和最远的 if 结合,即 if(x){if(y)a=4;}else a=6;。当 x 和 y 为下列值时,求出相应 a 的值,如表 3.1 所示。

表　3.1

x	y	a	x	y	a
flase	flase		true	flase	
flase	true		true	true	

3-5　已知文法 G:

1　＜语句＞→if 标识符 then＜语句＞else＜语句＞　　S→fitSeS

2　＜语句＞→if 标识符 then＜语句＞　　S→fitS

3　＜语句＞→标识符＝＜算术表达式＞　　S→i＝E

4　＜算术表达式＞→无符号整数　　E→x

(1) 用最左推导方法证明文法 G 是二义的。

(2) 消除文法的二义性。

3-6　用画语法树方法证明下列文法 G 是二义的。

S→fSeS|fS|i

3-7　文法如 3.3 节所示,用最左推导证明下列源程序语法正确。

```
1    Begin
2        x=-1;y=+1.0
3    End
```

3-8　已知文法 G:

1　＜程序＞→begin＜语句串＞end　　P→{L}

2　＜语句串＞→＜语句串＞;＜语句＞　　L→L;S

3　＜语句串＞→＜语句＞　　L→S

4　＜语句＞→integer＜标识符串＞　　S→aV

5　＜语句＞→real＜标识符串＞　　S→cV

6　＜标识符串＞→＜标识符串＞,标识符　　V→V,i

7　＜标识符串＞→标识符　　V→i

8　＜语句＞→标识符＝＜算术表达式＞　　S→i＝E

9　＜算术表达式＞→＜算术表达式＞＋＜项＞　　E→E＋T

10 ＜算术表达式＞→＜算术表达式＞－＜项＞ E→E－T
11 ＜算术表达式＞→＜项＞ E→T
12 ＜项＞→＜项＞＊＜因子＞ T→T＊F
13 ＜项＞→＜项＞/＜因子＞ T→T/F
14 ＜项＞→＜因子＞ T→F
15 ＜因子＞→(＜算术表达式＞) F→(E)
16 ＜因子＞→－＜因子＞ F→－F
17 ＜因子＞→＋＜因子＞ F→＋F
18 ＜因子＞→标识符 F→i
19 ＜因子＞→无符号整数 F→x
20 ＜因子＞→无符号实数 F→y

上述文法是在 3.3 节中的文法基础上,增加了说明语句。说明语句可出现在程序的任何地方,这一点和 C++ 相类似。用最左推导证明下列源程序是文法的一个合法句子。

```
1    Begin
2        real area,radius,height;
3        area=2*3.14*10*(radius+height)
4    End
```

3-9 修改习题 3-8 中的文法 G,使得在说明语句中可对变量赋初值。初值可以是常数,也可以是变量,还可以是由常数和变量构成的算术表达式。

3-10 设计一个文法,由文法产生的语言是一个奇数集合,集合中的每个奇数都不以 0 开头。

习 题 答 案

3-1 解(1):1＋1＊2^2＊1^2＝2＊2^2＊1^2＝4^2＊1^2＝4^2^2＝16^2＝256

解(2):1＋1＊2^2＊1^2＝2＊2^2＊1^2＝2＊4＊1^2＝2＊4＊1＝2＊4＝8

3-2 解(1):因为 E⇒E－T⇒E－T/F⇒T－T/F⇒F－T/F⇒i－T/F⇒i－F/F⇒i－F/i⇒i－i/i,所以 i－i/i 是文法 G 的一个句型。

解(2):i－i/i 的语法树如图 3.5 所示。

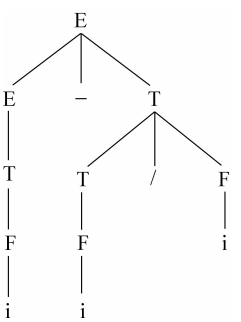

图 3.5

3-3 解(1):L(G)为无符号整数全体。

解(2):

$N \underset{L}{\Rightarrow} ND \underset{L}{\Rightarrow} DD \underset{L}{\Rightarrow} 3D \underset{L}{\Rightarrow} 34$

$N \underset{R}{\Rightarrow} ND \underset{R}{\Rightarrow} N4 \underset{R}{\Rightarrow} D4 \underset{R}{\Rightarrow} 34$

$N \underset{L}{\Rightarrow} ND \underset{L}{\Rightarrow} NDD \underset{L}{\Rightarrow} DDD \underset{L}{\Rightarrow} 5DD \underset{L}{\Rightarrow} 56D \underset{L}{\Rightarrow} 568$

$N \underset{R}{\Rightarrow} ND \underset{R}{\Rightarrow} N8 \underset{R}{\Rightarrow} ND8 \underset{R}{\Rightarrow} N68 \underset{R}{\Rightarrow} D68 \underset{R}{\Rightarrow} 568$

$N \underset{L}{\Rightarrow} ND \underset{L}{\Rightarrow} NDD \underset{L}{\Rightarrow} NDDD \underset{L}{\Rightarrow} DDDD \underset{L}{\Rightarrow} 0DDD \underset{L}{\Rightarrow} 01DD \underset{L}{\Rightarrow} 012D \underset{L}{\Rightarrow} 0127$

$$N \underset{R}{\Rightarrow} ND \underset{R}{\Rightarrow} N7 \underset{R}{\Rightarrow} ND7 \underset{R}{\Rightarrow} N27 \underset{R}{\Rightarrow} ND27 \underset{R}{\Rightarrow} N127 \underset{R}{\Rightarrow} D127 \underset{R}{\Rightarrow} 0127$$

3-4　解(1)：假设 else 和最近的 if 结合，相应 a 的值如表 3.2 所示。

表　3.2

x	y	a	x	y	a
flase	flase	2	true	flase	6
flase	true	2	true	true	4

解(2)：假设 else 和最远的 if 结合，相应 a 的值如表 3.3 所示。

表　3.3

x	y	a	x	y	a
flase	flase	6	true	flase	2
flase	true	6	true	true	4

3-5　证明(1)：

if x then if y then a＝4 else a＝6

的单词种别序列为：

fitfiti＝xei＝x

两个最左推导如下所示：

$S \underset{L}{\Rightarrow} fitSeS \underset{L}{\Rightarrow} fitfitSeS \underset{L}{\Rightarrow} fitfiti＝EeS \underset{L}{\Rightarrow} fitfiti＝xeS \underset{L}{\Rightarrow} fitfiti＝xei＝E \underset{L}{\Rightarrow} fitfiti＝xei＝x$

$S \underset{L}{\Rightarrow} fitS \underset{L}{\Rightarrow} fitfitSeS \underset{L}{\Rightarrow} fitfiti＝EeS \underset{L}{\Rightarrow} fitfiti＝xeS \underset{L}{\Rightarrow} fitfiti＝xei＝E \underset{L}{\Rightarrow} fitfiti＝xei＝x$

因为句子 fitfiti＝xei＝x 存在两个最左推导，所以文法 G 为二义文法。

解(2)：

1　＜语句＞→if 标识符 then＜语句＞else＜语句＞　　S→fitSeS

2　＜语句＞→if 标识符 then＜语句＞endif　　S→fitSj

3　＜语句＞→标识符＝＜算术表达式＞　　S→i＝E

4　＜算术表达式＞→无符号整数　　E→x

3-6　解：句子 ffiei 存在两棵语法树，如图 3.6 和图 3.7 所示，所以文法 G 是二义文法。

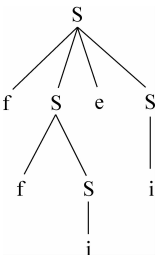

图　3.6

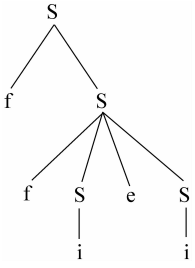

图　3.7

3-7　解：

```
1    Begin
```

```
2       x=-1;y=+1.0
3   End
```

的单词种别序列为：

$$\{i=-x;i=+y\}$$

最左推导如下所示：

$P \underset{L}{\Rightarrow} \{L\} \underset{L}{\Rightarrow} \{L;S\} \underset{L}{\Rightarrow} \{S;S\} \underset{L}{\Rightarrow} \{i=E;S\} \underset{L}{\Rightarrow} \{i=T;S\} \underset{L}{\Rightarrow} \{i=F;S\} \underset{L}{\Rightarrow} \{i=-F;S\} \underset{L}{\Rightarrow} \{i=-x;S\} \underset{L}{\Rightarrow} \{i=-x;i=E\} \underset{L}{\Rightarrow} \{i=-x;i=T\} \underset{L}{\Rightarrow} \{i=-x;i=F\} \underset{L}{\Rightarrow} \{i=-x;i=+F\} \underset{L}{\Rightarrow} \{i=-x;i=+y\}$

因为 $P \overset{+}{\Rightarrow} \{i=-x;i=+y\}$，所以源程序语法正确。

3-8 解：

```
1   Begin
2       real area,radius,height;
3       area=2*3.14*10*(radius+height)
4   End
```

的单词种别序列为：

$$\{ci,i,i;i=x*y*x*(i+i)\}$$

最左推导如下所示：

$P \underset{L}{\Rightarrow} \{L\} \underset{L}{\Rightarrow} \{L;S\} \underset{L}{\Rightarrow} \{S;S\} \underset{L}{\Rightarrow} \{cV;S\} \underset{L}{\Rightarrow} \{cV,i;S\} \underset{L}{\Rightarrow} \{cV,i,i;S\} \underset{L}{\Rightarrow} \{ci,i,i;S\} \underset{L}{\Rightarrow} \{ci,i,i;i=E\} \underset{L}{\Rightarrow} \{ci,i,i;i=T\} \underset{L}{\Rightarrow} \{ci,i,i;i=T*F\} \underset{L}{\Rightarrow} \{ci,i,i;i=T*F*F\} \underset{L}{\Rightarrow} \{ci,i,i;i=T*F*F*F\} \underset{L}{\Rightarrow} \{ci,i,i;i=F*F*F*F\} \underset{L}{\Rightarrow} \{ci,i,i;i=x*F*F*F\} \underset{L}{\Rightarrow} \{ci,i,i;i=x*y*F*F\} \underset{L}{\Rightarrow} \{ci,i,i;i=x*y*x*F\} \underset{L}{\Rightarrow} \{ci,i,i;i=x*y*x*(E)\} \underset{L}{\Rightarrow} \{ci,i,i;i=x*y*x*(E+T)\} \underset{L}{\Rightarrow} \{ci,i,i;i=x*y*x*(T+T)\} \underset{L}{\Rightarrow} \{ci,i,i;i=x*y*x*(F+T)\} \underset{L}{\Rightarrow} \{ci,i,i;i=x*y*x*(i+T)\} \underset{L}{\Rightarrow} \{ci,i,i;i=x*y*x*(i+F)\} \underset{L}{\Rightarrow} \{ci,i,i;i=x*y*x*(i+i)\}$

因为 $P \overset{+}{\Rightarrow} \{ci,i,i;i=x*y*x*(i+i)\}$，所以源程序是文法的一个合法句子。

3-9 解：

1	＜程序＞→begin＜语句串＞end	P→{L}
2	＜语句串＞→＜语句串＞;＜语句＞	L→L;S
3	＜语句串＞→＜语句＞	L→S
4	＜语句＞→integer＜标识符串＞	S→aV
5	＜语句＞→real＜标识符串＞	S→cV
6	＜标识符串＞→＜标识符串＞,标识符	V→V,i
7	＜标识符串＞→＜标识符串＞,标识符＝＜算术表达式＞	V→V,i=E
8	＜标识符串＞→标识符	V→i
9	＜标识符串＞→标识符＝＜算术表达式＞	V→i=E
10	＜语句＞→标识符＝＜算术表达式＞	S→i=E
11	＜算术表达式＞→＜算术表达式＞＋＜项＞	E→E+T
12	＜算术表达式＞→＜算术表达式＞－＜项＞	E→E－T
13	＜算术表达式＞→＜项＞	E→T

14　＜项＞→＜项＞＊＜因子＞　　T→T＊F

15　＜项＞→＜项＞/＜因子＞　　T→T/F

16　＜项＞→＜因子＞　　T→F

17　＜因子＞→(＜算术表达式＞)　　F→(E)

18　＜因子＞→－＜因子＞　　F→－F

19　＜因子＞→＋＜因子＞　　F→＋F

20　＜因子＞→标识符　　F→i

21　＜因子＞→无符号整数　　F→x

22　＜因子＞→无符号实数　　F→y

3-10　解：

1　＜奇数字＞→1|3|5|7|9

2　＜非零数字＞→2|4|6|8|＜奇数字＞

3　＜数字＞→0|＜非另数字＞

4　＜无符号整数＞→＜无符号整数＞＜数字＞|＜数字＞

5　＜奇数＞→＜奇数字＞|＜非零数字＞＜奇数字＞|＜非零数字＞＜无符号整数＞＜奇数字＞

第 4 章　自上而下的语法分析

先讨论自上而下的语法分析。顾名思义，自上而下就是从文法的开始符号出发，向下推导，最终推出句子。首先简单介绍自上而下语法分析的一般方法，这种方法是“带回溯”的。由此引出“不带回溯”的自上而下语法分析方法，它们是递归下降分析法和预测分析法。

4.1　带回溯的自上而下分析法概述

自上而下分析的宗旨是：对于输入串(由单词种别构成)，试图用一切可能的方法，从文法开始符号出发，自上而下地为输入串建立一棵语法树。或者说，为输入串寻找一个最左推导。这种分析过程本质上是一种试探过程，是反复使用不同的产生式，谋求匹配输入串的过程。

下面用一个简单例子来说明，设有文法 G：

1　S→xAy

2　A→＊＊｜＊

和输入串“x＊y”。

为了自上而下构造语法树，首先产生根结点，根结点由文法开始符号 S 标记，并让指示器 P 指向输入串的第 1 个符号“x”。然后用左部符号是 S 的产生式，把这棵树发展为如图 4.1 所示。

我们希望用 S 的子结点，从左至右匹配整个输入串。此树的最左子结点是用终结符 x 标记的，它和输入串的第 1 个符号相匹配。于是把指示器 P 调整为指向下一个输入符号“＊”，并让第 2 个子结点 A 去进行匹配。非终结符 A 有两个候选，先用它的第 1 个候选去推导，由 A 产生它的两个后代结点，两个后代结点的标记均为＊，如图 4.2 所示。

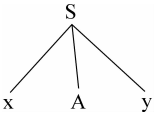

图　4.1

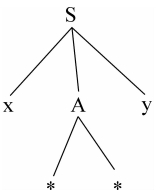

图　4.2

因子树 A 的最左子结点的标记和指示器 P 所指的输入符号相同，所以把指示器 P 调整为指向下一个输入符号“y”，并让 A 的第 2 个子结点进入工作。A 的第 2 个子结点的标记为＊，和指示器 P 所指的输入符号“y”不一致，这意味着 A 的第 1 个候选式不能用于构造输入串的语法树，此时应该回头(回溯)，查看 A 是否有其他候选。

为了这种回溯，一方面应把 A 的第 1 个候选所发展的子树注销掉，另一方面还应把指

示器 P 恢复为进入 A 时的状态，也就是让 P 重新指向第 2 个输入符号“*”。现在试探 A 的第 2 个候选，如图 4.3 所示。

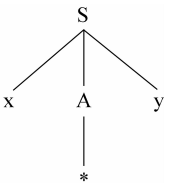

图　4.3

由于子树 A 只有一个子结点，而且它的标记和指示器 P 所指的输入符号“*”相一致，这样 A 完成了匹配任务。在 A 获得匹配后，指示器 P 应指向下一个输入符号“y”。

在 S 的第 2 个子结点 A 完成匹配后，接着轮到第 3 个子结点进行工作。由于这个子结点的标记和最后一个输入符号“y”相同，最终完成了为输入串构造语法树的任务，证明了输入串“x * y”是文法的一个句子。

上述这种自上而下的语法分析方法存在许多困难和缺点。首先是文法的左递归问题，设文法含有如下形式的左递归规则：

$$P \rightarrow P\alpha$$

其中 $P \in V_N$，$\alpha \in (V_T \cup V_N)^+$，则称该文法为左递归文法。左递归文法将使自上而下的分析过程陷入死循环中。当使用最左推导，试图用 P 去匹配输入串时，会发现在没有读入任何输入符号的情况下，需要重新要求 P 去进行新的匹配。因此，要使用自上而下的语法分析方法，文法不能含有左递归。

其次，在自上而下分析过程中，当一个非终结符使用某一候选式进行推导时，候选式中可能有部分子结点匹配于输入符号，有时这种匹配是虚假的。从上述分析中可以看到，A 首先使用第 1 个候选，该候选的第 1 个子结点和输入符号匹配，这个匹配就是虚假的。在检查下一个输入符号时，该匹配马上被推翻。由于这种虚假匹配现象，需要使用较为复杂的回溯技术。一般来说，要消除虚假匹配是困难的，可先使用较长的候选式进行推导，这样虚假匹配的现象就会减少。

第三，编译程序的语法分析和语义分析通常是同时进行的。由于回溯，所做的一大堆工作必须推倒重来，这样既麻烦又费时。

第四，如果选用所有的不同候选组合，都不能为输入串建立一棵语法树，或者说，都不能为输入串寻找到一个最左推导，那么输入串存在语法错误。这种分析法最终只能告知输入串不是文法的一个句子，而无法告知输入串错在什么地方。

最后，带回溯的自上而下分析法实际上是一种穷举法，是一种穷尽一切可能的试探法，因此效率很低，代价极高。严重的低效使得这种分析法只在理论上有意义，几乎没有实用价值。

综上所述，必须消除分析过程中的回溯，找到一个不带回溯的分析方法。只有这种不带回溯的自上而下的语法分析方法，才是实际可使用的。

4.2　直接左递归的消除

根据上述讨论，要进行自上而下的语法分析，必须消除文法的左递归。程序设计语言文法的左递归通常是由左递归规则直接引起的，由规则推导产生间接左递归的情况较少见。有部分左递归规则只要稍加调整，就可使其成为右递归规则。对于右递归文法所定义的语言，可以采用自上而下的语法分析方法。

例如,定义无符号整数的文法 G:

1 ＜无符号整数＞→＜无符号整数＞＜数字＞|＜数字＞ N→ND|D

2 ＜数字＞→0|1|2|3|4|5|6|7|8|9 D→0|1|2|3|4|5|6|7|8|9

因为 N→ND 是左递归规则,所以文法 G 是左递归文法。可将第 1 条规则改为右递归规则,如下所示:

1 ＜无符号整数＞→＜数字＞＜无符号整数＞|＜数字＞ N→DN|D

2 ＜数字＞→0|1|2|3|4|5|6|7|8|9 D→0|1|2|3|4|5|6|7|8|9

修改后的文法称为 G',显然文法 G 和 G'是等价的,而 G'是右递归文法。对于有些左递归规则,不能采用简单交换方式。例如:

＜算术表达式＞→＜算术表达式＞＋＜项＞ E→E＋T

若将它改为:

＜算术表达式＞→＜项＞＋＜算术表达式＞ E→T＋E

两者是不等价的。前者规定＋运算服从左结合,后者则规定＋运算服从右结合。

下面讨论消除文法直接左递归的方法,假定关于非终结符 P 的规则为:

$$P\rightarrow P\alpha|\beta$$

其中,β 不以 P 开头。那么,可以把 P 的规则改为如下形式:

1 $P\rightarrow\beta P'$

2 $P'\rightarrow\alpha P'|\varepsilon$

因为两者推导出的句型均为 $\beta\alpha^n(n\geqslant 0)$,所以变换是等价的。为此付出的代价是:引进新的非终结符 P'和产生式 $P'\rightarrow\varepsilon$(或称 ε 产生式)。

设有文法 G:

1 $E\rightarrow E+T|T$

2 $T\rightarrow T*F|F$

3 $F\rightarrow(E)|i|x|y$

消除直接左递归后如下所示:

1 $E\rightarrow TE'$

2 $E'\rightarrow +TE'|\varepsilon$

3 $T\rightarrow FT'$

4 $T'\rightarrow *FT'|\varepsilon$

5 $F\rightarrow(E)|i|x|y$

一般而言,假设关于非终结符 P 的全部产生式为:

$$P\rightarrow P\alpha_1|P\alpha_2|\cdots|P\alpha_m|\beta_1|\beta_2|\cdots|\beta_n$$

其中 $\beta_i(1\leqslant i\leqslant n)$都不以 P 开头。可将 P 的规则改成如下等价形式,即可消除左递归。

$$P\rightarrow\beta_1P'|\beta_2P'|\cdots|\beta_nP'$$

$$P'\rightarrow\alpha_1P'|\alpha_2P'|\cdots|\alpha_mP'|\varepsilon$$

上述变换等价性证明如下:

$$P\rightarrow P\alpha_1|P\alpha_2|\cdots|P\alpha_m|\beta_1|\beta_2|\cdots|\beta_n$$

等价于:

$$P\rightarrow P(\alpha_1|\alpha_2|\cdots|\alpha_m)|(\beta_1|\beta_2|\cdots|\beta_n)$$

令 $\alpha=\alpha_1|\alpha_2|\cdots|\alpha_m$、$\beta=\beta_1|\beta_2|\cdots|\beta_n$，则上式为：

$$P\rightarrow P\alpha|\beta$$

消除直接左递归后为：

$$P\rightarrow \beta P'$$
$$P'\rightarrow \alpha P'|\varepsilon$$

用 $\alpha_1|\alpha_2|\cdots|\alpha_m$ 替代 α，用 $\beta_1|\beta_2|\cdots|\beta_n$ 替代 β，则有：

$$P\rightarrow(\beta_1|\beta_2|\cdots|\beta_n)P'$$
$$P'\rightarrow(\alpha_1|\alpha_2|\cdots|\alpha_m)P'|\varepsilon$$

等价于：

$$P\rightarrow\beta_1P'|\beta_2P'|\cdots|\beta_nP'$$
$$P'\rightarrow\alpha_1P'|\alpha_2\ P'|\cdots|\alpha_m\ P'|\varepsilon$$

4.3　不带回溯的自上而下分析法的基本原理

带回溯的自上而下的分析法实际上是一种试探法。设文法 G 有产生式：

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

从 A 出发进行最左推导时，首先选用 α_1，若分析成功最好，若分析不成功则改用 α_2，…，以此类推。若使用了所有的候选，都不能为输入串寻找到一个最左推导，则认为输入串不是文法的一个句子。然而，对于文法某一个句型而言，只要该文法不是二义文法，从非终结符 A 出发的最左推导只有一个候选式是正确的。如果该候选式获得成功匹配，那么这个匹配绝不会是虚假的；若该候选式无法完成匹配任务，则任何其他候选式也肯定无法完成。所谓消除回溯，就是在最左推导时，根据面临的输入符号去找出 A 的那个唯一正确的候选式。基本原理如下：

(1) 引入候选式的 first 集。

候选式的 first 集定义如下：

$$first(\alpha)=\{a|\alpha\overset{*}{\Rightarrow}a\cdots,a\in V_T\}$$

first(α)直观意义是：从候选式 α 出发，所有可能推导出的符号串的第 1 个终结符都属于这个集合。

设有文法 G：

1　E→TE'

2　E'→+TE'|ε

3　T→FT'

4　T'→*FT'|ε

5　F→(E)|i|x|y

求候选式 TE'的 first 集。

因为 TE'⇒FT'E'⇒(E)T'E'，所以(∈first(TE')。

因为 TE'⇒FT'E'⇒iT'E'，所以 i∈first(TE')。

因为 TE'⇒FT'E'⇒xT'E',所以 x∈first(TE')。

因为 TE'⇒FT'E'⇒yT'E',所以 y∈first(TE')。

由此可得 first(TE')={i、(、x、y}。

(2) 根据定义,求出每个候选式 α_i 的 first 集。设:

$$first(\alpha_1)=\{a_1、b_1、\cdots\},first(\alpha_2)=\{a_2、b_2、\cdots\},\cdots,first(\alpha_n)=\{a_n、b_n、\cdots\}$$

根据输入符号 code,选择候选式进行推导。

```
1  if code∈first(αi) then
2      用 A→αi 推导(1≤i≤n)
3  else
4      报错
5  end if
```

由于推导的唯一性,要求 $first(\alpha_i)\cap first(\alpha_j)=\{\}$,其中 $1\leqslant i,j\leqslant n、i\neq j$。

(3) 进一步考虑和修正。

考虑更一般性,非终结符 A 的候选式可能是空字 ε,即:

$$A\rightarrow\alpha_1|\alpha_2|\cdots|\alpha_n|\varepsilon$$

因使用规则 A→ε 进行推导时,无须任何字符匹配(或称 A 匹配于空字 ε),所以当输入符号不属于 $\alpha_i(1\leqslant i\leqslant n)$ 的 first 集时,不能简单地处理为报错。需进一步分析,A 匹配于空字 ε 可能是一个正确的选择。

设有文法 G:

1 S→aA

2 A→cAd|ε

及输入串“acd”。显然 first(aA)={a},first(cAd)={c}。

识别过程可用语法树来描述。首先产生根结点,根结点由文法的开始符号 S 标记,并让指示器 P 指向输入串的第 1 个输入符号“a”。因输入符号“a”∈first(aA),故用 S 规则把这棵语法树发展为如图 4.4 所示。若输入符号不是“a”,即输入符号 code∉first(aA),则报错,没有必要再分析下去。

于是把指示器 P 调整为指向下一个输入符号“c”,并让第 2 个子结点 A 去进行匹配。非终结符 A 有两个候选,因输入符号“c”∈first(cAd),此时应使用 A 的第 1 个候选去推导,于是把这棵语法树发展为图 4.5 所示。

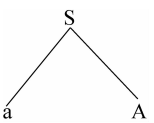

图 4.4

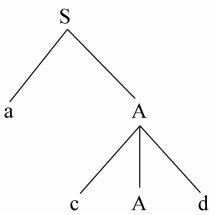

图 4.5

此时应把指示器 P 调整为指向下一个输入符号“d”,并让子树 A 的第 2 个标记为 A 的子结点进行工作。“d”∉first(cAd),显然不能用第 1 个候选去推导,也不能报错,应按 A→ε