第二章　形式文法和形式语言

形式语言与自动机理论是编译器设计的重要理论基础。高级程序设计语言是一种人造的形式语言，本章首先介绍形式语言与自然语言在词法、语法和语义方面的异同；接着介绍编译器设计中涉及到的一些有关形式文法和形式语言的基本概念，重点介绍如何采用形式化的方法描述程序设计语言，以及如何分析上下文无关文法的句型，这些内容是各类语法分析方法和语义分析方法的基础。本章对乔姆斯基形式语言体系也做了简要介绍。

2.1 形式语言与自然语言

自然语言（Natural Language）是人类在生产和生活中逐步进化和发展起来的，如汉语、英语、德语、西班牙语等。自然语言是信息的载体和人与人之间交换信息的媒介，也是人类思维的工具。据统计，全世界曾经存在的语言或者方言约有 5500 余种。自然语言是一种符号系统，有语言材料，有语法规则。自然语言的语法规则一般是有限的，语言材料也是有限的，而由语言材料按照语法规则组成的语句和篇章却是无限的。如汉字是汉语的基本语言材料，《中华大字典》收录汉字 48000 多个，计算机上常用的汉字国标码（GB2312-80）规定了一级汉字 3755 个，二级汉字 3008 个。有限的汉字可以组成“词语”，“词语”可以进一步组成无数个“语句”和“篇章”。

形式语言（Formal Language）是人们为了特定目的而设计的人工语言。与自然语言类似，形式语言也有语言材料和语法规则。高级程序设计语言（Advanced Programming Language）就是一种形式语言，用来编写计算机程序。高级程序设计语言都有自己的字符集，字符按照词法规则构成“单词”（如标识符、常数等），单词按照语法规则构成各级语法单位（如“表达式”、“赋值语句”等），并最终构成“程序”。“程序”是最高级的语法单位。

除了语法，自然语言和形式语言都有“语义”，即语法单位的含义。

（2.1）是高级语言程序中的两个表达式，它们的语义相同。

【例 2-1】：（1）a>b；

（2）b<a。　　（2.1）

（2.2）是两个自然语言语句，它们与（2.1）中的两个表达式结构相似，但是它们的意思却是相反的。

【例 2-2】：（1）中国队大胜美国队；

（2）美国队大败中国队。　　（2.2）

以上例子说明形式语言与自然语言有很大的不同，特别是在语义层面。自然语言的语义具有模糊性和歧义性，一字多义、一义多词的现象很普遍，如“盖”字在《现代汉语词典》中就有 10 种释义。自然语言语句和篇章的语义更加复杂。形式语言的“语义”一般不能有模糊性和歧义性。例如用程序设计语言编写的一个程序，其描述的算法和运行结果应该是唯一确定的，不能有歧义。

2.2 文法和语言的形式定义

2.2.1 一个自然语言的例子

在介绍形式文法和形式语言之前，先看一个自然语言（汉语）的例子。通过与自然语言的类比，可以更好地理解形式文法和形式语言中的相关概念。

一个微型汉语的语法规则：

（1）<句子>→<主语><谓语>

（2）<主语>→<代词>

（3）<主语>→<名词>

（4）<代词>→你

（5）<代词>→我

（6）<代词>→他

（7）<名词>→警察

（8）<名词>→小偷

（9）<名词>→儿童

（10）<谓语>→<动词><宾语>

（11）<动词>→抓捕

（12）<动词>→保护

（13）<宾语>→<代词>

（14）<宾语>→<名词>

以上语法规则中由“<”和“>”括起来的串表示一个有内部结构的语法单位。符号串如“警察”表示最小的不可分割的最小语法单位。符号“→”读作“由.....构成”或“定义为”，表示符号“→”左边的语法单位是由右边的语法单位序列构成的，或者左边的语法单位定义为右边的语法单位序列。以上语法中每条规则的意义为：

（1）<句子>由<主语>和<谓语>构成；

（2）<主语>由<代词>构成；

（3）<主语>由<名词>构成；

（4）<代词>定义为“你”；

（5）<代词>定义为“我”；

（6）<代词>定义为“他”；

（7）<名词>定义为“警察”；

（8）<名词>定义为“小偷”；

（9）<名词>定义为“儿童”；

（10）<谓语>由<动词>和<宾语>构成；

（11）<动词>定义为“抓捕”；

（12）<动词>定义为“保护”；

（13）<宾语>由<代词>构成；

（14）<宾语>由<名词>构成。

按照以上语法规则，可判断一个句子是否合法，如“警察抓捕小偷”、“他保护儿童”就是语法上合法的句子。注意，语法上合法的句子语义上未必正确，如“儿童保护小偷”。

可以通过反复使用语法规则来判断一个句子是否合法。以句子“警察抓捕小偷”为例，其判断过程为：

<句子>⇒<主语><谓语>（使用第 1 条规则<句子>→<主语><谓语>）

⇒<名词><谓语>（使用第 3 条规则<主语>→<名词>）

⇒警察<谓语>（使用第 7 条规则<名词>→警察）

⇒警察<动词><宾语>（使用第 10 条规则<谓语>→<动词><宾语>）

⇒警察抓捕<宾语>（使用第 11 条规则<动词>→抓捕）

⇒警察抓捕<名词>（使用第 14 条规则<宾语>→<名词>）

⇒警察抓捕小偷（使用第 8 条规则<名词>→小偷）

以上判断过程分为若干步骤，每一步使用一条语法规则，其中符号“⇒”表示一步变换，每步变换都是用语法规则中“→”的右部替换左部。以上判断过程可以用一棵树来刻画，这棵树常常被称为分析树，见图 2-1。这棵分析树也反映了该句子的层次结构或者说语法结构。

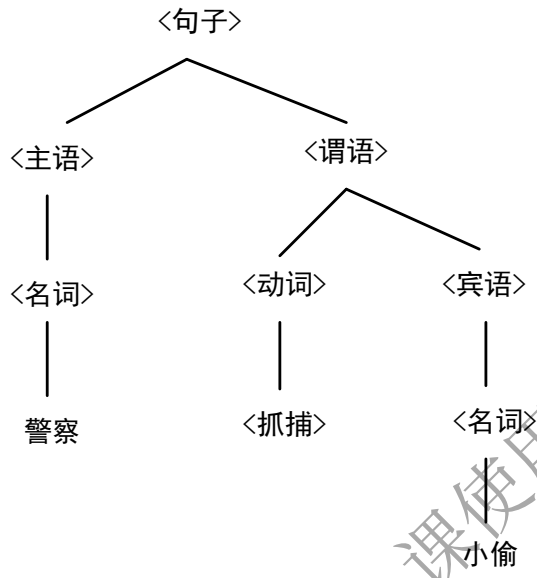

图 2-1 “警察抓捕小偷”的分析树

2.2.2 字母表和符号串

定义 2-1：字母表（Alphabet）是符号（Symbol）的非空有穷集合，记为 Σ 。符号是一个抽象实体，表示可以互相区分的记号或元素。字母表中至少包含一个元素，字母表中的元素，可以是字母、数字或其它符号。

例如：字母表 Σ ={a, b, c}，表示这个字母表由 a, b, c 三个符号组成。

在各种不同的应用及系统中有不同的字母表，如二进制的字母表是{0,1}。在早期的软件系统中经常使用 ASCII(American Standard Code for Information Interchange，美国标准信息交换代码)作为字母表。Unicode 包含大约 100000 个来自世界各地的字符，它是网络世界应用广泛的字母表。

就像英语的字母表是由 26 个英文字母构成的，每种形式语言也都有自己的字母表。C 语言的字母表是：{ A~Z, a~z, 0~9, +, -, *, /, <, =, >, ^, ~, |, &, !, #, ’, ”, ,, :, ;, ., (,), {, }, [,], _, ?, \, 空格 } 。除了符号串常量中可以出现其它字符，C 语言程序中不能出现除字母表之外的其它符号。

定义 2-2：符号串（Symbol String）是由字母表中的符号所组成的有穷序列，又称为句子（Sentence）或字（Word）。

注意，符号串总是建立在某个特定字母表上且只由字母表上的符号组成。

例如：Σ={a, b} ，则 a, b, aa, ab, aabba 等都是Σ上的符号串。而 abc 和 cb 不是Σ上的符号串，因为c ∉ Σ。

符号串中符号的顺序是很重要的。ab 和 ba 是两个不同的符号串。有一个特殊的符号串，它不包含任何一个符号，称之为**空符号串**，记为 ε 。 ε 是任何字母表上的符号串。

符号串的**长度**表示符号串中包含符号的个数，符号串 s 的长度记为|s|。

例如：Σ = {a, b}，aab 是该字母表上的一个符号串，则|aab|=3。

注意，空符号串 ε 的长度| ε |=0。

与符号串相关的还有几个概念，如前缀、后缀、子串、真前缀、真后缀、真子串、子序列等。这几个概念定义如下。

定义 2-3：

前缀（Prefix）：删去符号串 s 尾部的零个或多于零个符号得到的符号串；

后缀（Suffix）：删去符号串 s 头部的零个或多于零个符号得到的符号串；

子串（Substring）：从符号串 s 中删去一个前缀和一个后缀得到的符号串。

符号串 s 的**真（true）前缀**、**真后缀**和**真子串**定义为：既不等于 ε 也不等于 s 本身的前缀、后缀和子串。

符号串的**子序列**（Subsequence）是从 s 中删除 0 个或多个符号后得到的串，这些被删除的符号可能不相邻。

对符号串可以进行运算，下面定义符号串的两个常见运算。

定义 2-4：符号串的**连接**运算：符号串 α 、 β 的连接，是把 β 的符号写在 α 的符号之后得到的符号串，记为 α β 。

例如：α = ab, β = cd 则 α β = abcd， β α =cdab 。

注意，ε α = α ε = α 。

符号串的**方幂（指数）**运算：定义为符号串 α 进行自连接所得到的符号串。α^0 定义为空串，即 $\alpha^0 = \varepsilon$ ，并且对于 i>0， α^i 定义为 $\alpha^{i-1}\alpha$ 。即 $\alpha^1 = \alpha^0\alpha = \varepsilon\alpha = \alpha$， $\alpha^2 = \alpha^1\alpha = \alpha\alpha$，……， $\alpha^n = \alpha^{n-1}\alpha = \alpha\alpha\ldots\alpha\alpha$（n 个 α ）。

2.2.3 语言的非形式定义

定义 2-5：语言（ Language）的非形式定义：字母表Σ上的一个语言是Σ上的一些符号串的集合。即：语言是一个集合，是一个定义在某个字母表上的符号串的集合。

这个定义非常宽泛，只要是由符号串构成的集合都是语言。一个特殊的例子是只包含一个空串的集合{ ε }。这个集合也是语言，因为包含了一个空符号串 ε 。空集Φ是不包含任何一个符号串的集合，我们把空集Φ也看作是一个语言，称为**空语言**。

一般来说，语言中的每个符号串都要满足共同的构成规则。例如，所有语法上正确的 C 语言程序的集合是一个语言（即 C 语言）。

2.2.4 语言的运算

集合有很多运算，如并运算、交运算、差运算等。既然语言是符号串的集合，那么集合的这些运算对于语言同样适用。语言中的元素是符号串，有特殊性，因此语言也有一些特殊的运算。

下面给出针对语言的几个常见的运算的定义。

定义 2-6：语言的**并运算**：设 L 和 M 是两个语言，L 和 M 的并记为 L∪M，定义为：

$$L \cup M=\{s \mid s \in L \text{ 或者 } s \in M\}$$

定义 2-7：语言的**连接运算**：设 L 和 M 是两个语言，L 和 M 的连接记为 LM，定义为：

$$LM=\{st \mid s \in L \text{ 且 } t \in M\}$$

例如：集合 A ={ab, cde}，B ={0,1}，则 AB = { ab1,ab0,cde0,cde1 }。

注意，设 A 为任一语言，则{ ε }A = A { ε }= A。

定义 2-8：语言 L 的**正闭包运算**（也称+闭包运算）和 **Kleene 闭包**运算（也称*闭包运算），分别定义为：

$$L^{+}=L^{1} \cup L^{2} \cup \cdots \cup L^{n} \cdots=\bigcup_{i=1}^{\infty} L^{i}$$

$$L^{*}=L^{0} \cup L^{1} \cup L^{2} \cup \cdots \cup L^{n} \cdots=L^{0} \cup L^{+}=\bigcup_{i=0}^{\infty} L^{i}$$

特殊地，定义 $L^{0}=\{\varepsilon\}$。注意，除非 ε 属于 L，否则 ε 不属于 L^{+}。

如果把字母表中的每个符号看作是一个长度为 1 的“符号串”，字母表也就成为了“语言”，对字母表也就可以做语言的运算。后面经常对字母表Σ作+闭包运算和*闭包运算。

【例 2-3】：设Σ={ a, b }，则：

$$\Sigma^{+}=\{a, b, aa, ab, ba, bb, aaa, aab, \cdots\}$$

$$\Sigma^{*}=\{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \cdots\}$$

可见，字母表Σ的+闭包表示字母表中元素 a, b 构成的所有符号串的集合，字母表Σ的*闭包表示字母表中元素 a, b 构成的所有符号串加上一个空串 ε 所组成的集合。

下面看一个综合性的例子。

【例 2-4】：设字母表 L = { A，B，C，…，Z，a，b，c，…，z }，字母表 D = { 0，1，…，9 }，求以下运算的结果：（1） L ∪ D；（2）LD；（3）L^4；（4）L^*；（5）$L(L \cup D)^*$；（6）D^+。

解：（1）26 个大写英文字母，26 个小写英文字母加上 10 个阿拉伯数字组成的集合；

（2）第一个符号是字母，第二个符号是数字的长度为 2 的所有符号串组成的集合；

（3）由字母构成的长度为 4 的所有符号串组成的集合；

（4）由字母构成的任意长度的符号串加上空串 ε 所组成的集合；

（5）字母开始的由字母、数字构成的长度大于等于 1 的所有符号串组成的集合；

（6）所有由数字构成的长度大于等于 1 的符号串组成的集合。

2.2.5 语言的描述

语言是符号串的集合，那么如何来描述一种语言呢？

如果语言是有穷的（只含有有穷多个符号串），可以采用集合的列举法，即将语言中的符号串逐一列举出来表示。如果语言是无穷的，或者语言中的符号串个数多到难以列举，则需要另外寻找语言的表示方法。这种语言描述方法主要有两类，一类是所谓的识别方法，即为待描述的语言设计一个算法（或数学模型），当输入语言中的任意一个符号串时，该过程（或数学模型）经有限次计算后就会停止并回答“是”，若输入不属于该语言的符号串时，要么经有限次计算后停止并回答“不是”，要么永远计算下去。自动机就是以识别的方式来描述语言的，第三章介绍的有限自动机就是自动机的一种，对其它自动机的详细介绍超出了本书的范围。另外一类语言描述方法是所谓的生成方法，即为待描述的语言定义一套规则，语言中的任意一个符号串都可以用这套规则来构造，而语言之外的符号串根据这套规则都构造不出来。文法就是以生成的方式来描述语言的。

2.2.6 文法的形式定义

定义 2-9：一个**文法** G（Grammar），定义为一个四元组（V_T, V_N, S, P）。

其中，V_T 是一个非空有穷集合，其中的元素称为**终结符号**。终结符号代表语言中不可再分的最小语法单位，如 C 语言中的保留字。

V_N 也是一个非空有穷集合，其中的元素称为**非终结符号**。非终结符号代表语言中除最小语法单位之外的其它语法单位，如 C 语言中的语句、表达式、函数等。在这里要注意的是，V_T 和 V_N 中是没有共同元素的，即 $V_T \cap V_N = \phi$ 。

S 被称为**开始符号**，且 $S \in V_N$。它是一个特殊的非终结符号，代表最高级的语法单位。

P 是**产生式集合**，产生式又称为重写规则，或者生成式，是形如 $\alpha \rightarrow \beta$ 或 $\alpha ::= \beta$ 的（α, β）有序对，且 $\alpha \in V^+$，$\beta \in V^*$，其中 $V=(V_T \cup V_N)$，称为**文法符号集合**。α 称为产生式的**左部**，不能为空串 ε，且至少包含一个 V_N 中的元素，S 至少要在一条产生式中作为左部出现。β 称为产生式的**右部**，可以是空串 ε，如：$A \rightarrow \varepsilon$ 。

在后续文法书写中，通常遵循如下关于文法符号的约定。

终结符号的一般形式：

（1）在字母表里排在前面的小写字母，比如 a、b、c；

（2）运算符号，比如+、-、*、/；

（3）标点符号，比如括号、逗号等；

（4）阿拉伯数字 0、1、2、3、4、...、9；

（5）黑体加粗字符串，比如 **id**、**if** 等。

非终结符号的一般形式：

（1）在字母表中排在前面的大写字母，比如 A、B、C；

（2）大写字母 S。它出现时通常表示开始符号；

（3）小写、斜体的名字，比如 *expr* 或 *stmt*；

（4）用一对尖括号“<、>”括起来的一个字符串，如<表达式>；

（5）当讨论程序设计语言时，语法单位英文单词第一个字母的大写形式可以代表对应语法单位的非终结符号。比如，表达式（expression）、项（term）和因子（factor）的非终结符号通常用 E、T、F 表示。

在字母表中排在后面的大写字母（比如 X、Y、Z）表示单个文法符号，也就是说既可以表示非终结符号也可以表示终结符号。在字母表中排在后面的小写字母（主要是 u, v, ..., z）表示终结符号串（可能为空串）。小写的希腊字母，比如 α、β、γ，也可以表示文法符号串（可能为空串）。

下面介绍文法的几个例子。

【例 2-5】：文法 $G_1=(V_T, V_N, S, P)$，其中：

V_T={0, 1}

V_N={S}

P={S→0S1, S→01}

该文法只有一个非终结符号 S，它同时也是开始符号。

【例 2-6】：文法 G_2=（V_T, V_N, S, P），其中：

V_T={a,b,c,…,x,y,z,0,1,…,9}

V_N={<标识符>, <字母>, <数字>}

S=<标识符>

P={ <标识符>→<字母>,

<标识符>→<标识符><字母>,

<标识符>→<标识符><数字>,

<字母>→a,

<字母>→b,

…,

<字母>→z,

<数字>→0,

<数字>→1,

…,

<数字>→9 }

书写文法时可以遵循一些约定。一般来说，第一条产生式的左部是开始符号，根据文法符号使用的约定，在产生式中可以解析出文法的终结符号集合 V_T 和非终结符号集合 V_N。在不引起误解的情况下，后面书写文法时通常只列出文法的产生式。必要时在文法代号后面加上开始符号，如 G[E]。

对一组有相同左部的产生式，如：

$\alpha \rightarrow \beta_1$,

$\alpha \rightarrow \beta_2$,

…,

$$\alpha \rightarrow \beta_n$$

可以简写为：

$$\alpha \rightarrow \beta_1 | \beta_2 | \cdots | \beta_n$$

以上简写形式称为 α 的一组产生式，读作“α 产生 β_1，或者产生 β_2，…，或者产生 β_n”。

下面看一个常见的简单表达式文法。

【例 2-7】：考察文法 $G_3[E]$，产生式如下：

$$E \rightarrow E+E$$

$$E \rightarrow E*E$$

$$E \rightarrow (E)$$

$$E \rightarrow \mathbf{id}$$

以上文法可以简写为：

$$E \rightarrow E+E \mid E*E \mid (E) \mid \mathbf{id}$$

其中 E 是唯一的非终结符号，也是开始符号，**id** 表示变量标识符，是终结符号，终结符号还包括+、*、(、)。

文法可以描述大多数程序设计语言的语法结构，除 G_3 描述了表达式结构之外，可以再举两个例子。

例如，Java 语言中的 if-else 语句通常具有如下形式：

if (expression) statement **else** statement

即一个 if-else 语句是由关键字 if 、左括号、表达式、右括号、一个语句、关键字 else 和另一个语句连接构成。如果我们用非终结符号 *expr* 来表示表达式，用非终结符号 *stmt* 表示语句，那么这个语法结构可以用产生式描述为：

stmt → **if** (*expr*) *stmt* **else** *stmt*

在 Java 语言中，参数是包含在括号中的，例如 max(x,y)，表示使用参数 x 和 y 调用函数 max。生成这种结构的文法如下：

call → **id** (*optparams*)

optparams → *params* | ε　　（注：参数列表可为空）

params → *params*, **param** | **param**

2.2.7 推导与归约

定义 2-10：如果 A → γ 是文法 G 的一条产生式，则称用 α γ β 代替 α A β 为一步直接**推导**（Derivation），记为：

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

推导是文法符号串的一个变换过程，符号“⇒”读作“推出”，指用一条产生式的右部替换其左部。每一步推导都需要做两个选择，首先要选择替换哪个非终结符号，其次要选择使用该非终结符号的哪条产生式来推导，因为一个非终结符号可能有多条候选产生式。

（2.1）是基于【例 2-7】文法 G_3 的一次推导过程。

E⇒E+E（使用第 1 条产生式 E→E+E）

⇒E*E+E（使用第 2 条产生式 E→E*E）

⇒id*E+E（使用第 4 条产生式 E→id）　　（2.1）

⇒id*id+E（使用第 4 条产生式 E→id）

⇒id*id+id（使用第 4 条产生式 E→id）

定义 2-11：一次推导中直接推导的次数，称为**推导的长度**。如（2.1）这次推导长度为 5。如果只关心推导的起点和终点，则可以将推导过程简化。假设从 α 出发，经过一步或多于一步的推导得到 β，则可将这次推导记为：

$$A \overset{+}{\Rightarrow} \beta$$

假设从 α 出发，经过零步或多于零步的推导得到 β，可记为：

$$A \overset{*}{\Rightarrow} \beta$$

定义 2-12：如果 A → γ 是文法 G 的一条产生式，则称用 α A β 代替 α γ β 为一步直接**归约**（Reduce），记为：

$$\alpha\ \gamma\ \beta \Leftarrow \alpha A \beta$$

推导是用产生式的右部替换左部，而归约则是用产生式的左部替换右部，归约是推导的逆过程，同时推导也是归约的逆过程。

定义 2-13：从文法 G 的开始符号出发进行零步或多于零步的推导得到的文法符号串，即如果 $S \overset{*}{\Rightarrow} \alpha$，$\alpha \in (V_T \cup V_N)^*$，则称 α 为文法 G 的一个**句型**。只包含终结符号的句型，即如果 $S \overset{*}{\Rightarrow} \beta$，$\beta \in V_T^*$，则称 β 为文法 G 的一个**句子**。

例如：在推导（2.1）中，句型有：E+E、E*E+E、id*E+E、id*id+E 和 id*id+id，其中 id*id+id 是句子。

定义 2-14：最左推导（Left-most Derivation），即每次推导都施加在句型的最左边的非终结符号上的推导，记为 $\underset{lm}{\Rightarrow}$。**最右推导**（Right-most Derivation），即每次推导都施加在句型的最右边的非终结符号上的推导，记为 $\underset{rm}{\Rightarrow}$。最右推导又称为**规范推导**。

定义 2-15：最左归约（Left-most Reduce），即每次都是对句型中最左边的可归约串进行的归约。最左归约又称为**规范归约**。**最右归约**（Right-most Reduce），即每次都是对句型中最右边的可归约串进行的归约。

最左推导和最右归约互为逆过程，最右推导和最左归约互为逆过程。

基于文法 G_3，以句子(id+id)*id 作为推导目标，可以分别构造最左推导和最右推导，见（2.2）和（2.3）。

最左推导：

$$
\begin{aligned}
E &\underset{lm}{\Rightarrow} \underline{E*E} \\
&\underset{lm}{\Rightarrow} \underline{(E)}*E \\
&\underset{lm}{\Rightarrow} (\underline{E+E})*E \\
&\underset{lm}{\Rightarrow} (\underline{id}+E)*E \\
&\underset{lm}{\Rightarrow} (id+\underline{id})*E \\
&\underset{lm}{\Rightarrow} (id+id)*\underline{id}
\end{aligned}
\qquad (2.2)
$$

最右推导：

$$
\begin{aligned}
E &\underset{rm}{\Rightarrow} \underline{E*E} \\
&\underset{rm}{\Rightarrow} E*\underline{id} \\
&\underset{rm}{\Rightarrow} \underline{(E)}*id \\
&\underset{rm}{\Rightarrow} (\underline{E+E})*id \\
&\underset{rm}{\Rightarrow} (E+\underline{id})*id \\
&\underset{rm}{\Rightarrow} (\underline{id}+id)*id
\end{aligned}
\qquad (2.3)
$$

2.2.8 语言与文法

定义 2-16：语言的形式定义：文法 G 推导出的所有句子组成的集合，称为**语言**，记为 L(G)，即：

$$L(G)=\{x \mid S \overset{*}{\Rightarrow} x \text{ 且 } x \in V_T^*\}$$

L(G)是 V_T^*的子集，即属于 V_T^* 的符号串 x 不一定属于 L(G)。文法的作用是可以用有限的规则描述无限的语言现象，构成文法的终结符号集合、非终结符号集合、产生式集合都是有穷集合，但是却可以推导出无穷多个句子。文法给定了，它所描述的（唯一的一个）语言也就确定了，文法以生成的方式描述语言。

对于一个给定的文法 G，证明它描述了哪个语言 L 是很重要的。证明过程分两个步骤，一是证明文法 G 推导出的每个句子都在 L 中，二是证明 L 中的每个句子都可以由文法 G 推出。也就是要证明文法 G 不多不少正好推导出语言 L 中的句子。

【例 2-8】：考虑文法 G_4：

$$S \rightarrow (S)S \mid \varepsilon$$

这个文法只有两条产生式，但是却可以生成无穷多个句子。除了空串 ε，这些句子具有共同的特点，即它们都是由对称的括号对“(”和“)”构成的，并且它的每个前缀的左括号不少于右括号。如“()”（1 个括号对）、“()(())”（3 个括号对）等，空串 ε 可以看作是包含了零个括号对的句子。

要证明这个文法描述的是包含所有括号对称的句子的语言，首先需要证明从 S 推导得到的每个句子都是括号对称的，然后证明每个括号对称的句子都可以从 S 推导得到。

第一步：证明文法 G 推出的每个句子都是括号对称的。

归纳法：对推导步数 n 进行归纳。

（1）归纳基础：n=1。可以从 S 经过一步推导得到的句子只有一个，即空串 ε，它是括号对称的（含零个括号对）。

（2）归纳步骤：假设所有步数少于 n 的推导都能得到括号对称的句子，考察如下形式的包含 n 步的推导：

$$S \underset{lm}{\Rightarrow} (S)S \overset{*}{\underset{lm}{\Rightarrow}} (x)S \overset{*}{\underset{lm}{\Rightarrow}} (x)y$$

从 S 到 x 和 y 的推导过程都少于 n 步，根据归纳假设，x 和 y 都是括号对称的，因此，句子(x)y 也是括号对称的。

第二步：证明每个括号对称的句子都是可以由文法 G 推出的。

归纳法：对句子的长度进行归纳。

（1）归纳基础：如果句子的长度是 0 ，它必然是空串 ε 。 ε 是括号对称的，且可以从 S 一步推导得到。

（2）归纳步骤：容易理解，每个括号对称句子的长度是偶数。假设每个长度小于 2n 的括号对称的句子都能够从 S 推导得到，并考虑一个长度为 2n（n>=1）的括号对称的句子 w。w 一定以左括号开头。令(x)是 w 的最短的、左括号个数和右括号个数相同的非空前缀，那么 w 可以写成 w=(x)y 的形式，其中 x 和 y 都是括号对称的。因为 x 和 y 的长度都小于 2n ，根据归纳假设，它们可以从 S 推导得到。因此，我们可以找到一个如下形式的推导：

$$S \underset{lm}{\Rightarrow} (S)S \overset{*}{\underset{lm}{\Rightarrow}} (x)S \overset{*}{\underset{lm}{\Rightarrow}} (x)y$$

它证明 w=(x)y 也可以从 S 推导得到。

下面再举两个文法的例子，并考察其所描述的语言。

【例 2-9】：设有文法 G_5：

$S \rightarrow A$

$A \rightarrow 0A1$

$A \rightarrow 01$

问题：从开始符号 S 出发，将推出一些什么样的句子？也就是说，$L(G_5)$是由什么样的符号串组成的集合？

解：G_5 能推出的句子为 01, 0011, 000111,.....等等，句子都是以若干个 0 开头，后接相同数目的 1 构成，$L(G_5)$可以用如下式子描述：

$$L(G_5) = \{ 0^n 1^n \mid n \geqslant 1 \}$$

【例 2-10】：考察如下文法 G_6：

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

该文法可推出所有的由变量标识符 id，运算符号+、*，和括号(、)构成的算术表达式，如 id+id、id*id+id、id*(id+id)等。

给定一个文法如何确定其所描述的语言？可以从文法的开始符号出发，反复连续地使用产生式规则展开非终结符号以获得句子，并总结句子的结构特征，然后可以用式子或自然语言来描述该语言。

如果给定一个语言（句子的集合），如何设计生成该语言的文法呢？设计一个文法来描述一个语言，关键是设计一组产生式规则，用来生成语言中的句子。因此同样必须分析语言中句子的结构特征。

【例 2-11】：设字母表Σ={a, b}，试设计一个文法，描述语言 L_1= { $a^{2n}b^{2m}$ | n≥0，m≥0，n 和 m 不同时为 0 }。

解：L_1 中的句子都是以偶数个 a 开头，以偶数个 b 结尾的，可以设计文法 G_7 来生成这个语言，G_7 的文法产生式如下：

S→aa | aaB | bb | bbD

B→aa | aaB

D→bb | bbD

需要注意的是，描述一个语言的文法可能有多个，描述 L_1 的另一个文法为 G_8：

S→ B | D

B→aa | aBa

D→bb | bDb

定义 2-17：一个文法生成一个语言，但一个语言可以由若干个文法生成，生成同一个语言的文法是等价的，称为**等价文法**。

如【例 2-11】中文法 G_7 和 G_8 生成的语言相同，即 $L(G_7)=L(G_8)$，因此 G_7 和 G_8 是等价文法。同理，【例 2-7】中的 G_3 和【例 2-10】中的 G_6 也是等价文法，因为 G_3 也是产生所有的由变量标识符 id，运算符号+、*，和括号(、)构成的算术表达式。再次强调，判断两个文法是否等价的依据为是否产生了相同的符号串的集合。等价文法的应用场景主要是：经常需要对某个文法进行等价变换，以满足分析的需要，如后续章节中的提取公共左因子和消除左递归。

2.3 文法和语言的分类

美国著名语言学家艾弗拉姆·诺姆·乔姆斯基（Avram Noam Chomsky）在 1956 年提出了对形式文法进行分类的标准。通过对形式文法的产生式施加不同的限制条件，乔姆斯基将形式文法分为 4 类，即 0 型文法、1 型文法、2 型文法、3 型文法。每类文法对应一类语言，相应地形式语言也分为了 4 类。这就是所谓的乔姆斯基文法体系。

定义 2-18：对于文法 G=(V_T，V_N，S，P)，如果 P 中的每个产生式 α→β，都有 α∈$(V_T \cup V_N)^+$，β∈$(V_T \cup V_N)^*$，即|α|≠0 或 α≠ε，则称文法 G 为 **0 型文法**。0 型文法又称短语结构文法（PSG，Phrase Structure Grammar）。0 型文法描述的语言称为 **0 型语言**，或短语结构语言（PSL，Phrase Structure Language）。0 型语言由图灵机（TM，Turing Machine）识别。

定义 2-19：对于文法 G=(V_T，V_N，S，P)，如果 P 中的每个产生式 α→β，都有|α|≤|β|，则称文法 G 为 **1 型文法**。1 型文法又称上下文有关文法（CSG，Context Sensitive

Grammar）。1 型文法描述的语言称为 **1 型语言**，或上下文有关语言（CSL，Context Sensitive Language）。1 型语言由线性有界自动机（LBA，Linear Bounded Automata）识别。

定义 2-20：对于文法 G=(V_T，V_N，S，P)，如果 P 中的每个产生式 α→β，都有 α∈V_N，则称文法 G 为 **2 型文法**。2 型文法又称上下文无关文法（CFG，Context Free Grammar）。2 型文法描述的语言称为 **2 型语言**，或上下文无关语言（CFL，Context Free Language）。2 型语言由下推自动机（PDA，Push Down Automata）识别。高级程序设计语言的大部分语法结构可以由 2 型文法描述。

定义 2-21：对于文法 G=(V_T，V_N，S，P)，如果 P 中的每个产生式 α→β 的形式满足以下两种条件之一，则称文法 G 为 **3 型文法**。

（1）A→a 或 A→aB

（2）A→a 或 A→Ba

其中 A,B∈V_N，a∈V_T∪{ε}。

3 型文法又称为正规文法（RG，Regular Grammar）或者正则文法。其中满足条件（1）的称为右线性文法，满足条件（2）的称为左线性文法。3 型文法描述的语言称为 **3 型语言**，也称为正规语言或正则语言（RL，Regular Language）。3 型语言由有限自动机（FA，Finite Automata）识别。3 型文法能描述程序设计语言的多数单词。

由以上定义可知：

（1）一个 3 型文法，同时也是 0 型文法、1 型文法、2 型文法，反之不一定成立；

（2）一个 2 型文法，同时也是 0 型文法、1 型文法，反之不一定成立；

（3）一个 1 型文法，同时也是 0 型文法，反之不一定成立。

相应地：

（1）一个 3 型语言，同时也是 0 型语言、1 型语言、2 型语言，反之不一定成立；

（2）一个 2 型语言，同时也是 0 型语言、1 型语言，反之不一定成立；

（3）一个 1 型语言，同时也是 0 型语言，反之不一定成立。

下面举几个例子。

【例 2-12】：文法 G_9：

S→aBC|aSBC　　CB→BC

aB→ab　　bB→bb

bB→b　　bC→bc

cC→c　　cC→cc

该文法是 0 型文法。

【例 2-13】：文法 G_{10}：

S→CD　　Ab→bA

C→aCA　　Ba→aB

C→bCB　　Bb→bB

AD→aD　　BD→bD

Aa→bD

该文法是 1 型文法，同时也是 0 型文法。

【例 2-14】：文法 G_{11}：

S→aB | bA

A→a | aS | bAA

B→b | bS | aBB

该文法是 2 型文法，同时也是 0 型文法、1 型文法。

【例 2-15】：文法 G_{12}：

S→0A | 1B | 0

A→0A | 1B | 0S

B→1B | 1 | 0

该文法是 3 型文法（右线性文法），同时也是 0 型文法、1 型文法和 2 型文法。

除非特别指出，本书后面提到的文法均默认为上下文无关文法。

2.4 上下文无关文法的句型分析

2.4.1 用上下文无关文法描述高级语言

严格地说，高级程序设计语言不是上下文无关语言，无法完全用上下文无关文法来描述。例如，在用高级语言书写的程序中要引用一个标识符，一般来说需要在前面的程序文本中先定义（声明）。抽象地看，一个程序可以表示为形式 wcw，其中第一个 w 是标识符定义

语句中的标识符（符号串），c 表示中间的程序片段，第二个 w 是引用该标识符语句中的标识符。这是典型的上下文相关的语言结构，我们无法用上下文无关文法来描述这样一种语言特性。

高级语言上下文相关性的另一个例子是函数引用中的参数个数问题。高级语言在定义一个函数时一般会指明该函数参数的个数，在引用该函数时参数的个数应该和定义函数时一致。这个一致性也无法用上下文无关文法来描述。一个类 C 语言的函数引用结构的文法如下：

stmt → **id** (*expr_list*)

expr_list → *expr_list*, *expr* | *expr*

其中 id 是要引用的函数的名字，*expr_list* 是该函数的参数列表，*expr* 是一个单独的参数。使用参数列表 *expr_list* 的产生式可以生成用逗号隔开的 1~n 个参数，参数个数和函数定义时是否一致不作检查。这些高级语言的上下文相关性，可以在语义分析阶段来完成。

虽然上下文无关文法不能完全描述高级程序设计语言，但是现有的编译器大部分基于上下文无关文法来构建。上下文无关文法可以描述现今程序设计语言的大部分语法结构，如算术表达式、赋值语句、条件语句、循环语句等。

为区别同一条产生式中相同文法符号的多次出现，可以为文法符号引入下标。重新考察【例 2-7】中的算术表达式文法 G_3，见下例。

【例 2-16】：为产生式右部中的 E 引入下标，改写 G_3，产生式如下：

$E \rightarrow E_1+E_2 \mid E_1*E_2 \mid (E_1) \mid \mathbf{id}$

这是描述高级语言简单算术表达式的文法，其中 E（Expression）表示算术表达式，**id**（identifier）表示程序的“变量”。该文法定义了由变量 **id**，+，*，(和) 组成的算术表达式的语法结构，产生式的意义为：单独一个变量是算术表达式；若 E_1 和 E_2 是算术表达式，则 E_1+E_2，E_1*E_2 和 (E_1) 也是算术表达式。

描述一种简单赋值语句的产生式如下：

〈赋值语句〉→ id：= E

即赋值语句由一个变量标识符 id，后接一个赋值运算符“：=”，赋值运算符后面跟一个算术表达式 E 组成。

描述条件语句的产生式如下：

〈条件语句〉→ if〈条件〉then〈语句〉 |

if〈条件〉then〈语句〉else〈语句〉

条件语句有两类，即“if-then”语句和“if-then-else”语句。“if-then”语句由保留字“if”连接上条件表达式〈条件〉，接上保留字“then”，后跟上〈语句〉组成。“if-then-else”语句由保留字“if”连接上条件表达式〈条件〉，接上保留字“then”，后跟上〈语句〉，然后再接上保留字“else”，再跟上一个〈语句〉组成。

2.4.2 句型推导与分析树

从文法的开始符号出发，经过零步或者多于零步的推导，产生的符号串都是句型。所有的句型都可以由文法推导出来，推导过程包含若干步直接推导，每一步直接推导要用到一条文法产生式。

上下文无关文法的产生式，左部只有一个非终结符号。上下文无关文法的句型推导，可以用一种树状结构来表达。推导过程中用到的产生式的左部（非终结符号）总是对应这棵树中的某个内部结点（或者根结点），其子结点从左到右排列构成产生式的右部。推导过程中每用到一条产生式，就构造该树状结构的一部分。

树状结构的构造过程：第一步推导总是用到开始符号的一条产生式，首先构造标记为开始符号的结点，该结点作为树状结构的根结点；然后为产生式右部的每个文法符号（可能是终结符号也可能是非终结符号）构造一个结点，并且从左到右排列作为根结点的子结点。接着处理第二步直接推导用到的产生式，其左部（非终结符号）对应的结点在树状结构中已存在，为该产生式右部的每个文法符号分别构造结点，并且从左到右排列作为产生式左部结点的子结点。重复以上操作，直到处理完最后一条产生式，该树状结构构造完成。这个树状结构称为句型推导的**分析树**。

分析树是句型推导的图形化表示形式。分析树叶子结点的标号既可以是非终结符号，也可以是终结符号，分析树的叶子结点从左到右排列正好是推导出的句型。

再次以以下算术表达式文法为例，

$$E \rightarrow E+E \mid E*E \mid (E) \mid \mathbf{id}$$

基于此文法，构造一个最左推导：

$$E \Rightarrow \underline{E+E} \Rightarrow \underline{id}+E \Rightarrow id+\underline{E*E} \Rightarrow id+\underline{id}*E \Rightarrow id+id*\underline{id} \qquad (2.4)$$

其对应的分析树的构造过程见图 2-2。

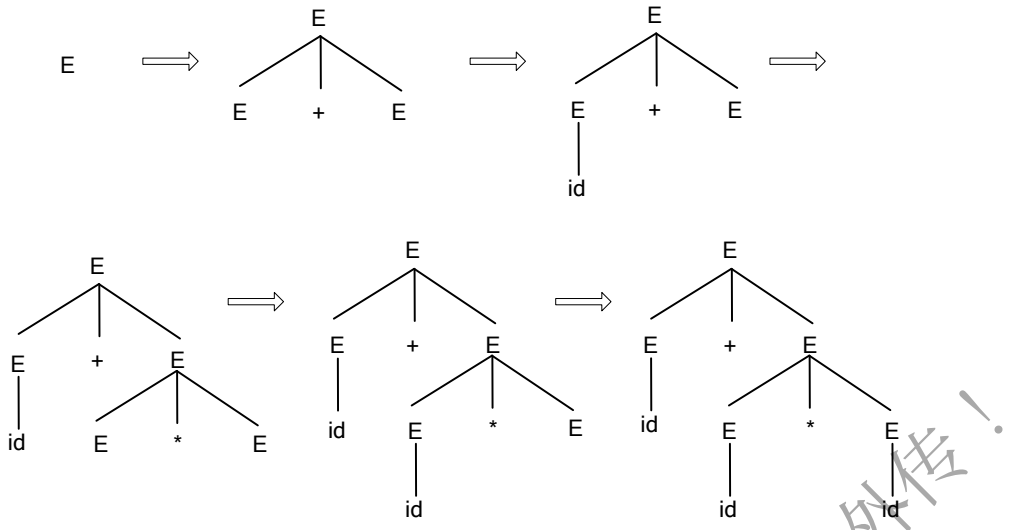

图 2-2 id+id*id 最左推导分析树的构造

一次句型推导，可以构造唯一的一棵分析树。但是分析树没有反映句型推导的顺序，一棵分析树可能对应多个推导，或者说同一个句型的多个推导有可能构造的是同一棵分析树。如以下两个推导（2.5）、（2.6）构造的分析树与图 2-2 构造的分析树是一样的，见图 2-3 和 2-4。其中（2.5）是最右推导，（2.6）则是一种混合推导。

$$E\Rightarrow\underline{E+E}\Rightarrow E+\underline{E*E}\Rightarrow E+E*\underline{id}\Rightarrow E+\underline{id}*id\Rightarrow\underline{id}+id*id \qquad (2.5)$$

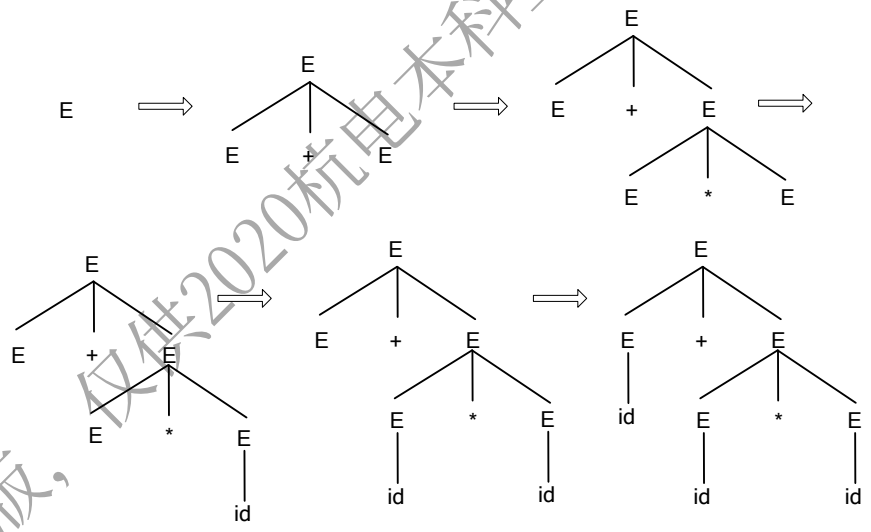

图 2-3 id+id*id 最右推导分析树的构造

$$E\Rightarrow\underline{E+E}\Rightarrow\underline{id}+E\Rightarrow id+\underline{E*E}\Rightarrow id+E*\underline{id}\Rightarrow id+\underline{id}*id \qquad (2.6)$$

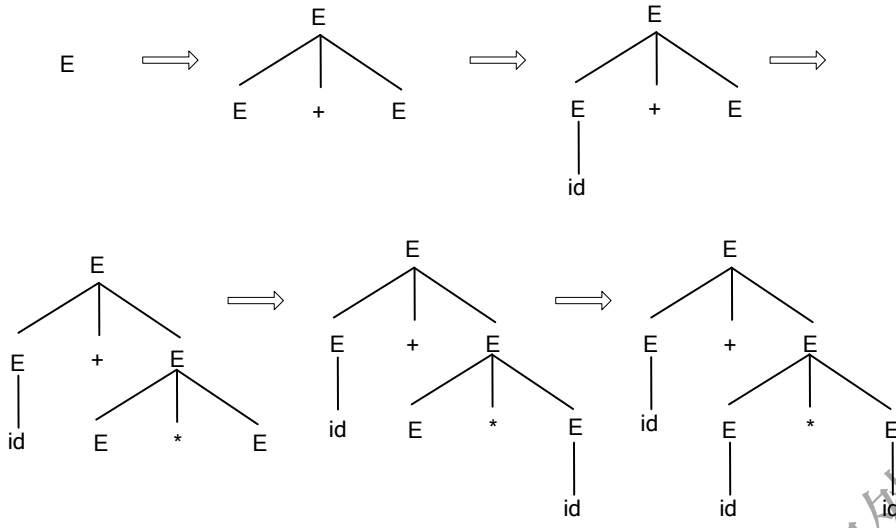

图 2-4 id+id*id 混合推导分析树的构造

不同的推导，在构造分析树时的顺序是不一样的。因为每次构建哪个内部结点的子结点的顺序不同。虽然一棵分析树可能与多个推导对应，但是一棵分析树只对应一个最左推导，也只对应一个最右推导。如上例，句子 id+id*id 的分析树对应的最左推导是（2.4），对应的最右推导是（2.5）。最左推导构造分析树时的顺序是自顶向下，从左往右，参见图 2-2；最右推导构造分析树时的顺序则是自顶向下、从右往左，参见图 2-3。

下面介绍句型分析中涉及到的几个概念，包括短语、直接短语和句柄。

定义 2-22：设 G 是一个文法，S 是该文法的开始符号，α β δ 是该文法的一个句型，如果有：

$$S \overset{*}{\Rightarrow} \alpha A \delta \quad 且 \quad A \overset{+}{\Rightarrow} \beta$$

则称 β 是一个关于非终结符号 A 的、句型 α β δ 的**短语**。

从分析树的角度来看，分析树中一棵子树的所有叶子结点从左到右排列起来构成一个相对于该子树的根的短语，见图 2-5。

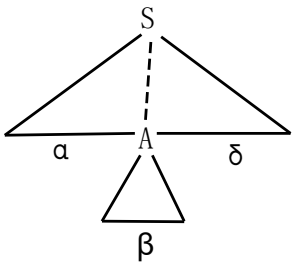

图 2-5 分析树的框架及短语子树

根据定义，句型本身也是该句型的短语，这个短语是相对于开始符号 S 的。

定义 2-23：设 G 是一个文法，S 是该文法的开始符号，α β δ 是它的一个句型，如果有：

$$S \overset{*}{\Rightarrow} \alpha A \delta \quad 且 \quad A \Rightarrow \beta$$

则称 β 是一个关于非终结符号 A 的、句型 α β δ 的**直接短语**。

从分析树的角度来看，分析树中仅有父子两代的一棵子树的所有叶子结点从左到右排列构成一个相对于父结点的直接短语。

定义 2-24：最左边的直接短语称为**句柄**。

【例 2-17】：考虑【例 2-10】中的文法 G_6，求：句型 F*id 的短语、直接短语和句柄。

解：首先构造句型 F*id 的最左推导：

$$E \Rightarrow \underline{T} \Rightarrow \underline{T*F} \Rightarrow \underline{F}*F \Rightarrow F*\underline{id} \qquad (2.7)$$

然后构造句型 F*id 的分析树，见图 2-6。

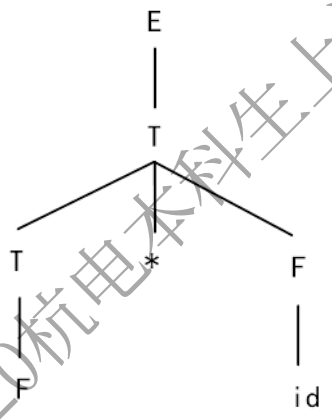

图 2-6 F*id 的分析树

句型 F*id 的短语包括 F*id、F 和 id，其中 F*id 是关于 E 的，同时也是关于 T 的，F 是关于 T 的，id 是关于 F 的。

F*id 的直接短语包括 F 和 id，它们都是只有父子两代的子树的叶子节点从左自有排列得到的。

句柄是 F，因为在所有直接短语中，它是最左边的这个。

注意，每个句型的句柄是唯一的。句柄是句型分析中最重要的概念之一，识别句型中的句柄是自底向上语法分析需要解决的关键问题。

2.4.3 句子、文法和语言的二义性

定义 2-25：如果一个文法的句子有两棵或两棵以上的分析树，则称此**句子是二义的**（ambiguous）。

因为每棵分析树都对应了一个最左推导，也对应了一个最右推导，所以二义性的句子有两个以上的最左推导，也有两个以上的最右推导。

例如，对于文法 G_3 的句子 id+id*id，有两个最左推导，见（2.8），也有两个最右推导，见（2.9），对应的两棵分析树分别见图 2-7(a)和(b)。

最左推导：

$$E\Rightarrow\underline{E+E}\Rightarrow\underline{id}+E\Rightarrow id+\underline{E*E}\Rightarrow id+\underline{id}*E\Rightarrow id+id*\underline{id} \quad (2.8)$$

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

最右推导：

$$E\Rightarrow\underline{E+E}\Rightarrow E+\underline{E*E}\Rightarrow E+E*\underline{id}\Rightarrow E+\underline{id}*id\Rightarrow\underline{id}+id*id \quad (2.9)$$

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

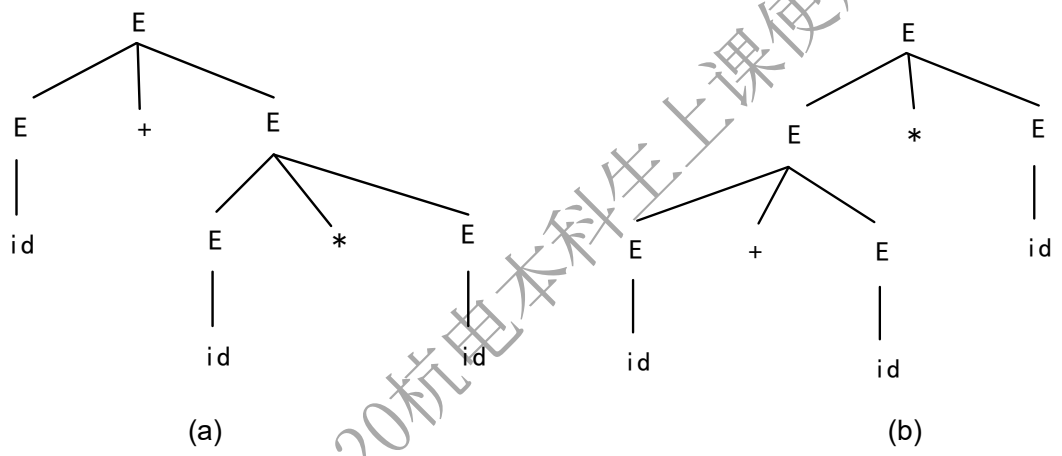

图 2-7 id+id*id 的两棵分析树

图 2-7(a)这棵分析树反映了+运算和*运算之间正常的优先级关系（即*运算的优先级高于+运算的优先级），而图 2-7(b)这棵分析树反映的两个运算之间的优先级关系是不正确的。对于一个表达式如：a+b*c，图 2-7(a)反映的是先计算 b*c，a 再去加 b*c 这样一个计算顺序，而图 2-7(b)反映的是先计算 a+b，再去乘上 c 这样一个计算顺序，显然前者是我们需要的。

定义 2-26：如果一个文法包含二义性的句子，则称这个**文法是二义性的**；否则，该文法是无二义性的。如 G_3 就是一个二义性的文法。

定义 2-27：产生某上下文无关语言的每一个文法都是二义的，则称此**语言是先天二义的**。我们希望文法是无二义的，因为希望对于每一个句子进行唯一确定的分析。

2.4.4 二义文法的改造

在某些情况下，使用经过精心设计的二义文法可以带来方便。但一般情况下，我们不能直接使用二义文法。可以通过对二义文法进行改造（等价变换）来消除文法的二义性，使得对于任何一个合法的句子，都只能生成唯一的一棵分析树。

对于二义文法我们需要使用消除二义性的规则来识别哪些是不合法的分析树，从而为每个句子留下一棵正确的分析树。

例如，通过规定运算的优先顺序和结合律，可以将二义文法 G_{13} 改写为无二义的文法 G_{14}。

G_{13} 的文法产生式如下：

E → E + E | E - E | E * E | E / E | (E) | id

G_{14} 的文法产生式如下：

E → E + T | E - T | T

T → T * F | T / F | F

F → (E) | id

文法 G_{14} 反映了+、-、*、/这 4 个运算都是左结合的，其中*、/ 的运算优先级高于 +、-的运算优先级。构造文法 G_{14} 的基本思路是：创建两个终结符号 E（Expression）和 T（Term），分别对应于两个优先级层次，并使用另一个非终结符号 F（Factor）来生成表达式中的基本单元。表达式的基本单元是变量标识符或带括号的表达式。使用这个文法时，一个表达式就是一个由+或-分隔开的项（T）的列表，而项是由*或/分割的因子（F）的列表。任意由括号括起来的表达式都是因子，可以使用括号构造出具有任意嵌套深度的表达式。

下面再看一个二义文法的例子。

【例 2-18】：文法 G_{15}：

stmt → **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| **other**

这是生成条件语句的文法，其中 other 表示任何其它语句。该文法是二义的，如对于一个句子：If E_1 then if E_2 then S_1 else S_2，有两棵分析树，见图 2-8。

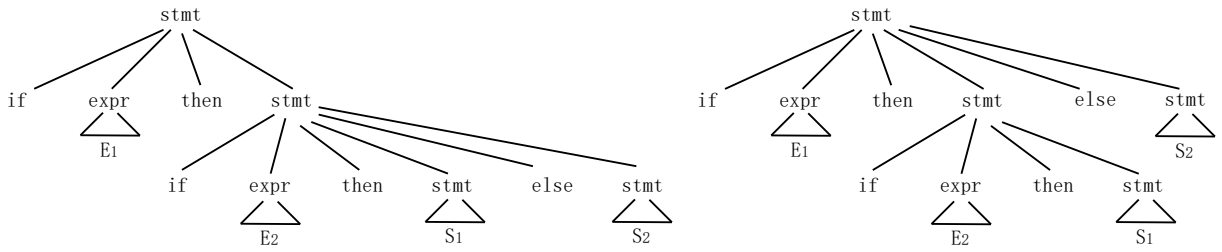

图 2-8 一个二义性句子的两棵分析树

在高级程序设计语言中，如果有 if-then-else 语句通常会选择第一棵分析树，因为默认的规则是“每个 else 和最近的尚未匹配的 then 匹配”。可以将以上文法改写为一个等价的无二义的文法。其基本思想是：在一个 then 和一个 else 之间出现的语句必须是“已匹配的”，也就是说中间的语句不能以一个尚未匹配的（或者说开放的）then 结尾。一个已匹配的语句要么是一个不包含开放语句的 if-then-else 语句，要么是一个非条件语句。改造后的文法如下：

stmt → *matched_stmt*

| *open_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt*

| **other**

open_stmt → **if** *expr* **then** *stmt*

| **if** *expr* **then** *matched_stmt* **else** *open_stmt*

对于句子 If E_1 then if E_2 then S_1 else S_2，其分析树是唯一的，见图 2-9。

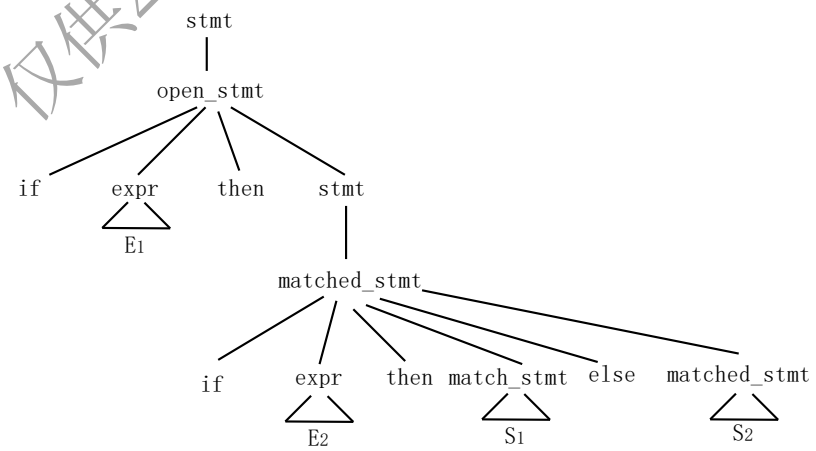

图 2-9 无二义性句子的分析树

2.5 小结

本书主要讲授编译器构造的原理与常见技术，形式文法和形式语言（特别是上下文无关文法与上下文无关语言）理论是编译器构造的理论基础。虽然高级程序设计语言并不是严格意义上的上下文无关语言，但是当前的编译器设计基本上都是基于上下文无关文法的。对于高级语言中上下文相关的语法结构，可以在语义分析阶段进行处理。与自然语言类似，形式语言也有语言材料和语法规则，形式语言的最高级语法单位是句子，句子由单词构成，单词由字母表中的符号构成。单词的构成规则称为词法规则，句子的构成规则称为语法规则。形式语言的语法规则通常用产生式的形式来描述，产生式是形式文法的主要组成部分。书写文法时通常只需列出文法的产生式序列，文法的其它组成部分，如开始符号、终结符号集合、非终结符号集合等可以按照一些约定从产生式中提取。形式语言定义为某个字母表上符号串的集合。一般来说，这些符号串是可以由某个文法推导出来，即它们应该遵循一些的共同的构成规则。推导是用产生式的右部替换左部，归约是用产生式左部替换右部，推导和归约互为逆过程。从一个文法的开始符号出发做零步或者多于零步的推导得到的文法符号串称为句型，只含有终结符号的句型称为句子，文法能推出的所有句子组成的集合称为该文法描述的语言。一个文法描述一个语言，但是一个语言可以对应多个描述它的文法，文法和语言是多对一的关系。乔姆斯基将形式文法和形式语言分为 4 类，编译器设计中主要涉及 2 型文法（即上下文无关文法），对上下文无关文法句型的分析具有特别重要的意义，因为它是语法分析和语义分析的基础。一个文法的句型（包括句子）总是可以从该文法的开始符号推导出来。某个句型的一次推导对应一棵分析树，一棵分析树却可能对应该句型的多个推导，但是一棵分析树只对应句型的一个最左推导，也只对应句型的一个最右推导。

本章要求理解掌握形式文法和形式语言的定义，及与形式文法和形式语言相关的几个重要概念，如推导、归约、句型、句子、句子与文法的二义性等，重点是掌握上下文无关文法句型推导与分析树之间的关系，理解短语、直接短语、句柄的概念，掌握从句型中解析短语、直接短语、句柄的方法。

习题

2.1 给定如下文法 G[A]，用自己的语言描述它定义的语言。（注：G[A]中的 A 是文法的开始符号）

A →aaA | aaB

B →Bcc | D#cc

D →bbbD | #

2.2 设有文法 G[S]：

S → B = E

B → C | D

C →a | b | c

D → m[1] | m[2] | m[3]

E → C O C | C O D | D O C | D O D

O → + | -

现有两个句子① b = a+b； ② m[2] = b + m[1]，分别完成以下题目：

(1) 分别给出这两个句子的最左推导或最右推导；

(2) 试画出对应的分析树；

(3) 指出每个句子中的短语、直接短语和句柄。

2.3 给定文法 G[E]：

E → E + T | E － T | T

T →F | T*F | T/F

F →F ^ P | P

P → c | id | (E)

现有文法符号串 E+T*(F-id)和 T*P ^ (id+c)，试完成如下题目：

(1) 证明这两个符号串都是该文法的句型；

(2) 画出相应的分析树；

(3) 指出每个句型的短语、直接短语、句柄。

2.4 考虑文法 G[S]：

S→ aSbS | bSaS | ε

(1) 为句子 abab 构造两个不同的最左推导，以此说明该文法是二义的。

(2) 为 abab 构造分析树。

2.5 给定文法 G[E]：

E →T | E + T | E - T

T →F | T*F | T/F

F →(E) | id

写出表达式 id*(id + id) + id 的最左或最右推导，并画出分析树。

2.6 考虑文法 G[S]：

S→(L) | a

L→ L , S | S

(1) 建立句子(a, (a, a))和(a, ((a, a), (a, a)))的分析树。

(2) 为上述两个句子构造最左推导。

(3) 为上述两个句子构造最右推导。

(4) 该文法产生的语言是什么？

2.7 设有文法 G[N]：

N→D | ND

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

(1) 该文法定义的语言是什么？

(2) 给出句子 0123 和 2468 的最左推导和最右推导。

2.8 证明如下文法 G[S]的二义性：

S→iSeS | iS | i

2.9 写出下面文法所描述的语言。

G_1[S]： S→AB

A→aA | ε

G_2[S]： S→aA | a

A→aS

B → bc | bBc

2.10 文法 G[S]=({S}, {a, b}, {S→bS | a}, S)所生成的语言是什么？