第3章　上下文无关文法及分析

本章要点

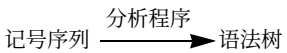
- 分析过程
- 上下文无关文法
- 分析树与抽象语法树
- 二义性
- 扩展的表示法：EBNF和语法图
- 上下文无关语言的形式特性
- TINY语言的语法

分析的任务是确定程序的语法，或称作结构，也正是这个原因，它又被称作语法分析（syntax analysis）。程序设计语言的语法通常是由上下文无关（context-free grammar）的文法规则（grammar rule）给出，其方式同扫描程序识别的由正则表达式提供的记号的词法结构相类似。上下文无关文法的确利用了与正则表达式中极为类似的命名惯例和运算。二者的主要区别在于上下文无关文法的规则是递归的（recursive）。例如一般来说，if 语句的结构应允许可其中嵌套其他的if语句，而在正则表达式中却不能这样做。这个区别造成的影响很大。由上下文无关文法识别的结构类比由正则表达式识别的结构类大大增多了。用作识别这些结构的算法也与扫描算法差别很大，这是因为它们必须使用递归调用或显式管理的分析栈。用作表示语言语义结构的数据结构现在也必须是递归的，而不再是线性的（如同用于词法和记号中的一样）了。经常使用的基本结构是一类树，称作分析树（parse tree）或语法树（syntax tree）。

同第2章相似，在学习分析算法和如何利用这些算法进行真正的分析之前需要先学习上下文无关文法的理论，但是又与在扫描程序中的情形不同——其中主要只有一种算法方法（表示为有穷自动机），分析涉及到要在许多属性和能力截然不同的方法中做出选择。按照它们构造分析树或语法树的方式，算法大致可分为两种：自顶向下分析（ top-down parsing）和由底向上分析（bottom-up parsing）。对这些分析方法的详细讨论将放到以后的章节中，本章只给出分析过程的一般性描述，之后还要学习上下文无关文法的基础理论。最后一节则从一个上下文无关文法的角度给出TINY语言的文法。对上下文无关文法理论和语法树熟悉的读者可以跳过本章中间的某些内容（或将其作为复习）。

3.1　分析过程

分析程序的任务是从由扫描程序产生的记号中确定程序的语法结构，以及或隐式或显式地构造出表示该结构的分析树或语法树。因此，可将分析程序看作一个函数，该函数把由扫描程序生成的记号序列作为输入，并生成语法树作为它的输出：



记号序列通常不是显式输入参数，但是当分析过程需要下一个记号时，分析程序就调用诸如**getToken**的扫描程序过程以从输入中获得它。因此，编译器的分析步骤可减为对分析程序的一个调用，如下所示：

```
syntaxTree = parse();
```

在单遍编译中，分析程序合并编译器中所有的其他阶段，这还包括了代码生成，因此也就不需要构造显式的语法树了（分析步骤本身隐式地表示了语法树），由此就发生了一个

```
parse();
```

调用。在编译器中更多的是多遍，此时后面的遍将语法树作为它们的输入。

语法树的结构在很大程度上依赖于语言特定的语法结构。这种树通常被定义为动态数据结构，该结构中的每个节点都由一个记录组成，而这个记录的域包括了编译后面过程所需的特性（即：并不是那些由分析程序计算的特性）。节点结构通常是节省空间的各种记录。特性域还可以是在需要时动态分配的结构，它就像一个更进一步节省空间的工具。

在分析程序中有一个比在扫描程序中更为复杂的问题，这就是对于错误的处理。在扫描程序中，如果遇到的一个字符是不正规记号的一部分，那么它只需生成一个出错记号并消耗掉这个讨厌的字符即可（在某种意义上，通过生成一个出错记号，扫描程序就克服了发生在分析程序上的困难）。但对于分析程序而言，它必须不仅报告一个出错信息，而且还须从错误状态恢复（recover）并继续进行分析（去找到尽可能多的错误）。分析程序有时会执行错误修复（error repair），此时它从提交给它的非正确的版本中推断出一个可能正确的代码版本（这通常是在简单情况下才发生的）。错误恢复的一个尤为重要的方面是有意义的错误信息报告以及在尽可能接近真正错误时继续分析下去。由于要到错误真正地已经发生了分析程序才会发现它，所以做到这一点并不简单。由于错误恢复技术依赖于所使用的特定分析算法，所以本章先就不学习它了。

3.2 上下文无关文法

上下文无关文法说明程序设计语言的语法结构。除了上下文无关文法涉及到了递归规则之外，这样的说明与使用正则表达式的词法结构的说明十分类似。例如在一个运算中，就可以使用带有加法、减法和乘法的简单整型算术表达式。这些表达式可由下面的文法给出：

$$exp \rightarrow exp\ op\ exp \mid \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}$$
$$op \rightarrow \mathbf{+} \mid \mathbf{-} \mid *$$

3.2.1 与正则表达式比较

在第2章中为**number**给出的正则表达式规则如下所示：

```
number = digit digit*
digit = 0|1|2|3|4|5|6|7|8|9
```

试与上面上下文无关文法的样本作一比较。基本正则表达式规则有 3种运算：选择（由竖线元字符表示）、并置（不带元符号）以及重复（由星号元符号提供）。此外还可用等号来表示正则表达式的名字定义，此时名字用斜体书写以示与真正字符序列的区别。

文法规则使用相似的表示法。名字用斜体表示（但它是一种不同的字体，所以可与正则表达式相区分）。竖线仍表示作为选择的元符号。并置也用作一种标准运算。但是这里没有重复的元符号（如正则表达式中的星号*），稍后还会再讲到它。表示法中的另一个差别是现在用箭头符号“→”代替了等号来表示名字的定义。这是由于现在的名字不能简单地由其定义取代，而需要更为复杂的定义过程来表示，这是由定义的递归本质决定的⊖。在我们的示例中，exp的

⊖ 参见本章后面的语法规则和等式。

规则是递归的，其中名字exp出现在箭头的右边。

读者还要注意到文法规则将正则表达式作为部件。在exp规则和op规则中，实际有6个表示语言中记号的正则表达式。其中5个是单字符记号：**+**、**-**、*****、**(** 和 **)**，另一个是名字***number***。记号的名字表示数字序列。

与这个例子的格式相类似的文法规则最初是用在 Algol60语言的描述中。其表示法是由John Backus为Algol60报告开发，之后又由Peter Naur更新，因此这个格式中的文法通常被称作**Backus-Naur**范式（Backus-Naur form）或**BNF**文法。

3.2.2 上下文无关文法规则的说明

同正则表达式类似，文法规则是定义在一个字母表或符号集之上。在正则表达式中，这些符号通常就是字符，而在文法规则中，符号通常是表示字符串的记号。在前一章中，我们利用C中的枚举类型定义了在扫描程序中的记号；本章为了避免涉及到特定实现语言（例如 C）中表示记号的细节，就使用了正则表达式本身来表示记号。此时的记号就是一个固定的符号，如同在保留字**while**中或诸如**+**或**:=**这样的特殊符号一样，使用在第2章曾用到的代码字体书写串本身。对于作为表示多于一个串的标识符和数的记号来说，代码字体为斜体，这就同假设这个记号是正则表达式的名字（这是它经常的表示）一样。例如，将 TINY语言的记号字母表表示为集合

```
{if, then, else, end, repeat, until, read, write,
identifier, number, +, -, *, /, =, <, (, ), ;, := }
```

而不是记号集（如在TINY扫描程序中定义的一样）：

```
{IF, THEN, ELSE, END, REPEAT, UNTIL, READ, WRITE, ID, NUM,
PLUS, MINUS, TIMES, OVER, EQ, LT, LPAREN, RPAREN, SEMI, ASSIGN }
```

假设有一个字母表，BNF中的上下文无关文法规则（context-free grammar rule in BNF）是由符号串组成。第1个符号是结构名字，第2个符号是元字符→，这个符号之后是一个符号串，该串中的每个符号都是字母表中的一个符号（即一个结构的名字）或是元符号｜。

在非正式术语中，对BNF的文法规则解释如下：规则定义了在箭头左边名字的结构。这个结构被定义为由被竖线分隔开的选择右边的一个选项组成。每个选项中的符号序列和结构名字定义了结构的布局。例如，前例中的文法规则：

$$exp \rightarrow exp\ op\ exp \mid \mathbf{(}exp\mathbf{)} \mid \mathbf{number}$$
$$op \rightarrow \mathbf{+} \mid \mathbf{-} \mid *$$

第1个规则定义了一个表达式结构（用名字exp）由带有一个算符和另一个表达式的表达式，或一个位于括号之中的表达式，或一个数组成。第 2个规则定义一个算符（利用名字 op）由符号**+**、**-**或*****构成。

这里所用的元符号和惯例与广泛使用中的类似，但读者应注意到这些惯例并没有统一的标准。实际上，用以代替箭头元符号→的通常有 =（等号）、:（冒号）和 ::=（双冒号等号）。在普通文本文件中，找到能替代斜体用法的方法也很有必要，通常的办法是在结构名字前后加尖括号<...>并将原来斜体的记号名字大写；因此，使用不同的惯例，上面的文法规则就可变为：

```
<exp> ::= <exp> <op> <exp> | ( <exp> ) | NUMBER
<op> ::= + | - | *
```

每一个作者都还有这些表示法的其他变形。本节后面将会谈到一些非常重要的变形（其中

一些有时也会碰到)。这里要先讲一下有关表示法方面的另外两个较小的问题。

在BNF的元符号中使用括号有时很有用，这同括号可在正则表达式中重新安排优先权很相似。例如，可将上面的文法规则重写为如下一个单一的文法规则：

$$exp \rightarrow exp\ (\text{"+"} \mid \text{"-"} \mid \text{"*"})\ exp \mid \text{"(" } exp \text{ ")"} \mid \textbf{number}$$

在这个规则中，括号很必要，它用于将箭头右边的表达式之间的算符选择组合在一起，这是因为并置优先于选择（同在正则表达式中一样）。因此，下面的规则就具有了不同（但不正确）的含义：

$$exp \rightarrow exp\ \text{"+"} \mid \text{"-"} \mid \text{"*"}\ exp \mid \text{"(" } exp \text{ ")"} \mid \textbf{number}$$

请读者再留意一下：当将括号包含为一个元符号时，就有必要区分括号记号与元符号，这一点是通过将括号记号放在引号中做到的，这同在正则表达式中的一样（为了具有连贯性，也将算符符号放在引号中)。

由于经常可将括号中的部分分隔成新的文法规则，所以在 BNF中的括号并不像元符号一样缺之不可。实际上如果允许在箭头左边的相同名字可出现任意次，那么由竖线元符号给出的选择运算在文法规则中也不是一定要有的。例如，简单的表达式文法可写作：

$$exp \rightarrow exp\ op\ exp$$
$$exp \rightarrow \text{(}\ exp\ \text{)}$$
$$exp \rightarrow \textbf{number}$$
$$op \rightarrow \text{+}$$
$$op \rightarrow \text{-}$$
$$op \rightarrow \text{*}$$

然而，通常把文法规则写成每个结构的所有选择都可在一个规则中列出来，而且每个结构名字在箭头左边只出现一次。

有时我们需要为说明简便性而给出一些用简短表示法写出的文法规则的示例。在这些情形中，应将结构名字大写，并小写单个的记号符号（它们经常仅仅是一个字符）；因此，按这种速记方法可将简单的表达式文法写作：

$$E \rightarrow E\ O\ E \mid \text{(}\ E\ \text{)} \mid n$$
$$O \rightarrow \text{+} \mid \text{-} \mid *$$

有时当正在将字符如同记号一样使用时，且无需使用代码字体来书写它们，则也可简化表示法如下：

$$E \rightarrow E\ O\ E \mid (\ E\ ) \mid a$$
$$O \rightarrow + \mid - \mid *$$

3.2.3 推导及由文法定义的语言

现在讨论文法规则如何确定一种“语言”或者是记号的正规串集。

上下文无关文法规则确定了为由规则定义的结构的记号符号符合语法的串集。例如，算术表达式

```
(34-3)*42
```

与7个记号的正规串相对应

```
( number - number ) * number
```

其中**number**记号具有由扫描程序确定的结构且串本身也是一个正规的表达式，这是因为

每一个部分都与由文法规则

$$exp \rightarrow exp\ op\ exp \mid (exp) \mid number$$
$$op \rightarrow + \mid - \mid *$$

给出的选择对应。另一方面，串

```
(34-3*42
```

就不是正规的表达式，这是因为左括号没有一个右括号与之匹配，且 exp的文法规则中的第2个选择要求括号需成对生成。

文法规则通过推导确定记号符号的正规串。推导（derivation）是在文法规则的右边进行选择的一个结构名字替换序列。推导以一个结构名字开始并以记号符号串结束。在推导的每一个步骤中，使用来自文法规则的选择每一次生成一个替换。

例如，图3-1利用同在上面简单表达式文法中的一样给出的文法规则为表达式 **(34-3)*42** 提供了一个推导。在每一步中，为替换所用的文法规则选择都放在了右边（还为便于在后面引用为每一步都编了号）。

(1)	*exp* ⇒ *exp op exp*	[*exp* → *exp op exp*]
(2)	⇒ *exp op* **number**	[*exp* → **number**]
(3)	⇒ *exp* * **number**	[*op* → *]
(4)	⇒ (*exp*) * **number**	[*exp* → (*exp*)]
(5)	⇒ (*exp op exp*) * **number**	[*exp* → *exp op exp*]
(6)	⇒ (*exp op* **number**) * **number**	[*exp* → **number**]
(7)	⇒ (*exp* - **number**) * **number**	[*op* → -]
(8)	⇒ (**number** - **number**)*number	[*exp* → **number**]

图3-1 算术表达式 **(34-3)*42** 的推导

请注意，推导步骤使用了与文法规则中的箭头元符号不同的箭头，这是由于推导步骤与文法规则有一点差别：文法规则定义（define），而推导步骤却通过替换来构造（construct）。在图3-1的第1步中，串exp op exp从规则exp→exp op exp的（在BNF中对于exp的第1个选择）右边替换了单个的exp。在第2步中，串exp op exp中的exp被符号***number***从选择exp→***number***的右边替换掉以得到串exp op ***number***。在第3步中，符号*从规则op→*中替换op（在BNF中对于op的第3个选择）以得到串exp * ***number***，等等。

由推导从exp符号中得到的所有记号符号的串集是被表达式的文法定义的语言（ language defined by the grammar）。这个语言包括了所有合乎语法的表达式。可将它用符号表示为：

$$L(G) = \{ s \mid exp \Rightarrow^{*} s \}$$

其中*G*代表表达式文法，*s* 代表记号符号的任意数组串（有时称为句子（ sentence）），而符号 ⇒*表示由如前所述的替换序列组成的推导（星号用作指示步骤的序列，这与在正则表达式中指示重复很相像）。由于它们通过推导“ 产生 ”*L* (*G*）中的串，文法规则因此有时也称作产生式（production）。

文法中的每一个结构名定义了符合语法的记号串的自身语言。例如，在简单表达式文法中由*op* 定义的语言定义了语言**{+, -, *}**只由3个符号组成。我们通常对由文法中最普通的结构定义的语言最感兴趣。用于程序设计语言的文法经常定义一个称作程序的结构，而该结构的语言是程序设计语言的所有符合语法的程序的集合（注意这里在两个不同的意思中所使用的“ 语言 ”）。

例如：Pascal的BNF以诸如

program → *program-heading* ; *program-block*.
program-heading → ...
program-block → ...
. . .

的文法规则开始（第1个规则认为程序由一个程序头、随后的分号、随后的程序块，以及位于最后的一个句号组成）。在诸如C的带有独立编译的语言中，最普通的结构通常称作编译单元。除非特别指出不是，在各种情况下都假定在文法规则中，第 1个列出的就是这个最普通的结构（在上下文无关文法的数学理论中，这个结构称作开始符号（ start symbol））。

通过更深一些的术语可更清楚地区分结构名和字母表中的符号（因为它们经常是编译应用程序的记号，所以我们一直调用记号符号）。由于在推导中必须被进一步替换（它们不终结推导），所以结构名也称作非终结符（nonterminal）。相反地，由于字母表中的符号终结推导，所以它们被称作终结符（terminal）。因为终结符通常是编译应用程序中的记号，所以这两个名字在使用时是基本同义的。终结符和非终结符经常都被认作是符号。

下面是一些由文法生成的语言示例。

例3.1　考虑带有单个文法规则的文法G

$$E \rightarrow (E) \mid a$$

这个文法有1个非终结符E、3个终结符(,) 以及a。这个文法生成语言$L(G) = \{a, (a), ((a)), (((a))),...\} = \{(^n a)^n \mid n$ 是一个≥0的整型 $\}$，即：串由零个或多个左括号、后接一个a，以及后面是与左括号相同数量的右括号组成。作为这些串的一个推导示例，我们给出 ((a))的一个推导：

$$E \quad (E) \quad ((E)) \quad ((a))$$

例3.2　考虑带有单个文法规则

$$E \rightarrow (E)$$

的文法G。除了减少了选项$E \rightarrow a$之外，这是与前例相同的文法。这个文法根本就不生成串，因此它的语言是空的：$L(G) = \{\ \}$。其原因在于任何以E开头的推导都生成总是含有E的串，所以没有办法推导出一个仅包含有终结符的串。实际上，与带有所有递归进程（如归纳论证或递归函数）相同，递归地定义一个结构的文法规则必须总是有至少一个非递归情况（称之为基础情况（base case））。本例中的文法并没有这样的情况，且任何潜在的推导都注定为无穷递归。

例3.3　考虑带有单个文法规则

$$E \rightarrow E + a \mid a$$

的文法G。这个文法生成所有由若干个“ + ”分隔开的a 组成的串：

$$L(G) = \{\ a, a + a, a + a + a, a + a + a + a, \ldots\}$$

为了（非正式地）查看它，可考虑规则$E \rightarrow E + a$ 的效果：它引起串 + a 在推导右边不断地重复：

$$E \quad E + a \quad E + a + a \quad E + a + a + a \quad \ldots$$

最后，必须用基础情况$E \rightarrow a$ 来替换左边的E。

若要更正式一些，则可如下来归纳证明：首先，通过归纳a 的数目来表示每个串$a + a + \ldots + a$ 都在$L(G)$中。推导$E \quad a$ 表示a 在$L(G)$ 中；现在假设$s = a + a + \ldots + a$ 在$L(G)$ 中，且有n-1个a，则存在推导$E \quad {}^* s$。现在推导$E \quad E + a \quad {}^* s + a$ 表示串$s + a$ 在$L(G)$中，且其中有n 个a。相反地，我们也表示出在$L(G)$ 中的任何串s 都必须属于格式$a + a + \ldots + a$。这是通过归纳推导

的长度得出的。假设推导的长度为1，则它属于格式$E \Rightarrow a$，而且s是正确格式。现在假设以下两个推导都为真：所有串都带有长度为$n-1$的推导，并使$E \Rightarrow^* s$是长度为$n>1$的推导；则这个推导开头必须将E改为$E+a$，格式$E \Rightarrow E+a \Rightarrow^* s'+a=s$也是这样。则$s'$有长度为$n-1$的推导，格式$a+a+\ldots+a$也是这样；因此，$s$本身必须也具有这个相同的格式。

例3.4 考虑下面语句的极为简化的文法：

statement → *if-stmt* | ***other***

if-stmt → **if (***exp***)** *statement*

| **if (***exp***)** *statement* **else** *statement*

exp → **0** | **1**

这个文法的语言包括位于类似C的格式中的嵌套if语句（我们已将逻辑测试表达式简化为0或1，且除if语句外的所有语句都被放在终结符***other***中了）。例如，这个语言中的串是：

```
other
if ( 0 ) other
if ( 1 ) other
if ( 0 ) other else other
if ( 1 ) other else other
if ( 0 ) if ( 0 ) other
if ( 0 ) if ( 1 ) other else other
if ( 1 ) other else if ( 0 ) other else other
...
```

请注意，if 语句中的可选else 部分由用于if-stmt 的文法规则中的单个选择指出。

在前面我们已注意到：BNF中的文法规则规定了并置和选择，但不具有与正则表达式的 * 相等同的特定重复运算。由于可由递归得到重复，所以这样的运算实际并不必要（如同在功能语言中的程序员所知道的）。例如，文法规则

$$A \rightarrow Aa \mid a$$

或文法规则

$$A \rightarrow aA \mid a$$

都生成语言$\{a^n \mid n$ 是≥1的整型$\}$（具有1个或多个a 的所有串的集合），该语言与由正则表达式**a+**生成的语言相同。例如，串*aaaa* 可由带有推导

$$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow Aaaa \Rightarrow aaaa$$

的第1个文法规则生成。一个类似的推导在第 2个文法规则中起作用。由于非终结符 A作为定义A的规则左边的第1个符号出现，所以这些文法中的第1个是左递归（left recursive）⊖，第2个文法则是右递归（right recursive）。

例3.3是左递归文法规则的另一个示例，它引出串“ $+a$ ”的重复。可将本例及前一个示例归纳如下。考虑一个规则形式

$$A \rightarrow A\alpha \mid \beta$$

其中α 和β 代表任意串，且β 不以A开头。这个规则生成形式β、βα、βαα、βααα、……的所有串（所有串均以 β开头，其后是零个或多个 α）。因此，这个文法规则在效果上与正则表达式βα*相同。类似地，右递归文法规则

$$A \rightarrow \alpha A \mid \beta$$

⊖ 这是左递归中的特殊情况，称作直接左递归（immediate left recursion），下一章将讨论一些更普通的情况。

（其中β并不在A处结束）生成所有串β、αβ、ααβ、αααβ、……

如果要编写生成与正则表达式 **a*** 相同语言的文法，则文法规则必须有一个用于生成空串的的表示法（因为正则表达式 **a*** 匹配空串）。这样的文法规则的右边必须为空，可在右边什么也不写，如在

$$empty \rightarrow$$

中，但大多数情况都使用ε 元符号表示空串（与在正则表达式中的用法类似）：

$$empty \rightarrow \varepsilon$$

这样的文法规则称作 ε- 产生式（ε-production）。生成包括了空串的文法必须至少有一个 ε-产生式。

现在可以将一个与正则表达式**a***相等的文法写作

$$A \rightarrow Aa \mid \varepsilon$$

或

$$A \rightarrow aA \mid \varepsilon$$

两个文法都生成语言 $\{a^n \mid n$ 是≥0的整型$\} = L$ (**a***)。ε-产生式在定义可选的结构时也很有用，我们马上就能看到这一点。

下面用更多的一些示例来小结这个部分。

例3.5　考虑文法

$$A \rightarrow (A)\,A \mid \varepsilon$$

这个文法生成所有“配对的括号”的串。例如，串 (( ) (( )) ( )就由下面的推导生成（利用ε-产生式去除无用的A）：

$$A \quad (A)\,A \quad (A)\,(A)\,A \quad (A)\,(A) \quad (A)\,() \quad ((A)\,A)\,()$$
$$((\,)\,A)\,() \quad (()\,(A)\,A)\,() \quad (()\,(A))\,()$$
$$((\,)\,((A)\,A))\,() \quad (()\,((\,)\,A))\,() \quad (()\,((\,)))\,()$$

例3.6　例3.4中的语句文法用ε-产生式还可写作：

$$statement \rightarrow if\text{-}stmt \mid \textbf{other}$$
$$if\text{-}stmt \rightarrow \textbf{if (}\ exp\ \textbf{)}\ statement\ else\text{-}part$$
$$else\text{-}part \rightarrow \textbf{else}\ \ statement \mid \varepsilon$$
$$exp \rightarrow \textbf{0} \mid \textbf{1}$$

请注意，ε-产生式指出结构*else part*是可选的。

例3.7　考虑一个语句序列的以下文法G：

$$stmt\text{-}sequence \rightarrow stmt\ ;\ stmt\text{-}sequence \mid stmt$$
$$stmt \rightarrow \textbf{s}$$

这个文法生成由分号分隔开的一个或多个语句序列（语句已被提练到单个终结符 **s**中了）：

$$L(G) = \{\ \textbf{s,}\ \ \textbf{s;s,}\ \ \textbf{s;s;s,}\ \ \textbf{..}\}$$

如要允许语句序列也可为空，则可写出以下的文法G'：

$$stmt\text{-}sequence \rightarrow stmt\ ;\ stmt\text{-}sequence \mid \varepsilon$$
$$stmt \rightarrow \textbf{s}$$

但是它将分号变为一个语句结束符号（terminator）而不是分隔符（separator）：

$$L(G') = \{\varepsilon,\ \textbf{s;,}\ \ \textbf{s;s;,}\ \ \textbf{s;s;s;,}\ \ \textbf{..}\}$$

如果允许语句序列可为空，但仍要求保留分号作为语句分隔符，则须将文法写作：

stmt-sequence → *nonempty-stmt-sequence* |ε

nonempty-stmt-sequence → *stmt* **;** *nonempty-stmt-sequence* | *stmt*

stmt → **s**

这个例子说明在构造可选结构时，必须留意 ε- 产生式的位置。

3.3 分析树与抽象语法树

3.3.1 分析树

推导为构造来自一个初始的非终结符的特定终结符的串提供了一个办法，但是推导并未唯一地表示出它们所构造的结构。总而言之，对于同一个串可有多个推导。例如，使用图 3-1中的推导从简单表达式文法构造出记号串

(number - number) * number

图3-2给出了这个串的另一个推导。二者唯一的差别在于提供的替换顺序，而这其实是一个很表面的差别。为了把它表示得更清楚一些，我们需要表示出终结符串的结构，而这些终结符将推导的主要特征抽取出来，同时却将表面的差别按顺序分解开来。这样的表示法就是树结构，它称作分析树。

(1)	*exp* ⇒ *exp op exp*	[*exp* → *exp op exp*]
(2)	⇒ **(***exp***)** *op exp*	[*exp* → **(** *exp* **)**]
(3)	⇒ **(***exp op exp***)** *op exp*	[*exp* → *exp op exp*]
(4)	⇒ **(number** *op exp***)** *op exp*	[*exp* → **number**]
(5)	⇒ **(number** − *exp***)** *op exp*	[*op* → **-**]
(6)	⇒ **(number - number)** *op exp*	[*exp* → **number**]
(7)	⇒ **(number - number) *** *exp*	[*op* → ***]
(8)	⇒ **(number - number) * number**	[*exp* → **number**]

图3-2 表达式**(34-3)*42** 的另一个推导

与推导相对应的分析树（parse tree）是一个作了标记的树，其中内部的节点由非终结符标出，树叶节点由终结符标出，每个内部节点的子节点都表示推导的一个步骤中的相关非终结符的替换。

以下是一个简单的示例，推导：

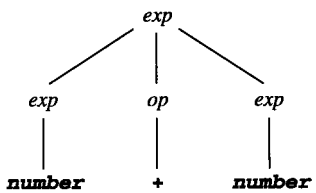
exp *exp op exp*

number *op exp*

number + *exp*

number + number

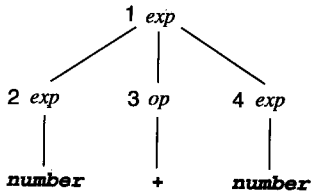
与分析树



相对应。推导中的第 1步对应于根节点的 3个孩子。第 2步对应于根下最左边的 exp的单个**number**孩子，后面的两步与上面的类似。通过将分析树中内部节点编号可将这个对应表示得更清楚一些，编号采用在相应的推导中，与其相关的非终结符被取代的步骤编号。因此，如果如下给前一个推导编号：

(1) *exp* *exp op exp*
(2) **number** *op exp*
(3) **number** + *exp*
(4) **number** + **number**

就可相应地将分析树中的内部节点编号如下：



请注意，该分析树的内部节点的编号实际上是一个前序编号（ preorder numbering)。

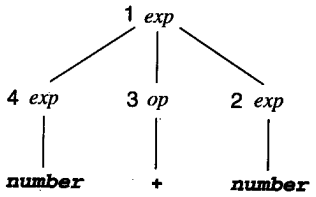
同一个分析树还可与推导

exp *exp op exp*
exp op **number**
exp + **number**
number + **number**

和

exp *exp op exp*
exp + *exp*
number + *exp*
number + **number**

相对应，但是它却提供了内部节点的不同编号。实际上，两个推导中的前一个与下面的编号相对应：



(我们将另一个推导的编号问题留给读者)。此时，该编号与分析树中的内部节点的后序编号(postorder numbering）相反（后序编号将按4、3、2、1的顺序访问内部节点)。

一般而言，分析树可与许多推导相对应，所有这些推导都表示与终结符的被分析串相同的基础结构，但是仍有可能找出那个与分析树唯一相关的推导。最左推导（ leftmost derivation）是指它的每一步中最左的非终结符都要被替换的推导。相应地，最右推导（ rightmost derivation）则是指它的每一步中最右的非终结符都要被替换的推导。最左推导和与其相关的分析树的内部节点的前序编号相对应；而最右推导则和后序编号相对应。

实际上，从刚刚给出的示例中的3个推导和分析中已可看到这个对应了。在3个推导的第1个中给出的是最左推导，而第2个则是最右推导（第3个推导既不是最左推导也不是最右推导）。

再看一个复杂一些的分析树与最左和最右推导的示例——表达式**(34-3)*42**与图3-1和图3-2中的推导。这个表达式的分析树在图3-3中，在其中也根据图3-1的推导为节点编了号。这个推导实际上是最左推导，且分析树的相应编号是相反的后序编号。另一方面，图3-2中的推导是最左推导（我们期望读者能够提供与这个推导相应的分析树的前序编号）。

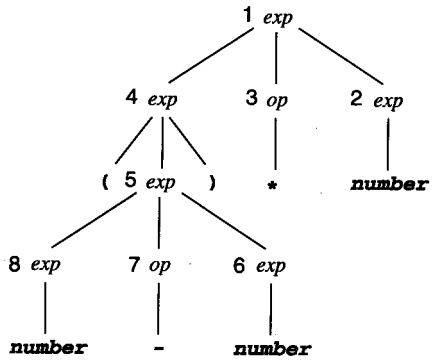
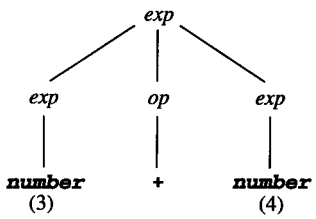


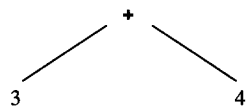
图3-3 算术表达式**(34-3)*42** 的分析树

3.3.2 抽象语法树

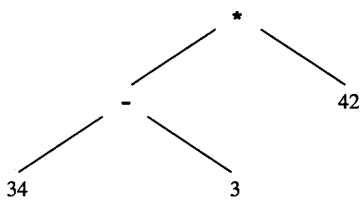
分析树是表述记号串结构的一种十分有用的表示法。在分析树中，记号表现为分析树的树叶（自左至右），而分析树的内部节点则表示推导的各个步骤（按某种顺序）。但是，分析树却包括了比纯粹为编译生成可执行代码所需更多的信息。为了看清这一点，可根据简单的表达式文法，考虑表达式**3+4**的分析树：



这是上一个例子的分析树。我们已经讨论了如何用这个树来显示每一个 ***number***记号的真实数值（这是由扫描程序或分析程序计算的记号的一个特征）。语法引导的转换原则（principle of syntax-directed translation）说明了：如同分析树所表示的一样，串**3+4**的含义或语义应与其语法结构直接相关。在这种情况中，语法引导的转换原则意味着在分析树表示中应加上数值3和数值4。实际上可将这个树看作：根代表两个孩子exp子树的数值相加。而另一方面，每个子树又代表它的每个 ***number***孩子的值。但是还有一个更为简单的方法能表示与这相同的信息，即如树：



这里的根节点仅是被它所表示的运算标出，而叶子节点由其值标出（不是 ***number***记号）。类似地，在图3-3中给出的表达式**(34-3)*42**的分析树也可由下面的树简单表示出来：



在这个树中，括号记号实际已消失了，但它仍然准确地表达着从 34中减去3，然后再乘以42的语义内容。

这种树是真正的源代码记号序列的抽象表示。虽然不能从其中重新得到记号序列（不同于分析树），但是它们却包含了转换所需的所有信息，而且比分析树效率更高。这样的树称作抽象语法树（abstract syntax tree）或简称为语法树（syntax tree）。分析程序可通过一个分析树表示所有步骤，但却通常只能构造出一个抽象的语法树（或与它等同的）。

我们可将抽象语法树想象成一个称作抽象语法（abstract syntax）的快速计数法的树形表示法，它很像普通语法结构的分析树表示法（当与抽象语法相比时，也称作具体语法（ concrete syntax））。例如，表达式**3+4**的抽象语法可写作*OpExp*(*Plus*,*ConstExp*(3),*ConstExp*(4))；而表达式**(34-3)*42**的抽象语法则可写作：

OpExp (*Times*, *OpExp* (*Minus*, *ConstExp* (34), *ConstExp* (3)), *ConstExp* (42))

实际上，可通过使用一个类似BNF的表示法为抽象语法给出一个正式的定义，这就同具体语法一样。例如，可将简单的算术表达式的抽象语法的相似 BNF规则写作：

exp → *OpExp* (*op*, *exp*, *exp*) | *ConstExp* (*integer*)
op → *Plus* | *Minus* | *Times*

对此就不再进一步探讨了，我们把主要的精力放在可被分析程序利用的语法树的真正结构上，它由一个数据类型说明给出⊖。例如，简单的算术表达式的抽象语法树可由C数据类型说明给出：

```
typedef enum {Plus, Minus, Times} OpKind;
typedef enum {OpKind, ConstKind} ExpKind;
typedef struct streenode
     { ExpKind kind;
       OpKind op;
       struct streenode * lchild, *rchild;
       int val;
     } STreeNode;
typedef  STreeNode *SyntaxTree;
```

请注意：除了运算本身（加、减和乘）之外，我们在语法树节点两种不同类型（整型常数和运算）中还使用了枚举类型。实际上是可以利用记号来表示运算，而无需定义一个新的枚举类型。此外还能够使用一个C **union**类型来节省空间，这是因为节点不能既是一个算符节点，同时又是一个常数节点。最后还要指出这些树的节点说明只包括了这些示例直接用到的特征。在实际应用中，编译中使用的特征还会更广泛，例如：数据类型、符号表信息等等，本章后面以及以后几章的示例都会谈到它。

下面用一些通过使用前面例子中谈到过的文法的分析树和语法树的示例来小结这一段。

例3.8　考虑例3.4中被简化了的if语句的文法：

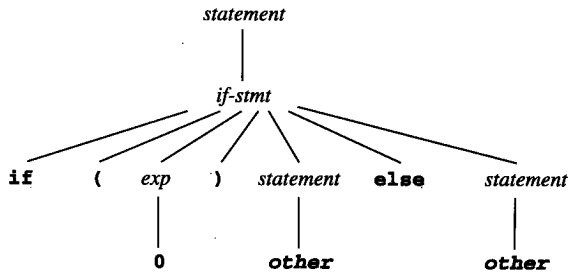
statement → *if-stmt* | ***other***
if-stmt → **if (***exp***)** *statement*
　　| **if (***exp***)** *statement* **else** *statement*
exp → **0** | **1**

串

⊖ 有一些刚刚给出的抽象语法的语言在本质上是一个类型说明，参见练习。

```
if (0) other else other
```

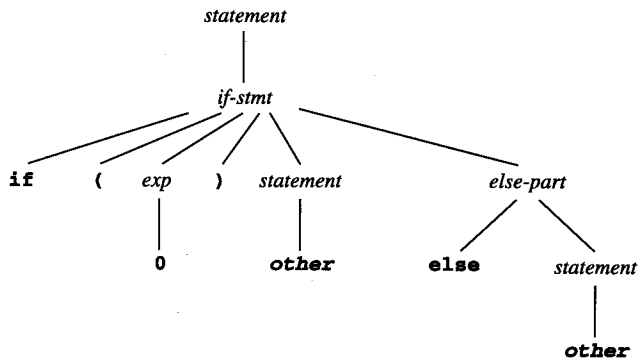
的分析树是：



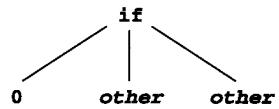
利用例3.6中的文法

statement → *if-stmt* | **other**
if-stmt → **if (***exp***)** *statement else-part*
else-part → **else** *statement* |ε
exp → **0** | **1**

这个串有以下的分析树：



if语句除了3个附属结构之外，它就什么也不需要了，这3个附属结构分别是：测试表达式、then-部分和else部分（如果出现），因此前面一个串的语法树（利用例3.4或例3.6中的文法）是



在这里我们将保留的记号**if**和**other**用作标记以区分语法树中的语句类型。利用枚举类型则会更为恰当一些。例如，利用一个C声明的集合将本例的语句结构和表达式相应地表示如下：

```
typedef enum { ExpK, StmtK} NodeKind;
typedef enum { Zero, One} ExpKind;
typedef enum { IfK, OtherK} StmtKind;
typedef struct streenode
  {NodeKind kind;
   ExpKind ekind;
   StmtKind skind;
   struct streenode
       *test, *thenpart, *elsepart;
```

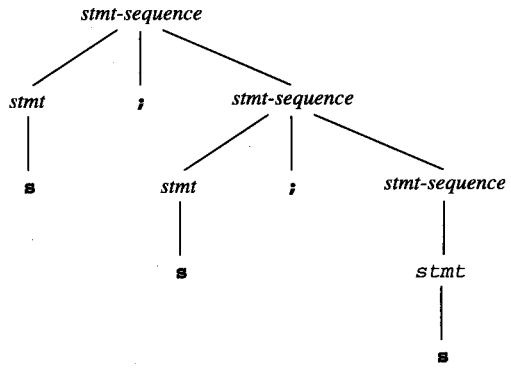
```
  } STreeNode;
typedef STreeNode * SyntaxTree;
```

例3.9　考虑将例3.7中的语句序列的文法用分号分隔开：

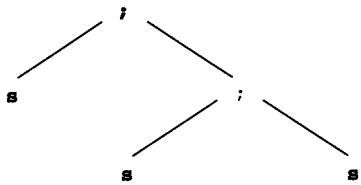
stmt-sequence → *stmt* **;** *stmt-sequence* | *stmt*

stmt → **s**

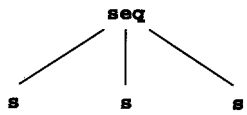
这个**s; s; s**串有关于这个文法的以下分析树：



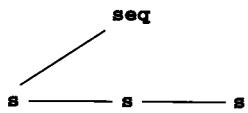
这个串可能有一个语法树：



除了它们的“运算”仅仅是将一个序列中的语句绑定在一起之外，这个树中的分号节点与算符节点（如算术表达式中的**+**节点）类似。我们可以尝试着将一个序列中的所有语句节点都与一个节点绑定在一起，这样前面的语法树就变成



但它存在一个问题：**seq**节点可以有任意数目的孩子，而又很难在一个数据类型说明中提供它。其解决方法是利用树的标准最左孩子右同属（leftmost-child right-sibling）来表示（在大多数数据结构文本中都有）。在这种表示中，由父亲到它的孩子的唯一物理连接是到最左孩子的。孩子则在一个标准连接表中自左向右连接到一起，这种连接称作同属（ sibling）连接，用于区别父子连接。在最左孩子右同属安排下，前一个树现在就变成了：



有了这个安排，我们还可以去掉连接的**seq**节点，那么语法树变得更简单了：

这很明显是用于表示语法树中一个序列的最简单和最方便的方法了。其复杂之处在于这里的连

接是同属连接，它必须与子连接相区分，而这又要求在语法树说明中有一个新的域。

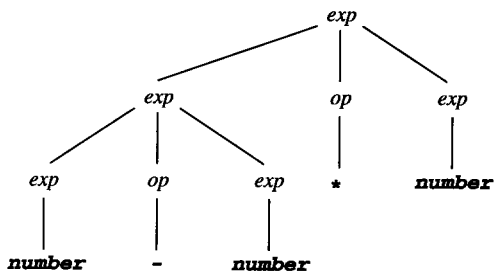
3.4 二义性

3.4.1 二义性文法

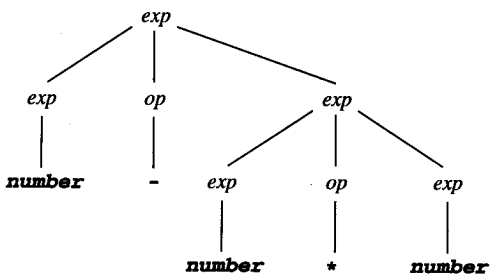
分析树和语法树唯一地表达着语法结构，它们与表达最左和最右推导一样，但并不是对于所有推导都可以。不幸的是，文法有可能允许一个串有多于一个的分析树。例如在前面作为标准示例的简单整型算术文法中

$$exp \rightarrow exp\ op\ exp \mid \texttt{(}\ exp\ \texttt{)} \mid \texttt{number}$$
$$op \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*}$$

和串**34-3*42**，这个串有两个不同的分析树：



和



它们与两个最左推导相对应：

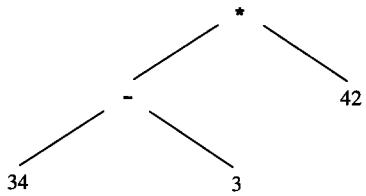
$exp \Rightarrow exp\ op\ exp$	[$exp \rightarrow exp\ op\ exp$]
$\Rightarrow exp\ op\ exp\ op\ exp$	[$exp \rightarrow exp\ op\ exp$]
$\Rightarrow \texttt{number}\ op\ exp\ op\ exp$	[$exp \rightarrow \texttt{number}$]
$\Rightarrow \texttt{number -}\ exp\ op\ exp$	[$op \rightarrow \texttt{-}$]
$\Rightarrow \texttt{number - number}\ op\ exp$	[$exp \rightarrow \texttt{number}$]
$\Rightarrow \texttt{number - number *}\ exp$	[$op \rightarrow \texttt{*}$]
$\Rightarrow \texttt{number - number * number}$	[$exp \rightarrow \texttt{number}$]

和

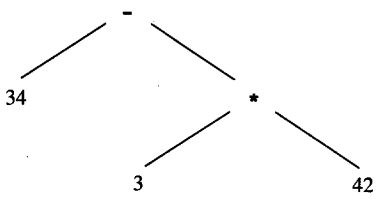
$exp \Rightarrow exp\ op\ exp$	[$exp \rightarrow exp\ op\ exp$]
$\Rightarrow \texttt{number}\ op\ exp$	[$exp \rightarrow \texttt{number}$]
$\Rightarrow \texttt{number -}\ exp$	[$op \rightarrow \texttt{-}$]
$\Rightarrow \texttt{number -}\ exp\ op\ exp$	[$exp \rightarrow exp\ op\ exp$]
$\Rightarrow \texttt{number - number}\ op\ exp$	[$exp \rightarrow \texttt{number}$]

⇒ **number - number * exp** [*op* → *]
⇒ **number - number * number** [*exp* → **number**]

则相应的语法树为：



和



可生成带有两个不同分析树的串的文法称作二义性文法（ambiguous grammar）。由于这个文法并不能准确地指出程序的语法结构（即使是完全确定正规串本身（它是文法的语言成员）），所以它是分析程序表示的一个严重问题。在某种意义上，二义性文法就像是一个非确定的自动机，此时两个不同的路径都可接收相同的串。但是因为没有一个合适的算法，因此就不能像自动机中的情形一样（第2章中讨论过的子集构造），文法中的二义性就不能如有穷自动机中的非确定性一样轻易地被解决⊖。

所以必须将二义性文法认为是一种语言语法的不完善说明，而且也应避免它。幸运的是，二义性文法在后面将介绍到的标准分析算法的测试中总是失败的，而且也开发出了标准技术体系来解决在程序设计语言中遇到的典型二义性。

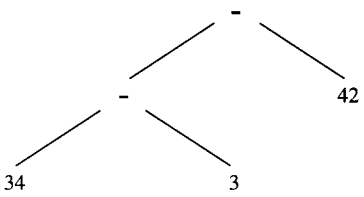
有两个解决二义性的基本方法。其一是：设置一个规则，该规则可在每个二义性情况下指出哪一个分析树（或语法树）是正确的。这样的规则称作消除二义性规则（disambiguating rule）。这样的规则的用处在于：它无需修改文法（可能会很复杂）就可消除二义性；它的缺点在于语言的语法结构再也不能由文法单独提供了。另一种方法是将文法改变成一个强制正确分析树的构造的格式，这样就可以解决二义性了。当然在这两种办法中，都必须确定在二义性情况下哪一个树是正确的。这就再一次涉及到语法制导翻译原则了。我们所需的分析（或语法）树应能够正确地反映将来应用到构造的意义，以便将其翻译成目标代码。

在前面的两个语法树中，哪一个是串**34-3*42**的正确解释呢？第1个树通过把减法节点作为乘法节点的孩子，指出可将这个表达式等于：先做减法 (34 - 3=31)，然后再做乘法 (31*42 = 1302)。相反地，第2个树指出先做乘法(3*42 =126)然后再做减法(34 - 126= - 92)。选择哪个树取决于我们认为哪一个计算是正确的。人们认为乘法比减法优先（ precedence）。通常地，乘法和除法比加法和减法优先。

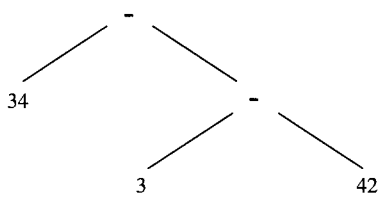
为了去除在这个简单表达式文法中的二义性，现在可以只需设置消除二义性规则，它建立了3个运算相互之间的优先关系。其标准解决办法是给予加法和减法相同的优先权，而乘法和除法则有高一级的优先权。

⊖ 情况甚至更糟，这是因为没有算法可以在一开始时就确定一个文法是否是二义性的，参见第 3.2.7节。

不幸的是，这个规则仍然不能完全地去除掉文法中的二义性。例如串 **34-3-42**，这个串也有两种可能的语法树：



和



第1个语法树表示计算(34-3)-42=-11，而第2个表示计算34-(3-42)=73。而哪一个算式正确又是一个惯例的问题，标准数学规定第1种选择是正确的。这是由于规定减法为左结合（left associative），也就是认为一个减法序列的运算是自左向右的。

因此，这又要求有一个消除二义性的规则：它能处理每一个加法、减法和乘法的结合性。这个消除二义性的规则通常都规定它们为左结合，它确实消除了简单表达式文法中其他的二义性问题（在后面再证明它）。

因为在表达式中不允许有超过一个算符的序列，有时还需要规定运算是非结合性的（nonassociative）。例如，可将简单表达式文法用下面的格式写出：

exp → *factor op factor* | *factor*
factor → **(** *exp* **)** | **number**
op → **+** | **-** | *****

在这种情况中，诸如**34-3-42**甚至于**34-3*42**的表达式都是正规的，但它们必须带上括号，例如**(34-3)-42**和**34-(3*42)**。这样的完全括号表达式（fully parenthesized expression）就没有必要说明其结合性或优先权了。上面的文法正如所写的一样去除了二义性。当然，我们不仅改变了文法，还更改了正被识别的语言。

我们不再讨论消除二义性的规则，现在来谈谈重写文法以消除二义性的方法。请注意，必须找到无需改变正被识别的基本串的办法（正如完全括号表达式的示例所做的一样）。

3.4.2 优先权和结合性

为了处理文法中的运算优先权问题，就必须把具有相同优先权的算符归纳在一组中，并为每一种优先权规定不同的规则。例如，可把乘法比加法和减法优先添加到简单表达式文法，如下所示：

exp → *exp addop exp* | *term*
addop → **+** | **-**
term → *term mulop term* | *factor*
mulop → *****
factor → **(** *exp* **)** | **number**

在这个文法中，乘法被归在*term*规则下，而加法和减法则被归在*exp*规则之下。由于*exp*的

基本情况是*term*，这就意味着加法和减法在分析树和语法树中将被表现地“更高一些”(也就是，更接近于根)，由此也就接受了更低一级的优先权。这样将算符放在不同的优先权级别中的办法是在语法说明中使用BNF的一个标准方法。这种分组称作优先级联（precedence cascade)。

简单算术表达式的最后一种文法仍未指出算符的结合性而且仍有二义性。它的原因在于算符两边的递归都允许每一边匹配推导（因此也在分析树和语法树）中的算符重复。解决方法是用基本情况代替递归，强制重复算符匹配一边的递归。这样将规则

$$exp \rightarrow exp\ addop\ exp \mid term$$

替换为

$$exp \rightarrow exp\ addop\ term \mid term$$

使得加法和减法左结合，而

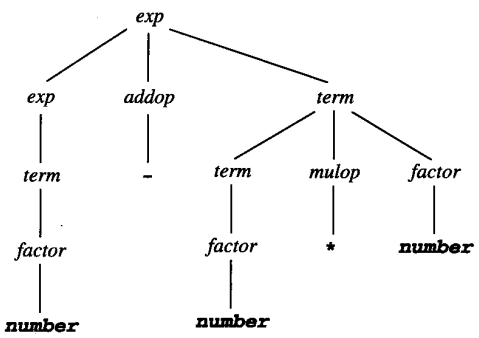
$$exp \rightarrow term\ addop\ exp \mid term$$

却使得它们右结合。换而言之，左递归规则使得它的算符在左边结合，而右递归规则使得它们在右边结合。

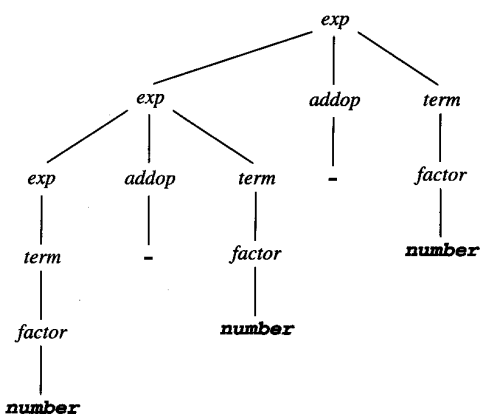
为了消除简单算术表达式BNF规则中的二义性，重写规则使得所有的运算都左结合：

$$exp \rightarrow exp\ addop\ term \mid term$$
$$addop \rightarrow \mathbf{+} \mid \mathbf{-}$$
$$term \rightarrow term\ mulop\ factor \mid factor$$
$$mulop \rightarrow \mathbf{*}$$
$$factor \rightarrow \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}$$

现在表达式**34-3*42**的分析树就是



表达式**34-3-42**的分析树是：



注意，优先级联使得分析树更为复杂。但是语法树并不受影响。

3.4.3 悬挂else问题

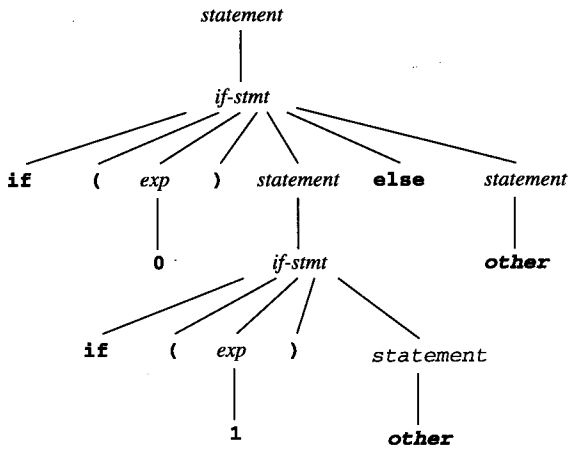
考虑例3.4中的文法：

statement → *if-stmt* | **other**
if-stmt → **if(** *exp* **)** *statement*
| **if(** *exp* **)** *statement* **else** *statement*
exp → **0** | **1**

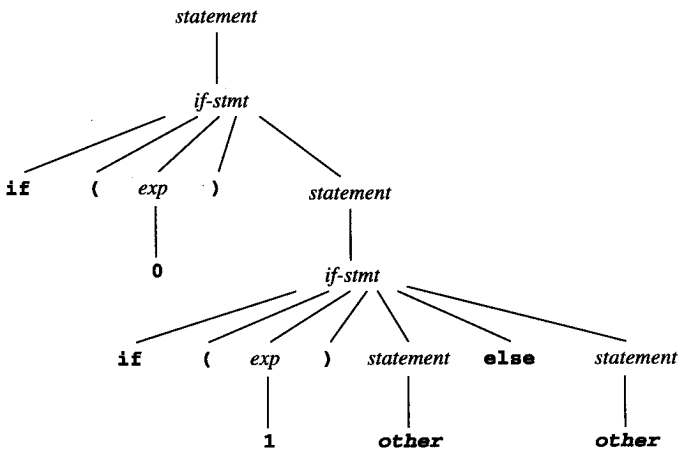
由于可选的else的影响，这个文法有二义性。为了看清它，可考虑下面的串：

```
if (0) if (1) other else other
```

这个串有两个分析树：



和



哪一个是正确的则取决于将单个 else部分与第1个或第2个if语句结合：第1个分析树将else部分与第1个if语句结合；第 2个分析树将它与第 2个if语句结合。这种二义性称作悬挂 else问题(dangling else problem)。为了分清哪一个分析树是正确的，我们必须考虑 if语句的隐含意思，请看下面的一段C代码：

```
if (x != 0)
  if  (y == 1/x) ok = TRUE
  else z = 1/x
```

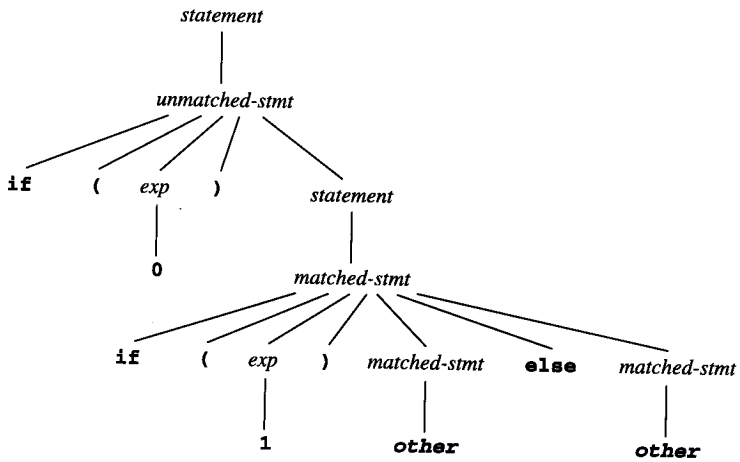
在这个代码中，如果else部分与第1个if语句结合，只要x等于0，则会发生一个除以零的错误。因此这个代码的含义（实际是else部分缩排的含义）是指一个else部分应总是与没有else部分的最近的if语句结合。这个消除二义性的规则被称为用于悬挂 else问题的最近嵌套规则（most closely nested rule），这就意味着上面第2个分析树是正确的。请注意：如要将else部分与第1个if语句相结合，则要用C中的**{...}**，如在

```
if (x != 0)
  {if (y == 1/x) ok = TRUE;}
else z = 1/x;
```

解决这个位于BNF本身中的悬挂else二义性要比处理前面的二义性困难。方法如下：

statement → *matched-stmt* | *unmatched-stmt*
matched-stmt → **if(** *exp*) *matched-stmt* **else** *matched-stmt* | ***other***
unmatched-stmt → **if(** *exp*) *statement*
| **if(** *exp*) *matched-stmt* **else** *unmatched-stmt*
exp → **0** | **1**

它允许在if语句中，只有一个*matched-stmt*出现在**else**之前，这样就迫使尽可能快地匹配所有的else部分。例如，经过结合了的简单串的分析树现在就变成了：



它确实将else 部分与第2个if 语句相连。

通常不能在BNF中建立最近嵌套规则，实际上经常所用的是消除二义性的规则。原因之一是它增加了新文法的复杂性，但主要原因却是分析办法很容易按照遵循最近嵌套规则的方法来配置（在无需重写文法时自动获得优先权和结合性有一点困难）。

悬挂else问题来源于Algol60的语法。我们还是有可能按照悬挂 else问题不出现的方法来设计语法。办法之一是要求出现 else部分，而该办法已在LISP和其他函数语言中用到了（但须返回一个值）。另一种办法是为if语句使用一个带括号关键字（bracketing keyword）。使用这种方法的语言包括了Algol68和Ada。例如，程序员写

```
if x /=0 then
  if y=1/x then  ok := true ;
```

```
    else z := 1/x ;
    end if;
  end if;
```

将else部分与第2个if语句结合在一起。程序员还可写出

```
  if x /=0 then
    if y = 1/x  then ok := true;
    end if;
  else z := 1/x;
  end if;
```

将它与第1个if语句结合在一起。Ada中相应的BNF（有一些简单化了）就是

if-stmt → **if** *condition* **then** *statement-sequence* **end if**
| **if** *condition* **then** *statement-sequence*
else *statement-sequence* **end if**

因此两个关键字**end if**就是Ada的括号关键字。在Algol68中，括号关键字是**fi**（反着写的**if**）。

3.4.4 无关紧要的二义性

有时文法可能会有二义性并且总是生成唯一的抽象语法树。例如，在例 3.9中的语句序列文法中，可选择一个类似于语法树的简单同属表。在这种情形下，右递归文法规则或左递归文法规则仍导致相同的语法树结构，且可将文法二义地写作

stmt-sequence → *stmt-sequence* **;** *stmt-sequence* | *stmt*
stmt → **s**

且仍然可得到唯一的语法树。由于相结合的语义不必依赖于使用的是哪种消除二义性的规则，所以可将这样的二义性称作无关紧要的二义性（ inessential ambiguity）。同样的情况出现在二进制算符中，例如算术加法或串的并置表现为可结合运算（ associative operation）（若对于所有的值a、b 和c，且有$(a \cdot b) \cdot c = a \cdot (b \cdot c)$，那么二进制算符“ · ”也是可结合的）。此时的语法树仍然各不相同，但是却表示相同的语义值，而且我们可能也无需在意到底使用的是哪一个。然而，分析算法却要提供一些消除二义性的规则，这些都是编译器编写者所需的。

3.5 扩展的表示法：EBNF和语法图

这一部分将对两种扩展表示法分别进行讲解。

3.5.1 EBNF表示法

重复和可选的结构在程序设计语言中极为普通，因此在 BNF文法规则中也是一样的。所以当看到BNF表示法有时扩展到包括了用于这两个情况的特殊表示法时，也不应感到惊奇。这些扩展包含了称作**扩展的BNF**（extended BNF）或**EBNF**的表示法。

首先考虑重复情况，如在语句序列中。我们已看到重复是由文法规则中的递归表达，并可能使用了左递归或右递归，它们由一般规则

$$A \rightarrow A\alpha|\beta \quad \text{（左递归）}$$

和

$$A \rightarrow \alpha A|\beta \quad \text{（右递归）}$$

指出，其中α和β是终结符和非终结符的任意串，且在第 1个规则中β不以*A*开始，在第2个规则中β不以*A*结束。

重复有可能使用与正则表达式所用相同的表示法，即：星号 *（在正则表达式中也称作Kleene闭包）。则这两个规则可被写作非递归规则

$$A \rightarrow \beta\alpha^*$$

和

$$A \rightarrow \alpha^*\beta$$

相反地，EBNF选择使用花括号{...}来表示重复（因此清晰地表达出被重复的串的范围），且可为规则写出

$$A \rightarrow \beta\{\alpha\}$$

和

$$A \rightarrow \{\alpha\}\beta$$

使用重复表示法的问题是：它使得分析树的构造不清楚，但是正如所看到的一样，我们对此并不在意，例如语句序列的情形（例3.9）。在右递归格式中写出如下文法：

stmt-sequence → *stmt* **;** *stmt-sequence* | *stmt*
stmt → **s**

这个规则具有格式*A*→α*A* | β，且*A* = *stmt-sequence*，α= *stmt* **;** β= *stmt*。在EBNF中，它表现为：

stmt-sequence → { *stmt* **;** } *stmt* （右递归格式）

同样也可使用一个左递归规则并得到EBNF

stmt-sequence → *stmt* { **;** *stmt* } （左递归格式）

实际上，第2个格式是通常所用的（原因在下一章再讲）。

出现在结合性中的更大的一个问题是发生在诸如二进制运算的减法和除法中。例如，前面减法的简单表达式文法中的第1个文法规则：

exp → *exp addop term* | *term*

它使得*A* → *A*α | β，且*A* = *exp*，α = *addop term*，β = *term*。因此，将这个规则在EBNF中写作：

exp → *term* { *addop term* }

尽管规则本身并未明显地说明出来，但现在仍可假设它暗示了左结合。我们还可假设通过写出

exp →{ *term addop* } *term*

来暗示一个右结合规则，但事实并不是这样。相反地，诸如

stmt-sequence → *stmt* **;** *stmt-sequence* | *stmt*

的右递归规则可被看作后接一个可选的分号和*stmt-sequence*的*stmt*。

EBNF中的可选结构可通过前后用方括号[. . .]表示出来。这同将问号放在可选部分之后的正则表达式惯例本质上是一样的，但它另有无需括号就可将可选部分围起来的优点。例如，用带有可选的else部分的if语句（例3.4和例3.6）的文法规则在EBNF中写作：

statement → *if-stmt* | ***other***
if-stmt → **if(** *exp* **)** *statement* [**else** *statement*]

$$exp \rightarrow \mathbf{0} \mid \mathbf{1}$$

而诸如

$$stmt\text{-}sequence \rightarrow stmt\ \mathbf{;}\ stmt\text{-}sequence \mid stmt$$

的右递归可写作：

$$stmt\text{-}sequence \rightarrow stmt\ [\ \mathbf{;}\ stmt\text{-}sequence\ ]$$

（请与前面使用花括号写在左递归格式中的这个规则作一比较）。

如果希望在右结合中写出一个诸如加法的算术运算，则可用

$$exp \rightarrow term\ [\ addop\ exp\ ]$$

来代替花括号的使用。

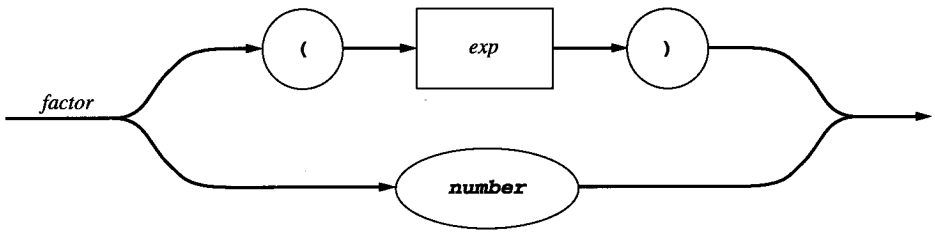
3.5.2 语法图

用作可视地表示EBNF规则的图形表示法称作语法图（syntax diagram）。它们是由表示终结符和非终结符的方框、表示序列和选择的带箭头的线，以及每一个表示文法规则定义该非终结符的图表的非终结符标记组成。圆形框和椭圆形框用来指出图中的终结符，而方形框和矩形框则用来指出非终结符。

例如，文法规则

$$factor \rightarrow \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}$$

用语法图表示则是：

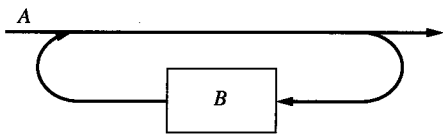


请注意，*factor* 并未放在框中，而是用作语法图的标记来指出该图表示该名称结构的定义。另外还需注意带有箭头的线用作指明选择和顺序。

语法图是从EBNF而并非BNF中写出来的，所以需要用图来表示重复和可选结构。给出下面的重复：

$$A \rightarrow \{\ B\ \}$$

相对应的语法图通常画作

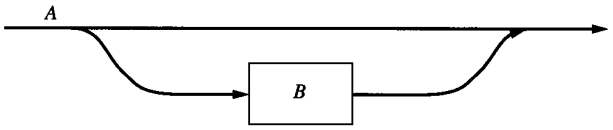


请留意该图必须允许根本没有出现*B*。

诸如

$$A \rightarrow [\ B\ ]$$

的可选结构可画作



下面的示例是几个使用了前面EBNF的例子，我们用它们来小结语法图的讨论。

例3.10　考虑简单算术表达式的运行示例。它具有BNF（包括结合性和优先权）。

exp → *exp addop term* | *term*
addop → **+** | **-**
term → *term mulop factor* | *factor*
mulop → ***
factor → **(** *exp* **)** | ***number***

相对应的EBNF是

exp → *trem* { *addop term* }
addop → **+** | **-**
term → *factor* { *mulop factor* }
mulop → ***
factor → **(** *exp* **)** | ***number***

图3-4是相对应的语法图（factor的语法图已在前面给出了）。

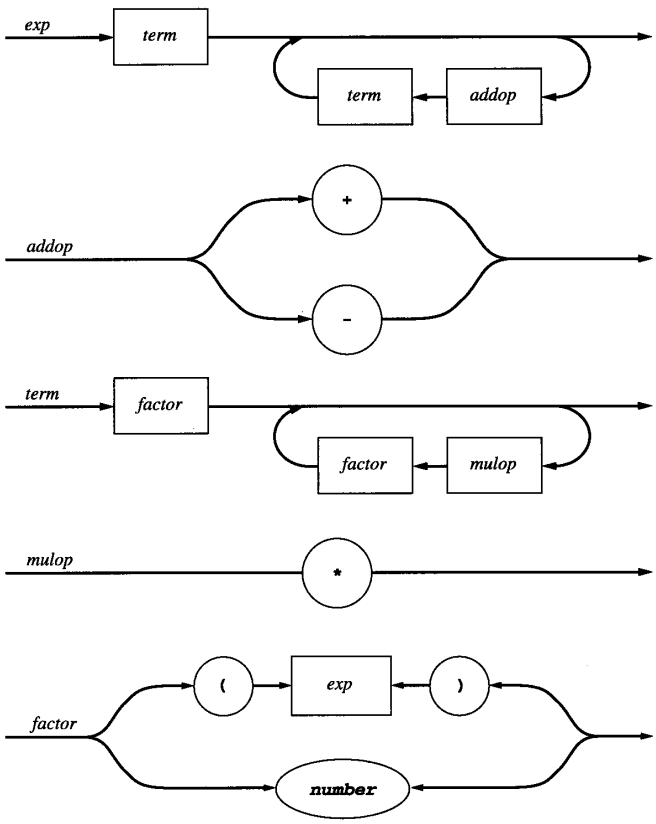


图3-4　例3.10中文法的语法图

例3.11 考虑例3.4中简化了的if语句的文法。它有BNF

statement → *if-stmt* | **other**
if-stmt → **if (***exp***)** *statement*
| **if (***exp***)** *statement* **else** *statement*
exp → **0** | **1**

EBNF为

statement → *if-stmt* | **other**
if-stmt → **if (** *exp* **)** *statement* [**else** *statement*]
exp → **0** | **1**

图3-5是相对应的语法图。

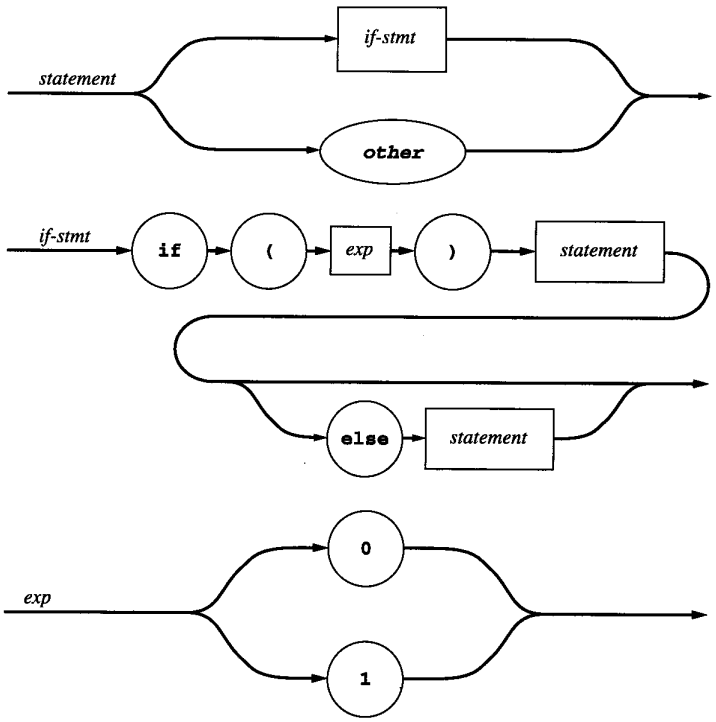


图3-5 例3.11中文法的语法图

3.6 上下文无关语言的形式特性

在这里用更正式的数学方法写出本章前面所提到过的一些术语和定义。

3.6.1 上下文无关语言的形式定义

本节给出上下文无关文法的形式定义。

定义 上下文无关文法由以下各项组成：
1) 终结符（terminal）集合T。
2) 非终结符（nonterminal）集合N（与T不相交）。

3) 产生式（production）或文法规则（grammar rule）$A \to \alpha$的集合P，其中A是N的一个元素，α是（$T \quad N$） 中的一个元素（是终结符和非终结符的一个可为空的序列）。

4) 来自集合N的开始符号（start symbol）。

令G是一个如上所定义的文法，则$G = (T, N, P, S)$。G上的推导步骤（derivation step）格式$\alpha A\gamma \quad \alpha\beta\gamma$，其中$\alpha$ 和γ 是$(T \quad N)^*$中的元素，且有$A \to \beta$在P中（终结符和非终结符的并集，有时被称作G的符号集，set of symbol），且$(T \quad N)^*$中的串α被称作句型（sentential form））。将关系$\alpha \quad {}^*\beta$定义为推导步骤关系 的传递闭包；也就是：假若有零个或更多个推导步骤，就有$\alpha \quad \beta$（$n \geq 0$）

$$\alpha_1 \quad \alpha_2 \quad \cdots \quad \alpha_{n-1} \quad \alpha_n$$

其中$\alpha=\alpha_1$，$\beta=\alpha_n$（如果$n=0$，则$\alpha=\beta$）。在文法G上的推导（derivation）形如$S \quad {}^*w$，且$w \; T^*$（即：w 是终结符的一个串，称作句子（sentence）），S是G的开始符号。

由G生成的语言（language generated by G）写作$L(G)$，它被定义为集合$L(G) = \{ w \quad T^* |$ 存在G的一个推导$S \quad {}^*w\}$，也就是：$L(G)$是由S推导出的句子的集合。

最左推导（leftmost derivation）$S \quad {}^*w$ 是一个推导，在其中的每一个推导步骤$\alpha A\gamma \quad \alpha\beta\gamma$ 都有$\alpha \quad {}^{lm}T^*$，换言之，α 仅由终结符组成。类似地，最右推导（rightmost derivation）就是每一个推导步骤$\alpha A\gamma \quad \alpha\beta\gamma$ 都有属性$\gamma \quad T^*$。

文法G上的分析树（parse tree）是一个带有以下属性的作了标记的树：

1) 每个节点都用终结符、非终结符或ε 标出。

2) 根节点用开始符号S标出。

3) 每个叶节点都用终结符或ε 标出。

4) 每个非叶节点都用非终结符标出。

5) 如带有标记$A \quad N$的节点有n 个带有标记$X_1, X_2, \ldots X_n$ 的孩子（可以是终结符也可以是非终结符），就有$A \to X_1, X_2, \ldots X_n \quad P$（文法的一个产生式）。

每一个推导都引出一个分析树，这个分析树中的每一个步骤 $\alpha A\gamma \quad \alpha\beta\gamma$ 都在推导中，且$\beta= X_1, X_2, \ldots X_n$ 与带有标记$X_1, X_2, \ldots X_n$ 的n 个孩子的结构相对应，其中$X_1, X_2, \ldots X_n$ 带有标记A。许多推导可引出相同的分析树。但每个分析树只有唯一的一个最左推导和一个最右推导。最左推导与分析树的前序编号相对应，而与之相反，最右推导与分析树的后序编号相对应。

若上下文无关文法 G有$L=L(G)$，就将串L的集合称作上下文无关语言（ context-free language）。一般地，许多不同的文法可以生成相同的上下文无关语言，但是根据所使用的文法的不同，语言中的串也会有不同的分析树。

若存在串$w \quad L(G)$，其中w 有两个不同的分析树（或最左推导或最右推导），那么文法G就有二义性（ambiguous）。

3.6.2 文法规则和等式

在本节的开始，我们注意到文法规则用箭头符号而不是等号来定义结构名称（非终结符），这与在正则表达式中用等号定义名称的表示法不同。其原因在于文法规则的递归本质使得定义关系（文法规则）并不完全与相等一样，而且我们确实也看到了从推导得出的由文法规则定义的串，而且在BNF中的箭头指示之后，使用了一个从左到右的替换方法。

但文法规则的左、右两边仍然保存某种等式关系，但由这个观点引出的语言定义过程与正则表达式中的不同。这个观点对于程序设计语言语义理论很重要，且由于它对诸如分析的递归

过程的重要作用（即使我们所学习的分析算法并不基于此），因此值得学习。

例如下面的文法规则是从简单表达式文法中抽取出来的（在简化了的格式中）：

$$exp \rightarrow exp + exp \mid \textbf{number}$$

我们已经看到如*exp*的非终结符名称定义了一个终结符的串集合（若非终结符是开始符号，那么它就是文法的语言）。假设将这个集合称作 E，并令N为自然数集（与正则表达式名称**number**对应），则给出的文法规则可解释为集合等式：

$$\mathbf{E} = (\mathbf{E} + \mathbf{E}) \quad \mathbf{N}$$

其中**E+E**是串{ $u + v \mid u, v \quad \mathbf{E}$ }的集合（这里并未添加串u 和串v，而只是用符号“ + ”将它们连接在一起）。

这是集合**E**的递归等式。思考一下它是如何定义 **E**的。首先，由于**E**是**N**与**E+E**的并集，所以**E**中包含了集合**N**（这是基本情况）。其次，等式**E**中也包含了**E+E**，就意味着**N+N**也在**E**中。而且又由于**N**和**N+N**均在**E**中，所以**N+N+N**也在**E**中，同理类推。我们可以把这看作是一个更长的串的归纳结构，且所有这些集合的和就是所需结果：

$$\mathbf{E} = \mathbf{N} \quad (\mathbf{N} + \mathbf{N}) \quad (\mathbf{N} + \mathbf{N} + \mathbf{N}) \quad (\mathbf{N} + \mathbf{N} + \mathbf{N} + \mathbf{N}) \quad \ldots$$

实际上，可以证明这个**E**满足正在讨论的等式，而**E**其实是最小的集合。如将**E**的等式右边看作是**E**的一个函数（集），则可定义出$f(s) = (s + s) \quad \mathbf{N}$，此时**E**的等式就变成了$\mathbf{E} = f(\mathbf{E})$。换而言之，**E**是函数$f$ 的一个固定点（fixed point），且（根据前面的注释）它确实是这样一个最小的点。我们将由这种办法定义的**E**看作是给定的最小的固定点语义（least-fixed point semantics）。

当根据本书后面要讲到的通用算法来完成它们时，被递归地定义的程序设计语言，例如语法、递归数据类型和递归函数，都能证明出它具有最小的固定点语义。因为以后将会用到这样的办法来证明编译的正确性，所以这十分重要。现在的编译器很少能被证明是正确的。而编译器代码的测试只能证实近似地正确，而且仍然保留了许多错误，即使是商品化生产的编译器也是这样的。

3.6.3 乔姆斯基层次和作为上下文无关规则的语法局限

对于表示程序设计语言的语法结构， BNF和EBNF中的上下文无关文法是一个有用且十分高效的工具，但是掌握BNF能够和应该表示什么同样也十分重要。我们已经遇到过故意使文法具有二义性的情形（悬挂else问题），那么因此也就不能直接表示出完整的语法来了。当试图在文法中表示太多的东西时，或在文法中表示一个极不可能的要求时，会出现其他情况。本节将讨论一些常见的情况。

当为某个语言编写 BNF时，会遇到一个常见的问题，那就是：词法结构的范围应表示在BNF中而不是在一个单独的描述中（可能使用正则表达式）。前面的讨论已指出上下文无关文法可以表达并置、重复和选择，这与正则表达式是一样的。因此我们能够为所有来自字符的记号结构写出文法规则，并将其与正则表达式一起分配。

例如，考虑一个使用正则表达式定义作为数字序列的数：

```
digit = 0|1|2|3|4|5|6|7|8|9
number = digit digit*
```

用BNF写出这个定义：

$$digit \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

$$number \rightarrow number\ digit \mid digit$$

注意，第2个规则中的递归仅仅是用来表达重复。人们认为带有这个属性的文法是一个正规文法（regular grammar），正规文法能够表达任何正则表达式可以表达的内容。因此，我们就可以设计一个分析程序，这个程序可以直接从输入源文件中接收字符并与扫描程序一起分配。

它确实是一个很好的办法。分析程序是一个比扫描程序更有效的机器，但相应地效率要差一些。然而我们还是有理由在BNF本身中包括记号的定义，文法会接着表达完整的语法结构，其中还包括了词法结构。当然，人们希望语言设备能从文法中抽取这些定义并将它们返回到一个扫描程序之上。

另一种情况的发生与上下文规则（context rule）有关，它经常发生在程序设计语言中。我们以前一直使用着术语“上下文无关”，但实际上却并未解释为什么这样的规则是“上下文无关”。其简单原因在于非终结符本身就出现在上下文无关规则中箭头的左边。因此，规则

$$A \rightarrow \alpha$$

就说明了无论在何处发生 A，它都可被任何地方的 α 所替代。另一方面，可以将上下文（context）非正式地定义为（终结符和非终结符的）一对串 β、γ，这样仅当β和γ分别在非终结符的前面和后面发生时，才提供一个规则。可将此写作

$$\beta A \gamma \rightarrow \beta \alpha \gamma$$

当$\alpha \neq \varepsilon$ 时，该规则称作上下文有关文法规则（context-sensitive grammar rule）。上下文有关文法规则比上下文无关文法规则更强大，但要作为分析程序的基础却仍有许多困难。

在程序设计语言中，对于上下文有关规则有哪些要求呢？典型的要求包括了名称的使用。要求在使用之前先说明（declaration before use）的C规则就是一个典型的例子。此时在允许语句或表达式使用名称之前必须在一个说明中先出现一个名称：

```
{ int x;
...
...x...
...
}
```

如果想用BNF规则处理这个要求，也是可以做到的。首先，文法规则必须包括了名称串本身，而不是包括不可区别的作为标识符记号的所有名称。其次，须为每个名称写出一个建立在可能的使用之前的说明规则。但是在许多语言中，标识符的长度是非限制性的，因此标识符的数量也是（至少是可能的）无限的。即使只允许名称为两个字符，仍然有可能有几百种新的文法规则。这很明显是不实际的。这种情形与消除二义性规则的情况类似：只需说出在文法中不是显式的规则（使用之前的声明）。但两者有区别：这样的规则不能作为分析程序本身，这是由于它超出了（可能的）上下文无关规则所能表达的能力。相反地，由于它取决于符号表的使用（它记录了声明的是哪一些标识符），因此该规则就变成了语义分析的一部分。

超过要分析程序检查的范围，但仍能被编译器检查的语言规则体称作语言的静态语义（static semantics）。它们包括类型检查（在一个静态分类的语言中）和诸如使用前先说明的规则。因此，只有当BNF规则可表达这些规则时才将其当作语法，而把其他的都当作语义。

除上下文有关文法之外，还有一种更常见的文法。这些文法称作非限制的文法（unrestricted grammar），它们具有格式$\alpha \rightarrow \beta$的文法规则，在其中对格式α和β没有限制（除了α不能是ε 之外）。4种文法——非限制的、上下文有关的、上下文无关的和正则的，分别被称为0型、1型、2型和3型文法。它们构成的语言类称为以乔姆斯基命名的乔姆斯基层次（Chomsky hierarchy），这是因为乔姆斯基最先将它们用来描述自然语言。这些文法表示计算

能力的不同层次。非限制的（或0型）文法与图灵机完全相等，这与正则文法和有穷自动机相等是一样的，而且由此表示了绝大多数已知的计算。上下文无关文法也有一个对应的相等机器，即：下推自动机，但是对于分析算法而言，并不需要这种机器的全部能力，所以也就不再讨论它了。

我们还应留意一些上下文无关语言和文法相关的难处理的计算问题。例如，在处理二义性文法时，如能找到一个可将二义性文法转变成非二义性文法的算法并且又不会改变语言本身，那就太好了。然而不幸的是，大家都认为这是一个无法决定的事情，因此这样的算法是不可能存在的。实际上，甚至还存在着没有非二义性文法的上下文无关语言（称之为先天二义性语言（inherently ambiguous language）），而且判断一种语言是否是先天二义性的也是无法做到的。

幸运的是，程序设计语言并不会引起诸如先天二义性的复杂问题，而且也证明了经常谈到的消除二义性的特殊技术在实际应用中很合适。

3.7 TINY语言的语法

3.7.1 TINY的上下文无关文法

程序清单3-1是TINY在BNF中的文法，我们可从中观察到一些内容：TINY程序只是一个语句序列，它共有5种语句：if 语句、repeat 语句、read 语句、write 语句和assignment 语句。除了if语句使用**end**作为括号关键字（因此在TINY中没有悬挂else 二义性）以及if 语句和repeat语句允许语句序列作为主体之外，它们都具有类似于 Pascal 的语法，所以也就不需要括号或**begin-end**对（而且**begin**甚至在TINY中也不是一个保留字）。输入/输出语句由保留字**read**和**write**开始。read 语句一次只读出一个变量，而write 语句一次只写出一个表达式。

程序清单3-1 BNF中的TINY的文法

```
program → stmt-sequence
stmt-sequence → stmt-sequence ; statement | statement
statement → if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt
if-stmt → if exp then stmt-sequence end
        | if exp then stmt-sequence else stmt-sequence end
repeat-stmt → repeat stmt-sequence until exp
assign-stmt → identifier := exp
read-stmt → read identifier
write-stmt → write exp
exp → simple-exp comparison-op simple-exp | simple-exp
comparison-op → < | =
simple-exp → simple-exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → * | /
factor → ( exp ) | number | identifier
```

TINY表达式有两类：在 if 语句和repeat 语句的测试中使用比较算符 =和<的布尔表达式，以及包括标准整型算符+、-、*和/（它代表整型除法，有时也写作div）的算术表达式（由文法中的*simple-exp*指出）。算术运算是左结合并有通常的优先关系。相反地，比较运算却是非结合的：每个没有括号的表达式只允许一种比较运算。比较运算比其他算术运算的优先权都低。

TINY中的标识符指的是简单整型变量，它没有诸如数组或记录构成的变量。 TINY中也没有变量声明：它只是通过出现在赋值语句左边来隐式地声明一个变量。另外，它只有一个（全局）作用域，且没有过程或函数（因此也就没有调用）。

还要注意TINY的最后一个方面。语句序列必须包括将语句分隔开来的分号，且不能将分号放在语句序列的最后一个语句之后。这是因为 TINY没有空语句（不同于Pascal和C）。另外，我们将stmt-sequence的BNF规则也写作一个左递归规则，但却并不真正在意语句序列的结合性，这是因为意图很简单，只需按顺序执行就行了。因此，只要将stmt-sequence编写成右递归即可。这个观点也出现在 TINY程序的语法树结构中，其中的语法序列不是由树而是由列表表示的。现在就转到这个结构的讨论上来。

3.7.2 TINY编译器的语法树结构

TINY有两种基本的结构类型：语句和表达式。语句共有 5类（if语句、repeat语句、assign语句、 read语句和write语句），表达式共有3类（算符表达式、常量表达式和标识符表达式）。因此，语法树节点首先按照它是语句还是表达式来分类，接着根据语句或表达式的种类进行再次分类。树节点最大可有 3个孩子的结构（仅在带有 else部分的if语句中才需要它们）。语句通过同属域而不是使用子域来排序。

必须将树节点中的属性保留如下（除了前面所提到过的域之外）：每一种表达式节点都需要一个特殊的属性。常数节点需要它所代表的整型常数的域；标识符节点应包括了标识符名称的域；而算符节点则需要包括了算符名称的域。语句节点通常不需要属性（除了它们的节点类型之外）。但为了简便起见，在assign语句和read语句中，却要保留在语句节点本身中（除了作为一个表达式子节点之外）被赋予或被读取的变量名。

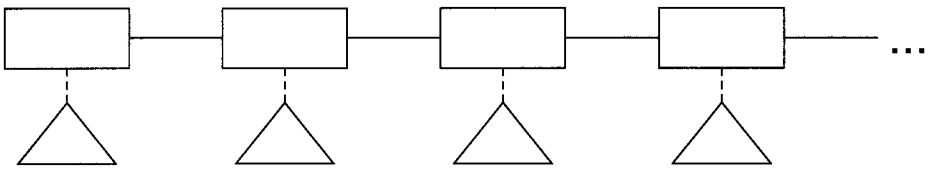
前面所描述的三个节点结构可通过程序清单 3-2中的C说明得到，该说明还可在附录 B（第198行到第217行）的globals.h文件的列表中找到。请注意我们综合了这些说明来帮助节省空间。它们还可帮助提醒每个节点类型的属性。现在谈谈说明中两个未曾提到过的属性。第 1个是簿记属性lineno；它允许在转换的以后步骤中出现错误时能够打印源代码行数。第 2个是type域，在后面的表达式（且仅是表达式）类型检查中会用到它。它被说明为枚举类型 ExpType，第6章将会完整地讨论到它。

程序清单3-2 一个TINY语法树节点的C声明

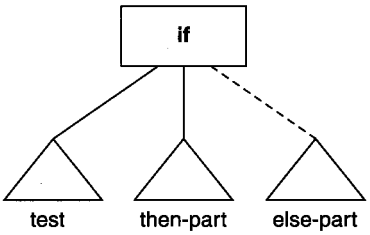
```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK}
             StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* ExpType is used for type checking */
typedef enum {Void,Integer,Boolean} ExpType;

#define MAXCHILDREN 3

typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN];
     struct treeNode * sibling;
     int lineno;
     NodeKind nodekind;
     union { StmtKind stmt; ExpKind exp;} kind;
```

```
    union { TokenType op;
            int val;
            char * name; } attr;
    ExpType type; /* for type checking of exps */
} TreeNode;
```

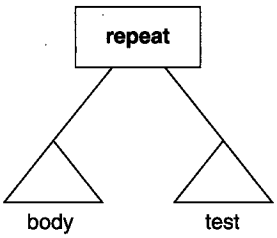
现在需要将语法树结构的描述用图形表示出来，并且画出示例程序的语法树。为了做到这一点，我们使用矩形框表示语句节点，用圆形框或椭圆形框表示表达式节点。语句或表达式的类型用框中的标记表示，额外的属性在括号中也列出来了。属性指针画在节点框的右边，而子指针则画在框的下面。我们还在图中用三角形表示额外的非指定的树结构，其中用点线表示可能出现也可能不出现的结构。语句序列由同属域连接（潜在的子树由点线和三角形表示）。则该图如下：



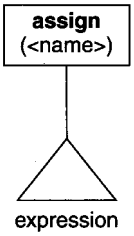
if 语句（带有3个可能的孩子）如下所示：



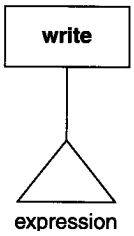
repeat 语句有两个孩子。第1个是表示循环体的语句序列，第2个是一个测试表达式：



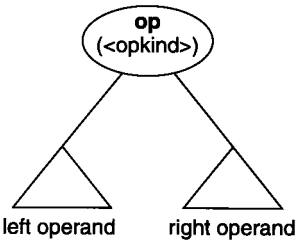
assign 语句有一个表示其值是被赋予的表达式的孩子（被赋予的变量名保存在语句节点中）：



write 语句也有一个孩子，它表示要写出值的表达式：



算符表达式有两个孩子，它们表示左操作数表达式和右操作数表达式：



其他所有的节点（read语句、标识符表达式和常量表达式）都是叶子节点。

最后准备显示一个TINY程序的树。程序清单3-3中有来自第1章计算一个整型阶乘的示例程序，它的语法树在图3-6中。

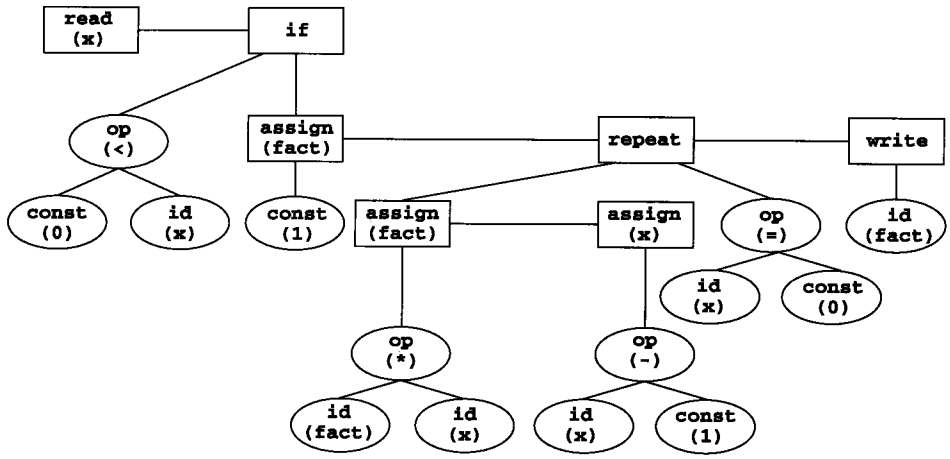


图3-6　程序清单3-3中TINY程序的语法树

程序清单3-3　TINY语言中的示例程序

```
{ Sample program
  in TINY language-
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
```

```
  until x = 0;
  write fact { output factorial of x }
end
```

练习

3.1 **a.** 写出一个生成串的集合

```
{s;, s;s;, s;s;s;, ... }
```

的非二义性的文法。

b. 利用你的文法为串`s;s;`给出一个最左推导和最右推导。

3.2 假设有文法$A \rightarrow AA \mid (A)\varepsilon$

a. 描述它生成的语言。

b. 说明它有二义性。

3.3 假设有文法

$$exp \rightarrow exp\ addop\ term \mid term$$
$$addop \rightarrow \texttt{+} \mid \texttt{-}$$
$$term \rightarrow term\ mulop\ factor \mid factor$$
$$mulop \rightarrow \texttt{*}$$
$$factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \textbf{\textit{number}}$$

则为下面的表达式写出最左推导、分析树以及抽象语法树：

```
a. 3+4*5-6          b. 3*(4-5+6)          c. 3-(4+5*6)
```

3.4 下面的文法生成字母表之上的所有正则表达式（以前曾在算符前后加上了引号，这是因为竖线既是一个算符又是一个元字符）：

$$rexp \rightarrow rexp\ \text{"|"}\ rexp$$
$$\mid rexp\ rexp$$
$$\mid rexp\ \text{"*"}$$
$$\mid \text{"(" } rexp \text{ ")"}$$
$$\mid \textbf{\textit{letter}}$$

a. 利用这个文法为正则表达式`(ab|b)*`给出一个推导。

b. 说明该文法有二义性。

c. 重写该文法以使算符建立正确的优先关系（参见第2章）。

d. `(`c`)` 的答案给二进制算符带来怎样的结合性？为什么？

3.5 为包括了常量`true`和`false`、算符`and`、`or`和`not`，以及括号的布尔表达式编写一个文法。确保给予`or`比`and`低的优先权，而`and`的优先权比`not`低，并允许`not`重复使用，如在布尔表达式中的`not not true`。另外还需保证该文法没有二义性。

3.6 考虑以下表示简化的类LISP表达式的文法：

$$lexp \rightarrow atom \mid list$$
$$atom \rightarrow \textbf{\textit{number}} \mid \textbf{\textit{identifier}}$$
$$list \rightarrow \texttt{(}\ lexp\text{-}seq\ \texttt{)}$$
$$lexp\text{-}seq \rightarrow lexp\text{-}seq\ lexp \mid lexp$$

a. 为串`(a 23 (m x y))`分别写出一个最左推导和一个最右推导。

b. 为**a**部分中的串画出一个分析树。

3.7 **a.** 为练习**3.6**中语法的抽象语法树结构编写一个C类型声明。

b. 为从**a**部分的声明得出的串**(a 23 (m x y))**画出一个语法树。

3.8 假设以下的文法

statement → *if-stmt* | ***other*** |ε
if-stmt → **if (*exp*)** *statement else-part*
else-part → **else** *statement* |ε
exp → **0** | **1**

a. 为串

if(0) if(1)*other* else else*other*

画出分析树。

b. 用两个**else**的目的是什么？

c. 在C中允许有这样的代码吗？请解释原因。

3.9 (Aho, Sethi andUllman) 显示以下的解决悬挂else二义性问题的尝试仍存在着二义性（与3.4.3节中的解法对比一下）：

statement → **if (*exp*)** *statement* | *matched-stmt*
matched-stmt → **if (*exp*)** *matched-stmt* **else** *statement* | ***other***
exp → **0** | **1**

3.10 **a.** 将练习**3.6**中的文法翻译成EBNF。

b. 为**a**部分中的EBNF画出语法图。

3.11 假设集合等式**X = (X + X) N**（**N**是自然数集：参见3.6.2节）。

a. 说明集合**E = N (N + N) (N + N + N) (N + N + N + N) ...** 满足等式。

b. 假设任意一个集合**E′** 满足等式，说明有**E**⊂**E′** 成立。

3.12 可以用多种方法将一目减添加到练习**3.3**的简单算术表达式文法中。修改BNF以使其符合每一个要求的规则。

a. 每个表达式至多允许有一个一目减，且一目减应在表达式的开头，则**-2-3**是合法的⊖（且答案应为-5），**-2-(-3)**也应是正规的，但**-2--3**却是非法的。

b. 在一个数或左括号之前至多有一个一目减，所以**-2--3**是合法的，但**--2**和**-2---3**却是非法的。

c. 在数字和左括号之前允许有任意数量的一目减，所以任何情况都是合法的。

3.13 考虑将练习**3.6**的文法如下简化：

lexp → ***number*** | **(** *op lexp-seq* **)**
op → **+** | **-** | *****
lexp-seq → *lexp-seq lexp* | *lexp*

这个文法可被认为是表示在类似于 LISP的前缀格式中的简单整型算术表达式。例如，表达式**34-3*42**按该文法就可写成**(-34 (*3 42))**。

a. 正规表达式**(- 2 3 4)**与**(- 2)**的解释是什么？表达式**(+ 2)**与**(* 2)**的解释是又什么呢？

⊖ 请注意这个表达式中的第2个减法是一个二目减，而不是一目减。

b. 在这个文法中，有没有优先权和结合性的问题？该文法有二义性吗？

3.14 **a.** 为上题中文法的语法树结构写出C类型的说明。

b. 利用**a**部分的答案为表达式`(- 34 (* 3 42))`画出语法树。

3.15 在3.3.2节中，一个抽象语法树

$$exp \rightarrow OpExp\ (op,exp,exp) \mid ConstExp(integer)$$
$$op \rightarrow Plus \mid Minus \mid Times$$

的类似于BNF的说明与一些语言中的真正类型说明很接近。ML和Haskell是其中的两种语言。这个练习是为那些了解这两种语言中任一种的读者编写的。

a. 写出完成上面抽象语法的数据类型说明。

b. 为表达式`(34* (42-3)`写出一个抽象语法表达式。

3.16 重写3.3.2节开始的语法树`typedef`以便使用一个`union`。

3.17 证明练习**3.5**中的文法生成了成对括号的所有串的集合，其中假设 w 具有以下两个属性，那么 w 就是一个成对括号的串：

a. w 确实包括了相同数量的左括号和右括号。

b. w 的每个前缀 u（对于某个 x，有 $w = ux$）至少有与右括号相同的左括号（提示：通过归纳一个推导的长度来证明）。

3.18 **a.** 写出一个生成格式 xcx 的串的文法，其中 x 是 a 和 b 的一个串。

b. 有没有可能为**a**部分的串写出一个上下文无关的文法？为什么？

3.19 在一些语言（例如Modula-2和Ada）中，希望一个过程说明用包括了过程名的语法结束，在Modula-2中，一个过程是这样说明的：

```
PROCEDURE P;
BEGIN
  ...
END P;
```

请注意在`END`之后的过程名`P`。分析程序能够检查这样的要求吗？为什么？

3.20 **a.** 写出一个生成与下面文法相同的语言的正则表达式：

$$A \rightarrow aA \mid B \mid \varepsilon$$
$$B \rightarrow bB \mid A$$

b. 写出一个生成与下面正则表达式相同的语言的文法：

```
(a|c|ba|bc)*(b|ε)
```

3.21 单元产生式（unit production）是公式 $A \rightarrow B$ 的一个文法规则选择，其中 A 和 B 均为非终结符。

a. 说明可系统地从文法中删除掉单元产生式，这样就产生了不带有单元产生式的文法，该单元产生式就生成了与原始文法相同的语言。

b. 你希望单元产生式在程序设计语言中经常出现吗？为什么？

3.22 循环文法（cyclic grammar）是在其中有一些非终结符 A 的一个推导 $A \Rightarrow^{+} A$ 的文法。

a. 说明循环文法是有二义性的。

b. 你希望定义程序设计语言的文法经常是循环的吗？请解释原因。

3.23 将练习**3.6**的TINY文法重写为在EBNF中的文法。

3.24 对于TINY程序

```
read x;
```

```
x:=x+1;
write x
```

a. 画出TINY分析树。

b. 画出TINY语法树。

3.25 为下面的程序画出TINY语法树：

```
read u;
read v; { input two integers }
if v=0 then v:=0 { do nothing }
else
  repeat
    temp:=v;
    v:=u-u/v*v;
    u:=temp
  until v=0
end;
write u {output gcd of original u & v}
```

注意与参考

上下文无关文法的许多理论都可在Hopcroft and Ullman [1979]中找到；在这里还可找到一些尚未解决的问题，例如先天二义性这样的许多属性；此外还有乔姆斯基层次。其他的内容则可在Ginsburg [1966, 1975]中找到。早期的一些理论在Chomsky [1956, 1959]中有提到，它为自然语言的学习提供了帮助。只有到后来对于程序语言的理解才成为了一个重要的论题，上下文无关文法的第1个应用是在Algol60的定义之中（Naur [1963]）。使用了上下文无关规则的有关3.4.3节中的悬挂else问题的解决方法是取自Aho、Hopcroft 和 Ullman [1986]的，练习**3.9**的答案也来自这儿。将上下文无关文法看作是递归等式（3.6.2节）是由指示语义得到的，读者可参看Schmidt [1986]。