第2章　一个简单的语法制导翻译器

本章的内容是对本书第3章至第6章中介绍的编译技术的总体介绍。通过开发一个可运行的Java程序来演示这些编译技术。这个程序可以将具有代表性的程序设计语言语句翻译为三地址代码(一种中间表示形式)。本章的重点是编译器的前端，特别是词法分析、语法分析和中间代码生成。在第7章和第8章将介绍如何根据三地址代码生成机器指令。

我们从小事做起，首先建立一个能够将中缀算术表达式转换为后缀表达式的语法制导翻译器。然后我们将扩展这个翻译器，使它能将某些程序片段(如图2-1所示)转换为如图2-2所示的三地址代码。

```
{
      int i; int j; float[100] a; float v; float x;
      while ( true ) {
            do i = i+1; while ( a[i] < v );
            do j = j-1; while ( a[j] > v );
            if ( i >= j ) break;
            x = a[i]; a[i] = a[j]; a[j] = x;
      }
}
```

图2-1　一个将被翻译的代码片段

这个可运行的Java程序见附录A。使用Java比较方便，但并非必须用Java。实际上，本章中描述的思想在Java和C语言出现之前就存在了。

```
 1:  i = i + 1
 2:  t1 = a [ i ]
 3:  if t1 < v goto 1
 4:  j = j - 1
 5:  t2 = a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
14:
```

图2-2　与图2-1中程序片段对应的经过简化的中间代码表示

2.1　引言

编译器在分析阶段把一个源程序划分成各个组成部分，并生成源程序的内部表示形式。这种内部表示称为中间代码。然后，编译器在合成阶段将这个中间代码翻译成目标程序。

分析阶段的工作是围绕着待编译语言的“语法”展开的。一个程序设计语言的*语法*(syntax)描述了该语言的程序的正确形式，而该语言的*语义*(semantics)则定义了程序的含义，即每个程序在运行时做什么事情。我们将在2.2节中给出一个广范使用的表示方法来描述语法，这个方法就是上下文无关文法或BNF(Backus-Naur范式)。使用现有的语义表示方法来描述一个语言的语义的难度远远大于描述语言的语法的难度。因此，我们将结合非形式化描述和启发性的示例来描述语言的语义。

上下文无关文法不仅可以描述一个语言的语法，还可以指导程序的翻译过程。在2.3节中，我们将介绍一种面向文法的编译技术，即*语法制导翻译*(syntax-directed translation)技术。语法扫描，或者说语法分析，将在2.4节中介绍。

本章的其余部分将快速浏览一下图 2-3 所示的编译器前端模型。我们将首先介绍语法分析器。为简单起见，我们首先考虑从中缀表达式到后缀表达式的语法制导翻译过程。后缀表达式是一种将运算符置于运算分量之后的表示方法。例如，表达式 9 - 5 + 2 的后缀形式是 95 - 2 +。将表达式翻译为后缀形式的过程可以充分演示语法分析技术，同时这个翻译过程又很简单，我们将在 2.5 节中给出这个翻译器的全部程序。这个简单的翻译器处理的表达式是由加、减号分隔的数位序列，如 9 - 5 + 2。我们之所以先考虑这样的简单表达式，主要目的是简化这个语法分析器，使得它在处理运算分量和运算符时只需要考虑单个字符。

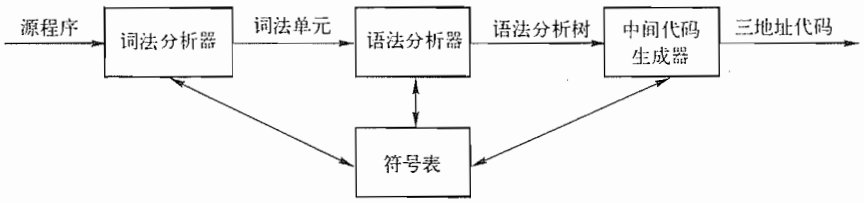


图 2-3 一个编译器前端的模型

词法分析器使得翻译器可以处理由多个字符组成的构造，比如标识符。标识符由多个字符组成，但是在语法分析阶段被当作一个单元进行处理。这样的单元称作*词法单元*(token)。例如，在表达式 *count* + 1 中，标识符 *count* 被当作一个单元。2.6 节中介绍的词法分析器允许表达式中出现数值、标识符和“空白字符”(空格、制表符和换行符)。

接下来我们考虑中间代码的生成。在图 2-4 中显示了两种中间代码形式。一种称为*抽象语法树*(abstract syntax tree)，或简称为*语法树*(syntax tree)。它表示了源程序的层次化语法结构。在图 2-3 的模型中，语法分析器生成一棵语法树，它又被进一步翻译为三地址代码。有些编译器会将语法分析和中间代码生成合并为一个组件。

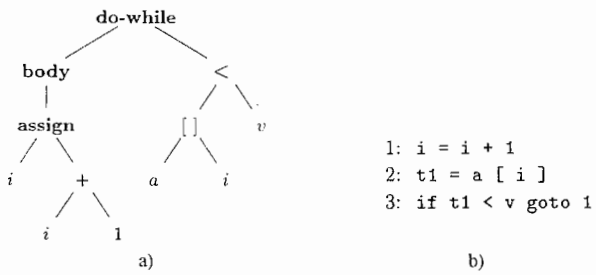


图 2-4 “do i = i + 1; while(a[i] < v);”的中间代码

图 2-4a 中的抽象语法树的根表示整个 do-while 循环。根的左子树表示循环的循环体，它仅包含赋值语句 $i = i + 1;$，根的右子树表示循环控制条件 $a[i] < v$。在 2.8 节中将介绍一个构造语法树的方法。

图 2-4b 中给出了另一种常见的中间表示形式，它是一组“三地址”指令序列，图 2-2 中显示了一个更加完整的示例。这个中间代码形式的名字源于它的指令形式：$x = y$ **op** z，其中 **op** 是一个二目运算符，y 和 z 是运算分量的地址，x 是运算结果的存放地址。三地址指令最多只执行一个运算，通常是计算、比较或者分支跳转运算。

在附录 A 中，我们将把本章中的技术集成在一起，构造出一个用 Java 语言编写的编译器前端。这个前端将语句翻译成汇编级的指令序列。

2.2 语法定义

在这一节，我们将介绍一种用于描述程序设计语言语法的表示方法——“上下文无关文法”，或简称“文法”。在本书中，文法将被用于组织编译器前端。

文法自然地描述了大多数程序设计语言构造的层次化语法结构。例如，Java 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** 和另一个语句连接而成。如果我们用变量 *expr* 来表示表达式，用变量 *stmt* 表示语句，那么这个构造规则可以表示为

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

其中的箭头(→)可以读作“可以具有如下形式”。这样的规则称为*产生式*(production)。在一个产生式中，像关键字 **if** 和括号这样的词法元素称为*终结符号*(terminal)。像 *expr* 和 *stmt* 这样的变量表示终结符号的序列，它们称为*非终结符号*(nonterminal)。

2.2.1 文法定义

一个*上下文无关文法*(context-free grammar)由四个元素组成：

1) 一个*终结符号*集合，它们有时也称为“词法单元”。终结符号是该文法所定义的语言的基本符号的集合。

2) 一个*非终结符号*集合，它们有时也称为“语法变量”。每个非终结符号表示一个终结符号串的集合。我们将在后面介绍这种表示方法。

3) 一个*产生式*集合，其中每个产生式包括一个称为*产生式头*或*左部*的非终结符号，一个箭头，和一个称为*产生式体*或*右部*的由终结符号及非终结符号组成的序列。产生式主要用来表示某个构造的某种书写形式。如果产生式头非终结符号代表一个构造，那么该产生式体就代表了该构造的一种书写方式。

4) 指定一个非终结符号为*开始符号*。

> **词法单元和终结符号**
>
> 在编译器中，词法分析器读入源程序中的字符序列，将它们组织为具有词法含义的词素，生成并输出代表这些词素的词法单元序列。词法单元由两个部分组成：名字和属性值。词法单元的名字是语法分析器进行语法分析时使用的抽象符号。我们常常把这些词法单元名字称为*终结符号*，因为它们在描述程序设计语言的文法中是以终结符号的形式出现的。如果词法单元具有属性值，那么这个值就是一个指向符号表的指针，符号表中包含了该词法单元的附加信息。这些附加信息不是文法的组成部分，因此在我们讨论语法分析时，通常将词法单元和终结符号当做同义词。

在描述文法的时候，我们会列出该文法的产生式，并且首先列出开始符号对应的产生式。我们假设数位、符号(如 < 、<=)和黑体字符串(如 **while**)都是终结符号。斜体字符串表示非终结符号，所有非斜体的名字或符号都可以看作是终结符号[⊖]。为表示方便，以同一个非终结符号为头部的多个产生式的体可以放在一起表示，不同体之间用符号 | (读作“或”)分隔。

例 2.1 在本章中，有多个例子使用由数位和 + 、- 符号组成的表达式，比如 9 - 5 + 2、3 - 1 或 7。由于两个数位之间必须出现 + 或 - ，我们把这样的表达式称为“由 + 、- 号分隔的数位序

⊖ 单个斜体字母在第 4 章中详细讨论文法时另有它用。例如，我们将使用 *X*、*Y* 和 *Z* 来表示终结符号或非终结符号。但是，包含两个或两个以上字符的任何斜体名字仍然表示一个非终结符号。

列”。下面的文法描述了这种表达式的语法。此文法的产生式包括：

$$list \rightarrow list + digit \tag{2.1}$$

$$list \rightarrow list - digit \tag{2.2}$$

$$list \rightarrow digit \tag{2.3}$$

$$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \tag{2.4}$$

以非终结符号 *list* 为头部的三个产生式可以等价地组合为：

$$list \rightarrow list + digit \mid list - digit \mid digit$$

根据我们的习惯，该文法的终结符号包括如下符号：

+ - 0 1 2 3 4 5 6 7 8 9

该文法的非终结符号是斜体名字 *list* 和 *digit*。因为 *list* 的产生式首先被列出，所以我们知道 *list* 是此文法的开始符号。□

如果某个非终结符号是某个产生式的头部，我们就说该产生式是该非终结符号的产生式。一个终结符号串是由零个或多个终结符号组成的序列。零个终结符号组成的串称为空串（empty string），记为 ϵ[⊖]。

2.2.2 推导

根据文法推导符号串时，我们首先从开始符号出发，不断将某个非终结符号替换为该非终结符号的某个产生式的体。可以从开始符号推导得到的所有终结符号串的集合称为该文法定义的语言（language）。

例 2.2 由例 2.1 中的文法定义的语言是由加减号分隔的数位列表的集合。非终结符号 *digit* 的 10 个产生式使得 *digit* 可以表示 0、1、…、9 中的任意数位。根据产生式(2.3)，单个数位本身就是一个 *list*。产生式(2.1)和(2.2)表达了如下规则：任何列表后跟一个符号 + 或 - 以及另一个数位可以构成一个新的列表。

产生式(2.1) ~ (2.4)就是我们定义所期望的语言时需要的全部产生式。例如，我们可以按照如下方法推导出 9 -5 +2 是一个 *list*。

1）因为 9 是 *digit*，根据产生式(2.3)可知 9 是 *list*。

2）因为 5 是 *digit*，且 9 是 *list*，由产生式(2.2)可知 9 -5 也是 *list*。

3）因为 2 是 *digit*，9 -5 是 *list*，由产生式(2.1)可知，9 -5 +2 也是 *list*。□

例 2.3 另一种稍有不同的列表是函数调用中的参数列表。在 Java 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`。这种列表的一个微妙之处是终结符号“(”和“)”之间的参数列表可能是空串。我们可以为这样的序列构造出具有如下产生式的文法：

$$\begin{aligned} call &\rightarrow \mathbf{id}\ (\ optparams\ ) \\ optparams &\rightarrow params \mid \epsilon \\ params &\rightarrow params\ ,\ param \mid param \end{aligned}$$

注意，在 *optparams*（“可选参数列表”）的产生式中，第二个可选规则体是 ϵ，它表示空的符号串。也就是说，*optparams* 可以被替换为空串，因此一个 call 可以是函数名加上两个终结符号“(”和“)”组成的符号串。请注意，*params* 的产生式和例 2.1 中 *list* 的产生式类似，只是将算术运算符 + 或 - 换成了逗号，并将 *digit* 换成 *param*。函数参数实际上可以是任意表达式，但是在这里

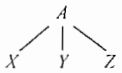
⊖ 从技术上讲，ϵ可以是任意字母表（符号的集合）上的零个符号组成的串。

我们没有给出 *param* 的产生式。稍后我们就会讨论用于描述不同的语言构造(比如表达式、语句等)的产生式。 □

语法分析(parsing)的任务是：接受一个终结符号串作为输入，找出从文法的开始符号推导出这个串的方法。如果不能从文法的开始符号推导得到该终结符号串，则报告该终结符号串中包含的语法错误。语法分析是所有编译过程中最基本的问题之一，主要的语法分析方法将在第 4 章中讨论。在本章中，为简单起见，我们首先处理像 9 - 5 + 2 这样的源程序，其中的每个字符均为一个终结符号。一般情况下，一个源程序中会包含由多字符组成的词素，这些词素由词法分析器组成词法单元，而词法单元的第一个分量就是被语法分析器处理的终结符号。

2.2.3 语法分析树

语法分析树用图形方式展现了从文法的开始符号推导出相应语言中的符号串的过程。如果非终结符号 A 有一个产生式 $A \rightarrow XYZ$，那么在语法分析树中就可能有一个标号为 A 的内部结点，该结点有三个子结点，从左向右的标号分别为 X、Y、Z：



正式地讲，给定一个上下文无关文法，该文法的一棵*语法分析树*(parse tree)是具有以下性质的树：

1) 根结点的标号为文法的开始符号。

2) 每个叶子结点的标号为一个终结符号或 ϵ。

3) 每个内部结点的标号为一个非终结符号。

4) 如果非终结符号 A 是某个内部结点的标号，并且它的子结点的标号从左至右分别为 X_1，X_2，…，X_n，那么必然存在产生式 $A \rightarrow X_1 X_2 \cdots X_n$，其中 X_1，X_2，…，X_n 既可以是终结符号，也可以是非终结符号。作为一个特殊情况，如果 $A \rightarrow \epsilon$ 是一个产生式，那么一个标号为 A 的结点可以只有一个标号为 ϵ 的子结点。

关于树型结构的术语

树型数据结构在编译系统中起着重要的作用。

- 一棵树由一个或多个结点(node)组成。结点可以带有标号(label)，在本书中标号通常是文法符号。当我们画一棵树时，我们常常只用这些标号来代表相应的结点。
- 树有且只有一个根(root)结点。每个非根结点都有唯一的父(parent)结点；根结点没有父结点。当我们画树的时候，将一个结点的父结点画在它的上方，并在父、子结点之间画一条边。因此根结点是最高的(顶层的)结点。
- 如果结点 N 是结点 M 的父结点，那么 M 就是 N 的子(child)结点。一个结点的各个子结点彼此称为*兄弟*(sibling)结点。它们之间是有序的，按照*从左向右*的方式排列。在我们画一棵树时也遵循这个顺序排列给定结点的子结点。
- 没有子结点的结点称为*叶子*(leaf)结点。其他结点，即有一个或多个子结点的结点，称为*内部结点*(interior node)。
- 结点 N 的*后代*(descendant)结点要么是结点 N 本身，要么是 N 的子结点，要么是 N 的子结点的子结点，依此类推(可以为任意层次)。如果结点 M 是结点 N 的后代结点，那么结点 N 是结点 M 的*祖先*(ancestor)结点。

例 2.4 例 2.2 中 9 - 5 + 2 的推导可以用图 2-5 中的树来演示。树中每个结点的标号都是一个

文法符号。每个内部结点和它的子结点都对应于一个产生式。其中，内部结点对应于产生式的头，它的子结点对应于产生式的体。

在图2-5中，根结点的标号为 *list*，即例2.1中文法的开始符号。根结点的子结点的标号从左向右分别为 *list*、+ 和 *digit*。请注意：

$$list \rightarrow list + digit$$

是例2.1中文法的产生式。根结点的左子结点和根结点类似，只是它的中间子结点的标号为 - 而不是 +。三个标号为 *digit* 的结点中，每个结点都有一个以具体数位为标号的子结点。 □

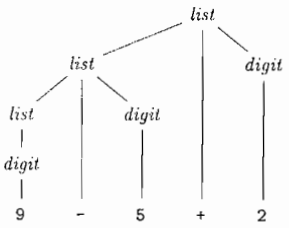


图2-5 根据例2.1中的文法得到的 9 -5 +2 的语法分析树

一棵语法分析树的叶子结点从左向右构成了树的结果(yield)，也就是从这棵语法分析树的根结点上的非终结符号推导得到(或者说生成)的符号串。在图2-5中的结果是 9 -5 +2。为了方便起见，我们将所有叶子结点都放在底层。以后我们不一定会把叶子结点按照这种方法排列。任何树的叶子结点都有一个自然的从左到右的顺序。这个顺序基于如下思想：如果 X 和 Y 是同一个父结点的子结点，并且 X 在 Y 的左边，那么 X 的所有后代结点都在 Y 的所有后代结点的左边。

一个文法的语言的另一个定义是指任何能够由某棵语法分析树生成的符号串的集合。为一个给定的终结符号串构建一棵语法分析树的过程称为对该符号串进行语法分析。

2.2.4 二义性

在根据一个文法讨论某个符号串的结构时，我们必须非常小心。一个文法可能有多棵语法分析树能够生成同一个给定的终结符号串。这样的文法称为具有二义性(ambiguous)。要证明一个文法具有二义性，我们只需要找到一个终结符号串，说明它是两棵以上语法分析树的结果。因为具有两棵以上语法分析树的符号串通常具有多个含义，所以我们需要为编译应用设计出没有二义性的文法，或者在使用二义性文法时使用附加的规则来消除二义性。

例2.5 假如我们使用一个非终结符号 *string*，并且不像例2.1中那样区分数位和列表，我们可以将例2.1中的文法改写如下：

$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

将符号 *digit* 和 *list* 合并为非终结符号 *string* 是有一些意义的，因为单个 *digit* 是 *list* 的一个特例。

但是，图2-6说明，在使用这个文法时，像 9 -5 +2这样的表达式会有多棵语法分析树。图中 9 -5 +2 的两棵语法分析树对应于两种带括号的表达式：(9 -5) +2 和 9 -(5 +2)。第二种方法给出的表达式值是意想不到的2，而不是通常的值6。例2.1的语法不支持这样的解释。 □

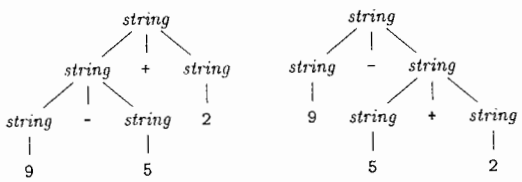


图2-6 9 -5 +2 的两棵语法分析树

2.2.5 运算符的结合性

依照惯例，9 +5 +2 等价于(9 +5) +2，9 -5 -2 等价于(9 -5) -2。当一个运算分量。(比如上式中的5)的左右两侧都有运算符时，我们需要一些规则来决定哪个运算符被应用于该运算分量。我们说运算符“ + ”是左结合(associate)的，因为当一个运算分量左右两侧都有“ + ”号时，它属于其左边的运算符。在大多数程序设计语言中，加、减、乘、除四种算术运算符都是

左结合的。

某些常用运算符是右结合的，比如指数运算。作为另一个例子，C 语言中的赋值运算符“=”及其后裔(即 +=、-= 等——译者注)也是右结合的。也就是说，对表达式 a=b=c 的处理和对表达式 a=(b=c)的处理相同。

带有右结合运算符的串，比如 a=b=c，可以由如下文法产生：

$$right \rightarrow letter = right \mid letter$$
$$letter \rightarrow a \mid b \mid \cdots \mid z$$

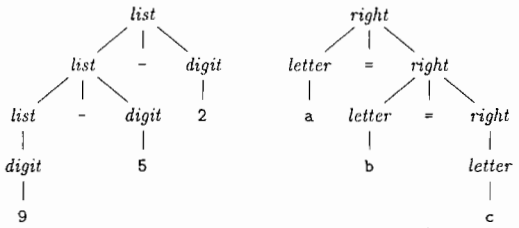


图 2-7 左结合运算符文法和右结合运算符文法的分析树

图 2-7 比较了一个左结合运算符(比如“-”)的语法分析树和一个右结合运算符(比如“=”)的语法分析树。注意，9-5-2 的语法分析树向左下端延伸，而 a=b=c 的语法分析树则向右下端延伸。

2.2.6 运算符的优先级

考虑表达式 9+5*2。该表达式有两种可能的解释，即(9+5)*2 或 9+(5*2)。+ 和 * 的结合性规则只能作用于同一运算符的多次出现，因此它们无法解决这个二义性。为此，当多种运算符出现时，我们需要给出一些规则来定义运算符之间的相对优先关系。

如果 * 先于 + 获得运算分量，我们就说 * 比 + 具有更高的优先级。在通常的算术中，乘法和除法比加法和减法具有更高的优先级。因此在表达式 9+5*2 和 9*5+2 中，都是运算分量 5 首先参与 * 运算，即这两个表达式分别等价于 9+(5*2)和(9*5)+2。

例 2.6 算术表达式的文法可以根据表示运算符结合性和优先级的表格来构建。我们首先考虑四个常用的算术运算符和一个优先级表。在此优先级表中，运算符按照优先级递增的顺序排列，同一行上的运算符具有相同的结合性和优先级：

左结合：+ -
左结合：* /

我们创建两个非终结符号 *expr* 和 *term*，分别对应于这两个优先级层次，并使用另一个非终结符号 *factor* 来生成表达式中的基本单元。当前，表达式的基本单元是数位和带括号的表达式。

$$\text{factor} \rightarrow \textbf{digit} \mid (expr)$$

现在我们考虑具有最高优先级的二目运算符 * 和 /。由于这些运算符是左结合的，因此其产生式和左结合列表的产生式类似：

$$\begin{aligned} term &\rightarrow term * factor \\ &\mid term / factor \\ &\mid factor \end{aligned}$$

类似地，*expr* 生成由加减运算符分隔的 *term* 列表：

$$\begin{aligned} expr &\rightarrow expr + term \\ &\mid expr - term \\ &\mid term \end{aligned}$$

因此最终得到的文法是：

$$\begin{aligned} expr &\rightarrow expr + term \mid expr - term \mid term \\ term &\rightarrow term * factor \mid term / factor \mid factor \\ factor &\rightarrow \textbf{digit} \mid (expr) \end{aligned}$$

例 2.6 中表达式文法的推广

我们可以将因子(factor)理解成不能被任何运算符分开的表达式。“不能分开”的意思是说当我们在任意因子的任意一边放置一个运算符，都不会导致这个因子的任何部分分离出来，成为这个运算符的运算分量。当然，因子本身作为一个整体可以成为该运算符的一个运算分量。如果这个因子是一个由括号括起来的表达式，那么括号将起到保护其不被分开的作用。如果因子就是一个运算分量，那么它当然不能被分开。

一个(不是因子的)项(term)是一个可能被高优先级的运算符 * 和/分开，但不能被低优先级运算符分开的表达式。一个(不是因子也不是项的)表达式可能被任何一个运算符分开。

我们可以把这种思想推广到具有任意 n 层优先级的情况。我们需要 $n+1$ 个非终结符号。首先，例 2.6 中描述的*factor* 不可被分开。通常，这个非终结符号的产生式体只能是单个运算分量或括号括起来的表达式。然后，对于每个优先级都有一个非终结符，表示能被该优先级或更高优先级的运算符分开的表达式。通常，这个非终结符的产生式有一些产生式体表示了该优先级的运算符的应用；另有一个产生式体只包含了代表更高一层优先级的非终结符号。

使用这个文法时，一个表达式就是一个由 + 或 - 分隔开的项(*term*)的列表，而项是由 * 或/分隔的因子(*factor*)的列表。请注意，任何由括号括起来的表达式都是一个因子。因此，我们可以使用括号来构造出具有任意嵌套深度的表达式(以及具有任意深度的语法分析树)。 □

例 2.7 由于大多数语句是由一个关键字或一个特殊字符开始的，因此关键字能够帮助我们识别语句。这一规则的例外情况包括赋值语句和过程调用语句。由图 2-8 中的(二义性)文法定义的语句都符合 Java 的语法。

在 *stmt* 的第一个产生式中，终结符号 **id** 表示任意标识符。非终结符号 *expression* 的产生式还没有给出。第一个产生式描述的赋值语句符合 Java 的语法，虽然 Java 将 = 号看作是可出现在表达式内部的赋值运算符。比如，在 Java 中允许出现 a = b = c，而这个文法不允许出现这样的形式。

```
stmt  →  id = expression ;
      |  if ( expression ) stmt
      |  if ( expression ) stmt else stmt
      |  while ( expression ) stmt
      |  do stmt while ( expression ) ;
      |  { stmts }

stmts →  stmts stmt
      |  ϵ
```

图 2-8 Java 语句的子集的文法

非终结符号 *stmts* 产生一个可能为空的语句列表。*stmts* 的第二个产生式生成一个空列表 ϵ。第一个产生式生成的是一个可能为空的列表再跟上一个语句。

分号的放置方式很微妙。它们出现在所有不以 *stmt* 结尾的产生式的末尾。这种方法可以避免在 if 或 while 这样的语句后面出现多余的分号，因为 if 和 while 语句的最后是一个嵌套的子语句。当嵌套子语句是一个赋值语句或 do-while 语句时，分号将作为这个子语句的一部分被生成。 □

2.2.7 2.2 节的练习

练习 2.2.1：考虑下面的上下文无关文法：

$$S \rightarrow SS + \mid SS * \mid \mathbf{a}$$

1) 试说明如何使用该文法生成串 aa + a * 。

2) 试为这个串构造一棵语法分析树。

3) 该文法生成的语言是什么？证明你的答案。

练习 2.2.2：下面的各个文法生成什么语言？证明你的每一个答案。

1) $S \rightarrow 0S1 \mid 01$

2) $S \rightarrow + \; S \; S \mid - \; S \; S \mid \mathbf{a}$

3) $S \rightarrow S \; (\; S \;) \; S \mid \epsilon$

4) $S \rightarrow \mathbf{a} \; S \; \mathbf{b} \; S \mid \mathbf{b} \; S \; \mathbf{a} \; S \mid \epsilon$

5) $S \rightarrow \mathbf{a} \mid S \; + \; S \mid S \; S \mid S \; * \mid (\; S \;)$

练习 2.2.3：练习 2.2.2 中哪些文法具有二义性？

练习 2.2.4：为下面的各个语言构建无二义性的上下文无关文法。证明你的文法都是正确的。

1) 用后缀方式表示的算术表达式。

2) 由逗号分隔开的左结合的标识符列表。

3) 由逗号分隔开的右结合的标识符列表。

4) 由整数、标识符、四个二目运算符 +、-、*、/构成的算术表达式。

! 5)在 4)的运算符中增加单目 + 和单目 - 构成的算术表达式。

练习 2.2.5：

1) 证明：用下面文法生成的所有二进制串的值都能被 3 整除。(提示：对语法分析树的结点数目使用数学归纳法。)

$$num \rightarrow 11 \mid 1001 \mid num \; 0 \mid num \; num$$

2) 上面的文法是否能够生成所有能被 3 整除的二进制串？

练习 2.2.6：为罗马数字构建一个上下文无关文法。

2.3 语法制导翻译

语法制导翻译是通过向一个文法的产生式附加一些规则或程序片段而得到的。比如，考虑由如下产生式生成的表达式 *expr*：

$$expr \rightarrow expr_1 \; + \; term$$

这里，*expr* 是两个子表达式 $expr_1$ 和 *term* 的和。($expr_1$ 中的下标仅仅被用于将产生式体中 *expr* 的实例和产生式头区别开来)。我们可以利用 *expr* 的结构，用如下的伪代码来翻译 *expr*：

```
翻译 expr1;
翻译 term;
处理 +;
```

我们将在 2.8 节中使用这段伪代码的一个变体，为 *expr* 构造一棵语法分析树：我们首先建立 $expr_1$ 和 *term* 的语法分析树，然后处理 + 运算符并构造得到一个和此运算符对应的结点。为方便起见，本节中的例子是从中缀表达式到后缀表达式的翻译。

本节介绍两个与语法制导翻译相关的概念：

- 属性(attribute)：属性表示与某个程序构造相关的任意的量。属性可以是多种多样的，比如表达式的数据类型、生成的代码中的指令数目或为某个构造生成的代码中第一条指令的位置等等都是属性的例子。因为我们用文法符号(终结符号或非终结符号)来表示程序构造，所以我们将属性的概念从程序构造扩展到表示这些构造的文法符号上。
- (语法制导的)翻译方案(translation scheme)：翻译方案是一种将程序片段附加到一个文法的各个产生式上的表示法。当在语法分析过程中使用一个产生式时，相应的程序片段就会执行。这些程序片段的执行效果按照语法分析过程的顺序组合起来，得到的结果就是这次分析/综合过程处理源程序得到的翻译结果。

语法制导的翻译方案将在本章中多次使用，它将用于把中缀表达式翻译成后缀表达式，还会用

于表达式求值，并用来构建一些程序构造的抽象语法树。第5章将更详细地讨论语法制导表示法。

2.3.1 后缀表示

本节中的例子处理的是中缀表达式到其后缀表示的翻译。一个表达式 E 的*后缀表示*(postfix notation)可以按照下面的方式进行归纳定义：

1）如果 E 是一个变量或常量，则 E 的后缀表示是 E 本身。

2）如果 E 是一个形如 E_1 **op** E_2 的表达式，其中 **op** 是一个二目运算符，那么 E 的后缀表示是 $E'_1E'_2$**op**，这里 E'_1 和 E'_2 分别是 E_1 和 E_2 的后缀表示。

3）如果 E 是一个形如 (E_1) 的被括号括起来的表达式，则 E 的后缀表示就是 E_1 的后缀表示。

例2.8 (9-5)+2 的后缀表示是 95-2+。也就是说，由规则1可知，9、5和2的翻译结果就是这些常量本身。然后，根据规则2，9-5 的翻译结果是 95-。由规则3可知，(9-5)的翻译结果与此相同。翻译完带括号的子表达式后，我们可以将规则2应用于整个表达式，(9-5)就是 E_1，2 为 E_2，由此得到结果 95-2+。

再举另外一个例子，9-(5+2)的后缀表达式是 952+-。也就是说，5+2 首先被翻译成 52+，然后这个表达式又成为减号的第二个运算分量。 □

运算符的位置和它的运算分量个数(arity)使得后缀表达式只有一种解码方式，所以在后缀表示中不需要括号。处理后缀表达式的“技巧”就是从左边开始不断扫描后缀串，直到发现一个运算符为止。然后向左找出适当数目的运算分量，并将这个运算符和它的运算分量组合在一起。计算出这个运算符作用于这些运算分量上后得到的结果，并用这个结果替换原来的运算分量和运算符。然后继续这个过程，向右搜寻另一个运算符。

例2.9 考虑后缀表达式 952+-3*。从左边开始扫描，我们首先遇到加号。向加号的左边看，我们找到运算分量 5 和 7。用它们的和 7 替换原来的 52+，这样我们得到串 97-3*。现在最左边的运算符是减号，它的运算分量是 9 和 7。将这些符号替换为它们的差，得到 23*。最后，将乘号应用在 2 和 3 上，得到结果 6。 □

2.3.2 综合属性

将量和程序构造关联起来(比如把数值及类型和表达式相关联)的想法可以基于文法来表示。我们将属性和文法的非终结符号及终结符号相关联。然后，我们给文法的各个产生式附加上语义规则。对于语法分析树中的一个结点，如果它和它的子结点之间的关系符合某个产生式，那么该产生式对应的规则就描述了如何计算这个结点上的属性。

语法制导定义(syntax-directed definition)把①每个文法符号和一个属性集合相关联，并且把②每个产生式和一组*语义规则*(semantic rule)相关联，这些规则用于计算与该产生式中符号相关联的属性值。

属性可以按照如下方式求值。对于一个给定的输入串 x，构建 x 的一个语法分析树。然后按照下面的方法应用语义规则来计算语法分析树中各个结点的属性。

假设语法分析树的一个结点 N 的标号为文法符号 X。我们用 $X.a$ 表示该结点上 X 的属性 a 的值。如果一棵语法分析树的各个结点上标记了相应的属性值，那么这棵语法分析树就称为*注释*(annotated)语法分析树(简称注释分析树)。比如，图2-9显示了 9-5+2 的一棵注释分析树，其中属性 t 与非终结符号 *expr* 和 *term* 关联。该属性在根结点处的值为 95-2+，也就是 9-5+2 的后缀表示。我们很快会看到这些表达式的计算方法。

如果某个属性在语法分析树结点 N 上的值是由 N 的子结点以及 N 本身的属性值确定的，那

么这个属性就称为综合属性(synthesized attribute)。综合属性具有一个很好的性质：只需要对语法分析树进行一次自底向上的遍历，就可以计算出属性的值。在 5.1.1 节中，我们将讨论另外一种重要的属性："继承"属性。非正式地讲，继承属性在某个语法分析树结点上的值是由语法分析树中该结点本身、父结点以及兄弟结点上的属性值决定的。

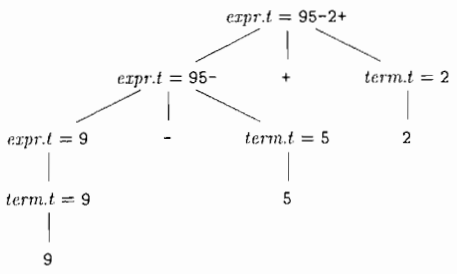


图 2-9 一个语法分析树的各个结点上的属性值

例 2.10 图 2-9 中的注释分析树是根据图 2-10 中的语法制导定义得到的。该语法制导定义用于把一个表达式翻译成为该表达式的后缀形式，待翻译的表达式是一个由加号和减号分隔的数位序列。图中每个非终结符号有一个值为字符串的属性 t，它表示由该非终结符号生成的表达式的后缀表示形式。语义规则中的符号 $\|$ 表示字符串的连接运算符。

产生式	语义规则
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \,\|\| \, term.t \,\|\| \, '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \,\|\| \, term.t \,\|\| \, '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

图 2-10 从中缀表示到后缀表示的翻译的语法制导定义

一个数位的后缀形式是该数位本身。例如，与产生式 $term \rightarrow 9$ 相关联的语义规则定义如下：当该产生式被应用在语法分析树的某个结点上时，$term.t$ 的值就是 9 本身。其他数位也按照类似的方法进行翻译。再如，当产生式 $expr \rightarrow term$ 被应用时，$term.t$ 的值成为 $expr.t$ 的值。

产生式 $expr \rightarrow expr_1 + term$ 推导出一个带有加号的表达式[⊖]。加法运算符的左运算分量由 $expr_1$ 给出，右运算分量由 $term$ 给出。与这个产生式关联的语义规则

$$expr.t = expr_1.t \,\|\, term.t \,\|\, '+'$$

定义了计算属性 $expr.t$ 的值的方式，它将分别代表左右运算分量后缀表示形式的 $expr_1.t$ 和 $term.t$ 连接起来，再在后面加上加号，就得到了属性 $expr.t$ 的值。这个规则是后缀表达式定义的一个公式化表示。 □

区分一个非终结符号的不同使用的规则

在规则中，我们经常要区分同一个非终结符号在一个产生式的头和/或体中的多次使用，在例 2.10 中就有这样的情况。原因是在语法分析树中，标号为同一个非终结符号的不同结点通常在翻译中具有不同的属性值。我们将采用下面的规则：出现在产生式头中的非终结符号没有下标，而在产生式体中的非终结符号带有不同的下标。同一个非终结符号的所有出现都按照这种方式区分，并且下标不是名字的组成部分。然而，读者应该注意使用了这种下标约定的特定翻译规则和 $A \rightarrow X_1 X_2 \cdots X_n$ 这样表示一般形式的产生式的区别。在后者中，带下标的 X 表示任意文法符号的列表，而不是某个名为 X 的非终结符号的不同实例。

⊖ 在这个规则以及很多其他的规则中，同一个非终结符号(这里是 $expr$)会在一个产生式中出现多次。$expr_1$ 中的下标 1 用于区分产生式中 $expr$ 的两次出现，但"1"并不是该非终结符号的一部分。在下面的"区分一个非终结符号的不同使用的约定"中有更加详细的描述。

2.3.3 简单语法制导定义

例 2.10 中的语法制导定义具有下面的重要性质：要得到代表产生式头部的非终结符号的翻译结果的字符串，只需要将产生式体中各非终结符号的翻译结果按照它们在非终结符号中的出现顺序连接起来，并在其中穿插一些附加的串即可。具有这个性质的语法制导定义称为简单(simple)语法制导定义。

例 2.11 考虑图 2-10 中的第一个产生式和语义规则：

产生式	语义规则	
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \parallel term.t \parallel \text{'+'}$	(2.5)

这里，翻译结果 $expr.t$ 是 $expr_1$ 和 $term$ 的翻译结果的连接，再跟一个加号。请注意，$expr_1$ 和 $term$ 在产生式体中和语义规则中的出现顺序是相同的。在它们的翻译结果之前和之间没有其他符号。在这个例子中，唯一的附加符号出现在结尾处。 □

当讨论翻译方案的时候，我们将看到，一个简单语法制导定义的实现很简单，只需要按照它们在定义中出现的顺序打印出附加的串即可。

2.3.4 树的遍历

树的遍历将用于描述属性的求值过程，以及描述一个翻译方案中的各个代码片段的执行过程。一个树的*遍历*(traversal)从根结点开始，并按照某个顺序访问树的各个结点。

一次深度优先(depth-first)遍历从根结点开始，递归地按照任意顺序访问各个结点的子结点，并不一定要按照从左向右的顺序遍历。之所以称之为深度优先，是因为这种遍历总是尽可能地访问一个结点的尚未被访问的子结点，因此它总是尽可能快地访问离根结点最远的结点(即最深的结点)。

图 2-11 中的过程 *visit*(N)就是一个深度优先遍历，它按照从左向右的顺序访问一个结点的子结点，如图 2-12 所示。在这个遍历中，完成某个结点的遍历之前(也就是在该结点的各个子结点的翻译结果都计算完毕之后)，我们加入了计算每个结点的翻译结果的动作。一般来说，我们可以任意选定和一次遍历过程相关联的动作，当然也可以选择什么都不做。

```
procedure visit(node N) {
    for (从左到右遍历N的每个子结点C) {
        visit(C);
    }
    按照结点N上的语义规则求值;
}
```

图 2-11 一棵树的深度优先遍历

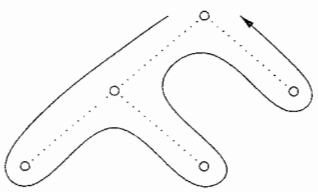


图 2-12 一棵树的深度优先遍历的例子

语法制导定义没有规定一棵语法分析树中各个属性值的求值顺序。只要一个顺序能够保证计算属性 a 的值时，a 所依赖的其他属性都已经计算完毕，这个顺序就是可以接受的。综合属性可以在自底向上遍历的时候计算。自顶向上遍历指在计算完成某个结点的所有子结点的属性值之后才计算该结点的属性值的过程。一般来说，当既有综合属性又有继承属性时，关于求值顺序的问题就变得相当复杂，参见 5.2 节。

2.3.5 翻译方案

图 2-10 中的语法制导定义将字符串作为属性值附加在语法分析树的结点上，从而得到翻译结果。我们现在来考虑另外一种不需要操作字符串的方法。它通过运行程序片段，逐步生成相同的翻译结果。

前序遍历和后序遍历

前序遍历和后序遍历是深度优先遍历的两种重要的特例。在这两种遍历中，我们都是从左到右递归地访问每个结点的子结点。

我们经常遍历一棵树，并在各个结点上执行某些特定的动作。如果动作在我们第一次访问一个结点时被执行，那么我们将这种遍历称为*前序遍历*（preorder traversal）。类似地，如果动作在我们最后离开一个结点前被执行，则称这种遍历为*后序遍历*（postorder traversal）。图 2-11 中的过程 *visit*(*N*) 就是一个后序遍历的例子。

前序遍历和后序遍历根据一个结点的动作执行时间来定义这些结点的相应次序。一棵以结点 *N* 为根的（子）树的前序排序由 *N*，跟上它的从左到右的每棵子树（如果存在）的前序排序组成。而一棵以结点 *N* 为根的（子）树的后序排序则由 *N* 的从左到右的每棵子树的后序排序，再跟上 *N* 自身组成。

语法制导翻译方案是一种在文法产生式中附加一些程序片段来描述翻译结果的表示方法。语法制导翻译方案和语法制导定义相似，只是显式指定了语义规则的计算顺序。

被嵌入到产生式体中的程序片段称为*语义动作*（semantic action）。一个语义动作用花括号括起来，并写入产生式的体中，它的执行位置也由此指定，如下面的规则所示：

$$rest \rightarrow + \ term \ \{\text{print}('+')\} \ rest_1$$

当我们考虑表达式的另一种形式的文法时，我们就会看到这样的规则，其中非终结符号 *rest* 代表“一个表达式中除第一个项之外的一切”。这种形式的文法将在 2.4.5 节中讨论。此外，$rest_1$ 中的下标将非终结符号 *rest* 在产生式体中的实例与产生式头部的 *rest* 实例区分开来。

当我们画出一个翻译方案的语法分析树时，我们为每个语义动作构造一个额外的子结点，并使用虚线将它和该产生式头部对应的结点相连。例如，表示上述产生式和语义动作的部分语法分析树如图 2-13 所示。对应于语义动作的结点没有子结点，因此在第一次访问该结点时就会执行这个动作。

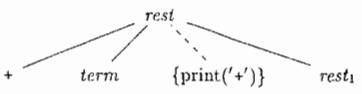


图 2-13 为一个语义动作创建一个额外的叶子结点

例 2.12 图 2-14 的语法分析树在额外的叶子结点中含有打印语句。这些叶子结点通过虚线与语法分析树的内部结点相连接。它的翻译方案如图 2-15 所示。该翻译方案的基础文法生成了由符号 + 和 - 分隔的数位序列组成的表达式。假设我们对整棵树进行从左到右的深度优先遍历，并在我们访问它的叶子结点时执行每个打印语句，那么产生式体中内嵌的语义动作将把这样的表达式翻译为相应的后缀表示形式。

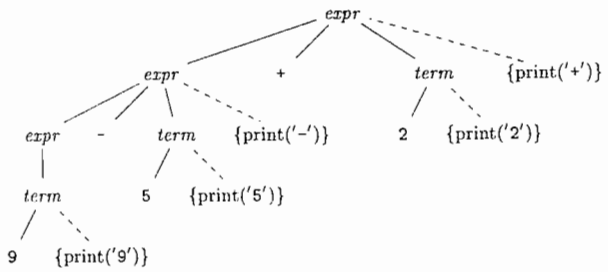


图 2-14 把 9 - 5 + 2 翻译成 95 - 2 + 的语义动作

图 2-14 的根结点代表图 2-15 中的第一个产生式。这个根结点的最左边的子树代表左边的运算分量，它的标号和根结点一样都是 *expr*。在一次后序遍历中，我们首先执行该子树中的所有语

义动作。然后我们访问没有语义动作的叶子结点 +。接下来，我们执行代表右运算分量 *term* 的子树中的所有语义动作。最后执行额外结点上的语义动作 {print('+')}。

由于 *term* 的产生式的右部只有一个数位，该产生式的语义动作把这个数位打印出来。产生式 *expr*→*term* 不需要产生输出，只有前面两个产生式的语义动作中的运算符才会打印出来。图 2-14 中的语义动作在对语法分析树的后序遍历中执行时会打印出 95 - 2 +。 □

expr	→	$expr_1$ + *term*	{print('+')}
expr	→	$expr_1$ - *term*	{print('-')}
expr	→	*term*	
term	→	0	{print('0')}
term	→	1	{print('1')}
		...	
term	→	9	{print('9')}

图 2-15 把表达式翻译成后缀形式的语义动作

注意，尽管图 2-10 和图 2-15 中的翻译方案产生相同的翻译结果，但它们构造结果的过程是不同的。图 2-10 是把字符串作为属性附加到语法分析树中的结点上，而图 2-15 通过语义动作把翻译结果以增量方式打印出来。

如图 2-14 所示的语法分析树中的语义动作将中缀表达式 9 - 5 + 2 翻译成 95 - 2 +，它恰好将 9 - 5 + 2 中的每个字符各打印一次。它不需要任何附加空间来存放子表达式的翻译结果。当按照这种方式递增地创建输出时，字符的打印顺序非常重要。

实现一个翻译方案时，必须保证各个语义动作按照它们在语法分析树的后序遍历中的顺序执行。这个实现不一定要真的构造出一棵语法分析树（通常也不会这么做），只要能够确保语义动作的执行过程等同于我们真的构建了语法分析树并在后序遍历中执行这些动作时的情形。

2.3.6 2.3 节的练习

练习 2.3.1：构建一个语法制导翻译方案，该方案把算术表达式从中缀表示方式翻译成运算符在运算分量之前的前缀表示方式。例如，$-xy$ 是表达式 $x-y$ 的前缀表示法。给出输入 9 - 5 + 2 和 9 - 5 * 2 的注释分析树。

练习 2.3.2：构建一个语法制导翻译方案，该方案将算术表达式从后缀表示方式翻译成中缀表示方式。给出输入 95 - 2 * 和 952 * - 的注释分析树。

练习 2.3.3：构建一个将整数翻译成罗马数字的语法制导翻译方案。

练习 2.3.4：构建一个将罗马数字翻译成整数的语法制导翻译方案。

练习 2.3.5：构建一个将后缀算术表达式翻译成等价的前缀算术表达式的语法制导翻译方案。

2.4 语法分析

语法分析是决定如何使用一个文法生成一个终结符号串的过程。在讨论这个问题时，我们可以想象我们正在构建一个语法分析树，这样可以帮助我们理解分析的过程，尽管在实践中编译器并没有真的构造出这棵树。然而，原则上语法分析器必须能够构造出语法分析树，否则将无法保证翻译的正确性。

本节将介绍一种称为“递归下降”的语法分析方法，该方法可以用于语法分析和实现语法制导翻译器。下一节将给出一个实现了图 2-15 中的翻译方案的完整 Java 程序。另一种可行的方法是使用软件工具直接根据翻译方案生成一个翻译器。4.9 节将描述一个这样的工具——Yacc。使用这个工具，无需修改就可以实现图 2-15 中的翻译方案。

对于任何上下文无关文法，我们都可以构造出一个时间复杂度为 $O(n^3)$ 的语法分析器，它最多使用 $O(n^3)$ 的时间就可以完成一个长度为 n 的符号串的语法分析。但是，三次方的时间代价一般来说太昂贵了。幸运的是，对于实际的程序设计语言而言，我们通常能够设计出一个可以被高效分析的文法。线性时间复杂度的算法足以分析实践中出现的各种程序设计语言。程序设计

语言的语法分析器几乎总是一次性地从左到右扫描输入，每次向前看一个终结符号，并在扫描时构造出分析树的各个部分。

大多数语法分析方法都可以归入以下两类：自顶向下(top-down)方法和自底向上(bottom-up)方法。这两个术语指的是语法分析树结点的构造顺序。在自顶向下语法分析器中，构造过程从根结点开始，逐步向叶子结点方向进行；而在自底向上语法分析器中，构造过程从叶子结点开始，逐步构造出根结点。自顶向下语法分析器之所以受欢迎，是因为使用这种方法可以较容易地手工构造出高效的语法分析器。不过，自底向上分析方法可以处理更多种文法和翻译方案，所以直接从文法生成语法分析器的软件工具常常使用自底向上的方法。

2.4.1 自顶向下分析方法

我们在介绍自顶向下的分析方法时考虑的文法适合使用自顶向下分析技术。在本节后面的内容中，我们将考虑构造自顶向下语法分析器的一般方法。图 2-16 中的文法生成 C 或 Java 语句的一个子集。我们分别用黑体终结符 **if** 和 **for** 表示关键字“`if`”和“`for`”，以强调这些字符序列被视为一个单元，也就是单个终结符号。此外，终结符 **expr** 代表表达式。一个更完整的文法将使用非终结符号 *expr*，并带有多个关于非终结符号 *expr* 的产生式。类似地，**other** 是一个代表其他语句构造的终结符号。

stmt	→	**expr** ;
	|	**if** (**expr**) *stmt*
	|	**for** (*optexpr* ; *optexpr* ; *optexpr*) *stmt*
	|	**other**
optexpr	→	ϵ
	|	**expr**

图 2-16 C 和 Java 中某些语句的文法

在自顶向下地构造一棵如图 2-17 所示的语法分析树时，从标号为开始非终结符 *stmt* 的根结点开始，反复执行下面两个步骤：

1）在标号为非终结符号 A 的结点 N 上，选择 A 的一个产生式，并为该产生式体中的各个符号构造出 N 的子结点。

2）寻找下一个结点来构造子树，通常选择的是语法分析树最左边的尚未扩展的非终结符。

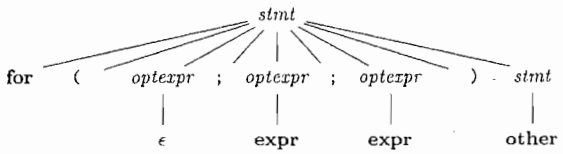


图 2-17 根据图 2-16 中的文法得到的语法分析树

对于某些文法，上面的步骤只需要对输入串进行一次从左到右的扫描就可以完成。输入中当前被扫描的终结符号通常称为向前看(lookahead)符号。在开始时，向前看符号是输入串的第一个(即最左的)终结符号。图 2-18 演示了构造如下输入串的语法分析树的过程：

for (; expr ; expr) other

得到的语法分析树如图 2-17 所示。一开始，向前看符号是终结符号 **for**，语法分析树的已知部分只包含标号为开始非终结符号 *stmt* 的根结点，如图 2-18a 所示。我们的目标是以适当的方法构造出语法分析树的其余部分，使得这棵树生成的符号串与输入符号串匹配。

为了与输入串匹配，图 2-18a 中的非终结符号 *stmt* 必须推导出一个以向前看符号 **for** 开头的串。在图 2-16 所示的文法中，*stmt* 只有一个产生式可以推导出这样的串，所以我们选择这个产生式，并构造出根结点的各个子结点，并使用该产生式体中的符号作为这些子结点的标号。这棵语法分析树的这次扩展如图 2-18b 所示。

在图 2-18 所示的三个快照中，都包含一个指向输入串中向前看符号的箭头和一个指向当前正被考虑的语法分析树结点的箭头。一旦一个结点的子结点全部构造完毕，我们就要考虑该结点的最左子结点。在图 2-18b 中，根结点的子结点刚刚构造完毕，下一个要考虑的结点是标号为 **for** 的最左子结点。

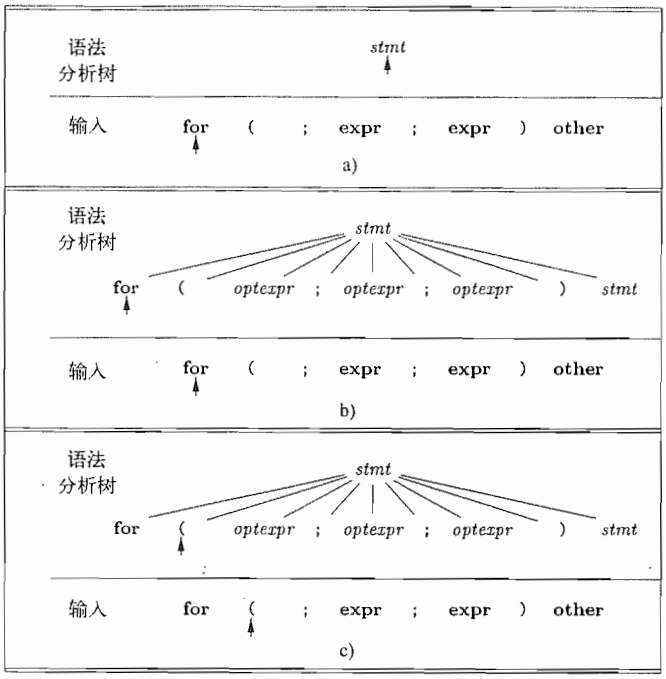


图 2-18 从左到右扫描输入串时进行的自顶向下语法分析

如果当前正考虑的语法分析树结点的标号是一个终结符号，而且此终结符号与向前看符号匹配，那么语法分析树的箭头和输入的箭头都前进一步。输入中的下一个终结符成为新的向前看符号，同时考虑语法分析树的下一个子结点。在图 2-18c 中，语法分析树的箭头指向根的下一个子结点，输入中的箭头已经前进到下一个终结符号，即“(”。再下一步将使得语法分析树的箭头指向标号为非终结符号 *optexpr* 的子结点，并将输入的箭头指向终结符号“;”。

在标号为 *optexpr* 的非终结符号结点上，我们需要再次为一个非终结符号选择产生式。以 ϵ 为体的产生式(即 ϵ 产生式)需要特殊处理。当前，我们将 ϵ 产生式当作默认选择，只有在没有其他产生式可用时才会选择它们。我们将在 2.4.3 节中再次讨论 ϵ 产生式。对于非终结符号 *optexpr* 和向前看符号“;”，我们使用 *optexpr* 的 ϵ 产生式，因为“;”和 *optexpr* 仅有的另一个产生式不匹配，那个产生式的体是终结符号 **expr**。

一般来说，为一个非终结符号选择产生式是一个“尝试并犯错”的过程。也就是说，我们首先选择一个产生式，并在这个产生式不合适时进行回溯，再尝试另一个产生式。一个产生式“不合适”是指使用了该产生式之后，我们无法构造得到一棵与当前输入串相匹配的语法分析树。但是在称为预测语法分析的特殊情形下不需要进行回溯。我们接下来将讨论这个方法。

2.4.2 预测分析法

递归下降分析方法(recursive-descent parsing)是一种自顶向下的语法分析方法，它使用一组递归过程来处理输入。文法的每个非终结符都有一个相关联的过程。这里我们考虑递归下降分析法的一种简单形式，称为*预测分析法*(predictive parsing)。在预测分析法中，各个非终结符号对应的过程中的控制流可以由向前看符号无二义地确定。在分析输入串时出现的过程调用序列隐式地定义了该输入串的一棵语法分析树。如果需要，还可以通过这些过程调用来构建一个显式的语法分析树。

图 2-19 的预测分析器包含了两个过程 *stmt*( )和 *optexpr*( )，分别对应于图 2-16 中文法的非终结符号 *stmt* 和*optexpr*。该分析器还包括一个额外的过程 *match*。这个额外过程用来简化 *stmt* 和 *optexpr*的代码。过程 *match*(*t*)将它的参数 *t* 和向前看符号比较，如果匹配就前进到下一个输入终结符号。因此，*match* 改变了全局变量 *lookhead* 的值，该变量存储了当前正被扫描的输入终结符号。

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other:
                match(other); break;
        default:
                report("syntax error");
        }
}

void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

图 2-19 一个预测分析器的伪代码

分析过程开始时，首先调用文法的开始非终结符号 *stmt* 对应的过程。在处理如图 2-18 所示的输入时，*lookhead* 被初始化为第一个终结符号 **for**。过程 *stmt* 执行和如下产生式对应的代码：

$$stmt \rightarrow \textbf{for}\ (\ optexpr\ ;\ optexpr\ ;\ optexpr\ )\ stmt$$

在对应于该产生式体的代码中——即图 2-19 的过程 *stmt* 中处理 **for** 语句的 case 分支——每个终结符都和向前看符号匹配，而每个非终结符都产生一个对相应过程的调用：

match(**for**)；*match*('(')；

optexpr( )；*match*(';')；*optexpr*( )；*match*(';')；*optexpr*( )；

match(')')；*stmt*( )；

预测分析需要知道哪些符号可能成为一个产生式体所生成串的第一个符号。更精确地说，令 α 是一个文法符号(终结符号或非终结符号)串。我们将 FIRST(α) 定义为可以由 α 生成的一个或多个终结符号串的第一个符号的集合。如果 α 就是 ϵ 或者可以生成 ϵ，那么 ϵ 也在 FIRST(α)中。

关于计算 FIRST(α)的算法的详细描述将在 4.4.2 节中给出。这里，我们将使用不具一般性的推导方法来求出 FIRST(α)中的符号。通常情况下，α 要么以一个终结符号开头，此时该终结符号就是 FIRST(α)中的唯一符号；要么 α 以一个非终结符号开头，且该非终结符的所有产生式体都以某个终结符号开头，那么这些终结符号就是 FIRST(α)的所有成员。

例如，对于图 2-16 中的文法，其 FIRST 的正确计算如下：

FIRST(*stmt*) = {**expr**, **if**, **for**, **other**}

FIRST(**expr** ;) = {**expr**}

如果有两个产生式 $A \rightarrow \alpha$ 和 $A \rightarrow \beta$，我们就必须考虑相应的 FIRST 集合。如果我们不考虑 ϵ 产生式，预测分析法要求 FIRST(α) 和 FIRST(β) 不相交，那么就可以用向前看符号来确定应该使用哪个产生式。如果向前看符号在 FIRST(α) 中，就使用 α。如果向前看符号在 FIRST(β) 中，就使用 β。

2.4.3 何时使用 ϵ 产生式

我们的预测分析器在没有其他产生式可用时，将 ϵ 产生式作为默认选择使用。处理图 2-18 所示的输入时，在终结符号 **for** 和“(”匹配之后，向前看符号为“;”。此时，过程 *optexpr* 被调用，其过程体中的代码：

```
if ( lookahead == expr ) match(expr);
```

被执行。非终结符号 *optexpr* 有两个产生式，它们的体分别是 **expr** 和 ϵ。向前看符号“;”与终结符号 **expr** 不匹配，因此不能使用以 **expr** 为体的产生式。事实上，该过程没有改变向前看符号，也没有做任何其他操作就返回了。不做任何操作就对应于应用 ϵ 产生式的情形。

对于更加一般化的情况，我们考虑图 2-16 中产生式的一个变体，其中 *optexpr* 生成一个表达式非终结符号，而不是终结符号 **expr**：

$$optexpr \rightarrow expr$$
$$\mid \ \epsilon$$

这样，*optexpr* 要么使用非终结符号 *expr* 生成一个表达式，要么生成 ϵ。在对 *optexpr* 进行语法分析时，如果向前看符号不在 FIRST(*expr*) 中，我们就使用 ϵ 产生式。

要更加深入地了解应该在何时使用 ϵ 产生式，请参见 4.4.3 节中关于 LL(1) 文法的讨论。

2.4.4 设计一个预测分析器

我们可以将 2.4.2 节中非正式介绍的技术推广应用到任意具有如下性质的文法上：对于文法的任何非终结符号，它的各个产生式体的 FIRST 集合互不相交。我们还将看到，如果我们有一个翻译方案，即一个增加了语义动作的文法，那么我们可以将这些语义动作当作此语法分析器的过程的一部分执行。

回顾一下，一个预测分析器(predictive parser)程序由各个非终结符对应的过程组成。对应于非终结符 A 的过程完成以下两项任务：

1）检查向前看符号，决定使用 A 的哪个产生式。如果一个产生式的体为 α(这里 α 不是空串 ϵ)且向前看符号在 FIRST(α) 中，那么就选择这个产生式。对于任何向前看符号，如果两个非空的产生式体之间存在冲突，我们就不能对这种文法使用预测语法分析。另外，如果 A 有 ϵ 产生式，那么只有当向前看符号不在 A 的其他产生式体的 FIRST 集合中时，才会使用 A 的 ϵ 产生式。

2）然后，这个过程模拟被选中产生式的体。也就是说，从左边开始逐个“执行”此产生式体中的符号。“执行”一个非终结符号的方法是调用该非终结符号对应的过程，一个与向前看符号匹配的终结符号的“执行”方法则是读入下一个输入符号。如果在某个点上，产生式体中的终结符号和向前看符号不匹配，那么语法分析器就会报告一个语法错误。

图 2-19 显示的是对图 2-16 的文法应用这些规则的结果。

就像通过扩展文法来得到一个翻译方案一样，我们也可以扩展一个预测分析器来获得一个语法制导的翻译器。在 5.4 节中将给出一个能够达到此目的的算法。下面的部分构造方法已经可以满足当前的要求：

1）先不考虑产生式中的动作，构造一个预测分析器。

2）将翻译方案中的动作拷贝到语法分析器中。如果一个动作出现在产生式 p 中的文法符号

X 的后面，则该动作就被拷贝到 p 的代码中 X 的实现之后。否则，如果该动作出现在一个产生式的开头，那么它就被拷贝到该产生式体的实现代码之前。

我们将在 2.5 节构造这样一个翻译器。

2.4.5 左递归

递归下降语法分析器有可能进入无限循环。当出现如下所示的“左递归”产生式时，分析器就会出现无限循环：

$$expr \rightarrow expr + term$$

在这里，产生式体的最左边的符号和产生式头部的非终结符相同。假设 *expr* 对应的过程决定使用这个产生式。因为产生式体的开头是 *expr*，所以 *expr* 对应的过程将被递归调用。由于只有当产生式体中的一个终结符号被成功匹配时，向前看符号才会发生改变，因此在对 *expr* 的两次调用之间输入符号没有发生改变。结果，第二次 *expr* 调用所做的事情与第一次调用所做的完全相同，这意味着会对 *expr* 进行第三次调用，并不断重复，进入无限循环。

通过改写有问题的产生式就可以消除左递归。考虑有两个产生式：

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

的非终结符号 A，其中 α 和 β 是不以 A 开头的终结符号/非终结符号的序列。例如，在产生式

$$expr \rightarrow expr + term \mid term$$

中，非终结符号 $A = expr$，串 $\alpha = + term$，串 $\beta = term$。

因为产生式 $A \rightarrow A\alpha$ 的右部的最左符号是 A 自身，非终结符号 A 和它的产生式就称为*左递归的*(left recursive)[⊖]。不断应用这个产生式将在 A 的右边生成一个 α 的序列，如图 2-20a 所示。当 A 最终被替换为 β 时，我们就得到了一个在 β 后跟有 0 个或多个 α 的序列。

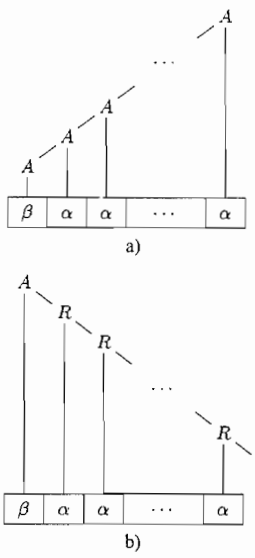


图 2-20 生成一个串的左递归方式和右递归方式

如图 2-20b 所示，使用一个新的非终结符号 R，并按照如下方式改写 A 的产生式可以达到同样的效果：

$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R \mid \epsilon$$

非终结符号 R 和它的产生式 $R \rightarrow \alpha R$ 是*右递归的*(right recursive)，因为这个产生式的右部的最后一个符号就是 R 本身。如图 2-20b 所示，右递归的产生式会使得树向右下方向生长。因为树是向右下生长的，对包含了左结合运算符(比如减法)的表达式的翻译就变得较为困难。然而，我们将在 2.5.2 节看到，通过仔细设计翻译方案，我们仍然可以将一个表达式正确地翻译成后缀表达式。

在 4.3.3 节，我们将考虑更一般的左递归形式，并说明如何从文法中消除左递归。

2.4.6 2.4 节的练习

练习 2.4.1：为下列文法构造递归下降语法分析器：

1) $S \rightarrow + SS \mid - SS \mid a$

2) $S \rightarrow S(S)S \mid \epsilon$

3) $S \rightarrow 0S1 \mid 01$

⊖ 在一般的左递归文法中，非终结符号 A 可能通过一些中间产生式推导出 $A\alpha$，而不一定存在产生式 $A \rightarrow A\alpha$。

2.5 简单表达式的翻译器

使用前面三节介绍的技术，现在我们可以用 Java 语言编写一个语法制导翻译器。这个翻译器可以把算术表达式翻译成等价的后缀形式。为了使最初的程序比较小且容易理解，我们首先处理最简单的表达式，即由二目运算符加号和减号分隔的数位序列。在 2.6 节中，我们将扩展这个程序，使它能够翻译包含数字和其他运算符的表达式。由于表达式是很多程序设计语言中的构造，因此深入研究表达式的翻译问题是有意义的。

语法制导翻译方案常常作为翻译器的规约。图 2-21（图 2-15 的重复）中的翻译方案定义了将要执行的翻译过程。

expr	→	*expr* + *term*	{ print('+') }
	|	*expr* - *term*	{ print('-') }
	|	*term*	
term	→	0	{ print('0') }
	|	1	{ print('1') }
		...	
	|	9	{ print('9') }

图 2-21 翻译为后缀表示形式的动作

在使用一个预测语法分析器进行语法分析时，我们常常需要修改一个给定翻译方案的基础文法。特别地，图 2-21 中的翻译方案的文法是左递归的。如上节所述，预测语法分析器不能处理左递归的文法。

现在我们看起来处在矛盾之中：一方面，我们需要一个能够支持翻译规约的文法；另一方面，我们又需要一个明显不同的能够支持语法分析过程的文法。解决的方法是首先使用易于翻译的文法，然后再小心地对这个文法进行转换，使之能够支持语法分析。通过消除图 2-21 中的左递归，我们可以得到一个适用于预测递归下降翻译器的文法。

2.5.1 抽象语法和具体语法

设计一个翻译器时，名为*抽象语法树*（abstract syntax tree）的数据结构是一个很好的起点。在一个表达式的抽象语法树中，每个内部结点代表一个运算符，该结点的子结点代表这个运算符的运算分量。对于更加一般化的情况，当我们处理任意的程序设计语言构造时，我们可以创建一个针对这个构造的运算符，并把这个构造的具有语义信息的组成部分作为这个运算符的运算分量。

9 - 5 + 2 的抽象语法树如图 2-22 所示，其中根结点代表运算符 +，根结点的子树分别代表子表达式 9 - 5 和 2。将 9 - 5 组成一个运算分量反映了在对优先级相同的运算符求值时，求值顺序总是从左到右的。因为 + 和 - 具有相同的优先级，因此 9 - 5 + 2 等价于（9 - 5）+ 2。

抽象语法树也简称*语法树*（syntax tree），在某种程度上和语法分析树相似。但是在抽象语法树中，内部结点代表的是程序构造；而在语法分析树中，内部结点代表的是非终结符号。文法中的很多非终结符号都代表程序的构造，但也有一部分是各种各样的辅助符号，比如那些代表项、因子或其他表达式变体的非终结符号。在抽象语法树中，通常不需要这些辅助符号，因此会将这些符号省略掉。为了强调它们之间的区别，我们有时把语法分析树称为*具体语法树*（concrete syntax tree），而相应的文法称为该语言的*具体语法*（concrete syntax）。

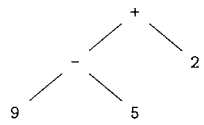


图 2-22 9 - 5 + 2 的语法树

在图 2-22 给出的语法树中，每个内部结点都和一个运算符关联。树中没有对应于 *expr*→*term* 这样的*单产生式*（即产生式体中仅包含一个非终结符号的产生式）的“辅助”结点，也没有对应于 ϵ 产生式（比如 *rest*→ϵ）的结点。

我们希望翻译方案的基础文法的语法分析树与抽象语法树尽可能相近。图 2-21 中的文法对子表达式进行分组的方式与语法树的分组方式相似。例如，加运算符的子表达式是由产生式体 *expr* + *tern* 中的 *expr* 和 *term* 给出的。

2.5.2 调整翻译方案

图 2-20 中简述的左递归消除技术同样可以应用于包含了语义动作的产生式。首先，该技术被扩展到 A 的多个产生式中。在我们的例子中，A 就是 *expr*，它有两个 *expr* 的左递归产生式和一

个非左递归的产生式。这个技术将产生式 $A\rightarrow A\alpha \mid A\beta \mid \gamma$ 转换成

$$A\rightarrow \gamma R$$
$$R\rightarrow \alpha R \mid \beta R \mid \epsilon$$

其次，我们要转换的产生式不仅包含终结符号和非终结符号，还包含内嵌动作。嵌入在产生式中的语义动作在转换时被当作终结符号直接进行复制。

例 2.13 考虑图 2-21 中的翻译方案。令

$$A = expr$$
$$\alpha = + \ term\{\text{print}('+')\}$$
$$\beta = - \ term\{\text{print}('-')\}$$
$$\gamma = term$$

那么进行左递归消除转换后将产生如图 2-23 所示的翻译方案。图 2-21 中的 *expr* 产生式已经转换成 *expr* 和新非终结符号 *rest* 的产生式，其中 *rest* 扮演了 R 的角色。*term* 的产生式就是图2-21 中 *term* 的产生式。图 2-24 展示了使用图 2-23 中的文法对 9 - 5 + 2 进行翻译的过程。 □

```
expr  →  term rest
rest  →  + term { print('+') } rest
      |  - term { print('-') } rest
      |  ϵ
term  →  0 { print('0') }
      |  1 { print('1') }
      |  ...
      |  9 { print('9') }
```

图 2-23 消除左递归后的翻译方案

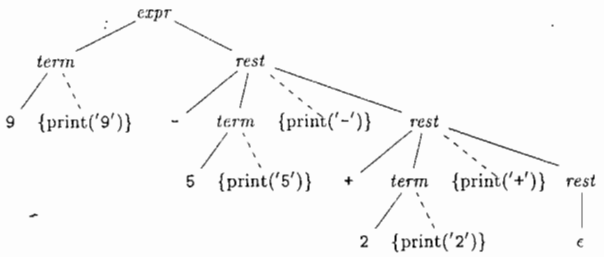


图 2-24 从 9 - 5 + 2 到 95 - 2 + 的翻译

左递归消除的工作必须小心进行，以确保消除后的结果保持语义动作的顺序。例如，在图 2-23 的翻译方案中，动作{print('+')}和{print('-')}都处于产生式体的中间，两边分别是非终结符号 *term* 和 *rest*。假如将这个动作放到产生式的末尾，即 *rest* 之后，那么这个翻译就是不正确的。请读者自己证明，假如这么做，9 - 5 + 2 就会被错误地转换成 952 + -，它是 9 - (5 + 2) 的后缀表示方式；而我们实际想要的是 952 + -，即(9 - 5)+ 2 的后缀表示方式。

2.5.3 非终结符号的过程

图 2-25 中的函数 *expr*、*term* 和 *rest* 实现了图 2-23 中的语法制导翻译方案。这些函数模拟了对应于非终结符号的各个产生式体。函数 *expr* 先调用 *term*( )再调用 *rest*( )，从而实现产生式 *expr* → *term rest*。

函数 *rest* 实现了图 2-23 中非终结符 *rest* 的三个产生式。如果向前看符号是加号，这个函数就使用第一个产生式；如果向前看符号是减号，就使用第二个产生式；在其他情况下使用产生式 *rest*→ϵ。非终结符号 *rest* 的前两个产生式是用过程 *rest* 中 if 语句的前两个分支实现的。如果向前看符号是 +，就调用 *match*('+')来匹配它。在调用 *term*( )之后，相应的语义动作通过输出一个加号来实现。第二个产生式与此类似，只是用 - 代替 +。因为 *rest* 的第三个产生式的右部是 ϵ，所以函数 *rest* 中最后一个 else 子句不做任何处理。

非终结符号 *term* 的十个产生式生成十个数位。因为每一个产生式都生成一个数位并打印，所以在图 2-25 中用相同的代码实现这些产生式。如果 *term*( )中的条件表达式成立，变量 *t* 中就保存 *lookahead* 代表的数位，它将在调用完 *match* 之后被打印出来。注意，*match* 会改变向前看符

号，所以我们需要 t 保存这个数位，以便稍后打印输出[⊖]。

```
void expr() {
        term(); rest();
}

void rest() {
        if ( lookahead == '+' ) {
                match('+'); term(); print('+'); rest();
        }
        else if ( lookahead == '-' ) {
                match('-'); term(); print('-'); rest();
        }
        else { } /* 不对输入作任何处理 */ ;
}

void term() {
        if ( lookahead是一个数位 ) {
                t = lookahead; match(lookahead); print(t);
        }
        else report("语法错误");
}
```

图 2-25　非终结符 *expr*、*rest* 和 *term* 的伪代码

2.5.4　翻译器的简化

在给出完整的程序之前，我们将对图 2-25 中的代码做两处简化。这个简化将把过程 *rest* 展开到过程 *expr* 中。在翻译具有多个优先级的表达式时，这样的简化处理可以减少需要使用的过程数目。

首先，某些递归调用可以被替换为迭代。如果一个过程体中执行的最后一条语句是对该过程的递归调用，那么这个调用就称为是*尾递归*的（tail recursive）。例如，在函数 *rest* 中，当向前看符号为 + 和 - 时对 *rest*( ) 的调用都是尾递归的。因为在每个分支中，对 *rest* 的递归调用都是调用 *rest* 时执行的最后一条语句。

对于没有参数的过程，一个尾递归调用可以被替换为跳转到过程开头的语句。过程 *rest* 的代码可以被改写为图 2-26 中的伪代码。只要向前看符号是一个加号或一个减号，过程 *rest* 就和该符号匹配，并调用 *term* 来匹配一个数位，然后重复这一过程。否则，它就跳出 while 循环并从 *rest* 返回。

```
void rest() {
        while( true ) {
                if( lookahead == '+' ) {
                        match('+'); term(); print('+'); continue;
                }
                else if ( lookahead == '-' ) {
                        match('-'); term(); print('-'); continue;
                }
                break ;
        }
}
```

图 2-26　消除图 2-25 中过程 *rest* 的尾递归

其次，整个 Java 程序还包含另一处修改。一旦图 2-25 中 *rest* 过程的尾递归调用被替换为迭代过程，那么对 *rest* 的调用仅仅出现在过程 *expr* 中。因此，将过程 *expr* 中对 *rest* 的调用替换为 *rest* 的过程体，就可以将这两个函数合二为一。

⊖ 作为一个小小的优化，我们可以在调用 *match* 之前打印这个数位，避免将这个数位保存起来。一般来说，改变语义动作和文法符号之间的顺序是有风险的，因为这么做可能改变这个翻译的结果。

2.5.5 完整的程序

我们的翻译器的完整 Java 程序显示在图 2-27 中。第一行以 import 开头，使得程序可以访问 java.io 包以进行系统输入和输出。其余的代码包括两个类：Parser 和 Postfix。类 Parser 包含变量 lookahead 和函数 Parser、expr、term 和 match。

```
import java.io.*;
class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void expr() throws IOException {
        term();
        while(true) {
            if( lookahead == '+' ) {
                match('+'); term(); System.out.write('+');
            }
            else if( lookahead == '-' ) {
                match('-'); term(); System.out.write('-');
            }
            else return;
        }
    }

    void term() throws IOException {
        if( Character.isDigit((char)lookahead) ) {
            System.out.write((char)lookahead); match(lookahead);
        }
        else throw new Error("syntax error");
    }

    void match(int t) throws IOException {
        if( lookahead == t ) lookahead = System.in.read();
        else throw new Error("syntax error");
    }
}

public class Postfix {
    public static void main(String[] args) throws IOException {
        Parser parse = new Parser();
        parse.expr(); System.out.write('\n');
    }
}
```

图 2-27 将中缀表达式翻译为后缀表达形式的 Java 程序

程序的执行从类 Postfix 中定义的函数 main 开始。函数 main 创建了一个 Parser 类的实例 parse，然后调用它的函数 expr 对一个表达式进行语法分析。

和类 Parser 同名的函数 Parser 是该类的*构造函数*(constructor)，它在创建该类的一个对象时自动调用。请注意，根据类 Parser 开始处的定义，构造函数 Parser 读入一个词法单元，并将变量 lookahead 初始化为这个词法单元。由单个字符组成的词法单元是由系统输入例程 read 提供的，该子程序从输入文件中读取下一个字符。注意，lookahead 被声明为整型变量，而不是字符型变量。这是为了便于在后面引入非单个字符的其他词法单元。

函数 expr 是 2.5.4 节中讨论的简化处理的结果。它实现了图 2-23 中的非终结符号 expr 和

rest。图 2-27 中 expr 的代码首先调用 term，然后用一个 while 循环不断测试 lookahead 是否和 + 或 - 匹配。当运行到代码中的 return 语句时，控制流离开这个 while 循环。在循环内部，System 类的输入/输出功能用来写一个字符。

函数 term 使用 Java 类 Character 中的例程 isDigit 来判断向前看符号是否为一个数位。例程 isDigit 的参数是一个字符。然而，为了方便将来的扩展，lookahead 被声明为整型变量。(char)lookahead 将 lookahead 的类型强制转化(cast)为字符。和图 2-25 相比，这里有一个小的改动，即输出向前看字符的语义动作在调用 match 之前就执行了。

函数 match 检查终结符号。如果向前看符号是匹配的，它就读取下一个输入终结符号，否则它执行下面的代码，发出出错消息。

```
throw new Error("syntax error");
```

上述代码创建了类 Error 的一个新异常，并将"syntax error"作为其错误消息。Java 并不强制要求在 throw 子句中声明 Error 异常，因为这些异常的本意是表示不应该发生的不正常事件。⊖

Java 的一些主要特征

对于不熟悉 Java 的读者来说，下面的一些注解有助于他们阅读图 2-27 中的代码：

- 一个 Java 的类由变量和函数定义的序列组成。
- 函数(例程)的参数列表用括号括起来，即使没有参数也需要写出括号，因此我们写成 expr()和 term()。这些函数实际上是过程，因为它们的函数名字前面的关键字 void 表示它们没有返回值。
- 函数之间通信时可以通过"值传递方式"传递参数，也可以通过访问共享数据进行通信。比如，函数 expr()和 term()使用类变量 lookahead 来检查向前看符号。这两个函数都可以访问这个类变量，因为它们同属于类 Parser。
- 和 C 语言一样，Java 语言使用 = 表示赋值，== 表示等于，!= 表示不等于。
- term()定义中的子句"throw IOException"声明该函数在执行时可能会出现一个名为 IOException 的异常。当函数 match 调用例程 read 时，如果无法读到输入就会出现这样的异常。任何调用了 match 的函数也必须声明在该函数运行时可能出现一个 IOException 异常。

2.6 词法分析

一个词法分析器从输入中读取字符，并将它们组成"词法单元对象"。除了用于语法分析的终结符号之外，一个词法单元对象还包含一些附加信息，这些信息以属性值的形式出现。至今为止，我们还不需要区分术语"词法单元"和"终结符号"，因为语法分析器忽略了词法单元中带有

⊖ 错误处理可以使用 Java 的异常处理机制来实现。方法之一是声明一个扩展了系统类 Exception 的新的异常，比如 SyntaxError。然后在 term 或 match 中检测到错误时抛出 SyntaxError 异常，而不是 Error 异常。然后在 main 中把对 parse.expr()的调用放在一个 try 语句中。该 try 语句可以捕获 SyntaxError 异常，输出一个消息并结束。如果这么做，我们将需要在图 2-27 的程序中加入一个类 SyntaxError。要完成这个扩展，我们还必须修改 match 和 term 的声明，使得它们不仅可以抛出 IOException，还可以抛出 SyntaxError。同时也必须重新声明调用它们的函数 expr，使得它可以抛出 SyntaxError 异常。

的属性值。在本节中，一个词法单元就是一个带有附加信息的终结符号。

构成一个词法单元的输入字符序列称为词素(lexem)。因此，我们可以说，词法分析器使得语法分析器不需要考虑词法单元的词素表示方式。

本节的词法分析器允许在表达式中出现数字、标识符和“空白”(空格、制表符和换行符)。它可以用于扩展上一节中介绍的表达式翻译器。要允许在表达式中出现数字和标识符，就必须扩展图 2-21 中的表达式文法。借此机会我们还将使扩展后的文法支持乘法和除法运算。扩展后的翻译方案如图 2-28 所示。

```
expr   →  expr + term      { print('+') }
       |  expr - term      { print('-') }
       |  term

term   →  term * factor    { print('*') }
       |  term / factor    { print('/') }
       |  factor

factor →  ( expr )
       |  num              { print(num.value) }
       |  id               { print(id.lexeme) }
```

图 2-28 翻译得到后缀表示方式的语义动作

在图 2-28 中，假定终结符号 **num** 具有属性 **num**. *value*，该属性给出了对应于 **num** 的本次出现的整数值。终结符号 **id** 有一个值为字符串类型的属性，写作 **id**. *lexeme*。我们假设这个字符串就是这个 **id** 实例的实际词素。

在本节结束时，这些被用来演示词法分析器的工作方式的伪代码片段将被组合成 Java 代码。本节中介绍的方法适合于手写的词法分析器。3.5 节描述了一个可根据一个词法规范生成词法分析器的工具 Lex。用于保存标识符相关信息的符号表或数据结构将在 2.7 节中讨论。

2.6.1 剔除空白和注释

2.5 节的表达式翻译器读取输入中的每个字符，所以任何无关字符，比如空格，都会使它运行失败。大部分语言允许词法单元之间出现任意数量的空白。在语法分析过程中同样会忽略源程序中的注释，所以这些注释也可以当作空白处理。

如果词法分析器消除了空白，那么语法分析器就不必再考虑它们了。当然，也可以修改文法使得语法中包含空白，但是实现这个方法远非易事。

图 2-29 中的伪代码在遇到空格、制表符或换行符时不断读取输入字符，从而跳过了空白部分。变量 *peek* 存放了下一个输入字符。在错误消息中加入行号和上下文有助于定位错误。这个代码使用变量 *line* 统计输入中的换行符个数。

```
for ( ; ; peek = next input character ) {
    if ( peek is a blank or a tab ) do nothing;
    else if ( peek is a newline ) line = line+1;
    else break;
}
```

图 2-29 跳过空白部分

2.6.2 预读

在决定向语法分析器返回哪个词法单元之前，词法分析器可能需要预先读入一些字符。例如，C 或 Java 的词法分析器在遇到字符 > 之后必须预先读入一个字符。如果下一个字符是 =，那么 > 就是字符序列 >= 的一部分，这个序列是代表“大于等于”运算符的词法单元的词素。否则 > 本身形成了“大于”运算符，词法分析器就多读了一个字符。

一个通用的预先读取输入的方法是使用输入缓冲区。词法分析器可以从缓冲区中读取一个字符，也可以把字符放回缓冲区。即使仅从效率的角度看，使用缓冲区也是有意义的，因为一次读取一块字符要比每次读取单个字符更加高效。我们可以用一个指针来跟踪已被分析的输入部分，向缓冲区放回一个字符可以通过回移指针来实现。输入缓冲技术将在 3.2 节中讨论。

因为通常只需预读一个字符，所以一种简单的解决方法是使用一个变量，比如 *peek*，来保存下一个输入字符。在读入一个数字的数位或一个标识符的字符时，本节的词法分析器会预读一个字符。例如，它在 1 后面预读一个字符来区分 1 和 10，在 t 后预读一个字符来区分 t 和 true。

词法分析器只在必要时才进行预读。像 * 这样的运算符不需预读就能够识别。在这种情况下，*peek* 的值被设置为空白符。词法分析器在寻找下一个词法单元时会跳过这个空白符。本节中的词法分析器的不变式断言如下：当词法分析器返回一个词法单元时，变量 *peek* 要么保存了当前词法单元的词素后的那个字符，要么保存空白符。

2.6.3 常量

在一个表达式的文法中，任何允许出现数位的地方都应该允许出现任意的整型常量。要使得表达式中可以出现整数常量，我们可以创建一个代表整型常量的终结符号，比如 **num**，也可以将整数常量的语法加入到文法中。将字符组成整数并计算它的数值的工作通常是由词法分析器完成的，因此在语法分析和翻译过程中可以将数字当作一个单元进行处理。

当在输入流中出现一个数位序列时，词法分析器将向语法分析器传送一个词法单元。该词法单元包含终结符号 **num** 及根据这些数位计算得到的整型属性值。如果我们把词法单元写成用〈〉括起来的元组，那么输入 `31 + 28 + 59` 就被转换成序列

⟨**num**, 31⟩⟨ + ⟩⟨**num**, 28⟩⟨ + ⟩⟨**num**, 59⟩

在这里，终结符号 + 没有属性，所以它的元组就是⟨+⟩。图 2-30 中的伪代码读取一个整数中的数位，并用变量 v 累计得到这个整数的值。

```
if ( peek holds a digit ) {
    v = 0;
    do {
        v = v * 10 + integer value of digit peek;
        peek = next input character;
    } while ( peek holds a digit );
    return token ⟨num, v⟩;
}
```

图 2-30 将数位组成整数

2.6.4 识别关键字和标识符

大多数程序设计语言使用 `for`、`do`、`if` 这样的固定字符串作为标点符号，或者用于标识某种构造。这些字符串称为*关键字*(keyword)。

字符串还可以作为标识符，来为变量、数组、函数等命名。为了简化语法分析器，语言的文法通常把标识符当作终结符号进行处理。当某个标识符出现在输入中时，语法分析器都会得到相同的终结符号，如 **id**。例如，在处理如下输入时

```
count = count + increment;
```
(2.6)

语法分析器处理的是终结符号序列 **id = id + id**。词法单元 **id** 有一个属性保存它的词素。将词法单元写作元组形式，我们看到输入流(2.6)的元组序列是

⟨**id**,"count"⟩ ⟨=⟩ ⟨**id**,"count"⟩ ⟨+⟩ ⟨**id**,"increment"⟩ ⟨;⟩

关键字通常也满足标识符的组成规则，因此我们需要某种机制来确定一个词素什么时候组成一个关键字，什么时候组成一个标识符。如果将关键字作为*保留字*，也就是说，如果它们不能被用作标识符，这个问题相对容易解决。此时，只有当一个字符串不是关键字时它才能组成一个标识符。

本节中的词法分析器使用一个表来保存字符串，从而解决了如下两个问题：

- *单一表示*。一个字符串表可以将编译器的其余部分和表中字符串的具体表示隔离开，因为编译器后面的步骤可以只使用指向表中字符串的指针或引用。操作引用要比操作字符串本身更加高效。
- *保留字*。要实现保留字，可以在初始化时在字符串表中加入保留的字符串以及它们对应的词法单元。当词法分析器读到一个可以组成标识符的字符串或词素时，它首先检查这个字符串表中是否有这个词素。如是，它就返回表中的词法单元，否则返回带有终结符号 **id** 的词法单元。

在 Java 中，使用类 *Hashtable* 可以将一个字符串表实现为一张散列表。下面的声明

Hashtable words = **new** *Hashtable*( );

将 *words* 初始化为一个将键映射到值的默认散列表。我们将使用它来实现从词素到词法单元的映射。图 2-31 中的伪代码使用 *get* 操作来查找保留字。这个伪代码从输入中读取一个以字母开头、由字母和数位组成的字符串 *s*。我们假定读取的 *s* 尽可能地长，即只要词法分析器遇到字母或数位，它就不断从输入中读取字符。当它遇到的不是字母或数位，比如它遇到了空白符，已读取的词素就被复制到缓冲区 *b* 中。如果字符串表中已经有一个 *s* 的条目，它就返回由 *words. get* 得到的词法单元。这里 *s* 可能是一个关键字，在表 *words* 初始化的时候这个 *s* 就已经在表中了；它也可能是一个之前被加入到表中的标识符。如果不存在 *s* 对应的条目，那么由 **id** 和属性值 *s* 组成的词法单元将被加入到字符串表中，并被返回。

```
if ( peek 存放了一个字母 ) {
        将字母或数位读入一个缓冲区b;
        s = b 中的字符形成的字符串 ;
        w = words.get(s) 返回的词法单元 ;
        if ( w 不是 null ) return w;
        else {
                将键- 值对 (s, (id, s)) 加入到 words ;
                return 词法单元 (id, s);
        }
}
```

图 2-31 区分关键字和标识符

2.6.5 词法分析器

将本节到目前为止给出的伪代码片段组合起来，就可以得到一个返回词法单元对象的函数 *scan*。如下所示：

```
Token scan( ) {
    跳过空白符，见 2.6.1 节；
    处理数字，见 2.6.3 节；
    处理保留字和标识符，见 2.6.4 节；
    / * 如果我们运行到这里，就将预读字符 peek 作为一个词法单元 * /
    Token t = new Token(peek) ;
    peek = 空白符 / * 按照 2.6.2 讨论的方法初始化 * /;
    return t;
}
```

本节的其余部分将函数 *scan* 实现为一个用于词法分析的 Java 程序包的一部分。这个叫做 *lexer* 的包中包含对应于各种词法单元的类和一个包含函数 `scan` 的类 `Lexer`。

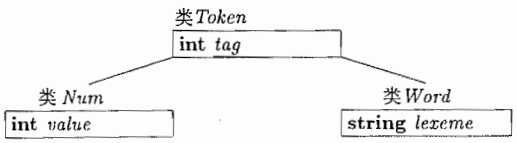


图 2-32 类 Token 以及子类 Num 和 Work

图 2-32 中显示了对应于各个词法单元的类及它们的字段，但图中没有给出它们的方法。类 `Token` 有一个 `tag` 字段，它用于做出语法分析决定。子类 `Num` 增加了一个用于存放整数值的字段 `value`；子类 `Word` 增加了一个字段 `lexeme`，用于保存关键字和标识符的词素。

每个类都在以它的名字命名的文件中。`Token` 类的文件内容如下：

```
1) package lexer;                    // 文件 Token.java
2) public class Token {
3)     public final int tag;
4)     public Token(int t) { tag = t; }
5) }
```

第一行指明了 *lexer* 包。第 3 行声明了字段 `tag` 为 `final` 的，即它一旦被赋值就不能再修改。第 4 行上的构造函数 `Token` 用于创建词法单元对象，比如

```
new Token('+')
```

创建了 Token 类的一个新对象，并且把它的 tag 字段初始化为“+”的整数表示。（为简洁起见，我们省略了常用的方法 toString。该方法将返回一个适于打印的字符串。）

在伪代码中使用诸如 **num**、**id** 这样的终结符号的地方，Java 代码中使用整型常量表示。类 Tag 实现了这些常量：

```
1) package lexer;                    // 文件 Tag.java
2) public class Tag {
3)    public final static int
4)       NUM = 256, ID = 257, TRUE = 258, FALSE = 259;
5) }
```

除了值为整数的字段 NUM 和 ID 外，这个类还定义了两个字段 TRUE 和 FALSE 以备后用，它们将用于演示如何处理保留的关键字。⊖

Tag 类中的字段是 public 的，因此它们可以在包的外面使用。它们同时也是 static 的，因此这些字段只能有一个实例，或者说拷贝。这些字段是 final 的，因此它们只能被赋值一次。事实上，这些常量就代表常量。在 C 语言中，可以使用 define 语句来获得类似的效果。这些 define语句使得 NUM 这样的名字可以被当作符号常量使用，例如：

```
#define NUM 256
```

在伪代码引用终结符号 **num** 和 **id** 的地方，Java 代码引用的是 Tag.NUM 和 Tag.ID。唯一的要求是 Tag.NUM 和 Tag.ID 必须被初始化为互不相同的值，且这些初始化值还必须不同于那些代表单字符词法单元（比如“+”或“*”）的常量。

```
1) package lexer;                    // 文件 Num.java
2) public class Num extends Token {
3)    public final int value;
4)    public Num(int v) { super(Tag.NUM); value = v; }
5) }
```

```
1) package lexer;                    // 文件 Word.java
2) public class Word extends Token {
3)    public final String lexeme;
4)    public Word(int t, String s) {
5)       super(t); lexeme = new String(s);
6)    }
7) }
```

图 2-33 Token 的子类 Num 和 Word

类 Num 和 Word 显示在图 2-33 中。类 Num 通过在第 3 行声明一个整数字段 value 而扩展了 Token。第 4 行的构造函数 Num 调用了 super (Tag.NUM)，该函数把其父类 Token 的 tag 字段设定为 Tag.NUM。

类 Word 既可用于保留字，也可用于标识符，因此第 4 行上的构造函数 Word 需要两个参数：一个词素和一个与 tag 对应的整数值。一个用于保留字 true 的对象可以通过以下语句创建：

```
new Word(Tag.TRUE, "true")
```

这个语句创建了一个新对象，该对象的 tag 字段被设为 Tag.TRUE，lexeme 字段被设为字符串“true”。

用于词法分析的类 Lexer 显示在图 2-34 和图 2-35 中。第 4 行上的整型变量 line 用于对输入行计数，第 5 行上的字符变量 peek 用于存放下一个输入字符。

保留字在第 6 行到第 11 行处理。第 6 行声明了表 words。第 7 行上的辅助函数 reserve 将一个字符串 - 字对放入这个表中。构造函数 Lexer 中的第 9 行和第 10 行初始化这个表。它们使用构造函数 Word 来创建字对象，这些对象被传递到辅助函数 reserve。因此，在第一次调用 scan 之前，这个表被初始化，并且预先加入了保留字“true”和“false”。

⊖ ASCII 字符通常被转化为 0 ~ 255 之间的整数。因此我们用大于 255 的整数来表示终结符号。

```
1) package lexer;                          //文件 Lexer.java
2) import java.io.*; import java.util.*;
3) public class Lexer {
4)    public int line = 1;
5)    private char peek = ' ';
6)    private Hashtable words = new Hashtable();
7)    void reserve(Word t) { words.put(t.lexeme, t); }
8)    public Lexer() {
9)       reserve( new Word(Tag.TRUE,  "true")  );
10)      reserve( new Word(Tag.FALSE, "false") );
11)   }
12)   public Token scan() throws IOException {
13)      for( ; ; peek = (char)System.in.read() ) {
14)         if( peek == ' ' || peek == '\t' ) continue;
15)         else if( peek == '\n' ) line = line + 1;
16)         else break;
17)      }
         /* 续见图 2-35*/
```

图 2-34 词法分析器的代码(第 1 部分)

```
18)      if( Character.isDigit(peek) ) {
19)         int v = 0;
20)         do {
21)            v = 10*v + Character.digit(peek, 10);
22)            peek = (char)System.in.read();
23)         } while( Character.isDigit(peek) );
24)         return new Num(v);
25)      }
26)      if( Character.isLetter(peek) ) {
27)         StringBuffer b = new StringBuffer();
28)         do {
29)            b.append(peek);
30)            peek = (char)System.in.read();
31)         } while( Character.isLetterOrDigit(peek) );
32)         String s = b.toString();
33)         Word w = (Word)words.get(s);
34)         if( w != null ) return w;
35)         w = new Word(Tag.ID, s);
36)         words.put(s, w);
37)         return w;
38)      }
39)      Token t = new Token(peek);
40)      peek = ' ';
41)      return t;
42)   }
43) }
```

图 2-35 词法分析器的代码(第 2 部分)

在图 2-34 和图 2-35 中，scan 的代码实现了本节中的各个伪代码片段。从第 13 行到第 17 行的 for 语句跳过了空格、制表符和换行符。当 peek 的值不是空白符时，控制流离开 for 循环。

第 18 行到第 25 行的代码读取一个数位序列。函数 isDigit 来自于 Java 的内置类 Character。它在第 18 行上用于检查 peek 是否为一个数位。如是，第 19 行到第 24 行的代码就会累积计算输入中的数位序列对应的整数值，然后返回一个新的 Num 对象。

第 26 行到第 38 行分析了保留字和标识符。关键字 **true** 和 **false** 已经在第 9 行和第 10 行被保留了。因此，如果字符串 s 不是保留字，则程序就会执行第 35 行，此时 s 一定是某个标识符的词素。因此第 35 行返回一个新的 word 对象，该对象的 lexeme 字段被设为 s，tag 字段被设为

Tag.ID。最后，第39行到第41行将当前字符作为一个词法单元返回，并把peek设为一个空格。当下一次调用scan时，这个空格会被删除。

2.6.6 2.6节的练习

练习2.6.1：扩展2.6.5节中的词法分析器以消除注释。注释的定义如下：

1）以//开始的注释，包括从它开始到这一行的结尾的所有字符。

2）以/*开始的注释，包括从它到后面第一次出现的字符序列*/之间的所有字符。

练习2.6.2：扩展2.6.5节中的词法分析器，使它能够识别关系运算符<、<=、==、!=、>=、>。

练习2.6.3：扩展2.6.5节中的词法分析器，使它能够识别浮点数，比如2.、3.14和.5等。

2.7 符号表

符号表(symbol table)是一种供编译器用于保存有关源程序构造的各种信息的数据结构。这些信息在编译器的分析阶段被逐步收集并放入符号表，它们在综合阶段用于生成目标代码。符号表的每个条目中包含与一个标识符相关的信息，比如它的字符串(或者词素)、它的类型、它的存储位置和其他相关信息。符号表通常需要支持同一标识符在一个程序中的多重声明。

从1.6.1节介绍的内容可知，一个声明的作用域是指该声明起作用的那一部分程序。我们将为每个作用域建立一个单独的符号表来实现作用域。每个带有声明的程序块[⊖]都会有自己的符号表，这个块中的每个声明都在此符号表中有一个对应的条目。这种方法对其他能够设立作用域的程序设计语言构造同样有效。例如，每个类也可以拥有自己的符号表，它的每个域和方法都在此表中有一个对应的条目。

本节包括一个符号表模块，它可以和本章中的Java翻译器代码片段一起使用。当我们在附录A中将这个翻译器集成到一起时可以直接使用这个模块。同时，为了简化问题，本节的主要例子是一个被简化了的语言，它只包含与符号表相关的关键构造，比如块、声明、因子等。所有其他的语句和表达式构造都被忽略了，这使得我们可以重点关注符号表的操作。一个程序由多个块组成，每个块包含可选的声明和由单个标识符组成的语句。每个这样的语句都表示对相应标识符的一次使用。下面是这个语言的一个例子程序：

```
{ int x; char y; { bool y; x; y; } x; y; }        (2.7)
```

1.6.3节中块结构的例子处理了名字的定义和使用。输入(2.7)仅仅由名字的定义和使用组成。

我们将要完成的任务是打印出一个修改过的程序，程序中的声明部分已经被删除，而每个“语句”中的标识符之后都跟着一个冒号和该标识符的类型。

谁来创建符号表条目?

符号表条目是在分析阶段由词法分析器、语法分析器和语义分析器创建并使用的。在本章中，我们让语法分析器来创建这些条目。因为语法分析器知道一个程序的语法结构，因此相对于词法分析器而言，语法分析器通常更适合创建条目。它可以更好地区分一个标识符的不同声明。

在有些情况下，词法分析器可以在它碰到组成一个词素的字符串时立刻建立一个符号表条目。但是在更多的情况下，词法分析器只能向语法分析器返回一个词法单元，比如**id**，以及指向这个词素的指针。只有语法分析器才能够决定是使用之前已创建的符号表条目，还是为这个标识符创建一个新条目。

⊖ 比如，在C语言中，程序块要么是一个函数，要么是函数中由花括号分隔的一个部分，这个部分中有一个或多个声明。

例 2.14 在处理上面的输入(2.7)时，目标是生成

```
{ { x:int; y:bool; } x:int; y:char; }
```

第一个 x 和 y 来自输入(2.7)的内层块。由于 x 的使用指向外层块中 x 的声明，因此第一个 x 后面跟的是 **int**，即该声明中的类型。内层块中对 y 的使用指向同一个块中的声明，因此具有布尔类型。我们同时看到，外层块中 x 和 y 的使用的类型分别为整型和字符型，也就是外层块中声明所指定的类型。 □

2.7.1 为每个作用域设置一个符号表

术语“标识符 x 的作用域”实际上指的是 x 的某个声明的作用域。术语作用域(scope)本身是指一个或多个声明起作用的程序部分。

作用域是非常重要的，因为在程序的不同部分，可能会出于不同的目的而多次声明相同的标识符。像 x 和 i 这样常见的名字会被重复使用。再例如，子类可以重新声明一个方法名字以覆盖父类中的相应方法。

如果程序块可以嵌套，那么同一个标识符的多次声明就可能出现在同一个块中。当 *stmts* 能生成一个程序块时，下面的语法规则会产生嵌套的块：

$$block \rightarrow \text{'}\{\text{'}\ decls\ stmts\ \text{'}\}\text{'}$$

(我们对这个语法中的花括号使用了引号，这么做的目的是将它们和用于语义动作的花括号区分开来。)在图 2-38 给出的文法中，*decls* 生成一个可选的声明序列，*stmts* 生成一个可选的语句序列。更进一步，一条语句可以是一个程序块，所以我们的语言支持嵌套的语句块。而标识符可以在这些块中重新声明。

块的符号表的优化

块的符号表的实现可以利用作用域的最近嵌套规则。嵌套的结构确保可应用的符号表形成一个栈。在栈的顶部是当前块的符号表。栈中这个表的下方是包含这个块的各个块的符号表。因此，符号表可以按照类似于栈的方式来分配和释放。

有些编译器维护了一个散列表来存放可访问的符号表条目。也就是说，存放那些没有被内嵌块中的某个声明掩盖起来的条目。这样的散列表实际上支持常量时间的查询，但是在进入和离开块时需要插入和删除相应的条目。在从一个块 B 离开时，编译器必须撤销所有因为 B 中的声明而对此散列表作出的修改。它可以在处理 B 的时候维护一个辅助的栈来跟踪对这个散列表的所做的修改。

语句块的最近嵌套(most-closely)规则是说，一个标识符 x 在最近的 x 声明的作用域中。也就是说，从 x 出现的块开始，从内向外检查各个块时找到的第一个对 x 的声明。

例 2.15 下列伪代码用下标来区分对同一标识符的不同声明：

```
1) { int x1; int y1;
2)   { int w2; bool y2; int z2;
3)     ···w2···; ···x1···; ···y2···; ···z2···;
4)   }
5)   ···w0···; ···x1···; ···y1···;
6) }
```

下标并不是标识符的一部分，它实际上是该标识符对应的声明的行号。因此，x 的所有出现都位于第 1 行上声明的作用域中。第 3 行上出现的 y 位于第 2 行上 y 的声明的作用域中，因为 y 在内

层块中被再次声明了。然而，第 5 行上出现的 y 位于第 1 行上 y 的声明的作用域中。

假设第 5 行上 w 的出现位于这个程序片段之外某个 w 的声明的作用域中，它的下标表示一个全局的或者位于这个块之外的声明。

最后，z 在最内层的块中声明并使用。它不能在第 5 行上使用，因为这个内嵌的声明只能作用于最内层的块。 □

实现语句块的最近嵌套规则时，我们可以将符号表链接起来，也就是使得内嵌语句块的符号表指向外围语句块的符号表。

例 2.16 图 2-36 显示了对应于例 2.15 中伪代码的符号表。B_1 对应于从第 1 行开始的语句块；B_2 对应着从第 2 行开始的语句块。图的顶端是符号表 B_0，它记录了全局的或由语言提供的默认声明。在我们分析第 2 行至第 4 行时，环境是由一个指向最下层的符号表(即 B_2 的符号表)的指针表示的。当我们分析第 5 行时，B_2 的符号表变得不可访问，环境指针转而指向 B_1 的符号表，此时我们可以访问上一层的全局符号表 B_0，但不能访问 B_2 的符号表。 □

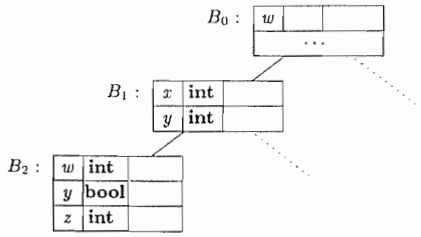


图 2-36 对应于例 2.15 的符号表链

图 2-37 中是链接符号表的 Java 实现。它定义了一个类 Env(环境“environment”的缩写)[⊖]。类 Env 支持三种操作：

- *创建一个新符号表。*图 2-37 中第 6 行至第 8 行所示的构造函数 Env(p)创建一个 Env 对象，该对象包含一个名为 table 的散列表。这个对象的字段 prev 被设置为参数 p，而这个参数的值是一个环境，因此这个对象被链接到环境。虽然形成链表的是 Env 对象，但是将它们说成是链接的符号表比较方便。

```
1) package symbols;                    // 文件 Env.java
2) import java.util.*;
3) public class Env {
4)    private Hashtable table;
5)    protected Env prev;

6)    public Env(Env p) {
7)       table = new Hashtable(); prev = p;
8)    }

9)    public void put(String s, Symbol sym) {
10)      table.put(s, sym);
11)   }

12)   public Symbol get(String s) {
13)      for( Env e = this; e != null; e = e.prev ) {
14)         Symbol found = (Symbol)(e.table.get(s));
15)         if( found != null ) return found;
16)      }
17)      return null;
18)   }
19) }
```

图 2-37 类 Env 实现了链接符号表

- *在当前表中加入一个新的条目。*散列表保存了键 - 值对，其中
  - 键(key)是一个字符串，也可以说是一个指向字符串的引用。我们也可以使用指向对应

⊖ “环境”是另一个用于表示与程序中某个点相关的符号表集合的术语。

于标识符的词法单元对象的引用作为键。

- 值(value)是一个 Symbol 类的条目。第 9 行到第 11 行的代码不需要知道一个条目的内部结构。也就是说，这个代码是独立于 Symbol 类的字段和方法的。
- 得到一个标识符的条目。它从当前块的符号表开始搜索链接符号表。第 12 行至第 18 行中这个操作的代码返回一个符号表条目或 null。

因为会有多个语句块嵌套在同一外围语句块中，所以将这些符号表链接起来就可以形成一个树形结构。图 2-36 中的虚线提醒我们链接的符号表可以形成一棵树。

2.7.2 符号表的使用

从效果看，一个符号表的作用是将信息从声明的地方传递到实际使用的地方。当分析标识符 x 的声明时，一个语义动作将有关 x 的信息“放入”符号表中。然后，一个像 *factor* →**id** 这样的产生式的相关语义动作从符号表中“取出”这个标识符的信息。因为对一个表达式 E_1 **op** E_1(其中 **op** 代表一般的运算符)的翻译只依赖于对 E_1 和 E_2 的翻译，不直接依赖于符号表，所以我们可以加入任意数量的运算符，而不会影响从声明通过符号表到达使用地点的基本信息流。

例 2.17 图 2-38 中的翻译方案说明了如何使用类 *Env*。这个翻译方案主要考虑作用域、声明和使用。它实现了例 2.14 中描述的翻译。如前面描述的，在处理输入

```
{ int x; char y; { bool y; x; y; } x; y; }
```

时，这个翻译方案过滤掉了各个声明，并生成

```
{ { x:int; y:bool; } x:int; y:char; }
```

请注意图 2-38 中各个产生式的体都已经对齐，因此所有的文法符号出现在同一列上，并且所有的语义动作都出现在第二列上。结果，一个产生式体的各个组成部分常常分开出现在多行上。

program	→		{ *top* = **null**; }
		block	
block	→	'{'	{ *saved* = *top*;
			top = **new** *Env*(*top*);
			print("{ "); }
		decls stmts '}'	{ *top* = *saved*;
			print("} "); }
decls	→	*decls decl*	
	|	ϵ	
decl	→	**type id** ;	{ *s* = **new** *Symbol*;
			s.type = **type**.*lexeme*;
			top.put(**id**.*lexeme*, *s*); }
stmts	→	*stmts stmt*	
	|	ϵ	
stmt	→	*block*	
	|	*factor* ;	{ print("; "); }
factor	→	**id**	{ *s* = *top.get*(**id**.*lexeme*);
			print(**id**.*lexeme*);
			print(":");
			print(*s.type*);}

图 2-38 使用符号表翻译带有语句块的语言

现在考虑语义动作。这个翻译方案在进入和离开块的时候将分别创建和释放符号表。变量 *top* 表示一个符号表链的顶部的顶层符号表。这个翻译方案的基础文法的第一个产生式是 *program*→*block*。在 *block* 之前的语义动作将 *top* 初始化为 **null**，即不包含任何条目。

第二个产生式 *block*→ '*'{*decls stmts*'}'中包含了进入和离开块时的语义动作。在进入块时，在 *decls* 之前，一个语义动作使用局部变量 *saved* 保存了对当前符号表的引用。这个产生式的每次使用都有一个单独的局部变量 *saved*，这个变量和这个产生式的其他使用中的局部变量都不同。在一个递归下降语法分析器中，*saved* 可以是 *block* 对应的过程的局部变量。对于递归函数中的局部变量的处理方法将在 7.2 节中讨论。代码

$$top = \textbf{new}\ Env(top);$$

将变量 *top* 设置为刚刚创建的新符号表。这个新符号表被链接到进入这个块之前一刻 *top* 的原值。变量 *top* 是类 *Env* 的一个对象，构造函数 *Env* 的代码显示在图 2-37 中。

在离开块时，'}'之后的一个语义动作将 *top* 的值恢复为进入块时保存起来的值。从实际效果看，这个表形成了一个栈，将 *top* 恢复为之前保存的值实际上是将该块中各个声明的结果弹出栈[⊖]。这样就使得该块中的声明在块外不可见。

声明 *decl*→**type id** 的结果是创建一个对应于已声明标识符的新条目。我们假设词法单元 **type** 和 **id** 都有一个相关的属性，分别是被声明标识符的类型和词素。我们不会讨论符号对象 *s* 的所有字段，但是我们假设对象中有一个字段 *type* 给出该符号的类型。我们创建一个新的符号对象 *s*，并通过代码 *s*. *type* = **type**. *lexeme* 为它赋予正确的类型。整个条目使用 *top*. *put*(**id**. *lexeme*, *s*) 加入到顶层的符号表中。

产生式 *factor*→**id** 中的语义动作通过符号表获取这个标识符的条目。操作 *get* 从 *top* 开始搜索符号表链中的第一个关于此标识符的条目。搜索得到的条目包含有关该标识符的所有信息，比如标识符的类型。 □

2.8 生成中间代码

编译器的前端构造出源程序的中间表示，而后端根据这个中间表示生成目标程序。在这一节里，我们考虑表达式和语句的中间表示形式，并给出一个如何生成中间表示的指导性的例子。

2.8.1 两种中间表示形式

正如我们在 2.1 节(特别是在图 2-4 中)指出的，两种最重要的中间表示形式是：

- 树型结构，包括语法分析树和(抽象)语法树。
- 线性表示形式，特别是“三地址代码”。

抽象语法树(或简称语法树)曾在 2.5.1 节中介绍过。我们将在 5.3.1 节中更加正式地探讨它。在语法分析过程中，将创建抽象语法树的结点来表示有意义的程序构造。随着分析的进行，信息以与结点相关的属性的形式被添加到这些结点上。选择哪些属性要依据待完成的翻译来决定。

另一方面，三地址代码是一个由基本程序步骤(比如将两个值的相加)组成的序列。和树形结构不一样，它没有层次化的结构。正如我们将在第 9 章中看到的那样，如果我们想对代码做出显著的优化，就需要这种表示形式。在那种情况下，我们可以把组成程序的很长的三地址语句序列分解

⊖ 我们也可以使用另一种方法来处理，可以在类 Env 中加入静态操作 *push* 和 *pop*，而不用显式地保存和恢复符号表。

为“基本块”。所谓基本块就是一个总是逐个顺序执行的语句序列，执行时不会出现分支跳转。

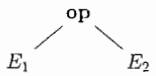
除了创建一个中间表示之外，编译器前端还会检查源程序是否遵循源语言的语法和语义规则。这种检查称为静态检查(static check)，“静态”一般是指“由编译器完成”[⊖]。静态检查确保一些特定类型的程序错误，包括类型不匹配，能在编译过程中被检测并报告。

编译器可以在创建抽象语法树的同时生成三地址代码序列。然而，在通常情况下，编译器实际上并不会创建出存放了整棵抽象语法树的数据结构，它仅仅“假装”构造了一棵抽象语法树，同时生成三地址代码。编译器在分析过程中只会保存将用于语义检查或其他目的的结点及其属性，同时也保存了用于语法分析的数据结构，而不会保存整棵抽象语法树。经过这样的处理，构造三地址代码时要使用到的那部分语法树在需要时都是可用的，一旦不再需要就会被释放。我们将在第 5 章详细讨论这个过程。

2.8.2 语法树的构造

我们将首先给出一个可以创建抽象语法树的翻译方案，然后在 2.8.4 节中说明如何修改这个翻译方案，使得它可以在构造语法树的同时生成三地址代码，或者让它只生成三地址代码。

回顾一下 2.5.1 节，下面的语法树



表示将运算符 **op** 应用于 E_1 和 E_2 所代表的子表达式而得到的表达式。我们可以为任意的构造创建抽象语法树，而不仅仅为表达式创建语法树。每个构造用一个结点表示，其子结点代表此构造中具有语义含义的组成部分。比如，在 C 语言的一个 while 语句

while(*expr*) *stmt*

中，具有语义含义的组成部分是表达式 *expr* 和语句 *stmt*[⊜]。这样的 while 语句的抽象语法树结点有一个运算符，我们称为 **while**，并有两个子结点——分别是 *expr* 和 *stmt* 的抽象语法树。

图 2-39 中的翻译方案为一个有代表性但却很简单的由表达式和语句组成的语言构造出一棵语法树。这个翻译方案中的所有非终结符都有一个属性 n，即语法树的一个结点。这些结点被实现为类 *Node* 的对象。

类 *Node* 有两个直接子类：一个是 *Expr*，代表各种表达式；另一个是 *Stmt*，代表各种语句。每一种语句都有一个对应的 *Stmt* 的子类。比如，运算符 **while** 对应于子类 *While*。一个对应于运算符 **while**，子结点为 x 和 y 的语法树结点可以由如下伪代码创建：

new *While*(x, y)

它通过调用构造函数 *While* 创建了类 *While* 的一个对象，其名称和类名相同。就和构造函数对应于运算符一样，构造函数的参数对应于抽象语法中的运算分量。

当我们研究附录 A 中的详细代码时，我们就会发现各个方法在这个类层次结构中的位置。在本节中，我们将简单讨论一下这些方法中的一小部分。

我们将依次考虑图 2-39 中的每一条产生式和规则。首先，我们将解释定义各种类型语句的产生式，然后再解释用于定义有限几种表达式的产生式。

⊖ 和它的对应“动态”指的是“当程序运行时”。很多语言也会进行某些动态检查。比如，像 Java 这样的面向对象语言有时必须在程序执行时检查类型，因为可能需要根据一个对象的特定子类来决定应该将哪个方法应用于该对象。

⊜ 其中的右括号的唯一作用是将表达式和语句分开。左括号实际上没有任何含义，把它放在那里只是为了让 while 语句看起来顺眼一些，因为如果没有左括号，C 语言中就会出现不匹配的括号对。

```
program → block                { return block.n; }

block → '{' stmts '}'          { block.n = stmts.n; }

stmts → stmts1 stmt            { stmts.n = new Seq(stmts1.n, stmt.n); }
      | ϵ                      { stmts.n = null; }

stmt → expr ;                  { stmt.n = new Eval(expr.n); }
     | if ( expr ) stmt1
                               { stmt.n = new If(expr.n, stmt1.n); }
     | while ( expr ) stmt1
                               { stmt.n = new While(expr.n, stmt1.n); }
     | do stmt1 while ( expr );
                               { stmt.n = new Do(stmt1.n, expr.n); }
     | block                   { stmt.n = block.n; }

expr → rel = expr1             { expr.n = new Assign('=', rel.n, expr1.n); }
     | rel                     { expr.n = rel.n; }

rel → rel1 < add               { rel.n = new Rel('<', rel1.n, add.n); }
    | rel1 <= add              { rel.n = new Rel('≤', rel1.n, add.n); }
    | add                      { rel.n = add.n; }

add → add1 + term              { add.n = new Op('+', add1.n, term.n); }
    | term                     { add.n = term.n; }

term → term1 * factor          { term.n = new Op('*', term1.n, factor.n); }
     | factor                  { term.n = factor.n; }

factor → ( expr )              { factor.n = expr.n; }
       | num                   { factor.n = new Num(num.value); }
```

图 2-39　为表达式和语句构造抽象语法树

语句的抽象语法树

我们在抽象语法中为每一种语句构造定义了相应的运算符。对于以关键字开头的构造，我们将使用这个关键字作为对应的运算符。因此，我们把 **while** 作为 while 语句的运算符，而把 **do** 作为 do-while 语句的运算符。对于条件语句，我们定义了两个运算符 **ifelse** 和 **if**，分别对应于带有和不带有 else 部分的 if 语句。在我们简单的示例性语言中，我们没有使用 **else**，所以仅有一种 **if** 语句。增加 **else** 会在语法分析过程中产生一些问题。我们将在 4.8.2 节中讨论这些问题。

每个语句运算符都有一个对应的同名的类，但是类名的首字符要大写。比如，类 *If* 对应于 **if**。此外，我们还定义了子类 *Seq*，它表示一个语句序列。这个子类对应于文法中的非终结符号 *stmts*。这些类都是 *Stmt* 的子类，而 *Stmt* 又是 *Node* 的子类。

图 2-39 中的翻译方案说明了抽象语法树结点的构建方法。一个典型的用于 if 语句的规则如下：

$$stmt \rightarrow \textbf{if}(expr)\, stmt_1 \{\ stmt.n = \textbf{new}\ If(expr.n, stmt_1.n);\ \}$$

if 语句中具有语义含义的成分是 *expr* 和 $stmt_1$。语义动作将结点 *stmt.n* 定义为子类 *If* 的一个新对象。我们没有给出 *If* 的构造函数的代码。它创建一个标号为 **if**，子结点为 *expr.n* 和 $stmt_1.n$ 的新结点。

表达式语句不以某个关键字开头，所以我们定义了一个新运算符 **eval** 及类 *Eval*（其中 *Eval* 是 *Stmt* 的一个子类）表示表达式语句。相关的规则如下：

$$stmt \rightarrow expr;\ \{\ stmt.n = \textbf{new}\ Eval(expr.n);\ \}$$

在抽象语法树中表示语句块

在图 2-39 中，另一个语句构造是由一系列语句组成的语句块。考虑下面的规则：

$$stmt \rightarrow block \qquad \{\ stmt.n = block.n;\ \}$$
$$block \rightarrow \text{'}\{\text{'}\ stmts\ \text{'}\}\text{'} \qquad \{\ block.n = stmts.n;\ \}$$

第一个规则说明当一个语句是一个语句块时，它的抽象语法树和这个语句块的相同；第二个规则说明非终结符号 *block* 对应的抽象语法树就是该块中的语句序列对应的语法树。

为简单起见，图 2-39 中的语言不包含声明。虽然在附录 A 中包含声明，但我们将看到一个语句块的抽象语法树仍然就是块中的语句序列的抽象语法树。因为声明中的信息已经加入到符号表中，所以它们不需要出现在抽象语法树中。因此，不管它是否包含声明，语句块在中间代码中看起来就是一个普通的语句构造。

一个语句序列的表示方法如下：用一个叶子结点 **null** 表示一个空语句序列，用运算符 **seq** 表示一个语句序列。规则如下：

$$stmts \rightarrow stmts_1\ stmt \ \{\ stmts.n = \textbf{new}\ Seq(stmts_1.n, stmt.n);\ \}$$

例 2.18 在图 2-40 中，我们可以看到表示一个语句块或语句列表的语法树的一部分。列表中有两个语句。第一个语句是一个 if 语句，第二个语句是 while 语句。我们没有显示在这个语句列表之上的那部分抽象语法树，并且将各棵子树用三角形表示，包括这个语句列表中对应于 if 语句和 while 语句的条件的抽象语法树，以及对应于这两个语句的子语句的语法树。 □

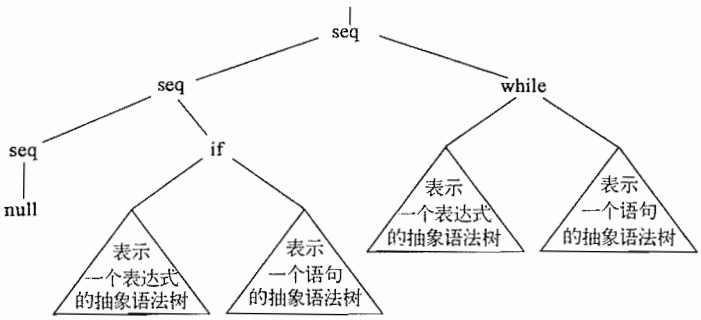


图 2-40 由一个 if 语句和一个 while 语句组成的语句列表的语法树的一部分

表达式的语法树

在以前的章节中，我们用三个非终结符号 *expr*、*term* 和 *factor* 使得乘法 * 相对加法 + 具有较高的优先级。我们在 2.2.6 节中指出，非终结符号的数目正好比表达式中优先级的层数多一。在图 2-39 中，我们增添了两个同优先级的比较运算符 < 和 <=，同时也保留了 + 和 * 运算符，故我们增加了一个新的非终结符号 *add*。

抽象语法允许我们将“相似的”运算符分为一组，以减少在实现表达式时需要处理的不同情况和需要设计的子类。在本章中，“相似的”意指运算符的类型检查规则和代码生成规则相近。比如，运算符 + 和 * 通常分为一组，因为它们可以用同一种方式进行处理——它们对运算分量类型的要求是一样的，且它们都会生成一个将一个运算符应用到两个数值之上的三地址指令。一般来说，在抽象语法中对运算符分组是根据编译器后期处理的需要来决定的。图 2-41 中的表描述了几种常见 Java 运算符的具体语法和抽象语法之间的对应关系。

具体语法	抽象语法
=	assign
||	cond
&&	cond
== !=	rel
< <= >= >	rel
+ -	op
* / %	op
!	not
$^{-}_{unary}$	minus
[]	access

图 2-41 几种常见 Java 运算符的具体语法和抽象语法

在具体语法中，几乎所有的运算符都是左结合的，只有赋值运算符 = 是右结合的。同一行中

的运算符具有同样的优先级，也就是说 == 和 != 具有同样的优先级。各行是按照优先级递增的方式排列的，比如 == 比 && 或 = 的优先级更高。$-_{unary}$ 中的下标 *unary* 用于区分单目减号（比如 -2 中的符号）和双目减号（比如 2 - a 中的符号）。运算符[]表示数组访问，例如 a[i]。

图中“抽象语法”列描述了运算符的分组方法。赋值运算符 = 所在的组仅包含它自己。组 **cond** 包含了条件布尔运算符 && 和 ||。组 **rel** 包含 == 和 < 所在行中的各个关系比较运算符。组 **op** 包含诸如 + 和 * 这样的算术运算符。单目减、逻辑非和数组访问运算符各自为一组。

图 2-41 中具体语法和抽象语法之间的映射关系可以通过编写翻译方案来实现。图 2-39 中的非终结符号 *expr*、*rel*、*add*、*term* 和 *factor* 的产生式描述了一些运算符的具体语法。这些运算符是图 2-41 中的运算符的一个代表性子集。这些产生式中的语义动作创建出相应的语法树结点。比如，规则

$$term \rightarrow term_1 * factor \ \{ \ term.n = \textbf{new } Op('*', term_1.n, factor.n); \ \}$$

创建了类 *Op* 的结点，这个类实现了图 2-41 中被分在 **op** 组中的运算符。构造函数 *Op* 的参数中包含了一个 '*'，它指明了实际的运算符。它的参数还包括对应于子表达式的结点 $term_1.n$ 和 $factor.n$。

2.8.3 静态检查

静态检查是指在编译过程中完成的各种一致性检查。这些检查不但可以确保一个程序被顺利地编译，而且还能在程序运行之前发现编程错误。静态检查包括：

- *语法检查*。语法要求比文法中的要求的更多。例如下面的这些约束：任何作用域内同一个标识符最多只能声明一次，一个 break 语句必须处于一个循环或 switch 语句之内。这些约束都是语法要求，但是它们并没有包括在用于语法分析的文法中。
- *类型检查*。一种语言的类型规则确保一个运算符或函数被应用到类型和数量都正确的运算分量上。如果必须要进行类型转换，比如将一个浮点数与一个整数相加时，类型检查器就会在语法树中插入一个运算符来表示这个转换。下面我们将使用常用的术语“自动类型转换”来讨论类型转换的问题。

左值和右值

现在我们考虑一些简单的静态检查，它们可以在源程序的抽象语法树构造过程中完成。一般来说，在进行复杂的静态检查时，首先要生成源程序的某个中间表示，然后再分析这个中间表示。

赋值表达式左部和右部的标识符的含义是不一样的。在下面的两个赋值语句

```
i = 5;
i = i + 1;
```

中，表达式的右部描述了一个整数值，而左部描述的是用来存放该值的存储位置。术语*左值*（l-value）和*右值*（r-value）分别表示可以出现在赋值表达式左部和右部的值。也就是说，右值是我们通常所说的“值”，而左值是存储位置。

静态检查要确保一个赋值表达式的左部表示的是一个左值。一个像 i 这样的标识符是一个左值，像 a[2]这样的数组访问也是左值，但 2 这样的常量不可以出现在一个赋值表达式的左部，因为它有一个右值，但不是左值。

类型检查

类型检查确保一个构造的类型符合其上下文对它的期望。比如说，在 if 语句

if(*expr*) *stmt*

中，期望表达式 *expr* 是 **boolean** 型的。

类型检查规则按照抽象语法中运算符/运算分量的结构进行描述。假设运算符 **rel** 表示关系运算符，如 <= 。那么运算符组 **rel** 的类型规则是：它的两个运算分量必须具有相同的类型，而其结果为布尔类型。用属性 *type* 来表示一个表达式的类型，令 E 表示将 **rel** 应用于 E_1 和 E_2 的表达式。那么 E 的类型检查可以在创建它对应的抽象语法树的结点时进行，执行如下所示的代码即可：

if(E_1. *type* == E_2. *type*) E. *type* = boolean;
else error ;

即使在下面的情况下，仍可以运用将实际类型和期望类型相匹配的思想：

- *自动类型转换*。当一个运算分量的类型被自动转换为运算符所期望的类型时，就发生了*自动类型转换*(coercion)。在一个像 2 * 3.14 这样的表达式中，常见的转换是将整数 2 转换为一个等值的浮点数 2.0，然后对得到的两个浮点运算分量执行相应的浮点运算。程序设计语言的定义指明了允许的自动类型转换方式。比如，上面讨论的 rel 的实际规则可能是这样的；E_1. *type* 和 E_2. *type* 可以被转换成相同的类型。如果是那样，把一个整数和一个浮点数比较就是合法的。
- *重载*。Java 中的运算符 + 应用于整数运算分量时表示相加，而应用于字符串型运算分量时表示连接。如果一个符号在不同上下文中有不同的含义，那么我们说这个符号是*重载*(overloading)的。因此，在 Java 中 + 是重载的。我们可以通过已知的运算分量类型和结果类型来判断一个重载的运算符的含义。比如，如果我们知道 x、y 或 z 中的任意一个是字符串类型，那么表达式 z = x + y 中的运算符 + 的含义就是连接。然而，如果我们还知道其中另一个运算分量是整型的，那么我们就找到了一个类型错误，+ 的这次使用就没有意义。

2.8.4 三地址码

一旦抽象语法树构造完成，我们就可以计算树中各结点的属性值并执行各结点中的代码片段，进行进一步的分析和综合。我们将说明如何通过遍历语法树来生成三地址代码。具体地说，我们将显示如何编写一个抽象语法树的函数，并且同时生成必要的三地址代码。

三地址指令

三地址代码是由如下形式的指令组成的序列

x = y **op** z

其中 x、y 和 z 可以是名字、常量或由编译器生成的临时量；而 **op** 表示一个运算符。

数组将由下面的两种变体指令来处理：

x [y] = z
x = y [z]

前者将 z 的值保存到 x [y]所指示的位置上，而后者则将 y [z]的值放到位置 x 上。

三地址指令将被顺序执行，但是当遇到一个条件或无条件跳转指令时，执行过程就会跳转。我们选择下面的指令来控制程序流：

ifFalse x goto L	如果 x 为假，下一步执行标号为 L 的指令
ifTrue x goto L	如果 x 为真，下一步执行标号为 L 的指令
goto L	下一步执行标号为 L 的指令

在一个指令前加上前缀 L：就表示将标号 L 附加到该指令。同一指令可以同时拥有多个标号。

最后，我们还需要一个拷贝值的指令。如下的三地址指令将 y 的值拷贝至 x 中：

x = y

语句的翻译

通过利用跳转指令实现语句内部的控制流，我们就可以将语句转换成为三地址代码。图 2-42 的代码布局说明了对语句 **if** *expr* **then** $stmt_1$ 的翻译。该代码布局中的跳转指令

```
ifFalse x goto after
```

将在 *expr* 的值为 **false** 时跳过语句 $stmt_1$ 对应的翻译结果。其他语句的翻译方法是类似的：我们将使用一些跳转指令在其各个组成部分对应的代码之间进行跳转。

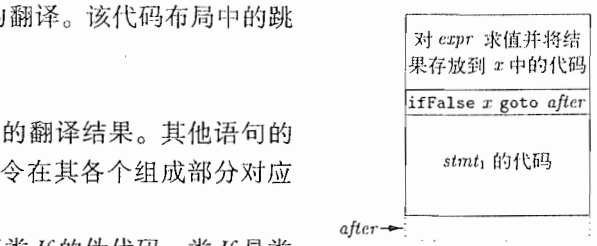


图 2-42 if 语句的代码布局

为了具体说明，我们在图 2-43 中给出了类 *If* 的伪代码。类 *If* 是类 *Stmt* 的一个子类，对应于其他语句的类也是 *Stmt* 的子类。*Stmt* 的每一个子类（这里是 *If*）都有一个构造函数及一个为此类语句生成三地址代码的函数 *gen*。

```
class If extends Stmt {
        Expr E; Stmt S;
        public If(Expr x, Stmt y) { E = x; S = y; after = newlabel(); }
        public void gen() {
                Expr n = E.rvalue();
                emit( "ifFalse " + n.toString() + " goto " + after);
                S.gen();
                emit(after + ":");
        }
}
```

图 2-43 类 *If* 中的函数 *gen* 生成三地址代码

图 2-43 中的构造函数 *If* 构建了 if 语句的语法树结点。它有两个参数，一个表达式结点 x 和一个语句结点 y。它们被分别存放在属性 E 和 S 中。同时，这个构造函数调用了函数 *newlable*( )，给属性 *after* 赋予一个唯一的新标号。这个标号将按照图 2-42 所示的布局被使用。

一旦源程序的整个抽象语法树被创建完毕，函数 *gen* 在此抽象语法树的根结点处被调用。在我们的简单语言中，一个程序就是一个语句块，所以这棵抽象语法树的根结点就代表这个语句块中的语句序列。所有的语句类都有一个 *gen* 函数。

图 2-43 中类 *If* 的 *gen* 函数的伪代码具有代表性。它调用 *E*. *rvalue*( ) 函数来翻译表达式 E（即作为 if 语句的组成部分的布尔值表达式），并保存 *E*. *rvalue*( ) 返回的结果结点。我们稍后会讨论表达式的翻译。然后，*gen* 函数发生一个条件跳转指令，并且调用 *S*. *gen*( ) 来翻译子语句 S。

表达式的翻译

我们将考虑包含二目运算符 **op**、数组访问和赋值运算，并包含常量及标识符的表达式，以此来说明对表达式的翻译。为了简单起见，我们要求在数组访问 $y[z]$ 中，y 必须为标识符[⊖]。关于表达式的中间代码生成的详细讨论请见 6.4 节。

我们将采用一种简单的方法，为一个表达式的语法树中的每个运算符结点都生成一个三地址指令。不需要为标识符和常量生成任何代码，因为它们可以作为地址出现在指令中。如果一个结点 x 的类为 *Expr*，其运算符为 **op**，我们就发出一个指令来计算结点 x 上的值，并将此值存放到一个由编译器生成的“临时”名字（比如 t）中。因此，`i - j + k` 会被翻译成为两条指令

⊖ 这个简单语言支持 `a[a[n]]`，但是不支持 `a[m][n]`。请注意，`a[a[n]]`是形如 `a[E]`的访问，其中的 E 是 `a[n]`。

```
t1 = i - j
t2 = t1 + k
```

在处理数组访问及赋值运算时要区分左值和右值。例如，对于 2 * a[i]，可以通过计算 a[i]的右值并存放在一个临时量中而得到翻译结果，如下所示：

```
t1 = a [ i ]
t2 = 2 * t1
```

但是，当 a[i]出现在一个赋值表达式的左边时，我们不能简单地以一个临时量来替换 a[i]。

我们的简单方法使用了两个函数 *lvalue* 及 *rvalue*，它们分别显示在图 2-44 和图 2-45 中。当函数 *rvalue* 被应用于一个非叶子结点 x 时，它生成一些指令，这些指令对 x 求值并存放到一个临时量中，然后该函数返回一个表示此临时量的新结点。当函数 *lvalue* 被应用于一个非叶子结点 x 时，它也会生成一些指令，这些指令计算 x 之下的各个子树。然后这个函数返回代表 x 的“地址”的新结点。

因为函数 *lvalue* 要处理的情况相对较少，我们首先对它进行描述。当将它应用于一个结点 x 时，如果此结点对应于一个标识符(即 x 的类是 *Id*)，那么它直接返回 x。在我们的简单语言中，除此之外只存在一种情况会使一个表达式拥有左值，即结点 x 代表一个数组访问，比如 a[i]。在这种情况下，结点 x 形如 *Access*(y, z)，其中类 *Access* 是类 *Expr* 的子类，y 表示被访问数组的名字，而 z 表示被访问元素在该数组中的偏移量(下标)。在图 2-44 所示的伪代码中，函数 *lvalue* 会在必要时调用 *rvalue*(z)来生成计算 z 的右值的指令。然后它创建并返回一个新的 *Access* 结点，此结点包含两个子结点，分别对应于数组名 y 及 z 的右值。

```
Expr lvalue(x : Expr) {
    if ( x是一个Id 结点 ) return x;
    else if ( x 是一个Access(y, z) 结点 , 且 y是一个 Id 结点 ) {
        return new Access(y, rvalue(z));
    }
    else error;
}
```

图 2-44 函数 *lvalue* 的伪代码

例 2.19 当结点 x 表示数组访问 a[2 * k]时，*lvalue*(x)的调用将生成指令

```
t = 2 * k
```

并返回一个表示 a[t]的左值的新结点 x'，其中 t 是一个新的临时名字。

具体来说，lvalue 函数将运行到代码

return new *Access*(y, *rvalue*(z));

处，此时 y 是对应于 a 的结点，z 是对应于表达式 2 * k 的结点。对 *rvalue*(x)的调用生成了表达式 2 * k 的代码(即三地址语句 t = 2 * k)，并返回表示临时名字 t 的新结点 z'。这个结点就成为新的 *Access* 结点 x'的第二个字段的值。 □

图 2-45 中的函数 *rvalue* 生成指令并返回一个(可能是新生成的)结点。当 x 代表一个标识符或常量时，*rvalue* 返回 x 本身。在其他情况下，它都返回一个对应于新的临时名字 t 的 *Id* 结点。各种情况的处理如下：

- 如果结点 x 表示 y **op** z，则代码首先计算 y' = *rvalue* (y) 及 z' = *rvalue* (z)。它创建一个新的临时名字 t 并产生一个指令 t = y' **op** z'(更精确地说，生成了一个由代表 t、y′、**op** 和 z′的字符串组合而成的指令字符串)。它返回一个对应于标识符 t 的结点。
- 如果结点 x 表示一个数组访问 y[z]，我们可以复用函数 *lvalue*。函数调用 *lvalue*(x)返回一个数组访问 y[z']，其中 z'代表一个标识符，它保存了该数组访问的偏移量。函数 *rvalue*

会创建一个临时变量 t，并按照 $t = y[z']$ 生成一个指令，最后返回一个对应于 t 的结点。

- 如果 x 表示 $y = z$，那么代码将首先计算 $z' = rvalue(z)$。它生成一条计算 $lvalue(y) = z'$ 的指令，并返回结点 z'。

```
Expr rvalue(x : Expr) {
    if ( x是一个 Id 或者 Constant 结点 ) return x;
    else if ( x是一个 Op(op, y, z) 或者 Rel(op, y, z) 结点 ) {
        t = 新的临时名字;
        生成对应于 t = rvalue(y) op rvalue(z)的指令串;
        return 一个代表 t 的新结点;
    }
    else if ( x是一个 Access(y, z) 结点 ) {
        t = 新的临时名字;
        调用 lvalue(x), 它返回一个 Access(y, z')的结点;
        生成对应于 t = Access(y, z')的指令串;
        return 一个代表 t 的新结点;
    }
    else if ( x是一个 Assign(y, z) 结点 ) {
        z' = rvalue(z);
        生成对应于 lvalue(y) = z'的指令串;
        return z';
    }
}
```

图 2-45　函数 *rvalue* 的伪代码

例 2.20　当将函数 *rvalue* 应用于

```
a[i] = 2*a[j-k]
```

的语法树时，它将生成

```
t3 = j - k
t2 = a [ t3 ]
t1 = 2 * t2
a [ i ] = t1
```

这棵语法树的根是 *Assign* 结点，它的第一个参数是 `a[i]`，第二个参数是 `2 * a[j - k]`。因此，适用 *rvalue* 函数的第三种情况，函数被递归地应用于 `2 * a[j - k]`。这棵子树的根结点是表示 * 的 *Op* 结点，因此 *rvalue* 首先创建一个临时变量 `t1`，然后处理左运算分量 2，再后是右运算分量。常量 2 没有生成三地址代码，*rvalue* 返回它的右值，即一个值为 2 的 *Constant* 结点。

右运算分量 `a[j - k]` 是一个 *Access* 结点，因此 *rvalue* 创建一个新的临时变量 `t2`，然后在这个结点上调用 *lvalue* 函数。函数 *rvalue* 被递归地调用来处理表达式 `j - k`。这个调用的副作用是创建临时变量 `t3`，然后生成三地址语句 `t3 = j - k`。接着，函数的执行返回到正在处理 `a[j - k]` 的函数 *lvalue* 的活动中，临时名字 `t2` 被赋予整个数组访问表达式的右值，即 `t2 = a[t3]`。

现在，我们返回到处理 *Op* 结点 `2 * a[j - k]` 的 *rvalue* 的活动中。这次调用已经创建了临时变量 `t1`。作为一个副作用，*rvalue* 生成了一条执行这个乘法表达式的三地址指令。最后，应用于整个表达式的 *rvalue* 的调用活动在最后调用 *lvalue* 来处理左部 `a[i]`，然后生成了一条三地址指令 `a[i] = t1`。这个指令把这个赋值表达式的右部赋给左部。 □

改进表达式的代码

使用如下几种方法，我们可以改进图 2-45 中的函数 *rvalue*，使它生成更少的三地址指令：

- 在之后的优化阶段减少拷贝指令的数目。例如，对于指令 `t = i + 1; i = t`，如果 `t` 没有再被使用，我们就可以将它们合并为 `i = i + 1`。

- 充分考虑上下文的情况，在最初生成指令时就减少生成的指令。例如，如果一个三地址赋值指令的左部是一个数组访问 a[t]，那么其右部必然是一个名字、常量或临时变量，它们都只使用了一个地址。但如果左部是一个名字 x，那么其右部可以是一个使用两个地址的运算 y op z。

我们可以按照如下的方式来避免一些拷贝指令。首先修改翻译函数，使之生成一个部分完成的指令，该指令只进行计算，比如计算 j + k，但并不确定将结果保存在哪里，而是用 **null** 来替代结果地址：

```
null = j + k                                    (2.8)
```

随后，这个空的结果地址会被替换为适当的标识符或临时量。如果 j + k 位于一个赋值表达式的右部，如 i = j + k，那么 **null** 就会被替换为标识符。此时(2.8)就变成

```
i = j + k
```

但如果 j + k 是一个子表达式，比如它在 j + k +1 中，那么这个空的结果地址会被替换成一个新的临时变量 t，并且生成一个新的部分指令：

```
t = j + k
null = t + 1
```

很多编译器想方设法使得它生成的代码和汇编代码专家手写的一样好，甚至更好。如果使用第 9 章中讨论的代码优化技术，那么一个有效的策略是首先使用一个简单的中间代码生成方法，然后依靠代码优化器来消除不必要的指令。

2.8.5 2.8 节的练习

练习 2.8.1：C 语言和 Java 语言中的 for 语句具有如下形式：

for($expr_1$; $expr_2$; $expr_3$) *stmt*

第一个表达式在循环之前执行，它通常被用来初始化循环下标。第二个表达式是一个测试，它在循环的每次迭代之前进行。如果这个表达式的结果变成 0，就退出循环。循环本身可以被看作语句{ *stmt* $expr_3$; }。第三个表达式在每一次迭代的末尾执行，它通常用来使循环下标递增。故 for 语句的含义类似于

$expr_1$; while($expr_2$) { *stmt* $expr_3$; }

仿照图 2-43 中的类 *If*，为 for 语句定义一个类 *For*。

练习 2.8.2：程序设计语言 C 中没有布尔类型。试说明 C 语言的编译器可能使用什么方法将一个 if 语句翻译成为三地址代码。

2.9 第 2 章总结

本章介绍的语法制导翻译技术可以用于构造如图 2-46 所示的编译器的前端。

- 构造一个语法制导翻译器要从源语言的文法开始。一个文法描述了程序的层次结构。文法的定义使用了称为*终结符号*的基本符号和称为*非终结符号*的变量符号。这些符号代表了语言的构造。一个文法的规则，即*产生式*，由一个作为*产生式头*或*产生式左部*的非终结符，以及称为*产生式体*或*产生式右部*的终结符号/非终结符号序列组成。文法中有一个非终结符被指派为*开始符号*。
- 在描述一个翻译器时，在程序构造中附加属性是非常有用的。*属性*是指与一个程序构造关联的任何量值。因为程序构造是使用文法符号来表示的，因此属性的概念也被扩展到文法符号上。属性的例子包括与一个表示数字的终结符号 **num** 相关联的整数值，或与一个表示标识符的终结符号 **id** 相关联的字符串。

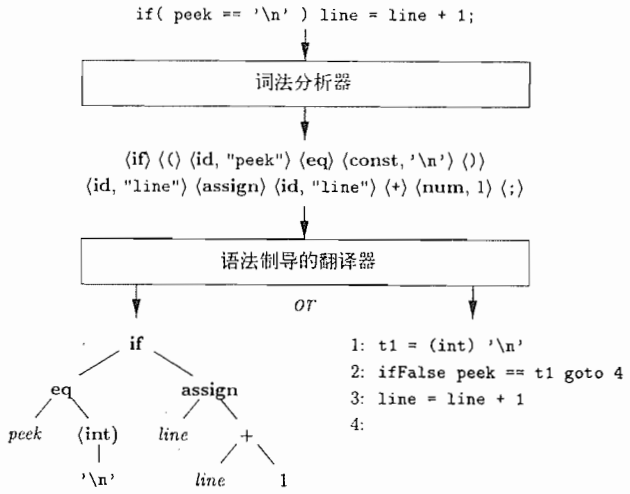


图 2-46 一个语句的两种可能的翻译结果

- 词法分析器从输入中逐个读取字符，并输出一个词法单元的流，其中词法单元由一个终结符号以及以属性值形式出现的附加信息组成。在图 2-46 中，词法单元被写成用⟨⟩括起的元组。词法单元⟨**id**, "peek"⟩由终结符号 **id** 和一个指向包含字符串"peek"的符号表条目的指针构成。翻译器使用符号表来存放保留字和已经遇到的标识符。
- 语法分析要解决的问题是指出如何从一个文法的开始符号推导出一个给定的终结符号串。推导的方法是反复将某个非终结符替换为它的某个产生式的体。从概念上讲，语法分析器会创建一棵语法分析树。该树的根结点的标号为文法的开始符号，每个非叶子结点对应于一个产生式，每个叶子结点的标号为一个终结符号或空串 ϵ。语法分析树推导出由它的叶子结点从左到右组成的终结符号串。
- 使用被称为预测语法分析法的自顶向下(从语法分析树的根结点到叶子结点)方法可以手工建立高效的语法分析器。预测分析器有对应于每个非终结符的子过程。该过程的过程体模拟了这个非终结符号的各个产生式。只要在输入流中向前看一个符号，就可以无二义地确定该过程体中的控制流。其他语法分析方法见第 4 章。
- 语法制导翻译通过在文法中添加规则或程序片段来完成。在本章中，我们只考虑了综合属性。任意结点 x 上的一个综合属性的值只取决于 x 的子结点(如果有的话)上的属性值。语法制导定义将规则和产生式相关联，这些规则用于计算属性值。语法制导的翻译方案在产生式体中嵌入了称为语义动作的程序片段。这些语义动作按照语法分析中产生式的使用顺序执行。
- 语法分析的结果是源代码的一种中间表示形式，称为中间代码。图 2-46 列出了中间代码的两种主要形式。抽象语法树中的各个结点代表了程序构造，一个结点的子结点给出了该构造有意义的子构造。另一种表示方法是三地址代码，它是一个由三地址指令组成的序列，其中每个指令只执行一个运算。
- 符号表是存放有关标识符的信息的数据结构。当分析一个标识符的声明的时候，该标识符的信息被放入符号表中。当在后来使用这个标识符时，比如它作为一个表达式的因子使用时，语义动作将从符号表中获取这些信息。