第5章

中间代码生成

在编译过程的词法和语法阶段对源程序的结构进行了分析。在第4章对静态语义进行检查的基础上，本章将进一步对每个语法上正确的语法单位的内部逻辑含义加以解释，生成中间代码。执行中间代码生成的程序称为中间代码生成程序，也称为中间代码生成器。

为了能够将高级语言翻译为中间代码，必须解决以下三个问题：首先是中间代码的形式是什么样的，其次是高级语言(或者其中的语法单位)的语义如何表示，再次是翻译方法是什么。本章使用第4章介绍的属性文法来表示各个语法单位的含义，利用语法制导的语义处理方法来进行翻译。因此，本章首先介绍常见的几种中间代码表示形式，然后介绍Sample语言中几种常见语法单位的中间代码翻译方法，最后介绍Sample语言中间代码生成器的设计。

5.1 中间代码生成概述

在第4章中介绍了利用属性文法来描述Sample语言的语义。通过对Sample语言中各种类型名字的声明构造属性文法，进行语义处理，建立了符号表，将源程序中出现的各个名字的相关信息登记在符号表中，然后对各个语法单位的静态语义进行了检查。

这就保证了源程序中各个语法单位的静态语义是正确的，下面就可以执行真正的翻译了。翻译就是要“忠实地”将源代码所表达的含义用另一种形式的代码表达出来，另一种形式的代码可以是某种格式统一的中间代码，或者是目标代码。中间代码是编译程序内部使用的对源程序的一种表示形式，不依赖于目标机，复杂性介于源语言和机器语言之间，从它能更容易地生成目标代码。

中间代码生成这一阶段在编译过程中是可以没有的，如果没有这一阶段，在语法和静态语义正确的基础上就可以直接产生目标代码。直接生成机器语言或汇编语言形式的目标代码的优点是编译时间短且无须中间代码到目标代码的翻译，但这样生成的目标代码执行效率和质量都较低，移植性差。因此，生成中间代码是有益处的，许多编译程序都选择生成中间代码，即翻译为等效的中间代码。生成中间代码的优点如下。

(1) 便于进行与机器无关的代码优化工作，提高目标代码的效率。

(2) 使编译程序改变目标机更容易。

(3) 使编译程序的结构在逻辑上更为简单明确。以中间语言为界面，编译前端和后端的接口更清晰，也更易于生成不同语言在不同机器上的编译程序。

中间代码生成在编译器中的地位如图5.1(a)所示。

生成中间代码是为了缩小高级语言和机器语言之间的语义鸿沟，如果有必要，则可以设

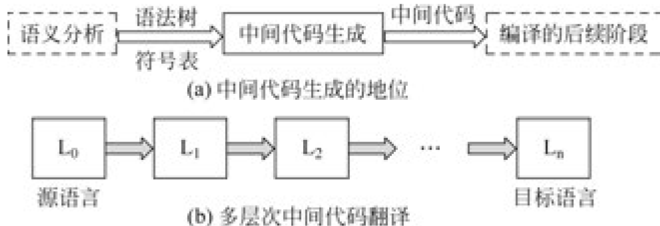

图 5.1 中间代码的翻译

计多层中间语言,采取逐层翻译为不同的中间语言的方式,如图 5.1(b)所示。其中,L_0 是源语言,L_n 是目标语言,即机器语言。可以采取首先将 L_0 翻译为 L_1,再将 L_1 翻译为 L_2,…,最后将 L_{n-1} 翻译为 L_n。L_1、L_2、…、L_{n-1} 都是中间语言。相邻层之间的语义更接近。为了反映不同层次中间语言的特征,每种中间语言可以用不同的形式来表示。

翻译的依据是语法单位的含义,能够进行翻译生成另一种形式的代码的语法单位包括表达式、各种可执行语句、函数和程序等。因此,需要利用属性文法来描述待翻译的语法单位的真正含义。这就需要针对每个语法单位的文法,定义各个文法符号的相关属性,用属性的计算和传递规则来定义各个产生式表达的含义,作为产生式的语义规则。

将源程序翻译为中间代码的方法很多。第 4 章介绍了基于属性文法描述语义的基础上,采用语法制导的语义处理方法,本章继续利用语法制导的语义处理方法来生成各个语法单位的中间代码。语法制导的翻译方法是指对语义的处理是由语法结构驱动的。当语法分析正确建立了语法树后,就可以遍历语法树,在树的各个结点上,根据相应的语义规则进行语义计算,生成中间代码。如果属性文法满足 L-属性文法的要求,还可以将语法分析和翻译同步进行。

这样,语法制导的翻译就和语法分析方法有关。利用 S-属性文法进行自下而上的语法制导的翻译时,遵循“句柄归约在先,语义规则调用在后”的原则。因此,在 5.3~5.5 节介绍为各个语法单位设计属性文法时,很多时候会为了适应自下而上的语法分析,对文法进行改造,利用和语法分析同步操作的语义栈来存放归约过程中产生的语义信息。而在自上而下的语法制导翻译时,如递归下降的翻译器,在利用产生式推导时递归调用其他文法符号的函数,可以利用函数内部的局部变量来表示文法符号的语义信息,利用函数之间的参数传递来传递语义信息,利用产生式的语义规则来计算语义信息,这就不必为了存储和计算语义信息来改造文法。具体方法将在 5.6 节介绍。

5.2 中间代码

中间代码是源程序的不同表示形式,有时候又称为中间语言、中间表示。例如,P_code 是用于 Pascal 语言编译器的一种中间代码,Java 编译器输出的 Bytecode 也是一种中间代码,是 Java 虚拟机的输入。中间代码有多种形式,常见的有后缀式(逆波兰式)、三地址代码(Three Address Code,TAC)(包括三元式和四元式)、抽象语法树(Abstract Syntax Tree,AST)、有向无环图(Directed Acyclic Graph,DAG)等,用得较多的是三地址代码。

5.2.1 逆波兰式

逆波兰表示法是波兰逻辑学家卢卡西维奇(Lukasiewicz)发明的一种表示表达式的方法,这种表示法把运算对象(操作数)写在前面,把运算符写在后面,因而又称后缀式表示法(Postfix)。

例 5.1 将下列各表达式用逆波兰式表示(见表 5.1)。

表 5.1 例 5.1 表

表达式	逆波兰式
a+b	ab+
(a+b)*c	ab+c*
a+b*c	abc*+
a=b*c+b*d	abc*bd*+=

表达式 E 的后缀形式表示的递归定义如下。

(1) 如果 E 是变量或常数,则 E 的后缀表示就是 E 自身。

(2) 如果 E 为 E_1 op E_2 形式,则它的后缀表示为 $E_1'E_2'$op。其中,op 是双目运算符,E_1'、E_2'分别是 E_1 和 E_2 的后缀表示。若 op 为单目运算符,则视 E_1 和 E_1'为空。

(3) 如果 E 为(E_1)形式,则 E 的后缀表示就是 E_1 的后缀表示。

上述递归定义的实质是:在后缀表示中,操作数出现的顺序与原来一致,运算符按运算的先后顺序放入相应的操作数之后(即运算符相对于运算对象的顺序发生了变化),不需要用括号来规定运算顺序。例如,把(a+b)*c 表示成 ab+c*。

后缀表示的优点是计算机易于处理。其常用方法是使用一个栈,自左至右扫描后缀表达式,每碰到运算对象就压栈,每碰到 K 目运算符就把它作用于栈顶的 K 个运算对象,并用运算的结果(即一个新的运算对象)来取代栈顶的 K 个运算对象。

例 5.2 B@CD*+(中缀表示形式为-B+C*D,@表示单目减)的计算过程为:

(1) B 进栈。

(2) 对栈顶元素施行@运算,将结果代替栈顶,即-B 的值置于栈顶。

(3) C 进栈。

(4) D 进栈。

(5) 栈顶两元素相乘,两元素退栈,相乘结果置栈顶。

(6) 栈顶两元素相加,两元素退栈,相加结果进栈,现在栈顶存放的是整个表达式的值。

由于后缀式表示的简洁和计算的方便,特别适用于解释执行的程序设计语言的中间表示,也便于具有堆栈体系的计算机的目标代码生成。

5.2.2 三地址代码

三地址代码是指每条代码包含一个运算和三个地址,两个地址用于存放运算对象,一个地址用于存放运算结果。其一般形式为:

$$x = y \text{ op } z$$

其中，y和z为名字、常量或编译时产生的临时变量；x为名字或临时变量；op为运算符，如定点运算符、浮点运算符和布尔运算符等。三地址代码类似于汇编代码中的三地址指令。由于每条三地址代码只含有一个运算符，因此多个运算符组成的表达式必须用三地址代码序列来表示，如表达式 x+y*z 的三地址代码为：

$$T_1 = y * z$$

$$T_2 = x + T_1$$

其中，T_1 和 T_2 是编译时产生的临时变量。在实际实现中，用户定义的名字将由指向符号表中该名字项的指针所取代。

三地址代码也可以表示多种语句形式，有符号标号和各种控制流语句等。常用的三地址代码有以下几种。

（1）x=y op z，其中op为双目的算术运算符或布尔运算符。其含义是y和z进行op所指定的操作后，结果存放到x中。

（2）x=op z，其中op为单目运算符，如单目求负@、逻辑非not等。其含义是对z进行op指定的操作，结果存放到x中。

（3）x=y，将y的值赋给x。

（4）无条件转移goto L，即直接跳转执行标号为L的三地址代码。

（5）条件转移if x rop y goto L或if a goto L。在第一种形式中，rop为关系运算符(如<、<=、==、<>、>和>=)，若x和y满足关系rop就转去执行标号为L的三地址代码，否则继续按顺序执行。在第二种形式中，a为单个的表达式，若a为非0，则执行标号为L的三地址代码，否则继续顺序执行。

三地址代码是中间代码的一种抽象形式。在编译程序中，三地址代码的具体实现通常有四元式和三元式。

1. 四元式

一个四元式是具有四个域的结构体，表示为：

(op,arg1,arg2,result)

其含义是arg1和arg2进行op指定的操作，结果存放到result中。其中，op为运算符；arg1、arg2及result分别为第一、第二运算对象和结果，它们可以是用户定义的变量或临时变量，arg1、arg2还可以是常量。如果op是单目运算符，只需要一个运算对象，则arg2用空格或用下画线来代替，本书都使用空格，只是为它留出位置。四元式的表示形式非常类似于汇编或机器语言的三地址指令，这更有利于下一步翻译为目标代码。

常见的三地址代码与四元式的对应关系如表5.2所示。

表5.2 常见的三地址代码与四元式的对应关系

三地址代码	四元式	三地址代码	四元式
x=y op z	(op,y,z,x)	goto L	(j, , ,L)
x=op z	(op,z, ,x)	if x rop y goto L	(jrop,x,y,L)
x=y	(=,y, ,x)	if x goto L	(jnz,x, ,L)

例5.3 写出赋值表达式 a=b*(-c)+b*(-c)的四元式。结果如表5.3(a)所示。

表5.3 赋值表达式 a=b*(-c)+b*(-c)的四元式和三元式表示

(a) 四元式

	op	arg1	arg2	result
(0)	@	c		T_1
(1)	*	b	T_1	T_2
(2)	@	c		T_3
(3)	*	b	T_3	T_4
(4)	+	T_2	T_4	T_5
(5)	=	T_5		a

(b) 三元式

	op	arg1	arg2
(0)	@	c	
(1)	*	b	(0)
(2)	@	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

2. 三元式

为了避免把临时变量填入符号表中,可以通过计算这个临时变量的值的代码位置来引用这个临时变量,这样表示的三地址代码称为三元式。三元式是只具有三个域的结构体,表示为:

(op,arg1,arg2)

其中,op 是运算符;arg1、arg2 可以的变量或临时变量,也可以是常量,还可以指向三元式表中的某一个三元式编号。同样地,如果 op 是单目运算符,arg2 用空格来代替。

例 5.4 写出赋值表达式 a=b*(-c)+b*(-c)的三元式。结果如表 5.3(b)所示。

在三元式表示中,每个语句的位置同时有两个作用:一是可作为该三元式的结果被其他三元式引用;二是三元式的位置顺序即为运算顺序。在代码优化阶段需要调整三元式的运算顺序时会很麻烦,它意味着必须改变其中一系列指针的值。因此,变动一张三元式表是很困难的。

对四元式来说,引用另一语句的结果可以通过引用该语句的 result(通常是一个临时变量)来实现。它不存在语句位置同时具有两种功能的现象,代码调整时要做的改动只是局部的,因此,当需要对中间代码表进行优化处理时,四元式比三元式方便得多。

5.2.3 抽象语法树

通过语法分析,已经为源程序建立了语法树。抽象语法树是语法树的一种简化形式,是源程序的抽象语法结构的树状表示,树的每个结点都表示源代码中的一种结构,之所以说是抽象的,是因为抽象语法树并不会表示出真实语法出现的每一个细节,例如,嵌套括号被隐含在树的结构中,并没有以结点的形式呈现。

抽象语法树不依赖于源语言的语法,与语法树的建立过程无关。对上下文无关文法进行语法分析时,经常会对文法进行一些等价变换(如消除左递归、回溯、二义性等),这给文法分析引入了一些多余的成分,对后续阶段造成不利影响。抽象语法树去掉那些对后续阶段的翻译不必要的信息,得到源程序的一种有效的中间表示,作为编译器前后端的一个清晰的接口。

抽象语法树在很多领域有广泛的应用,如浏览器、智能编辑器、编译器等。

在抽象语法树中,运算符和关键字都作为树的内部结点,运算对象作为树的叶子结点出现。如产生式 S→if B then S1 else S2 的抽象语法树如图 5.2 所示,表达式 3*5+4 的抽象语法树如图 5.3 所示。

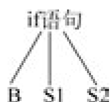

图 5.2 产生式 S→if B then S1 else S2 的抽象语法树

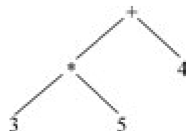

图 5.3 表达式 3＊5＋4 的抽象语法树

抽象语法树的建立过程是对一个表达式中的每一个运算符和运算分量都建立一个结点，运算分量可以是带运算符的子表达式，也可以是单个运算对象（常数和标识符）。运算符结点作为运算分量子树的根。因此，在抽象语法树中有两种不同的结点：运算符结点（内部结点）和运算对象结点（叶子结点）。不同的结点中包含的信息是不同的，可以用不同的结构来表示，例如，运算符结点至少应包含运算符和指向左右子树根的指针，后期还可以对结点添加其他属性；运算对象结点至少应包含常数的词法值或者标识符指向符号表的入口。可以使用下面的函数来实现抽象语法树的建立。每一个函数都返回一个指向新建立的结点的指针。

（1）mknode(op,left,right)，建立一个运算符结点，标号是运算符 op，left 和 right 分别指向左右子树。

（2）mkleaf(entry(id).name)，建立一个标识符结点，标号就是标识符的名字，如果符号表已经建立，entry(id)是一个语义函数，表示查找标识符在符号表中的入口。

（3）mkleaf(val(num))，建立一个常数结点，标号是 num，val()表示取常数的值。

例 5.5 使用上述几个函数为表达式 3＊(a－1)＋c 建立抽象语法树。

建立过程的函数调用序列为：

（1）p1＝mkleaf(entry(id).name)//a

（2）p2＝mkleaf(val(num))//1

（3）p3＝mknode("－",p1,p2)

（4）p4＝mkleaf(val(num))//3

（5）p5＝mknode("＊",p4,p3)

（6）p6＝mkleaf(entry(id).name)//c

（7）p7＝mknode("＋",p5,p6)

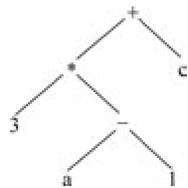

图 5.4 表达式 3＊(a－1)＋c 的抽象语法树

这棵抽象语法树是自下而上建立起来的。由 p1 指向的 entry(id)和 p6 指向的 entry(id)是指向符号表中标识符 a 和 c 的入口。首先建立叶子结点 a 和 1，然后建立内部结点“－”，再建立叶子结点 3，再建立内部结点“＊”，一步一步建立一棵树，p1，p2，…，p7 是指向结点的指针，最终建立的树根由 p7 指向。

可以看出，逆波兰式是抽象语法树的线性表示形式，逆波兰式是树结点的一个线性序列，其中的每个结点都是在它的所有子结点之后立即出现的，所以从图 5.4 可以很容易写出表达式 3＊(a－1)＋c 的逆波兰式是 3 a 1－＊ c＋，即是树的后序遍历的结果。

5.2.4 有向无环图

有向无环图(DAG)也是源代码的一种中间表示形式。与抽象语法树一样，对表达式中的每个子表达式，DAG 中都有一个结点，内部结点表示运算符，它的孩子代表运算分量。与抽象语法树不同的是，DAG 中代表公共子表达式的结点只出现一次，具有多个父结点，抽象

语法树中公共子表达式被表示为重复的子树。例如,表达式 x+x*(a-b)+(a-b)*y 的DAG如图5.5所示,抽象语法树如图5.6所示。

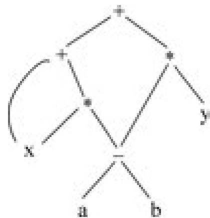

图5.5 表达式 x+x*(a-b)+(a-b)*y 的DAG

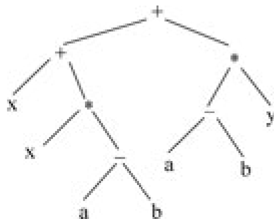

图5.6 表达式 x+x*(a-b)+(a-b)*y 的抽象语法树

从图5.5可以看出,叶子结点 x 是一个公共子表达式,有两个父结点,a-b 也是一个公共子表达式,在两个地方使用,因此也有两个父结点。所以 DAG 在描述源程序的层次结构时,比抽象语法树更紧凑,可以标出公共子表达式。

DAG 的特点是每个结点都带上各自的标记。

(1) 图中叶子结点由独特的标识符所标记。所谓独特的标识符是指它或者是变量名,或者是常数。根据作用到一个名字上的算符可以决定需要的是一个名字的左值还是右值。大多数叶子结点代表右值,叶子结点代表名字的初始值。

(2) 图的内部结点由运算符标记,代表进行这种运算和计算出来的值。

(3) 图中各个结点可能附加一个或多个标识符,表示这些标识符都具有该结点所代表的值。

构造 DAG 的方法与构造抽象语法树的方法类似:表达式的每个子表达式对应 DAG 中的一个结点,内部结点代表运算符,其左右孩子代表运算对象。注意,每次构造结点之前需要查找该结点是否已经构建过。

5.3 表达式的翻译

表达式是高级语言中唯一对数据进行处理的语法成分,表达式的执行将对其他变量或数据结构等产生影响,因此需要生成相应的中间代码,从而最终生成可执行的机器指令。对表达式的翻译应根据表达式的目标结构及其含义给出相应的语义规则,构成属性文法。

在 Sample 语言中,表达式的实现方式类似于 C 语言中的表达式,所有的表达式要么是赋值表达式,赋值符号右边还是一个表达式;或者如果缺少左边的标识符和赋值符号,就直接是一个简单表达式。由于 Sample 语言从 C 语言中裁剪了一部分成分,因此,在 Sample 语言中,简单表达式就是布尔表达式,布尔表达式由关系表达式和布尔运算符构成,关系表达式由算术表达式和关系运算符构成,算术表达式由常数、常量、变量、函数调用和运算符等构成。

5.3.1 几个语义变量和语义函数

由于表达式的翻译需要涉及翻译为四元式代码,因此在翻译过程中需要涉及中间代码表和临时变量区,除了用到前面介绍的语义函数 entry(id)外,还需要用到一些语义变量和函数,这里先介绍下面几个,其余的以后用到的时候再介绍。

(1) NXQ：将要生成的四元式的编号。在生成中间代码时，一般先输出到一个中间代码表中，等翻译结束再将中间代码表一起输出。四元式形式的中间代码表如表5.4所示。每一行都存放一条中间代码，用NXQ来表示即将要生成的四元式的编号，每生成一条新的中间代码，就将它插入NXQ所指向的位置，然后让NXQ自动加1，指向下一个空表项。中间代码表开始为空，NXQ=1，如当翻译X=B*C+1这个表达式时，先进行B*C的运算，生成第(1)条四元式，然后NXQ自动加1指向下一个空表项，随着代码生成过程的进行，新产生的四元式将逐步填入中间代码表中，直至结束，中间代码表中存放了中间代码生成的结果。

表5.4 四元式形式的中间代码表

	编号	运算符	运算对象1	运算对象2	结果
	(1)	*	B	C	T_0
NXQ=(2) →					

(2) X.PLACE：文法符号X的语义变量，表示与X对应的变量的存放位置或值。

(3) NewTemp()：语义函数，用来申请一个新的临时变量。编译程序管理着一个临时变量区，用于存放翻译过程中建立的临时语义变量。临时语义变量需要进行管理，NewTemp()用来申请一个新的临时变量，每调用一次，就生成一个，如第一次调用生成的临时变量为T_1，第二次调用生成的临时变量为T_2，等等。

(4) GenCode(op,arg1,arg2,result)：语义函数，用来产生一条四元式，并将该四元式插入四元式表中由NXQ指向的位置。如表5.4中就是在翻译表达式B*C时运行了GenCode('*',B,C,T_0)而得到的。

5.3.2 表达式的通用翻译方法

从上面的介绍可知，算术表达式是表达式的基础，由算术表达式构成关系表达式，再构成布尔表达式，再构成赋值表达式，按照统一求值的方法，先从算术表达式的翻译开始，逐步进行翻译。

1. 算术表达式的翻译

简单算术表达式是一种仅含普通常量、变量和常数的算术表达式，不含数组元素及结构引用，函数调用放在后面专门介绍。简单算术表达式的计算顺序与四元式出现的顺序相同，因此很容易将其翻译为四元式。

根据第3章的文法定义，简单算术表达式的文法如下(省略了函数调用)。

(1) <算术表达式>→<算术表达式>+<项>|<算术表达式>-<项>|<项>

(2) <项>→<项>*<因子>|<项>/<因子>|<项>%<因子>|<因子>

(3) <因子>→-<因子>|!<因子>|<初等量>

(4) <初等量>→<常数>|<常量名>|<变量名>|(<表达式>)

为简化起见，对文法进行简写。由于减(-)的语义处理方法和加(+)的语义处理方法相同，除(/)的语义处理方法和乘(*)的语义处理方法相同，因此，只选用+和*的运算的产生式。另外，常数可以直接使用其词法值，常量与变量都是标识符，其处理方式相同，因此，这里只考虑变量的情况。用< AExpr >表示<算术表达式>，它是算术表达式文法的开始符

号,用< Term >表示<项>,用< Factor >表示<因子>,用< Primary >表示<初等量>。这样,简单算术表达式的文法就可以简化为文法 G5.1。

(1) < AExpr >→< AExpr >+< Term >|< Term >

(2) < Term >→< Term >*< Factor >|< Factor >

(3) < Factor >→−< Factor >|!< Factor >|< Primary >

(4) < Primary >→id|(< AExpr >) (G5.1)

由于每个产生式都有相应的语义规则,因此需要将各个产生式分开,如表 5.5 是为文法 G5.1 编写的语义规则(假定只做整数运算)。

表 5.5 文法 G5.1 的属性文法

编号	产 生 式	语 义 规 则
(1)	< AExpr >→ < AExpr >$^{(1)}$ +< Term >	{< AExpr >.PLACE=NewTemp(); GenCode($+^{i}$,< AExpr >$^{(1)}$.PLACE,< Term >.PLACE,< AExpr >.PLACE)}
(2)	< AExpr >→< Term >	{< AExpr >.PLACE=< Term >.PLACE }
(3)	< Term >→ < Term >$^{(1)}$ *< Factor >	{< Term >.PLACE=NewTemp(); GenCode($*^{i}$,< Term >$^{(1)}$.PLACE,< Factor >.PLACE,< Term >.PLACE)}
(4)	< Term >→< Factor >	{< Term >.PLACE=< Factor >.PLACE }
(5)	< Factor >→ −< Factor >$^{(1)}$	{< Factor >.PLACE=NewTemp(); GenCode($@^{i}$,< Factor >$^{(1)}$.PLACE,,< Factor >.PLACE) }
(6)	< Factor >→!< Factor >$^{(1)}$	{< Factor >.PLACE= NewTemp(); GenCode(!,< Factor >$^{(1)}$.PLACE, ,< Factor >.PLACE) }
(7)	< Factor >→< Primary >	{< Factor >.PLACE=< Primary >.PLACE }
(8)	< Primary >→id	{< Primary >.PLACE=entry(id).name }
(9)	< Primary >→(< AExpr >)	{< Primary >.PLACE=< AExpr >.PLACE }

其中,$+^{i}$、$*^{i}$ 分别表示整数加法与乘法,$@^{i}$ 表示整数单目求负操作,其运算优先级高于乘法运算。

对应于产生式(1),是实现加法运算,首先要申请一个对应于产生式左部文法符号的临时语义变量< AExpr >.PLACE,然后对右部两个文法符号的属性值< AExpr >$^{(1)}$.PLACE 和< Term >.PLACE 进行加法运算,把值赋给它,作为< AExpr >的语义值保留下来,以供下一次调用产生式时使用,此功能需要产生一条四元式来实现。对于产生式(2),其语义是将右部文法符号的属性值< Term >.PLACE 赋给左部文法符号,< AExpr >的语义值保留下来,以供下一次调用产生式时使用,它的作用是传递语义属性的值。因为下一次使用其他产生式时,< Term >的信息已经在栈中消失了,所以对应于每个产生式的语义规则,必须在本条产生式中把后续需要用到的语义值进行保存。产生式(4)、(7)、(8)、(9)的语义规则与产生式(2)类似,都是传递语义属性。产生式(3)的语义规则与(1)类似,产生式(5)是单目求负运算,因此其语义规则是:先申请一个对应于产生式左部文法符号的临时语义变量< Factor >.PLACE,然后产生一条四元式,将右部文法符号的属性值求负后赋给它。产生式(6)的语义规则与产生式(5)类似,进行的是布尔运算求非!操作。

例 5.6 对 B+C＊(－D)进行自下而上的分析，同步产生四元式，最后几个移进-归约动作如图 5.7 所示。

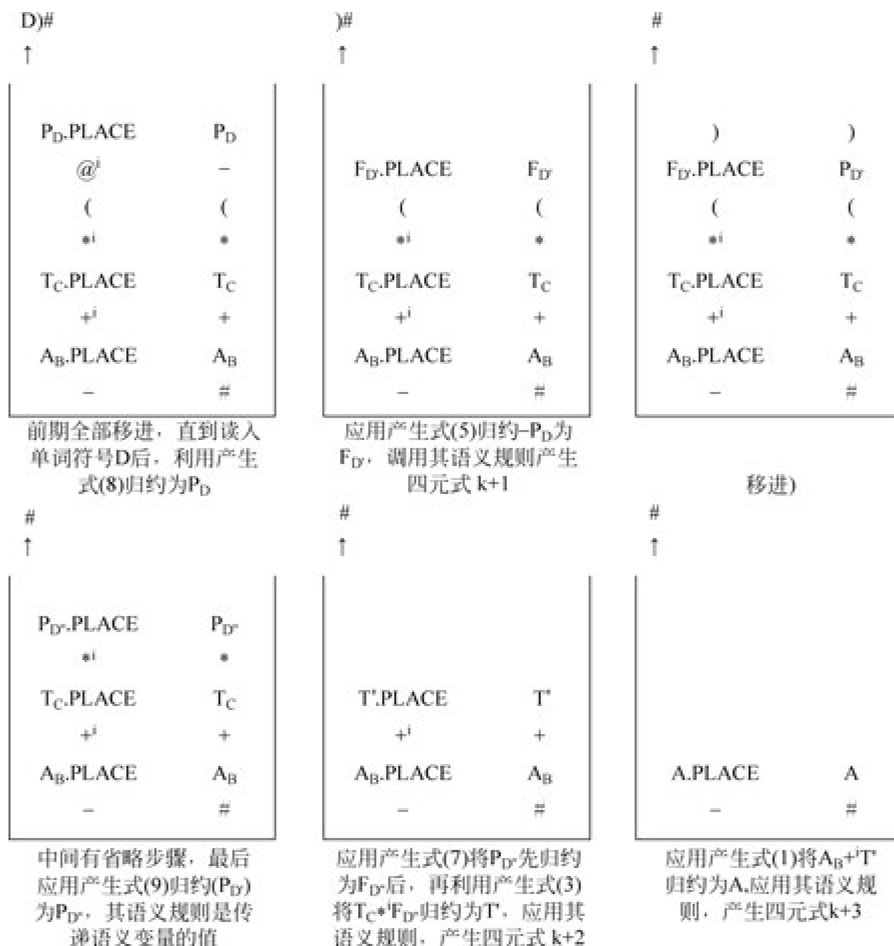

图 5.7 算术表达式 B+C＊(－D)的规范归约及翻译过程

在图 5.7 中，与归约的符号栈同步设置一个语义栈，为了让读者更清楚当前用来归约的产生式对应的文法符号，将读取的每个单词符号与文法符号联系起来，用文法符号的首字母来表示归约的符号，用读取的单词符号做下标，如 Tc 表示对应于文法符号< Term >，当前读入的符号是 C。

设开始分析该表达式时，中间代码表中四元式的最大编号为 k，之前已经申请了 4 个临时变量，那这个表达式生成的第一条四元式的编号是 k+1，申请的临时变量从 T_5 开始。随着语法分析的进行，在每一次使用某个产生式进行归约时，就调用表 5.5 中对应这个产生式的语义规则，同步完成语义的计算，语义栈的内容也同步发生变化，产生相应的四元式。注意，为了观察中间代码调用语义规则生成四元式，图 5.7 中省略了刚开始只进行语义传递的几个步骤，如在移进)之前，需要将 F 归约为 T，T 归约为 A，然后才能用产生式(9)将(A)归约为 P……当该赋值表达式翻译完后，四元式表中的内容如表 5.6 所示，最大的四元式编号为 k+3。A. PLACE 中存储了整个表达式的值，也就是在 T_7 中。

表 5.6　算术表达式 B+C*(-D)的四元式表

编　　号	用语义变量表示的四元式形式	实际生成的四元式
(k)	⋮	
(k+1)	($@^{i}$,P_{D}.PLACE,,$F_{D'}$.PLACE)	($@^{i}$,D,,T_5)
(k+2)	($*^{i}$,T_{C}.PLACE,$P_{D''}$.PLACE,T'.PLACE)	($*^{i}$,C,T_5,T_6)
(k+3)	($+^{i}$,A_{B}.PLACE,T'.PLACE,A.PLACE)	($+^{i}$,B,T_6,T_7)

2. 关系表达式的翻译

关系表达式的基础是算术表达式,由算术表达式和关系运算符构成关系表达式,关系运算符有两种不同的优先级。根据第3章的文法定义,Sample语言中与关系表达式相关的文法如下。

(1) <关系表达式>→<关系表达式><关系运算符><简单关系表达式>|<简单关系表达式>

(2) <关系运算符>→==|!=

(3) <简单关系表达式>→<简单关系表达式><简单关系运算符><算术表达式>|<算术表达式>

(4) <简单关系运算符>→＞|＜|＞＝|＜＝

按照上面介绍的算术表达式的语义处理方法,直接将关系运算符当作四元式的运算符,如关系运算 a＜b 可以翻译为这样一条四元式。

(100)(＜,a,b,T_0)

为了书写方便,对该文法进行简化,用<RE>表示<关系表达式>,它是关系表达式文法的开始符号,用 rop 代表相等/不相等的关系运算符==和!=,用<SRE>表示<简单关系表达式>,用 srop 代表简单关系运算符＞、＞＝、＜、＜＝,把 rop 和 srop 当作终结符。<AExpr>是算术表达式,这里使用的是这个表达式最终的值,用<AExpr>.PLACE来表示。

这样,上述关系表达式的文法就可以简写为文法G5.2。

(1) <RE>→<RE> rop <SRE>|<SRE>

(2) <SRE>→<SRE> srop <AExpr>|<AExpr>　　　　(G5.2)

可以参照算术表达式文法G5.1的语义动作,为G5.2配上合适的语义动作,如表5.7所示。

表 5.7　文法 G5.2 的属性文法

编　　号	产　生　式	语 义 规 则
(1)	<RE>→ <RE>(1) rop <SRE>	{<RE>.PLACE=NewTemp(); GenCode(rop,<RE>(1).PLACE,<SRE>.PLACE,<RE>.PLACE)　}
(2)	<RE>→<SRE>	{<RE>.PLACE=<SRE>.PLACE　}
(3)	<SRE>→ <SRE>(1) srop <AExpr>	{<SRE>.PLACE=NewTemp(); GenCode(srop,<SRE>(1).PLACE,<AExpr>.PLACE,<SRE>.PLACE)}
(4)	<SRE>→<AExpr>	{<SRE>.PLACE =<AExpr>.PLACE　}

关系表达式的结果有0或者1两种情况,如果用在控制语句中,则可能会根据不同的结果有不同的转向执行点,如用在if语句或者while语句中,到时候再根据其值判断具体的

转向。

关系运算的翻译也可以不使用关系运算符＜、＜＝、＞、＞＝、＝＝、!＝作为四元式的运算符，而是用另一种方式来实现，把比较的结果存入一个取值为0或者1的变量中，把关系运算翻译为比较后直接跳转的四元式。四元式的运算符为jsrop(其中srop是＜、＜＝、＞、＞＝)和jrop(rop是＝＝、!＝)，以及无条件跳转的四元式j。例如，对于关系运算a＜b，它的含义是：如果a＜b结果为1，否则结果为0，它等价于**T＝(a＜b)**，当a＜b时，临时变量T赋值为1，否则T赋值为0。可以翻译为四元式(j＜,a,b,k＋4)，表示的含义是当a＜b成立时，跳转到第k＋4条四元式，另外一条无条件跳转四元式为(j,,,k)，表示的含义是无条件跳转到第k条四元式。

如果在翻译a＜b这个关系表达式之前已经有k条四元式，那么关系运算a＜b就可以等价地翻译为如下四条四元式。

(k＋1)　(j＜,a,b,k＋4)
(k＋2)　(＝,0 ,_ ,T)
(k＋3)　(j,_,_,k＋5)
(k＋4)　(＝,1 ,_ ,T)
(k＋5)

其中，第k＋1条四元式的含义是对a和b进行比较，如果结果为真，跳转到第k＋4条四元式执行，置T为1；否则，a不小于b，则顺序执行第k＋2条四元式，置T为0，这时，就必须跳过设置T为1的代码，因此应该加入第k＋3条四元式，跳过第k＋4条四元式，这个关系运算结束后，即将要产生的四元式的编号是k＋5。其含义和执行流程如图5.8所示。这样，后续就可以直接用T进行计算。

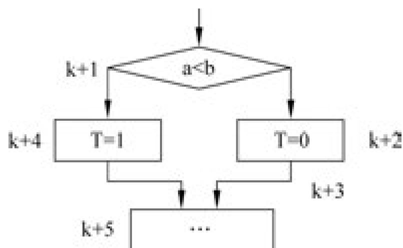

图5.8　关系运算a＜b的目标结构

这样，针对表5.7中的产生式(3)，也可以应用下面的语义规则，生成4条四元式。可以验证，它和表5.7中的语义规则生成的四元式是等价的。

(3) < SRE >→< SRE >$^{(1)}$ srop < AExpr >
{　< SRE >. PLACE＝ NewTemp()；
GenCode(jsrop,< SRE >$^{(1)}$. PLACE,< AExpr >. PLACE,NXQ＋2)；
GenCode(＝,0,_,< SRE >. PLACE)；
GenCode(j,_ ,_ ,NXQ＋1)；
GenCode(＝,1,_,< SRE >. PLACE)　}

同样可以得到产生式(1)也可以生成4条四元式。这种方法实际上是在中间代码阶段实现了比较，而前面介绍的第一种方法实际就是把比较工作推迟到目标代码阶段再进行。

3. 布尔表达式的翻译

由关系表达式和布尔运算符||和&&可以构成布尔表达式。按照算术表达式的语义处理方法，对布尔表达式中的每个布尔项和布尔因子都计算其布尔值，最后求得整个表达式的布尔值。

布尔运算中将非零当作真，表示为1，否则当作假，表示为0。例如，布尔表达式1||(!

0 && 0)|| 0 的计算过程是：

1||(! 0 && 0)|| 0

=1||(1 && 0)|| 0

=1|| 0|| 0

=1|| 0

=1

根据第 3 章的文法定义，Sample 语言中与布尔表达式相关的文法如下。

(1) <布尔表达式>→<布尔表达式>||<布尔项>|<布尔项>

(2) <布尔项>→<布尔项> && <布尔因子>|<布尔因子>

(3) <布尔因子>→<关系表达式>

为了书写方便，对该文法进行简化，用<BE>表示<布尔表达式>，它是布尔表达式文法的开始符号，用<BT>表示<布尔项>，用<BF>表示<布尔因子>。<RE>是关系表达式，这里使用的是这个表达式最终的值<RE>.PLACE。这样，布尔表达式文法就可以简写为文法 G5.3。

(1) <BE>→<BE>||<BT>|<BT>

(2) <BT>→<BT> && <BF>|<BF>

(3) <BF>→<RE> (G5.3)

按照上面介绍的算术表达式的语义处理方法，将每个布尔运算符当作四元式的运算符，可以参照算术表达式文法 G5.1 的语义动作，为 G5.3 配上合适的语义动作，如表 5.8 所示。

表 5.8 文法 G5.3 的属性文法

编号	产生式	语义规则
(1)	<BE>→<BE>$^{(1)}$ || <BT>	{<BE>.PLACE=NewTemp()； GenCode(||,<BE>$^{(1)}$.PLACE,<BT>.PLACE, <BE>.PLACE) }
(2)	<BE>→<BT>	{<BE>.PLACE=<BT>.PLACE }
(3)	<BT>→ <BT>$^{(1)}$ && <BF>	{<BT>.PLACE=NewTemp()； GenCode(&&,<BT>$^{(1)}$.PLACE,<BF>.PLACE, <BT>.PLACE) }
(4)	<BT>→<BF>	{<BT>.PLACE=<BF>.PLACE }
(5)	<BF>→<RE>	{<BF>.PLACE=<RE>.PLACE }

这种翻译方法最后求得整个布尔表达式的值，当使用这个表达式时，如在 if 语句或者 while 语句的表达式结束时，再根据其值来判断具体的转向。

例 5.7 根据布尔表达式的语义规则，写出布尔表达式 a<b|| c<d && e>f 生成的四元式代码。

严格按照表 5.7 和表 5.8 的语义规则，可以直接根据运算符和运算符之间的优先级，得到的四元式代码如下。

(101) (<,a,b,T_0)

(102) (<,c,d,T_1)

(103) (>,e,f,T_2)

(104) (&&,T_1,T_2,T_3)

(105) (||,T_0,T_3,T_4)

由于关系运算的优先级比布尔运算的优先级高，首先翻译 a<b，如果关系运算按照表5.7中的语义规则，利用产生式(3)，申请一个临时变量 T_0，生成四元式101；然后翻译c<d，利用产生式(3)，申请一个临时变量 T_1，生成四元式102；然后翻译 e>f，利用产生式(3)，申请一个临时变量 T_2，生成四元式103。接下来是进行布尔运算的翻译，由于 && 运算的优先级高，翻译 T_1&&T_2，利用表5.8中的产生式(3)的语义规则，申请一个临时变量 T_3，生成四元式104；最后，翻译 T_0||T_3，利用产生式(1)的语义规则，申请一个临时变量 T_4，生成四元式105。该表达式的最终结果存储在 T_4 中。

如果关系运算按照第二种先求值的翻译方法，这个表达式将会生成如下的四元式列表。

(101) (j<,a,b,104)

(102) (=,0,_,T_0)

(103) (j,_,_,105)

(104) (=,1,_,T_0)

(105) (j<,c,d ,108)

(106) (=,0,_,T_1)

(107) (j,_,_,109)

(108) (=,1,_,T_1)

(109) (j>,e,f ,112)

(110) (=,0,_,T_1)

(111) (j,_,_,113)

(112) (=,1,_,T_2)

(113) (&&,T_1,T_2,T_3)

(114) (||,T_0,T_3,T_4)

a<b 生成4条四元式(101)～(104)；同样 c<d 生成4条四元式(105)～(108)； e>f 生成4条四元式(109)～(112)，后面再利用表5.8中产生式(3)和产生式(1)的语义规则，生成2条四元式(113)和(114)。表达式的最终结果存储在 T_4 中。

从这个例子可以看出，利用不同的翻译方法得到了不同的四元式列表，其功能是相同的，只是四元式的求值方法不同。第一种翻译直接用关系运算符作为四元式的运算符，将比较推迟到目标代码生成阶段，第二种翻译方法是在这里就直接进行了比较，根据比较结果选择进行不同的后续运算。可以验证，这两组四元式序列是等价的。

和关系表达式的第二种翻译方法类似，也可以把布尔表达式直接翻译为比较跳转指令，同时还可以根据布尔运算的特殊性对其进行优化，这部分内容将在5.3.3节中介绍。

4. 表达式和赋值表达式的翻译

根据第3章的文法，Sample 语言中的表达式有两种形式：有赋值符号和没有赋值符号。有赋值符号的表达式称为赋值表达式，赋值符号也是一个运算符，优先级最低，含义是把赋值号右边的表达式的值赋值给左边的一个变量，没有赋值符号的表达式称为简单表达式，在 Sample 语言中对C语言进行了裁剪，它就是一个布尔表达式。文法描述如下。

(1) <表达式>→<赋值表达式>|<简单表达式>

(2) <赋值表达式>→<变量名>=<表达式>

(3) <简单表达式>→<布尔表达式>

为方便起见,可以对表达式的文法进行简写。用< Expr >表示<表达式>,它是表达式文法的开始符号,用< SExpr >表示<简单表达式>,用< AssignExpr >表示<赋值表达式>,<变量名>就是标识符。用< BE >表示<布尔表达式>,不在此处翻译,这里只作为一个整体来使用其值< BE >. PLACE。这样,表达式文法就可以简写为文法G5.4。

(1) < Expr >→< AssignExpr >|< SExpr >

(2) < AssignExpr >→id=< Expr >

(3) < SExpr >→< BE > (G5.4)

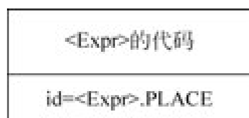

图5.9 赋值表达式的目标结构

赋值表达式 id=< Expr >是将表达式< Expr >的值计算出来,再赋给 id,其目标结构如图5.9所示。其中,"< Expr >的代码"是表达式< Expr >翻译后的一系列顺序执行的四元式,其翻译不在这条产生式中进行。目标结构中的最后一条四元式是把表达式的结果赋给赋值表达式的左变量。

表5.9是为文法G5.4编写的语义规则(假定只做整数运算)。

表5.9 文法G5.4的属性文法

编号	产生式	语义规则
(1)	< Expr >→< AssignExpr >	{ < Expr >. PLACE=< AssignExpr >. PLACE }
(2)	< Expr >→< SExpr >	{ < Expr >. PLACE=< SExpr >. PLACE }
(3)	< AssignExpr >→id=< Expr >	{ < AssignExpr >. PLACE=< Expr >. PLACE; GenCode(=,< Expr >. PLACE,,entry(id). name) }
(4)	< SExpr >→< BE >	{ < SExpr >. PLACE=< BE >. PLACE }

对于产生式(1)、(2)、(4)的语义规则主要是进行语义信息传递,把右部文法符号的属性值PLACE赋给左部文法符号的语义变量保留下来,以供下一次归约时使用。对于产生式(3)的语义规则,除了将其语义信息保存到左部文法符号的语义变量中,它的功能是赋值,要将赋值号右部的表达式的值赋值给左部的变量,产生一条四元式。

例如,要对赋值表达式A=B+C*(-D)进行翻译。根据优先级和文法,需要先翻译右边的表达式,然后赋值。在例5.6中已经将右边的算术表达式翻译为3条四元式了,根据赋值表达式的翻译,最后还需要将右部表达式的值赋值给变量A。得到的四元式序列如下。

(k) …

(k+1) ($@^i$,D,,T_5)

(k+2) ($*^i$,C,T_5,T_6)

(k+3) ($+^i$,B,T_6,T_7)

(k+4) (=,T_7, ,A)

5.3.3 布尔表达式的优化翻译方法

5.3.2节介绍了通用表达式的翻译方法,由于布尔表达式有其特殊性,因此可以在运算过程中对求值进行优化,所以下面介绍布尔表达式的优化翻译方法。

1. 真假出口的概念

虽然在C语言中，不特别强调布尔值，非零表示真，零表示假。5.3.2节介绍的表达式的通用翻译方法，每一次都对表达式中的每个项和因子计算其值，最后求得整个表达式的值，把判断其值为真和为假的时机推迟到目标代码阶段进行处理。

如果表达式用作控制语句中的条件，其作用是选择下一个执行点。有时候并不需要把整个表达式的值计算出来就可以判断出表达式为真或为假，那就可以提前确定跳转，节省计算量，尤其是表达式是一个布尔表达式时，可以根据布尔运算的特殊性采用优化措施来进行翻译。假如有while语句形如while (E) $S^{(1)}$，其中布尔表达式E的作用是选择执行$S^{(1)}$语句，还是跳过$S^{(1)}$执行while语句后面的语句。假如E= E_1 && E_2，通过计算已经发现E_1的值为0，那就没有必要去计算E_2了。为了进行优化翻译，需要为布尔表达式E定义两个出口。

(1) 真出口：E为真时应转向执行的位置，指向$S^{(1)}$语句的开始，表明如果E的值为真，则执行$S^{(1)}$的代码。用E.TC表示，在上例中，等于$S^{(1)}$的代码的第一个四元式编号。

(2) 假出口：E为假时应转向执行的位置，指向这个while语句后面的语句，表明如果E的值为假，则跳过$S^{(1)}$的代码，跳出循环。用E.FC表示，在上例中，等于while语句后面一个语句的第一个四元式编号。

while语句的目标结构如图5.10所示。正常情况下，$S^{(1)}$执行完后，需要转移到E的开始位置(用W.HEAD标记)，重新测试E的值，可以用一条无条件跳转语句(图中用GOTO W.HEAD表示)跳转到while语句的开始。考虑$S^{(1)}$本身也可能是一个控制语句，当某种条件不满足或者遇到continue等语句时，应从$S^{(1)}$语句中间的某点跳过后面的代码，直接转移到E的开始位置，重新测试E的值。如果在循环中遇到break，和布尔表达式E为假一样，直接跳出循环，执行后面的语句。

因此，翻译布尔表达式的关键是确定E为真和为假时跳转到的位置，即确定E.TC和E.FC的值。

每个布尔表达式都可以看作由布尔因子经过!、&&、||等符号组合而成的表达式。布尔因子的基本形式有两种：单个的初等量(如标识符、常数等)和关系表达式。

如果初等量A作为布尔表达式，其目标结构应包括两个出口：一个表示真出口A.TC，另一个表示假出口A.FC，目标结构如图5.11所示。

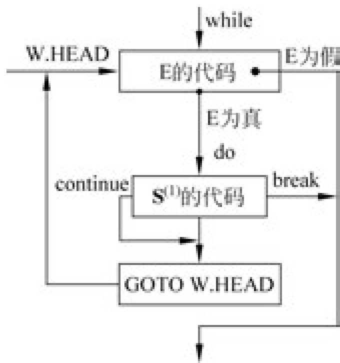

图5.10 while语句的目标结构

图5.11 初等量A的目标结构

因此,对初等量作为布尔表达式使用,应翻译为如下两条相继出现的四元式。

(1) (jnz,A,,P):真出口,当 A 为真时,则跳转到四元式 P。

(2) (j,,,Q):假出口,无条件跳转到四元式 Q。

四元式的第 4 个分量表示转移去向,即 P 和 Q 均为某个四元式的编号,暂时未知,需要在后续处理中去填写。

同样,对关系运算 i_1 rop i_2,如果作为布尔表达式使用,也可以翻译为如下两条相继出现的四元式。

(1) (jrop,i_1,i_2,P):真出口,当 i_1 rop i_2 为真时转四元式 P(如果 rop 是<,则 jrop 写作 j<,其余关系运算符类推)。

(2) (j,,,Q):假出口,无条件跳转到四元式 Q。

同样,四元式的第 4 个分量 P 和 Q 是暂时未知的。

布尔因子经过||和 && 运算后,组成布尔表达式。由于布尔运算的特殊性,&& 和||运算可以进行优化处理。

对于表达式 E||T,只要 E 为真,不必计算 T,就知道布尔表达式 E||T 为真;只有当 E 为假时才读取 T,E||T 的值由 T 值决定。这样就可以确定布尔表达式 E||T 的假出口和真出口。

对于布尔项 T&&F,只要 T 为假,不必计算 F,就知道布尔项 T&&F 为假;只有当 T 为真时才读取 F,T&&F 的值由 F 值决定。这样就可以确定布尔项 T&&F 的假出口和真出口。

2. 回填技术

在由多个因子组成的布尔表达式中,可能有多个因子的真出口或假出口的转移去向相同,但又不能立刻知道具体转向位置。在这种情况下,需要把这些转移去向相同的四元式链在一起,形成四元式链,以便后续知道转移地址后再回填。

对于给定的布尔表达式,其翻译方法如下。

(1) 若已知转移地址则直接填入;若不知道,则先填入 0,等知道后再回填。

(2) 如果多个因子的转移去向相同,但又不知道具体位置,应该用一个链表将这些未知且出口相同的四元式链在一起。

例 5.8 写出布尔表达式 A && B &&(C>D)的四元式序列。

首先分析该布尔表达式,当扫描到 A 后的 && 时,对布尔量 A 进行归约,根据上面对布尔量的翻译的介绍,将产生两条四元式,假定为(1)和(2)。四元式(1)的第 4 个分量表示真出口,由于 A 为真时应计算 B,因此 A 的真出口的值为 B 翻译后的第一条四元式的编号,也是即将产生的四元式的编号,为 3(即 A 为真时转向 3)。四元式(2)的第 4 个分量表示假出口,其值未知,先填入 0;当扫描到 B 后的 && 时,对布尔量 B 进行归约,又将产生两条四元式,假定编号为(3)和(4)。(3)后的第 4 个分量表示真出口,由于 B 为真时计算 C>D,因此 B 的真出口的值应为 5(当 B 为真时转向 5)。四元式(4)的第 4 个分量表示假出口,其值仍未知,但可以知道它与 A 的假出口相同,则将它与四元式(2)链接起来,因此将四元式(4)的第 4 个分量填入 2。当扫描到最后,对关系表达式 C>D 进行归约,又产生两条四元式(5)和(6)。此时四元式(5)的第 4 个分量表示真出口,其值未知(即暂时不知道 C>D 时转向哪里),填入 0。四元式(6)的第 4 个分量表示假出口,其值未知,但它与 A 和 B 的假出口相同,

则将它们链接起来，填入4。得到最后的四元式列表为：

(1) (jnz,A,,3)

(2) (j,,,0)

(3) (jnz,B,,5)

(4) (j,,,2)

(5) (j>,C,D,0)

(6) (j,,,4)

这样上述布尔表达式就生成了真、假出口两个链，每个链用一个头指针来指向。其中四元式(5)构成了真出口链，(5)也是这个链的头指针，四元式(6)、(4)、(2)形成一条假出口链，(6)作为这个链的头指针，每个真出口链和假出口链的链尾的四元式第4个分量都为0，作为结束标记，如上面的(2)和(5)的第4个分量为0。用E.TC指向真出口的链首(其值为5)，用E.FC指向假出口的链首(其值为6)。以后在语义处理的过程中，一旦发现具体的转向目标，则应把转向的目标四元式编号回填到对应的真出口或假出口链上所有四元式的第4个分量处。

例如，对于下面的if语句

```
if (A && B &&(C > D) ) { S1 } else { S2 };
```

当布尔表达式分析结束，遇到{时就知道布尔式的真出口位置，此时可以将E.TC链(5)上的每个四元式的第4个分量的0填入S_1翻译后的第一个四元式的编号。同样，只有当遇到else时，才能回填E.FC链(6,4,2)上每个四元式的第4个分量。在不知道转移去向时，上面四元式(6,4,2)通过第4个分量链在一起，一旦知道转移去向，如else后的语句生成的四元式编号为t，则需要查找链首为6的链表，将该链表中的每个四元式的第4个分量都填为t。

为按上述优化方法进行布尔表达式的翻译，在为文法G5.3设计语义动作时，需要为布尔表达式中的每个非终结符X设置两个语义变量，分别代表X的两个出口：真出口X.TC和假出口X.FC，同时它们又是X这个布尔表达式的真出口和假出口的链首。此外，还需要用到下面两个语义函数。

(1) merge(P1,P2)是一个函数，把以P1、P2为链首的两个四元式链合并为一个链，返回合并后的链首。

$$\text{合并后的链首}=\begin{cases}P1(P2=0)\\P2(P2\neq 0)\end{cases}$$

merge(P1,P2)的函数描述如下。

```
//合并两个四元式链
merge(P1, P2)
{
     if ( P2 == 0) return (P1);
          else {
               P = P2;
                    while (四元式 P 的第 4 个分量内容不为 0)
                         P = 四元式 P 的第 4 个分量内容;
                    把 P1 填入四元式 P 的第 4 个分量;
               return (P2);
          }
}
```

(2) backpatch(P,t)是一个回填函数,把链首P所链接的每个四元式的第4个分量都改写为编号t。这个函数的描述如下。

```
//回填四元式编号
backpatch(P,t)
{
    Q=P;
    while (Q!=0) {
        m = 四元式Q的第4个分量内容;
        把t填入四元式Q的第4个分量;
        Q = m;
    }
}
```

3. 翻译方法

按照前面介绍的布尔表达式的优化处理方法,对于表达式 E_1||T 和 T_1&&F 可以画出其目标结构。图5.12(a)中的E.FC和E.TC表示整个布尔表达式 E_1||T 的假出口和真出口。图5.12(b)中的T.FC和T.TC表示整个布尔项 T_1&&F 的假出口和真出口。

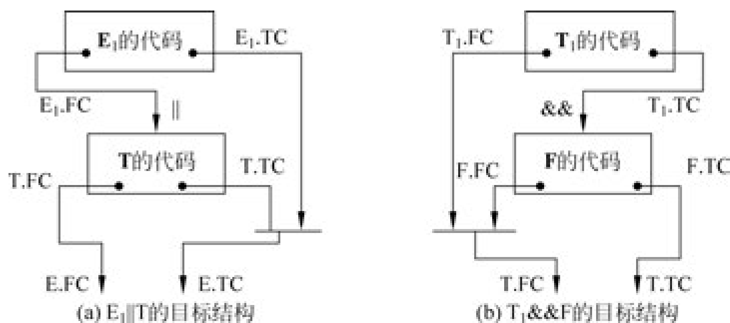

图5.12　E_1||T 和 T_1&&F 的目标结构

从图5.12(a)中可以看出,E_1 的假出口应转向布尔项T的第一个四元式的位置。由于语法分析程序分析到运算符"||"时才知道 E_1 已分析完毕,开始生成T的四元式,这样,当分析程序扫描到"||"时,应该执行一个语义动作,把即将生成的下一个四元式的编号(即T的第一条四元式的编号)回填给 E_1 的假出口。为此,产生式<BE>→<BE>|| <BT>应做如下改造,以便能在归约时执行这个语义动作。

$$<BE>^{or} \rightarrow <BE>||$$

$$<BE> \rightarrow <BE>^{or} <BT>$$

这样,当用产生式<BE>^{or}→<BE>|| 进行归约时,就能立即执行回填动作。类似地,产生式<BT>→<BT> && <BF>也应改造为下面两个产生式。

$$<BT>^{and} \rightarrow <BT> \&\&$$

$$<BT> \rightarrow <BT>^{and} <BF>$$

当使用产生式<BT>^{and}→<BT> && 归约时,就可以及时把<BF>的第一个四元式编号回填给<BT>的真出口链。

根据上面的分析,为进行布尔表达式的优化翻译,把文法G5.3和G5.2合并后构成完

整的布尔表达式的文法，经过改造和简写后，得到文法 G5.5。

(1) <BE>→<BE>^{or} <BT>

(2) <BE>^{or}→<BE>||

(3) <BE>→<BT>

(4) <BT>→<BT>^{and} <BF>

(5) <BT>^{and}→<BT> &&

(6) <BT>→<BF>

(7) <BF>→<RE>

(8) <RE>→<RE> rop <SRE>

(9) <RE>→<SRE>

(10) <SRE>→<SRE> srop <AExpr>

(11) <SRE>→<AExpr>　　(G5.5)

现在来详细分析各个产生式，以便得到各个产生式对应的语义规则。

产生式(11)是算术表达式的值作为一个布尔量(注：算术表达式本身产生四元式的方法已在 5.3.2 节中介绍，此处只是将其结果作为一个整体来看待)，其值为零或非零，语义规则是产生两个四元式，第一个四元式的编号为 NXQ，它是相应的真出口；第二个四元式的编号为 NXQ+1，它是相应的假出口。当将右部归约为左部的<SRE>时，其真假出口由左部符号<SRE>的语义变量<SRE>.TC 和<SRE>.FC 携带，由于真假出口暂时都不能确定，因此四元式的第 4 个分量的值为 0。产生式(10)和(8)是关系运算，其语义规则也是生成两个四元式。

当利用产生式(9)进行归约时，语义规则是仅把右部非终结符<SRE>的真假出口链(链首)传递给左部非终结符<RE>的语义变量<RE>.TC 和<RE>.FC。产生式(7)、(6)和(3)的语义动作也可以用相同的方法给出。

当利用产生式(5)进行归约时，根据图 5.12(b)，当用<BT> && 归约时，<BT>真出口的转向已知(即下一个四元式的编号，也就是<BF>的第一条四元式的编号)，可以回填；而<BT>的假出口的转向暂时还不能填入，需要继续向后传递，用左部文法符号<BT>^{and} 的假出口来保存。

当利用产生式(4)进行归约时，根据图 5.12(b)，当<BF>已归约出来后，<BF>的假出口就是<BT> && <BF>的假出口，应把它与<BT>的假出口合并为一个，作为整个布尔式<BT> && <BF>的假出口；<BF>的真出口作为整个布尔式的真出口。

用同样的方法，根据图 5.12(a)可以给出(1)、(2)两个产生式的语义动作。

总结前面的分析，可以为文法 G5.5 中各产生式配上对应的语义规则如表 5.10 所示。

表 5.10　文法 G5.5 的属性文法

编　号	产　生　式	语 义 规 则
(1)	<BE>→<BE>^{or} <BT>	{ <BE>.FC=<BT>.FC; <BE>.TC=merge(<BE>^{or}.TC,<BT>.TC) }
(2)	<BE>^{or}→<BE>||	{ backpatch(<BE>.FC,NXQ); <BE>^{or}.TC=<BE>.TC }
(3)	<BE>→<BT>	{ <BE>.TC=<BT>.FC; <BE>.FC=<BT>.FC }

续表

编　　号	产　生　式	语 义 规 则
(4)	<BT>→<BT>^{and} <BF>	{ <BT>.TC=<BF>.TC; <BT>.FC=merge(<BT>^{and}.FC,<BF>.FC) }
(5)	<BT>^{and}→<BT> &&	{ backpatch(<BT>.TC,NXQ); <BT>^{and}.FC=<BT>.FC }
(6)	<BT>→<BF>	{ <BT>.TC=<BF>.FC; <BT>.FC=<BF>.FC }
(7)	<BF>→<RE>	{ <BF>.TC=<RE>.TC; <BF>.FC=<RE>.FC }
(8)	<RE>→ <RE>$^{(1)}$ rop <SRE>	{ <RE>.TC=NXQ; <RE>.FC=NXQ+1; GenCode(jrop,<RE>$^{(1)}$.PLACE,<SRE>.PLACE,0); GenCode(j,,,0) }
(9)	<RE>→<SRE>	{ <RE>.TC=<SRE>.FC; <RE>.FC=<SRE>.FC }
(10)	<SRE>→ <SRE>$^{(1)}$ srop <AExpr>	{ <SRE>.TC=NXQ; <SRE>.FC=NXQ+1; GenCode(jsrop,<SRE>$^{(1)}$.PLACE,<AExpr>.PLACE,0); GenCode(j,,,0) }
(11)	<SRE>→<AExpr>	{ <SRE>.TC=NXQ; <SRE>.FC=NXQ+1; GenCode(jnz,<AExpr>.PLACE,,0); GenCode(j,,,0) }

当整个布尔表达式归约为开始符号<BE>后,该表达式的真假出口链的链首分别保存在<BE>.TC和<BE>.FC中。由于作为条件式的布尔表达式仅属于语句的一部分,因此必须等到分析了语句的其余部分后才能确定真假出口的具体转向。例如,对while (E) S$^{(1)}$语句而言,当扫描到表达式后的右括号)时,才能回填E的真出口。

另外,布尔运算求非(!)操作,优先级高,也可以进行带真假两个出口链的翻译,只需调换真假出口链的头指针即可,不产生新的四元式。

5.4 控制语句的翻译

高级语言中通过语句来控制程序执行的顺序。Sample语言中的语句主要包括表达式语句、控制语句和复合语句三种,其中,表达式语句是在表达式后加上语句分隔符构成的,5.3节中已介绍了表达式的翻译方法,加上分隔符,不产生新的四元式。复合语句是把多个语句用{}括在一起,本身不产生新的四元式。因此,语句的翻译主要是翻译控制语句。Sample语句中的控制语句主要有if语句、while语句、do…while语句和for语句,还有一些和函数相关的语句,放在5.5节中介绍。翻译控制语句时,需要根据语句的语义画出目标代码结构,找出源与目标的对应关系,设计属性文法。

5.4.1　if 语句的翻译

if 语句是控制语句的一种，第 3 章中描述 if 语句的文法如下。

< if 语句> → if (<表达式>) <语句>$^{(1)}$

< if 语句> → if (<表达式>) <语句>$^{(1)}$ else <语句>$^{(2)}$

其含义是根据表达式的值来决定执行后面的<语句>$^{(1)}$还是执行<语句>$^{(2)}$，或者没有<语句>$^{(2)}$就不执行。if 语句可以是嵌套的，即<语句>本身又可以是 if 语句或其他语句。

if 语句的<表达式>的真出口应为<语句>$^{(1)}$的第一条四元式的编号，当分析到<表达式>后面的右括号)准备分析后面的<语句>$^{(1)}$时，就知道<语句>$^{(1)}$的入口位置，就可以回填<表达式>的真出口。而<表达式>的假出口必须等待<语句>$^{(1)}$归约完之后才能回填，关键字 else 是<语句>$^{(1)}$归约完和<语句>$^{(2)}$开始的标记。因此，当扫描到 else 时，应及时把<语句>$^{(2)}$的入口四元式编号及时回填给<表达式>的假出口。由于在自下而上的分析中，只有在某产生式归约时才能调用相应的语义动作，因此，需要对文法进行改造，使得在需要完成语义动作的地方，能够对产生式进行归约。

为简化起见，用< IFS >表示<if 语句>，它是 if 语句文法的开始符号，用< C >表示 if 语句的条件部分 if (<表达式>)，用< T >代表 else 及之前的内容 if (<表达式>) <语句>$^{(1)}$ else，<表达式>仍然使用< Expr >来表示，用< S >表示<语句>，为了区分，后面的两个语句分别用< S >$^{(1)}$和< S >$^{(2)}$表示。改造和简写后的 if 语句的文法如文法 G5.6 所示。

(1) < IFS >→< C >< S >$^{(1)}$

(2) < C >→if (< Expr >)

(3) < IFS >→< T >< S >$^{(2)}$

(4) < T >→< C >< S >$^{(1)}$ else　　　　(G5.6)

其中，(1)、(2)两个产生式生成无 else 的 if 语句，而(2)、(3)、(4) 3 个产生式则生成带 else 的 if 语句。if 语句的目标结构如图 5.13 所示，其中图 5.13(a)为带有 else 的 if 语句，图 5.13(b)为不带 else 的 if 语句。

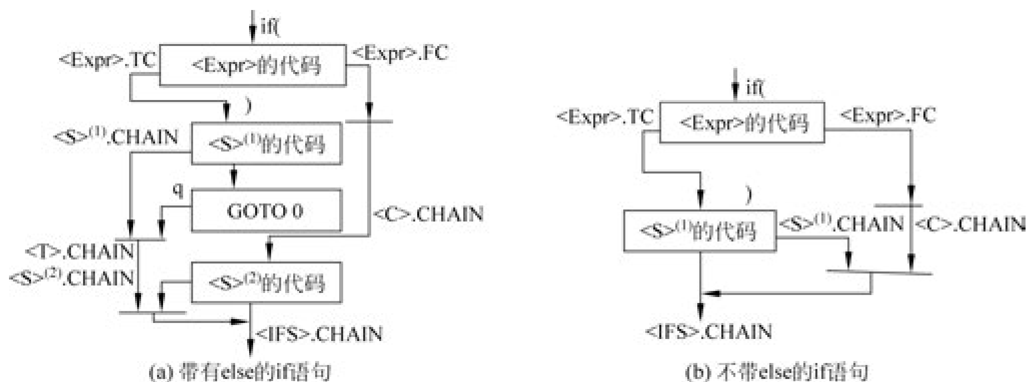

图 5.13　if 语句的目标结构

在为 G5.6 编写语义规则之前，先来分析图 5.13 的 if 语句结构。当用产生式(2)进行归约时，生成了表达式< Expr >的代码，由于< Expr >.FC 此时还不能回填，因此要通过一个变量来记录它是需要回填的。由于左边的文法符号是< C >，因此为方便起见，就使用非终结符< C >的语义变量< C >.CHAIN 向后传递，以便在后续产生式中使用。当用产生式(4)

归约时,已读取到 else,<S>$^{(1)}$的代码已经生成,即将要生成的是<S>$^{(2)}$的第一条四元式。由于<S>$^{(1)}$本身又可能是一个控制语句(例如,多个 if 语句嵌套的情况),当某种条件不满足时也需要从<S>$^{(1)}$中间某个位置跳出,而且还需要跳过<S>$^{(2)}$的范围,即跳过本条 if 语句去执行 if 语句后面的语句。由于此时<S>$^{(2)}$尚未归约出来,if 语句的后面一个语句的位置还不知道,于是需要设置语义变量<S>$^{(1)}$.CHAIN 来记录<S>$^{(1)}$的转出链。由于<S>$^{(1)}$.CHAIN 和<S>$^{(1)}$后面的 GOTO 0(表示 if 语句的表达式为真时执行语句<S>$^{(1)}$结束后,应跳出 if 语句,图中编号为 q)的四元式的转移地址相同,所以把它们合并成一条链,并由左部非终结符<T>的语义变量<T>.CHAIN 向后传递。而且,由于此时已读取 else,就已知<Expr>的假出口,应回填<C>.CHAIN(即<Expr>.FC),其中使用 backpatch(<C>.CHAIN,NXQ) 函数,表示将 NXQ 回填给以<C>.CHAIN 为链表头的四元式链表中每个四元式的第 4 个分量,参见 5.3.3 节。<S>$^{(2)}$.CHAIN 的含义与<S>$^{(1)}$.CHAIN 相同。产生式(1)和(3) 的语义动作就是将待回填的假出口链合并后向外传递,其中的函数 merge()的功能是合并两个以头指针代表的四元式链,参见 5.3.3 节。此时还不能立刻回填<T>.CHAIN 或<C>.CHAIN,这是考虑语句可能嵌套的情况,转移目标暂且不能确定。因此,最后建立 if 语句总的待填链<IFS>.CHAIN,它是该语句对外的接口,是假出口的链表头,留待转移目标确定后(如遇到";")再回填。

根据上面的分析,文法 G5.6 各产生式的语义规则如表 5.11 所示。

表 5.11 文法 G5.6 的属性文法

编号	产生式	语义规则
(1)	<IFS>→<C><S>$^{(1)}$	{ <IFS>.CHAIN = merge(<C>.CHAIN,<S>$^{(1)}$.CHAIN) }
(2)	<C>→if (<Expr>)	{backpatch(<Expr>.TC,NXQ); <C>.CHAIN=<Expr>.FC }
(3)	<IFS>→<T><S>$^{(2)}$	{ <IFS>.CHAIN= merge(<T>.CHAIN,<S>$^{(2)}$.CHAIN) }
(4)	<T>→<C><S>$^{(1)}$ else	{ q=NXQ; GenCode(j,,,0); backpatch(<C>.CHAIN,NXQ); <T>.CHAIN=merge(<S>$^{(1)}$.CHAIN,q) }

例 5.9 将下面的 if 语句翻译为四元式序列:

```
if (A && B && (C>D) )
    if (A<B) F=1;
        else F=0;
else G=G+1;
```

翻译后四元式序列为:

(1) (jnz, A, , 3) /* A 的四元式 1,2 */

(2) (j, , , 13)

(3) (jnz, B, , 5) /* B 的四元式 3,4 */

(4) (j, , , 13)

(5) (j>, C, D, 7) /* C>D 的四元式 5,6 */

(6) (j, , , 13)

(7)　(j<, A, B, 9)　　/ * A<B 的四元式 7,8 * /

(8)　(j, , , 11)

(9)　(=, 1, , F)　　/ * F=1 的四元式 9 * /

(10) (j, , , 15)　　/ * 第 2 个 if 为真执行的语句结束,应跳出 * /

(11) (=, 0, , F)　　/ * F=0 的四元式 * /

(12) (j, , , 15)　　/ * 第 1 个 else 语句结束,应跳出 * /

(13) (+, G, 1, T_0)　　/ * G=G+1 的四元式 13,14 * /

(14) (=, T_0, , G)

(15)　　/ * 15 为该 if 语句之后的语句的四元式 * /

注意,上面的第(10)条四元式(j,,,15)不仅应跳过 F=0,而且还应跳过 G=G+1,第(12)条四元式的作用是为跳过外层的 else 部分。

另外,根据 if 语句的语义规则,第(10)、(12)条四元式的第 4 个分量应该是不知道转移去向的,也就是应该是 0。但是当读取到最后的分号时,转移去向已知,就可以回填了。所以在翻译各个语句时,该语句总的回填应该是在一个语句的结束(;)处要添加一条回填的处理。

5.4.2 do…while 语句的翻译

Sample 语言中 do…while 语句形如:

<do…while 语句> → do <语句> while (<表达式>);

为了书写方便,将上述文法改写为:

<DOS> → do <S>$^{(1)}$ while(<Expr>);

do…while 语句的执行流程是:先执行<S>$^{(1)}$语句,再计算表达式<Expr>,当其值为假时,不再进行循环,应跳出 do…while 语句。这就需要记录几个位置,其一是当表达式<Expr>为真时,应循环执行<S>$^{(1)}$的代码,因此必须记住<S>$^{(1)}$的开始位置;其二是<Expr>的开始位置也是一个转移目标,因为<S>$^{(1)}$本身也可能是控制语句,当某种条件不满足时,需要从<S>$^{(1)}$的中间某位置跳出,重新计算表达式<Expr>的值,因此必须记录这一位置。这两个位置,可以根据关键字 do 和 while 来确定。

do…while 语句的目标结构如图 5.14 所示。

根据分析,就需要对文法进行改造,在读取到 do 和 while 关键字时进行归约,正好执行相应的语义动作。同时,为简化起见,用<D>表示关键字 do,用<U>代表关键字 while 及之前的部分。这样,do…while 语句经过改造和简写后的文法如 G5.7 所示。

(1) <D>→do

(2) <U>→<D><S>$^{(1)}$ while

(3) <DOS>→U(<Expr>)　　(G5.7)

在分析过程中,当扫描到关键字 do 时,表示下面就要开始生成<S>$^{(1)}$的四元式,用产生式(1)进行归约,此时用语义变量<D>.HEAD 来记录<S>$^{(1)}$的入口。

当扫描到关键字 while 时,就已知<S>$^{(1)}$已经翻译完毕,接下来是<Expr>的代码,此时应该用产生式(2)进行归约,应该用表达式<Expr>的开始位置去回填<S>$^{(1)}$.CHAIN。

当扫描到表达式最后的右括号时,应该用产生式(3)进行归约,此时<Expr>的值已知,就需要回填它的真出口和假出口。当表达式<Expr>为真时,应循环执行<S>$^{(1)}$的代码,因此表达式<Expr>的真出口<Expr>.TC 应为<D>.HEAD,所以需要用<D>.HEAD 回填

< Expr >. TC 链。当< Expr >为假时，应跳出这个循环语句，也就是说，< Expr >的假出口< Expr >. FC 是 do…while 语句之后的语句的第一条四元式的编号，暂时不知道，同时考虑语句嵌套的情况，最后建立 do…while 语句最后总的< DOS >. CHAIN，表示该语句对外的接口，即假出口的链表头，留待向外传递转移目标(如遇到“;”)再回填。

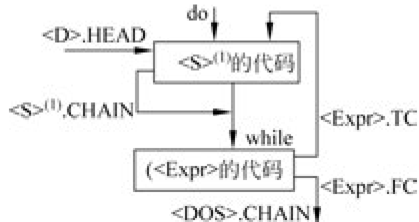

图 5.14　do…while 语句的目标结构

注：图 5.14 没有考虑在循环中遇到 break 的情况，如果< S >$^{(1)}$中遇到了 break，将跳出循环语句，如何添加语义信息，留待读者思考。

文法 G5.7 的各个产生式的语义规则如表 5.12 所示。

表 5.12　文法 G5.7 的属性文法

编　　号	产　生　式	语 义 规 则
(1)	< D >→do	{　< D >. HEAD=NXQ　}
(2)	< U >→< D >< S >$^{(1)}$ while	{　< U >. HEAD= < D >. HEAD; backpatch(< S >$^{(1)}$. CHAIN,NXQ)　}
(3)	< DOS >→< U >(< Expr >)	{　backpatch(< Expr >. TC,< U >. HEAD); < DOS >. CHAIN=< Expr >. FC　}

例 5.10　将下面的语句翻译为四元式序列。

```
if (w<1) a=b*c+d;
    else do a=a-1;
        while( a<0 );
```

解：翻译后四元式序列为：

(1) (j<,w,1,3)　/* w<1 生成两个四元式 1 和 2,真出口回填为 3 */
(2) (j,,,7)　/* 读到 else 时,假出口回填为 7 */
(3) (*,b,c,T_1)　/* a=b*c+d 的四元式 3,4,5 */
(4) (+,T_1,d,T_2)
(5) (=,T_2,,a)
(6) (j,,,10)　/* 跳过 else */
(7) (-,a,1,T_3)　/* a=a-1 的四元式 7,8 */
(8) (=,T_3,,a)
(9) (j<,a,0,7)　/* a<0 的真出口的四元式 9,跳转到 7 进行循环,a<0 的假出口是接下来的一个四元式,即 10 */
(10)

5.4.3　for 语句的翻译

Sample 语言中 for 语句的形式如下：

< for 语句> → for (<表达式>;<表达式>;<表达式>) <语句>

为书写方便，将该文法产生式改写为：

< FORS > → for ($E^{(1)}$; $E^{(2)}$; $E^{(3)}$) $S^{(1)}$

它的目标结构如图 5.15 所示。for 语句的执行顺序是：首先计算初值表达式 $E^{(1)}$；再计算终值表达式 $E^{(2)}$，并将 $E^{(2)}$ 的值存放到一个临时变量 T 中；根据 T 的值是否为 0，来决

定是否执行 $S^{(1)}$ 的代码，当 T 为 0 时，跳出该语句。当 $S^{(1)}$ 的代码执行完毕或从 $S^{(1)}$ 中遇到 continue 语句跳出或者当 $S^{(1)}$ 的某种条件不满足时，需要从 $S^{(1)}$ 的中间某位置跳出，应计算表达式 $E^{(3)}$ 的值；然后循环计算终值表达式 $E^{(2)}$ 的值，重新判断循环条件，决定是否执行 $S^{(1)}$ 的代码。当 $S^{(1)}$ 中遇到 break 等语句时，应跳出循环语句，跳出后转移的目标地址和 T 的值为 0 时的目标地址应该一致，因此，将它们合并为一条链，最后构成该 for 语句对外的出口链<FORS>. CHAIN。

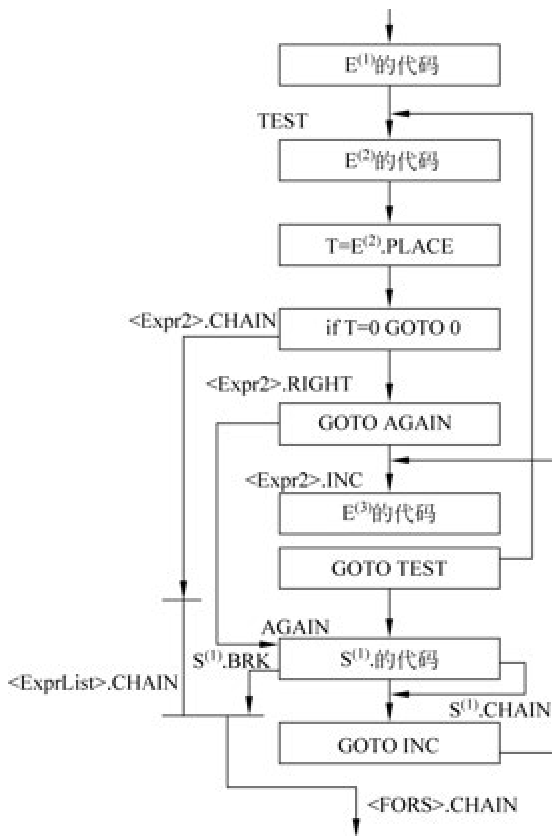

图 5.15　for 语句的目标结构

在这个过程中，应注意以下几点。

(1) 由于读取 token 文件生成中间代码时是从左到右读取的，生成中间代码时是顺序生成 $E^{(1)}$、$E^{(2)}$、$E^{(3)}$、$S^{(1)}$ 的代码，而实际的代码执行顺序是 $E^{(1)}$、$E^{(2)}$、$S^{(1)}$、$E^{(3)}$，再返回执行 $E^{(2)}$ 进行判断，代码翻译的顺序和实际执行的顺序不同，因此应通过相应的跳转语句改变代码的顺序执行流程，一共需要增加 4 条跳转语句。①在 $E^{(2)}$ 执行完后应进行判断增加两条跳转语句：一条是条件跳转，判断 $E^{(2)}$ 的结果是否为 0，如果为 0，则跳出循环；否则无条件跳转到 $S^{(1)}$ 的代码，即跳过 $E^{(3)}$ 的代码；②在 $S^{(1)}$ 的代码执行结束后，应跳转到 $E^{(3)}$ 代码的开始，改变循环变量的值，构成循环；③在 $E^{(3)}$ 代码执行结束后，应跳转到 $E^{(2)}$ 代码的开始，重新进行计算和测试，以决定是否继续循环。

(2) 由于有跳转语句，凡是跳转的目标点必须使用标号记录，因此应记录如下三个位置：表达式 $E^{(2)}$ 代码的入口地址，记为 TEST；表达式 $E^{(3)}$ 代码的入口地址，记为 INC；$S^{(1)}$ 代码的入口地址，记为 AGAIN。

(3) 如果在翻译过程中有暂时不知道的跳转位置,需要后续待填,此时也需要记录待填四元式的编号,还需要用相应的语义变量做记录,如在翻译 $E^{(2)}$ 的代码后,判断 $E^{(2)}$ 的结果不为 0,应跳转到 $S^{(1)}$ 代码处,但此时还没有翻译 $S^{(1)}$ 代码,不知道跳转的位置,因此应将判断 $E^{(2)}$ 的结果不为 0 的四元式编号记录下来,设为 RIGHT,后续知道后再回填。

为了在合适的位置进行归约,如在每个表达式计算完毕,就要进行归约,执行一些语义动作,需要对文法进行改造。同时为简化起见,用< ForHead >表示 for 语句中不带后续语句的部分,用< ExprList >表示 for 语句中括号中的三个表达式,用< Expr2 >表示括号中的前两个表达式,用< Expr1 >表示括号中的第一个表达式,各个表达式以分号为分隔符。改造和简写后的 for 语句文法描述如文法 G5.8 所示,< FORS >是 for 语句文法的开始符号。

(1) < FORS >→< ForHead > $S^{(1)}$

(2) < ForHead >→for (< ExprList >)

(3) < ExprList >→< Expr2 > $E^{(3)}$

(4) < Expr2 >→< Expr1 > $E^{(2)}$;

(5) < Expr1 >→$E^{(1)}$; (G5.8)

文法 G5.8 的语义规则如表 5.13 所示。

表 5.13 文法 G5.8 的属性文法

编号	产生式	语义规则
(1)	< FORS >→< ForHead > $S^{(1)}$	{ backpatch($S^{(1)}$.CHAIN,NXQ); GenCode(j,,,< ForHead >.INC); // $S^{(1)}$ 执行结束也应跳转到 $E^{(3)}$ 执行 < FORS >.CHAIN= merge(< ForHead >.CHAIN,$S^{(1)}$.BRK) }
(2)	< ForHead >→for (< ExprList >)	{ < ForHead >.INC =< ExprList >.INC; < ForHead >.CHAIN =< ExprList >.CHAIN }

(3)	< ExprList >→< Expr2 > $E^{(3)}$	{ GenCode(j,,,< Expr2 >.TEST);
		backpatch(< Expr2 >.RIGHT,NXQ);
		< ExprList >.CHAIN=< Expr2 >.CHAIN ;
		//产生式之间的语义变量的传递
		< ExprList >.INC=< Expr2 >.INC
		//产生式之间的语义变量的传递 }

(4)	< Expr2 >→< Expr1 > $E^{(2)}$;	{ < Expr2 >.PLACE=NewTemp();
		GenCode(=,$E^{(2)}$.PLACE, ,< Expr2 >.PLACE);
		< Expr2 >.CHAIN=NXQ; //对外的出口链
		GenCode(jz,< Expr2 >.PLACE,,0);
		//< Expr2 >.CHAIN,需回填
		< Expr2 >.RIGHT=NXQ ;
		GenCode(j,,,0) ;
		//< Expr2 >.RIGHT,还不知道位置,需回填
		< Expr2 >. INC=NXQ ;
		< Expr2 >.TEST=< Expr1 >.TEST
		//产生式之间语义变量的传递 }

(5)	< Expr1 >→$E^{(1)}$;	{ < Expr1 >.TEST= NXQ }

产生式(5)的语义规则是：当 $E^{(1)}$ 计算完毕，读取到分号时，下一条即将产生的四元式的编号就是 $E^{(2)}$ 的入口地址，应该在此处记录 TEST，使用产生式左部文法符号的语义变量来记录，以便后续传递。

产生式(4)的语义规则是：当 $E^{(2)}$ 计算完毕，读取到分号时，应将 $E^{(2)}$ 的值保存到临时变量 T 中(此处不一定是 T，是用 NewTemp()函数申请的)，然后需要对 T 的值进行判定，如果为 0，则生成一条为 0 跳转的四元式 jz，但需回填，因此应该将其四元式编号保存起来才能回填，用<Expr2>.CHAIN 来记录待回填的四元式编号；否则 T 不为 0，应该生成一条跳转到 $S^{(1)}$ 的无条件跳转四元式 j，但其跳转位置是 $S^{(1)}$ 的开始位置，也还不知道，因此也需要保存其编号，用<Expr2>.RIGHT 来记录。下一条即将产生的四元式的编号是 $E^{(3)}$ 的入口地址，需要用<Expr2>.INC＝NXQ 来记录。同时，还需要传递 TEST 这个语义变量。

产生式(3)的语义规则是：当表达式 $E^{(3)}$ 计算完毕，需要跳转到 $E^{(2)}$ 的开始位置重新计算，这需要生成一条无条件跳转指令 j，此时即将产生的四元式是循环语句 $S^{(1)}$ 的入口，因此，应该回填<Expr2>.RIGHT，也就是 $E^{(2)}$ 为真时的跳转位置已知。同时，还需要传递两个语义变量 INC 和 CHAIN。

产生式(2)的语义规则是：当用产生式(2)进行归约时，没有多余的语义动作，只需要传递两个语义变量 INC 和 CHAIN。

产生式(1)的语义规则是：当遇到中间不满足条件或 continue 语句退出时，应跳转到 $S^{(1)}$ 结束的位置，因此，应该回填 $S^{(1)}$.CHAIN。然后跳转去计算 $E^{(3)}$，这就需再生成一条无条件跳转四元式，跳转到 INC。同时将表达式 $E^{(2)}$ 的假出口与 $S^{(1)}$ 中间遇到 break 语句跳出的待填链合并为 for 语句总的出口链<FORS>.CHAIN，表示该语句对外的接口，即待填信息的链表头，留待向外传递转移目标，如遇到“;”再回填。例如，下面的语句

```
if (A < B)
    for (E(1); E(2); E(3)) S(1)
else S(2);
```

中，for 语句归约为<FORS>之后，其<FORS>.CHAIN 不能立刻回填，必须等待 $S^{(2)}$ 的代码生成之后才能确定，<FORS>.CHAIN 的转向目标是 $S^{(2)}$ 之后的第一个四元式的编号。

例 5.11　将下面的语句翻译为四元式序列：

```
for (i = a+b*2; i < c+d+10; i = i+1)
    if (h > g)
        p=p+1;
```

解：整个语句的翻译可用图 5.16 表示，图中左边的序号表示四元式的编号。依据图示可以写出翻译后四元式序列如下。

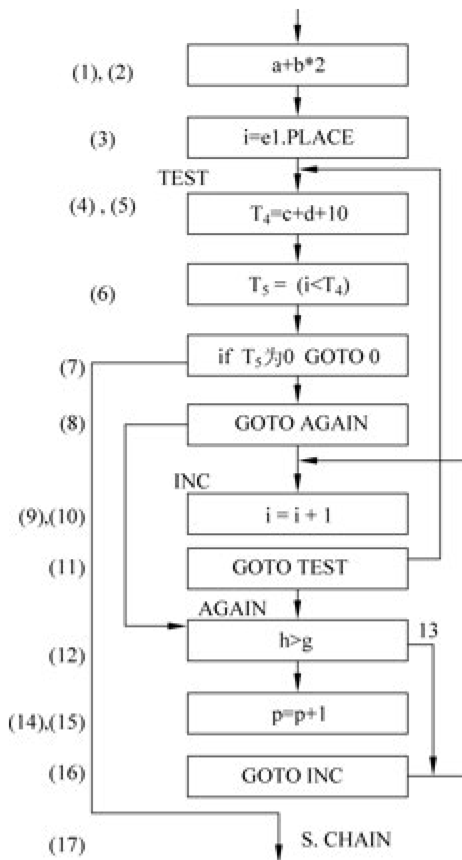

图 5.16　例 5.11 的 for 语句的翻译图示

(1) (*,b,2,T_1)
(2) (+,a,T_1,T_2)
(3) (=,T_2,,i)
(4) (+,c,d,T_3)
(5) (+,T_3,10,T_4)
(6) (<,i,T_4,T_5)
(7) (jz,T_5,,17)
(8) (j,,,12)
(9) (+,i,1,T_6)
(10) (=,T_6,,i)
(11) (j,,,4)
(12) (j>,h,g,14)
(13) (j,,,16)
(14) (+,p,1,T_7)
(15) (=,T_7,,p)
(16) (j,,,9)
(17)

至此,已经给出了高级语言中主要的几种语句的翻译方法。解决问题的基本策略是先研究各种语句的代码结构,然后根据代码结构的特点和只有在归约时才调用语义动作这一关系,对原文法进行适当的改造,使得在归约时能及时调用相应的语义规则,执行语义操作。其他语句(如 while、switch 等)的翻译,留待读者扩展。

5.5 函数定义及函数调用的翻译

函数是程序设计语言中最常用的一种结构,是实现模块化程序设计的基本单位。Sample 语言中和函数相关的有函数声明、函数定义和函数调用。函数声明已经在 4.7.3 节介绍过,所以本节主要介绍函数调用和函数定义的翻译方法。

5.5.1 函数调用过程

Sample 语言的函数调用格式是直接写上函数名,并将实参写在括号中,如 max(1+2,3)。max()称为被调函数,调用 max()的函数称为主调函数。函数调用的过程是:首先计算出各个实参表达式的值,然后根据参数传递方式,将实参的值或地址传递给形参,再跳转到函数对应的代码处去执行,函数执行结束后,返回主程序调用它的下一条指令继续执行。

函数调用的实质是把程序的控制权转移到函数。因此,函数调用应该解决的问题有两个:第一是如何把实参传递给被调用函数的形参;第二是如何告诉函数在它运行结束时应该返回到什么地方继续执行。

关于第一个问题,函数调用的参数传递方式有很多种,参见 6.2 节。不管哪种参数传递方式,所有实参的值或地址都应存放在被调用函数能够取到的地方。如果采用传值的方式,图 5.17 所示的函数调用流程是:首先在主函数中计算实参表达式的值 second+1,并存储

到一个临时变量 T_0 中，然后将实参的值 first 和 T_0 传递给形参 x 和 y，再跳转到函数 max()对应的代码处去执行，x 和 y 就具有了相应的值。

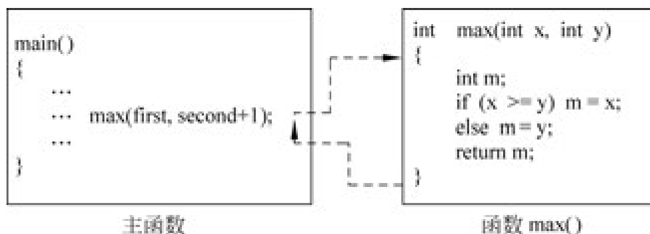

图 5.17 函数调用过程

关于第二个问题，现代计算机的指令系统中一般都有函数调用指令 CALL，这条指令在实现将控制权转移给函数的同时，把返回主函数后执行的指令的地址同时放入相应的寄存器(或内存地址)中保存，一旦遇到函数返回，就能够返回到主函数中。

理解函数调用的语义，需要涉及许多程序运行时的细节。在高级语言层面，我们面对的是变量、函数、表达式和语句等语言元素，在程序运行前，所有的高级语言元素都已经转换为二进制的形式。程序运行时除了要为二进制代码分配内存空间，还要为程序运行所需要的数据分配空间，不同的高级语言分配空间的方案不同。详细的内存组织和分配方式可以参看第 6 章。

Sample 语言的内存分配方式和 C 语言类似，采取栈式存储分配方式，函数调用中实参传递以及局部变量的存储都是由栈来完成的。栈空间是一块连续的内存空间，向低地址方向增长。CPU 的栈指针寄存器 SP 总是指向栈顶位置，当数据入栈时，SP 减小，当数据出栈时，SP 增加。当栈空间不足时，操作系统会自动增加栈的大小。

因此，在函数调用前(程序还在执行 main()函数)，需要将实参压入栈中，当有多个实参时，根据函数调用规则可以从右往左入栈，也可以从左往右入栈。Sample 语言采取的是从右往左入栈的方式。例如，在调用 max(first，second＋1)函数时，需要将 T_0(即 second＋1)和 first 的值依次压入栈中，然后使用 CALL 指令调用 max()函数。

5.5.2 函数调用的翻译

根据上述介绍，在函数调用中需要定义两个新的四元式：

(1) (para，，，value)

其含义是将函数的一个参数压入栈中，para 是运算符，value 是值。

(2) (call，，，entry(id). name)

其含义是调用名为 id. name 的函数。

根据第 3 章中的描述，函数调用的文法如下。

(1) <函数调用>→<函数名>(<实参列表>)

(2) <实参列表>→<实参表达式>|ε

(3) <实参表达式>→<表达式>|<表达式>,<实参表达式>

为书写方便，用< FuncCall >表示<函数调用>，它是函数调用文法的开始符号，<函数名>就是标识符，用< ActualParaList >表示<实参列表>，用< ActualPara >表示<实参表达式>，用< Expr >表示<表达式>，其翻译方法在 5.4 节中已介绍，此处将其值< Expr >. PLACE 作为

一个整体来使用。函数调用文法简写为 G5.9。

(1) <FuncCall>→id(<ActualParaList>)

(2) <ActualParaList>→<ActualPara>|ε

(3) <ActualPara>→<Expr>|<Expr>,<ActualPara> (G5.9)

根据上述介绍,可以为每个产生式配上合适的语义规则,如表 5.14 所示。

表 5.14 文法 G5.9 的属性文法

编号	产 生 式	语 义 规 则
(1)	<FuncCall>→id(<ActualParaList>)	{ GenCode(call,,,entry(id).name) }
(2)	<ActualParaList>→<ActualPara>	{ }
(3)	<ActualPara>→<Expr>	{GenCode(para,,,<Expr>.PLACE) }
(4)	<ActualPara>→<Expr>,<ActualPara>[(1)]	{GenCode(para,,,<Expr>.PLACE) }

根据上述语义规则可知,存入实参时是从右往左传递的,当然,也可以实现为从左往右传递的方式,这要和后续函数执行过程中将其读取到形参时的顺序联系起来。

例 5.12 假定在某主函数中以 max(1+x,y)形式调用了函数,写出生成的四元式序列。

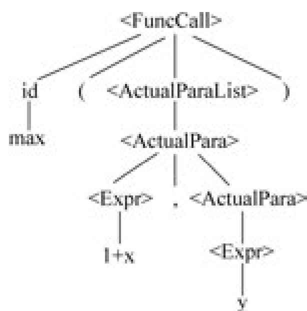

图 5.18 函数 max(1+x,y)的语法树

假定在调用函数之前,已生成了 k 条四元式,函数调用的四元式是从 k+1 条开始的。假定从左到右读入输入符号串进行 LR 分析,语法树如图 5.18 所示。读入 1+x 后,就会调用表达式的产生式,将其归约为<Expr>(注意,这里的 1+x 的语法树没有完全画出,将其作为一个整体,读者可以参考表达式的文法画出细节),调用对应的语义规则,生成一条四元式 k+1,计算 1+x 的值,存入 T_0 中,然后继续读入 y,将 y 归约为<Expr>,继续利用表 5.14 中的产生式(3)将其归约为<ActualPara>时,调用对应的语义规则,将 y 压入栈中,再利用产生式(4)将<Expr>,<ActualPara>归约为<ActualPara>时,调用对应的语义规则,将 T_0 压入栈中,最后利用产生式(1)进行归约,产生一条 call 四元式。得到的完整的四元式序列如下。

(k+1) (+,1,x,T_0)

(k+2) (para,,,y)

(k+3) (para,,,T_0)

(k+4) (call,,,max)

5.5.3 函数定义的翻译

Sample 语言的函数定义均位于主函数定义之后,第 3 章介绍了函数定义的文法如下。

(1) <函数定义>→<函数类型> <函数名>(<函数定义形参列表>) <复合语句>

(2) <函数定义形参列表>→<函数定义形参>|ε

(3) <函数定义形参>→<单形参声明>|<单形参声明>,<函数定义形参>

(4) <单形参声明>→<数据类型> <形参名>

(5) <函数类型>→int|char|float|void

（6）<数据类型>→int|char|float

函数定义的语义是要将函数名和其后的复合语句中的代码对应起来，以便后续函数调用时直接用函数名及参数就可以执行对应的代码。因此，函数定义翻译的关键是确定该函数的代码范围，即从哪条四元式开始，到哪条四元式结束，如何返回主调函数。

第一个问题是如何标记函数代码的开始。假定有函数定义：

```
int sum(int sum_x,int sum_y){
    int result ;
    result = sum_x + sum_y ;
    return result ;
}
```

在读取到函数定义的第一行时，就表明后续就是函数 sum()的代码，因此，在中间代码中也可以定义一个以函数名为运算符的四元式，用来标记函数的开始。

上述函数定义的翻译产生一条四元式代码：

```
(sum,,,)
```

函数的代码范围就是后续复合语句的范围，复合语句中的代码是前面介绍过的语句，可以独立翻译的。

第二个问题是函数执行到哪里结束。在源程序中，函数定义是到复合语句的“}”或者在函数中遇到 return 语句时函数返回，表示函数执行结束。现代计算机中一般有一条机器指令 ret，可以自动恢复函数调用时存入的返回地址，将控制转移回主调函数中。因此，在中间代码中，也可以定义一条函数返回四元式 ret。

```
(ret,,,)
```

这样，在翻译时，当遇到函数定义后面的复合语句结束的“}”时，需要添加这条函数返回四元式，实现函数返回。

有时，函数定义中本身也包含有 return 语句，有如下两种形式。

（1）return;

（2）return return_value;

第一种形式和函数执行结束时的处理一致，不返回值；第二种形式需要向主调函数返回一个值，其值为 return_value。因此，在翻译为中间代码时，也需要记录返回值，定义 ret 四元式的一种新的形式为：

```
(ret,,, return_value )
```

这样就可以将返回值返回到主调函数中。

为书写方便，用< FuncDef >表示<函数定义>，它是函数定义文法的开始符号，用< FuncType >表示<函数类型>，取值为 int、char、float 和 void，用< FuncFParaList >表示<函数定义形参列表>，用< FuncFPara >表示<函数定义形参>，用< SPara >表示<单形参声明>，用< ParaType >表示形参的<数据类型>，取值为 int、char、float。函数名、形参名都是标识符，用< CST >表示<复合语句>，此处当作一个整体来使用，其翻译不在此处，文法简写为 G5.10。

（1）< FuncDef >→< FuncType > id(< FuncFParaList >) < CST >

（2）< FuncType >→int|char|float|void

（3）< FuncFParaList >→< FuncFPara >|ε

(4) < FuncFPara >→< SPara >|< SPara >,< FuncFPara >

(5) < SPara >→< ParaType > id

(6) < ParaType >→int|char|float (G5.10)

函数定义要求在读入函数名后就必须产生四元式将函数名存入四元式表中,这样才能将相应的形参信息,如名字和类型记录在这个函数中;在继续读入每个形参的类型和名字后,将其插入符号表中。后面的复合语句< CST >将产生该函数对应的代码,复合语句结束产生 ret 四元式。为了在合适的位置能够进行语义处理,需要对该文法的第一个产生式进行改写。用< FuncHead >表示函数定义中不带复合语句的函数头部,用< FuncTID >表示函数类型和名字标识符部分。

(1) < FuncDef >→< FuncHead >< CST >

(2) < FuncHead >→< FuncTID >(< FuncFParaList >)

(3) < FuncTID >→< FuncType > id

改写后的文法就可以在需要完成相应的语义动作时进行产生式归约。对应的语义规则如表 5.15 所示。

表 5.15 改写后的函数定义文法的属性文法

编 号	产 生 式	语 义 规 则
(1)	< FuncDef >→< FuncHead >< CST >	{ GenCode(ret,,,) }
(2)	< FuncHead >→ < FuncTID >(< FuncFParaList >)	{ }
(3)	< FuncTID >→< FuncType > id	{ GenCode(entry(id).name,,,) }
(4)	< FuncFParaList >→< FuncFPara >	{ }
(5)	< FuncFPara >→ < SPara >,< FuncFPara >(1)	{ }
(6)	< FuncFPara >→< SPara >	{ }
(7)	< SPara >→< ParaType > id	{ InsertVarible(id.name); //将形参插入符号表中 entry(id).type =< ParaType >.type }
(8)	< ParaType >→int	{ < ParaType >.type=int }
(9)	< FuncType >→int	{ < FuncType >.type=int }

其中,产生式(8)和(9)的多种数据类型均以 int 为例,其他类型的语义规则类似,其作用是记录类型信息。产生式(2)、(4)、(5)、(6)本身不产生多余的语义动作。

在读入函数返回值类型和函数名后,利用产生式(3)进行归约,就可以生成对应的四元式 GenCode(entry(id).name,,,),表明从这一条四元式以后就是这个函数对应的代码。在处理到后面的形参列表时,每读入一个形参名,就可以在符号表中添加相应形参的类型信息,如表中产生式(7)的语义规则。从形参列表的语义规则可知,处理形参列表中的变量是从右往左的。

在利用产生式(1)进行归约时,表示函数定义的代码翻译结束,遇到"}",需要返回,因此需要产生一条表示返回的四元式代码 GenCode(ret,,,)。

例 5.13 有如下函数定义。

```
int sum(int sumx,int sumy,char sumz){
…
}
```

写出其生成的中间代码序列。

该函数定义的语法树如图5.19所示。

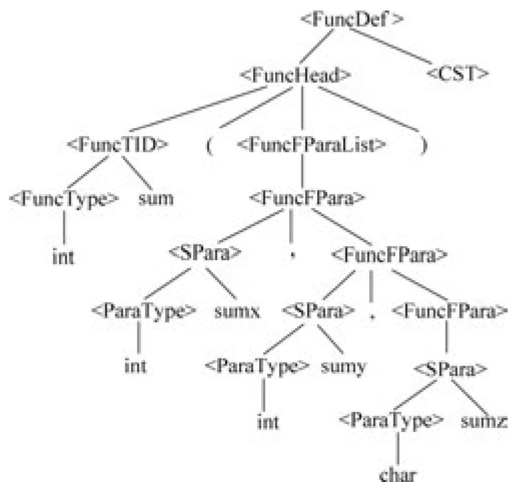

图5.19　函数定义 int sum(int sumx,int sumy,char sumz){…}的语法树

从左到右扫描输入符号串，在读到 int sum 时，进行 LR 归约的过程中，利用产生式(3)归约，调用对应产生式的语义动作，生成一条四元式(i)。

(i) (sum,,,)

然后继续读入后续符号，当(int sumx,int sumy,char sumz)读入完毕，进行归约，将在符号表中插入 sumz、sumy 和 sumx 三个变量，并填入类型信息。

到读取到该函数最后的"}"时，利用第一个产生式归约，再生成一条四元式。

(j) (ret,,,)

四元式的编号(i)和(j)是生成四元式的过程中自动加1得到的。(i)和(j)之间的代码是复合语句中的语句生成的，是函数 sum()生成的中间代码。

5.6　中间代码生成器的设计

本节介绍 Sample 语言的中间代码生成器的设计，主要采用语法制导的翻译方法。Sample 语言的语法分析已经在第3章中介绍过，各个语法成分的文法描述和语义规则在本章前几节中也已经阐述了。只要定义的属性文法满足 L-属性文法的要求，语法制导的翻译就可以和语法分析在一遍中完成，一边进行语法分析，一边进行语义处理。因此，语法制导的翻译程序是在语法分析程序的基础上添加相应的语义处理，并填写符号表，生成四元式表。语法制导翻译程序的接口如图5.20所示。

图5.20　语法制导翻译程序的接口

第3章已经按照递归下降的分析方法进行了语法分析，下面介绍在递归下降分析程序的适当位置添加语义处理，改造为递归下降的翻译器。语法制导翻

译程序的处理流程与语法分析程序的处理流程相同，如图3.10所示。其中，处理常量声明、变量声明和函数声明部分只访问符号表，将相应的名字及相关信息填入符号表。各种表达式和语句的处理要访问符号表，进行静态语义检查，生成四元式序列，存入四元式表中。语句的处理可以嵌套，同时还需要调用表达式的处理。

高级语言程序有开始和结束标记，如C语言中以main()开始，与main()匹配的复合语句结束的右大括号"}"表示整个程序的结束。在转换为中间代码时，也必须表示整个程序的开始和结束，这样才能在最后翻译为目标代码时能够确定程序执行的开始和结束位置。因此在中间代码中，设计了如下两条新的四元式。

(1) (main,,,)，表示主程序开始，是程序的入口。

(2) (sys,,,)，表示主程序结束。

语法制导翻译程序处理主程序的流程是在递归下降语法分析器分析main()函数结构的基础上，添加< MAIN函数定义>的产生式的语义动作。当匹配了"main"后，应生成程序开始的四元式(main,,,)，以便后续在翻译为目标程序时，在执行前能够做一些准备工作，这些工作通常和机器有关，包括保护一些寄存器，设置保护区，为用户程序分配一定的运行空间等。当main()函数结束时，遇到复合语句结束的右大括号"}"，则生成程序结束四元式(sys,,,)，以便将它翻译为汇编代码或机器代码时，能够做一些程序退出处理，如恢复寄存器、释放保护区、退出程序的运行状态等操作。具体内容将在第8章中详细介绍。

在3.3.3节对main()函数进行递归下降分析的函数MainFunctionAnalyzer()中添加语义处理，构成语法制导的翻译程序如下。需要在主函数开始和结束处分别生成表示主程序开始和结束的四元式，在下面的伪代码中用下画线来表示添加的生成四元式的代码。

```
void MainFunctionAnalyzer( ) /* 语法制导翻译的主程序分析部分 */
{
    ...
    if (token == "main" ) {              //分析器读入词法符号 main,主函数开始
        GenCode("main", , , );           // 生成程序开始的四元式
        token = GetNextToken();
        if (token 不是 ''('' ) SyntaxError(); //调用错误处理
        token = GetNextToken();
        if (token 不是 '')'' ) SyntaxError(); //调用错误处理
        CSTAnalyzer();                   //处理复合语句
        GenCode("sys", , ,);             //主函数结束,生成程序结束的四元式
        }
    }
```

然后针对每个函数和语句，将翻译模式中的语义规则加入递归下降分析程序中。具体的设计方法是：(1)在每个非终结符U对应的函数中，对U的语义规则中涉及的每个继承属性都设置一个形参，对涉及的每个综合属性均设置一个变量，整个函数的返回值是U的综合属性（可能有多个），函数的形式为datatype $U(b_1, b_2, \cdots, b_n)$，其中datatype是U的综合属性的返回类型，$b_1, b_2, \cdots, b_n$是为U的n个继承属性设置的形参；(2)在U对应的程序代码中，如果终结符U_i带有综合属性a，把a的值存入为该综合属性设置的变量$U_i.a$中；对非终结符U_j，添加一条对非终结符U_j的综合属性赋值的语句$c = U_j(d_1, d_2, \cdots,$

d_m)，其中 c 是为 U_j 的综合属性设置的变量，d_1，d_2，…，d_m 是为 U_j 的 m 个继承属性设置的形参；对于语义规则中的其他代码，直接写入翻译器中，用代表属性的变量来代替对属性的每一次引用。

下面以 if 语句为例来说明语法制导翻译程序是如何在递归下降分析程序基础上添加语义的。if 语句的递归下降的分析程序参见 3.3.3 节。根据 if 语句的语法制导的翻译方法，将表 5.11 中各个产生式的语义规则加入语法分析程序的适当位置即可。在下面的伪代码中用下画线来标记添加的语义处理部分，用下画波浪线来表示需要修改的部分。由于添加了语义处理，每个函数都要发生变化，如需要有一个返回值，通过它返回待填的真出口或假出口链，因此 if 语句也定义为一个返回 int * 的函数。

```
int *ifs( )                                          //if 语句的语法制导翻译程序
{
    token = GetNextToken();
    if (token 不是 ''('' ) SyntaxError();            //调用错误处理
    token = GetNextToken();
    (e_tc, e_fc) = ExpressionAnalyzer();             //调用分析表达式的函数,返回真假出口*/
    token = GetNextToken();
    if (token 不是 '')'' ) SyntaxError();            //调用错误处理
    backpatch(e.tc, NXQ);                            /*回填布尔表达式的真出口 e.tc*/
    s1.chain = StatementAnalyzer();                  //处理语句,返回待填链首
    token = GetNextToken();
    if(token 是 "else") {                            /*带有 else 的 if 语句,处理 else 部分*/
        q = NXQ;
        GenCode(j, , , 0);                           /* 跳过 s2*/
        backpatch(e.fc, NXQ);                        /*回填布尔表达式的假出口 e.fc*/
        t.chain = merge(s1.chain, q);                /*合并两个链*/
        token = GetNextToken();
        s2.chain =StatementAnalyzer();               //处理 else 后的语句
        return (merge(t.chain, s2.chain));           /*传递整个语句的链*/
    }
    else if (token 是 ";")                           /*无 else 的 if 语句*/
    return(merge(s1.chain, e.fc));                   /*传递整个语句的链*/
}
```

其他的语法成分也可以参考上述方法在语法分析程序的基础上添加语义处理。

5.7 小　　结

中间代码生成的任务是把经过前期分析后正确的源代码翻译为某种形式的中间代码。中间代码是复杂性介于高级语言和低级语言之间的一种表示形式，生成中间代码是为了缩小源语言和目标语言之间的语义鸿沟，如果有必要，则可以设计多层中间代码，采取逐层翻译为不同的中间代码的方式。常见的中间代码形式有逆波兰表示、三地址代码、抽象语法树和有向无环图。三地址代码又包括三元式和四元式，本章主要使用四元式。

语言之间的翻译必须保证语义相同，即高级语言、中间代码和目标语言三者之间的语法结构虽然不同，但它们所表示的含义是一致的。为了更好地进行翻译，需要理解和表示源程序的语义。

高级语言中常见的语法单位包括表达式及表达式语句、各种控制语句、函数和程序等。需要针对每个语法单位的上下文无关文法设计相应的L-属性文法,将每个文法产生式的语义用语义规则的方式表示出来,然后采用语法制导的翻译方法,在语法分析过程中,每推导或归约出一个语法单位时调用相应的语义规则生成中间代码。

针对不同的语法分析方法,设计属性文法的方法不同。与自下而上语法分析同步进行翻译时,需要先归约,再调用语义规则,所以设计属性文法时,需要先改造文法产生式,再对改造后的产生式设计相应的语义规则,在5.3节~5.5节进行了详细介绍。与自上而下语法分析同步进行翻译时,可以采用递归函数内部的局部变量来表示文法符号的语义信息,利用函数之间的参数传递来传递语义信息,利用产生式的语义规则来计算语义信息,在5.6节进行了详细介绍。

本章中用到的所有四元式代码形式参见附录B。

5.8 习　　题

1. 为什么要生成中间代码?常见的中间代码有哪几种形式?

2. 请把逆波兰表示 ab+cde3-/+8*+复原成中缀表达式。

3. 给出下面表达式的逆波兰表示和抽象语法树。

(1) a*(-b+c)

(2) !A|| !(C|| !D)

(3) a+b*(c+d/e)

(4) (A && B)|| (!C|| D)

(5) -a+b*(-c+d)

(6) (A|| B) && (C|| !D && E)

4. 分别给出下述表达式的三元式、四元式序列和DAG。

(1) -(a+b)*(c+d)-(a+b+c)

(2) A|| (B && !(C|| D))

5. 根据while语句的目标结构写出while语句的属性文法。

6. C语言中没有布尔类型,试说明C语言编译器可能使用什么方式将一个if语句翻译为四元式。

7. 将下面的语句翻译为四元式序列。

(1)
```
if ((A<C) && (B<D))
    if (A==1)
        c=c+1;
    else if (A<=D)
        A=A+2;
```

(2)
```
if ((x>0) && (y>0))
        z=x+y;
    else {
        x=x+2;
        y=y+3;
    }
```

(3)
```
do {
    A=A+3;
    B=C*A*2;
}while(X<0);
```

(4)
```
for (i = b*2;i<=100;i=i+1) {
        x= (a+b)*(c+d)-(a+b+c);
    if (x < 0) break;
    }
```

(5)
```
int count = 0,x=0;
    while(x<100) {
      y = x%2;
      if (x =!0) {
            count=count+1;
            continue;
        }
        x=x+1;
    }
```

(6)
```
int sum(int,int);
main()
{
        int x,y,r;
    r=sum(x*y+x,x+y);
  }
 int sum(int a,int b) {
    return a+b;
}
```

8. 用Sample语言编写下述程序,并将其翻译为四元式序列。

(1) 求斐波那契数列。

(2) 判定给定的一个数是否是水仙花数。

(3) 求2到给定数N之间的所有素数。

(4) 用递归的方式求n的阶乘。

(5) 求两个数的最大公约数。

(6) 用函数的方式判定给定年份是否是闰年。

(7) 用函数的方式打印1000以内的所有完数。

(8) 将一个数分解为多个质因数的乘积。

9. 算法程序题。

(1) 用自己熟悉的语言编写程序,其功能是将布尔表达式翻译为四元式代码。

(2) 用自己熟悉的语言编写程序,其功能是将赋值表达式翻译为四元式代码。

(3) 在第3章写的各种控制语句的递归下降分析程序的基础上,实现其递归下降的语法制导翻译程序,翻译为四元式代码。

(4) 理解函数定义和调用,用自己熟悉的语言编写程序,实现函数的四元式代码生成。

10. 延伸阅读与思考。

(1) 查阅资料,了解LLVM和Clang。

(2) 2018年图灵奖获得者John Hennessey和David Patterson在一次演讲上说过:"几十年来的RISC和CISC孰优孰劣之争可以终结了,新一轮计算机架构的黄金时代已经到来。"接着有人说:"目前,编译器迎来了它的黄金时代。"你是如何理解的?

(3) 查阅资料,了解抽象语法树,其作为中间代码的优势在哪里? 它和语法分析树有什么差别? 你还了解其他的中间代码吗?

(4) 查阅资料,选择一款国产编译系统,了解其使用了哪种中间代码表示。

(5) 查阅资料,了解除了语法制导的翻译方法外,还有其他的语义处理和翻译方法吗?