第4章 语义分析和中间代码生成

4.1 概　述

4.1.1 语义分析的概念

一个源程序通过了词法分析、语法分析的审查和处理，表明该源程序在书写上是正确的，符合程序语言所规定的语法。但是语法分析并未对程序内部的逻辑含义加以分析，因此编译程序接下来的工作是语义分析，即审查每个语法成分的静态语义。如果静态语义正确，则生成与该语言成分等效的中间代码，或者直接生成目标代码。

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需中间代码到目标代码的翻译，而中间代码的优点是使编译结构在逻辑上更为简单明确，特别是使目标代码的优化比较容易实现。

如同在进行词法分析、语法分析的同时也进行着词法检查、语法检查一样，在语义分析时也必然要进行语义检查。动态语义检查需要生成相应的目标代码，它是在运行时进行的；静态语义检查是在编译时完成的，它涉及以下几个方面：

(1) 类型检查，如参与运算的操作数其类型应相容。

(2) 控制流检查，用以保证控制语句有合法的转向点。如C语言中不允许goto语句转入switch语句中；break语句需寻找包含它的最小switch、while或for语句方可找到转向点，否则出错。

(3) 一致性检查，如在相同作用域中标识符只能说明一次，switch语句中所有case后的标号不能相同等。

语义分析阶段只产生中间代码而不生成目标代码的方法使编译程序的开发变得较为容易，但语义分析不像词法分析和语法分析那样可以分别用正规文法和上下文无关文法描述。由于语义是上下文有关的，因此语义的形式化描述是非常困难的，目前较为常见的是用属性文法作为描述程序语言语义的工具，并采用语法制导翻译的方法完成对语法成分的翻译工作。

4.1.2 语法制导翻译方法

语法制导翻译的方法就是为每个产生式配上一个翻译子程序(称语义动作或语义子程序)，并在语法分析的同时执行这些子程序。语义动作是为产生式赋予具体意义的手段，它一方面指出了一个产生式所产生的符号串的意义，另一方面又按照这种意义规定了生成某种中间代码应做哪些基本动作。在语法分析过程中，当一个产生式获得匹配(对于自顶向下分析)或用于归约(对于自底向上分析)时，此产生式相应的语义子程序就进入工作，完成既定的翻译任务。

语法制导翻译分为自底向上语法制导翻译和自顶向下语法制导翻译，我们重点介绍自底向上语法制导翻译。

假定有一个自底向上的LR分析器，我们可以把这个LR分析器的能力加以扩大，使它能在用某个产生式进行归约的同时调用相应的语义子程序进行有关的翻译工作。每个产生式的语义子程序执行之后，某些结果(语义信息)必须作为此产生式的左部符号的语义值暂时保存下来，以便以后语义子程序引用这些信息。

此外，原LR分析器的分析栈也加以扩充，以便能够存放与文法符号相对应的语义值。这样，分析栈可以存放三类信息：分析状态、文法符号及文法符号对应的语义值。扩充后的分析栈如图4-1所示。

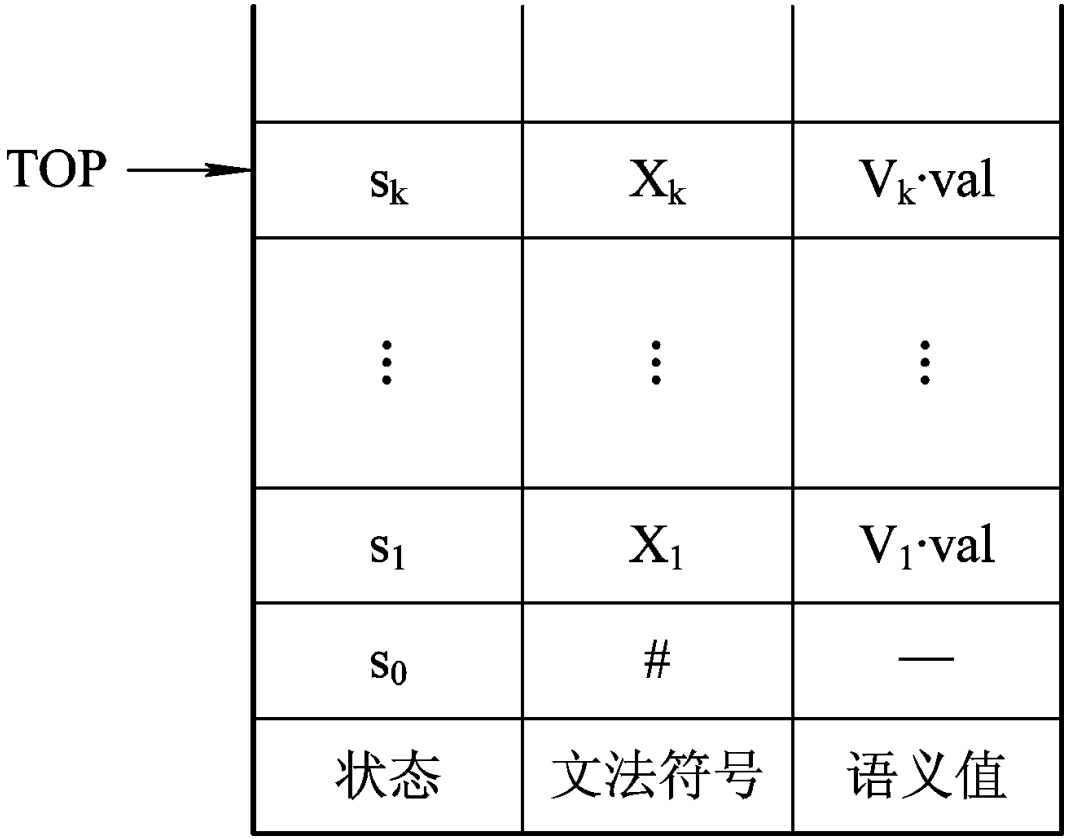


图4-1 扩充后的LR分析栈

作为一个例子，我们考虑下面的文法及语义动作所执行的程序：

作为一个例子，我们考虑下面的文法及语义动作所执行的程序：

产生式	语义动作
(0) $S' \rightarrow E$	print val[TOP]
(1) $E \rightarrow E^{(1)}+E^{(2)}$	val[TOP] =val[TOP]+val[TOP+2]
(2) $E \rightarrow E^{(1)}*E^{(2)}$	val[TOP] =val[TOP]*val[TOP+2]
(3) $E \rightarrow (E^{(1)})$	val[TOP] =val[TOP+1]
(4) $E \rightarrow i$	val[TOP] =lexval (注：lexval 为 i 的整型内部值)

这个文法的 LR 分析表见表 3.21。

我们扩充分析栈工作的总控程序功能，使其在完成语法分析的同时也能完成语义分析工作(这时的语法分析栈已成为语义分析栈)；即在用某一个规则进行归约之后，调用相应的语义子程序完成与所用产生式相应的语义动作，并将每次工作后的语义值保存在扩充后的“语义值”栈中。图4-2表示算术表达式7 + 9*5#的语法树及各结点值，而表4.1则给出了根据表3.21用LR语法制导翻译方法得到的该表达式的语义分析和计值过程。只不过在分析成功到达acc后还要添加执行产生式(0)所对应的语义动作，即输出语义值栈顶的val[TOP] 值52。

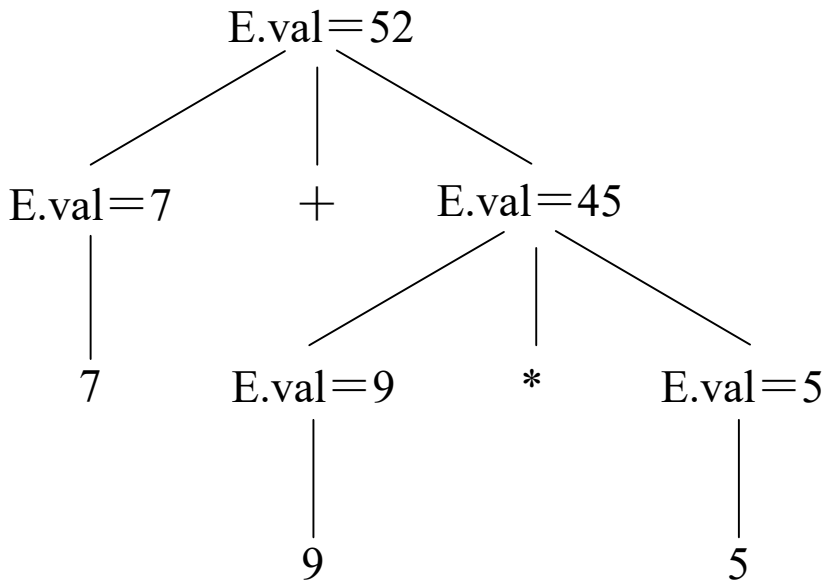


图4-2 语法制导翻译计算表达式7＋9*5# 的语法树

表 4.1 表达式 7 + 9*5# 的语义分析和计值过程

步骤	状态栈	符号栈	语义栈	输入串	主要动作
1	0	#	_	7＋9*5#	s_3
2	03	# 7	_ _	＋9*5#	r_4
3	01	# E	_7	＋9*5#	s_4
4	014	# E+	_7_	9*5#	s_3
5	0143	# E+9	_7_ _	*5#	r_4
6	0147	# E+E	_7_9	*5#	s_5
7	01475	# E+E*	_7_9_	5#	s_3
8	014753	# E+E*5	_7_9_ _	#	r_4
9	014758	# E+E*E	_7_9_5	#	r_2
10	0147	# E+E	_7_45	#	r_1
11	01	# E	_52	#	acc

4.2 属性文法

4.2.1 文法的属性

属性是指与文法符号的类型和值等有关的一些信息，在编译中用属性描述处理对象的特征。随着编译的进展，对语法分析产生的语法树进行语义分析，且分析的结果用中间代码描述出来。对于一棵等待翻译的语法树，它的各个结点都是文法中的一个符号X，该X可以是终结符或非终结符。根据语义处理的需要，在用产生式A→αXβ进行归约或推导时，应能准确而恰当地表达文法符号X在归约或推导时的不同特征。

例如，判断变量X的类型是否匹配，要用X的数据类型来描述；判断变量X是否存在，要用X的存储位置来描述；而对X的运算，则要用X的值来描述。因此，语义分析阶段引入X的属性，如X.type、X.place、X.val等来分别描述变量X的类型、存储位置以及值等不同的特征。

文法符号的属性可分为继承属性与综合属性两类。

继承属性用于“自顶向下”传递信息。继承属性由相应语法树中结点的父结点属性计算得到，即沿语法树向下传递，由根结点到分枝(子)结点，它反映了对上下文依赖的特性。继承属性可以很方便地用来表示程序语言上下文的结构关系。

综合属性用于“自底向上”传递信息。综合属性由相应语法分析树中结点的分枝结点(即子结点)属性计算得到，其传递方向与继承属性相反，即沿语法分析树向上传递，从分枝结点到根结点。

4.2.2 属性文法

属性文法是一种适用于定义语义的特殊文法，即在语言的文法中增加了属性的文法，它将文法符号的语义以“属性”的形式附加到各个文法的符号上(如上述与变量X相关联的属性X.type、X.place和X.val等)，再根据产生式所包含的含义，给出每个文法符号属性的求值规则，从而形成一种带有语义属性的上下文无关文法，即属性文法。属性文法也是一种翻译文法，属性有助于更详细地指定文法中的代码生成动作。

例如，简单算术表达式求值的属性文法如下：

产生式	语义规则
(1) $S \rightarrow E$	print (E.val)
(2) $E \rightarrow E^{(1)}+T$	$E.val = E^{(1)}.val + T.val$
(3) $E \rightarrow T$	E.val =T.val
(4) $T \rightarrow T^{(1)}*F$	$T.val = T^{(1)}.val * F.val$
(5) $T \rightarrow T^{(1)}$	$T.val = T^{(1)}.val$
(6) $F \rightarrow (E)$	F.val =E.val
(7) $F \rightarrow i$	F.val =i.lexval

上面的一组产生式中，每一个非终结符都有一个属性val来表示整型值，如E.val表示E的整型值，而i.lexval则表示i的整型内部值。与产生式关联的每一个语义规则的左部符号E、T、F等的属性值的计算由其各自相应的右部符号决定，这种属性也称为综合属性。与产生式S→E关联的语义规则是一个函数print (E.val)，其功能是打印E产生式的值。S在语义规则中没有出现，可以理解为其属性是一个虚属性。

我们再举一例说明属性文法。一简单变量类型说明的文法G[D] 如下：

G[D]：D→int L | float L
L→L, id | id

其对应的属性文法为

	产生式	语义规则
(1)	D→TL	L.in=T.type
(2)	T→int	T.type=int
(3)	T→float	T.type=float
(4)	L→L$^{(1)}$，id	L$^{(1)}$.in=L.in; addtype (id.entry, L.in)
(5)	L→id	addtype (id.entry, L.in)

注意到与文法G[D]相应的说明语句形式可为

int id_1，id_2，…，id_n 或者 float id_1，id_2，…，id_n

为了把扫描到的每一个标识符id都能及时地填入符号表中，而不必等到待所有标识符都扫描完后再归约为一个标识符表，可以将文法G[D] 改写为属性文法，并在与之关联的语义规则中，用函数addtype把每个标识符id的类型信息(由L.in继承得到)登录在符号表的相关项id.entry中。

非终结符T有一个综合属性type，其值为int或float。语义规则L.in=T.type表示L.in的属性值由相应说明语句指定的类型T.type决定。属性L.in被确定后将随语法树的逐步生成而传递到下边的有关结点使用，这种结点属性称为继承属性。由此可见，标识符的类型可以通过继承属性的复写规则来传递。例如，对输入串int a,b，根据上述的语义规则，可在其生成的语法树中看到用“→”表示的属性传递情况，如图4-3所示。

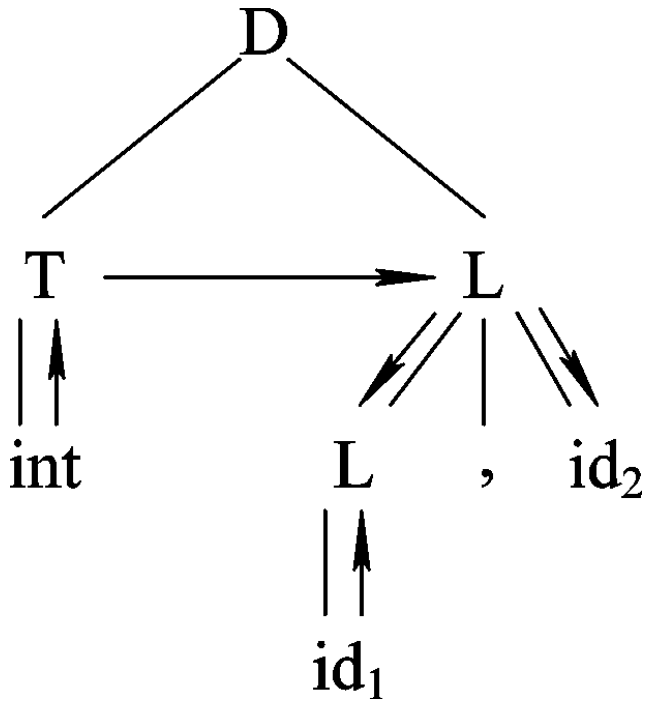


图4-3 属性信息传递情况示意

4.3 几种常见的中间语言

4.3.1 抽象语法树

抽象语法树也称图表示，是一种较为流行的中间语言表示形式。在抽象语法树表示中，每一个叶结点都表示诸如常量或变量这样的运算对象，而其它内部结点则表示运算符。抽象语法树不同于前述的语法树，它展示了一个操作过程并同时描述了源程序的层次结构。

注意：语法规则中包含的某些符号可能起标点符号作用也可能起解释作用。如赋值语句语法规则：

$$S \rightarrow V=e$$

其中的赋值号“=”仅起标点符号作用，其目的是把V与e分开；而条件语句语法规则：

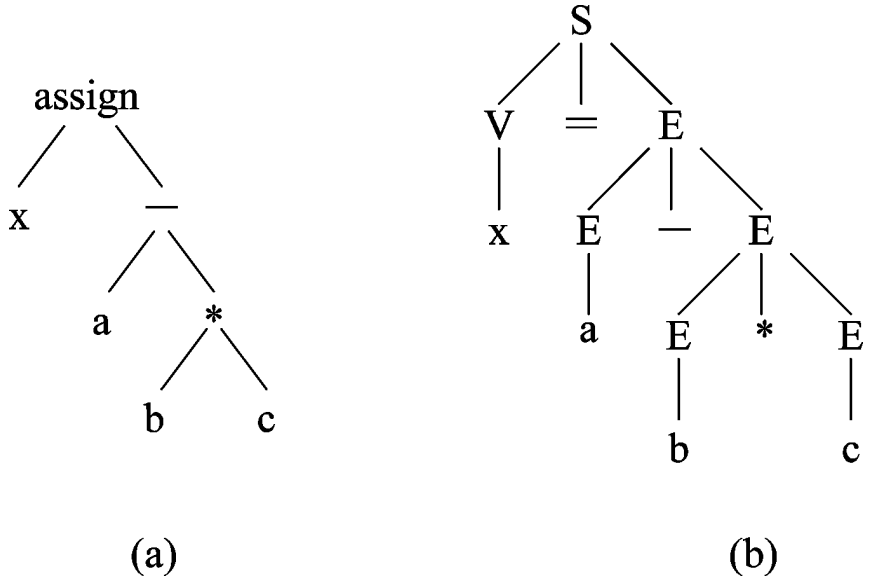
$$S \rightarrow if(e)S_1; else\ S_2$$

其中的保留字符号if和else起注释作用，说明当布尔表达式e为真时执行S_1，否则执行S_2；而“；”仅起标点符号作用。可以看出，上述语句的本质部分是V、e和S_i。在把语法规则中本质部分抽象出来而将非本质部分去掉后，便得到抽象语法规则。这种去掉不必要信息的做法可以获得高效的源程序中间表示。上述语句的抽象语法规则为

(1) 赋值语句：左部 表达式

(2) 条件语句：表达式 语句1 语句2

抽象语法相对应的语法树称为抽象语法树或抽象树，如赋值语句x=a−b*c的抽象语法树如图4-4(a) 所示，而图4-4(b)则是该赋值语句的普通语法树。



(a) (b)

(a) 抽象语法树；(b) 普通语法树

图4-4 x=a−b*c的语法树

抽象语法树的一个显著特点是结构紧凑，容易构造且结点数较少。图4-4(b) 所示的普通语法树的结点为14个；而图4-4(a) 所示的抽象语法树的结点仅有7个，且每个内部结点最多只有两个分支，因此可以将每个赋值语句或表达式表示为一棵二叉树。对于含有多元运算的更为复杂的语法成分，相应的抽象语法树则为一棵多叉树，但我们总可以将其转变为一棵二叉树。

4.3.2 逆波兰表示法

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，这种表示法把运算量(操作数)写在前面，把运算符写在后面，因而又称后缀表示法。例如，把a+b写成ab+，把a*(b+c)写成abc+*。

1．表达式的逆波兰表示

表达式E的后缀表示的递归定义如下：

(1) 如果E是变量或常数，则E的后缀表示即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是二元运算符，而E_1 '、E_2'分别又是E_1 和E_2的后缀表示。若op为一元运算符，则视E_1 和E_1 '为空。

(3) 如果E为 (E_1) 形式，则E_1 的后缀表示即为E的后缀表示。

上述递归定义的实质是：后缀表示中，操作数出现的顺序与原来一致，而运算符则按运算先后的顺序放入相应的操作数之后(即运算符先后的顺序发生了变化)。这种表示已不再需要用括号来规定运算的顺序了。后缀表示中的计值用栈实现非常方便。一般的计值过程是自左至右扫描后缀表达式，每碰到运算量就把它推进栈，每碰到K目运算符就把它作用于栈顶的K个运算量，并用运算的结果(即一个运算量)来取代栈顶的K个运算量。

例**4.1** 将 ((a+b)*(a-c)-d) * (x+y) 翻译成逆波兰表示。

[解答] 按后缀表示递归定义翻译如下：

(1) E：$\underbrace{((a+b)*(a-c)-d)}_{E_1} * \underbrace{(x+y)}_{E_2} \Rightarrow E_1E_2*$

(2) E_1：$\underbrace{(a+b)*(a-c)}_{E_3}-d \Rightarrow E_3d-$

(3) E_2：$x+y \Rightarrow xy+$

(4) E_3：$\underbrace{(a+b)}_{E_4} * \underbrace{(a-c)}_{E_5} \Rightarrow E_4E_5*$

(5) E_4：$a+b \Rightarrow ab+$

(6) E_5：$a-c \Rightarrow ac-$

(7) 将(5)、(6)代入(4)得：$E_3=E_4E_5*=ab+ac-*$

(8) 将 E_3 代入(2)得：$E_1=E_3d-=ab+ac-*d-$

(9) 将(8)、(3)代入(1)得：$E=E_1E_2*=ab+ac-*d-xy+*$

布尔运算符的运算顺序一般为¬ 、∧、∨，因此布尔表达式也可用逆波兰表示。

2．程序语句的逆波兰表示

为了用逆波兰式表示一些控制语句，我们定义转移操作如下：

(1) BL：转向某标号；

(2) BT：条件为真时转移；

(3) BF：条件为假时转移；

(4) BR：无条件转移。

部分程序语句的逆波兰表示如下：

(1) 赋值语句。赋值语句“<左部>=<表达式>”的逆波兰表示为

<左部><表达式>=

例如，赋值语句“x=a+b*c”可按逆波兰式写为“xabc*+=”。

(2) GOTO语句。转向语句“GOTO<语句标号>”的逆波兰表示为

<语句标号>BL

其中，“BL”为单目后缀运算符，“<语句标号>”则为BL的一个运算分量。

(3) 条件语句。BR表示无条件转移单目后缀运算符。例如，“<顺序号>BR”表示无条件转移到“<顺序号>”处，这里的顺序号是BR的一个特殊运算分量，用来表示逆波兰式中单词符号的顺序号(即第几个单词)，它不同于GOTO语句中的语句标号。BT和BF表示按条件转移的两个双目后缀运算符。例如：

<布尔表达式e的逆波兰式><顺序号>BT

<布尔表达式e的逆波兰式><顺序号>BF

分别表示当e为真或假时转移到顺序号处。其中，布尔表达式e的逆波兰式和顺序号是两个特殊的运算分量。若使用BT和BF两个运算符，则条件语句if(e) S_1;else S_2的逆波兰式为

<e的逆波兰式> <顺序号1> BF　　//e为假则转到S_2的第一个单词的顺序号处

<S_1的逆波兰式>　　//e为真则执行S_1

<顺序号2>BR　　//S_1执行结束后无条件转出该条件语句

<S_2的逆波兰式>

例如，条件语句if(m<n) k=i+1；else k=i−1的逆波兰式表示为((1)～(18)为单词编号)

(1) mn<
(4) 13BF
(6) ki1+=
(11) 18BR
(13) ki1−=
(18) {if 语句的后继语句}

此逆波兰式也可写在一行上，即mn<13BFki1+=18BRki1−=。

(4) 循环语句。for循环语句为for(i=m；i<=n；i++)S。其中，i为循环控制变量，m为初值，n为终值，S为循环体。循环语句不能直接用逆波兰表示，因而将其展开为等价的条件语句后再用逆波兰表示，即

```
    i=m;
10: if( i<=n )
    {   S;
        i=i+1;
    }
    if(i<=n) goto 10;
```

4.3.3 三地址代码

1. 三地址代码的形式

三地址代码语句的一般形式为

$$x=y\ op\ z$$

其中，x、y和z为名字、常量或编译时产生的临时变量；op为运算符，如定点运算符、浮点运算符和逻辑运算符等。三地址代码的每条语句通常包含三个地址，两个用来存放运算对象，一个用来存放运算结果。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。

由于三地址语句只含有一个运算符，因此多个运算符组成的表达式必须用三地址语句序列来表示，如表达式x+y*z的三地址代码为

$$t1=y*z$$

$$t2= x+t_1$$

其中，t_1和t_2是编译时产生的临时变量。三地址代码是语法树的一种线性表示，如图4-4(a)所示的语法树用三地址代码表示为

$$t_1=b*c$$

$$t_2=a-t_1$$

$$x=t_2$$

2．三地址语句的种类

作为中间语言的三地址语句非常类似于汇编代码，它可以有符号标号和各种控制流语句。常用的三地址语句有以下几种：

(1) x=y op z形式的赋值语句，其中op为二目的算术运算符或逻辑运算符。

(2) x=op y形式的赋值语句，其中op为一目运算符，如一目减uminus、逻辑否定not、移位运算符以及将定点数转换成浮点数的类型转换符。

(3) x=y形式的赋值语句，将y的值赋给x。

(4) 无条件转移语句goto L，即下一个将被执行的语句是标号为L的语句。

(5) 条件转移语句if x rop y goto L，其中rop为关系运算符，如<、<=、==、!=、>、>=等。若x和y满足关系rop就转去执行标号为L的语句，否则继续按顺序执行本语句的下一条语句。

(6) 过程调用语句par X和call P,n。源程序中的过程调用语句P(X_1, X_2, …, X_n) 可用下列三地址代码表示：

par X_1
par X_2
⋮
par X_n
call P,n

其中，整数n为实参个数。

过程返回语句为return y，其中y为过程返回值。

(7) 变址赋值语句x=y[i]，其中x、y、i均代表数据对象，表示把从地址y开始的第i个地址单元中的值赋给x。x[i] =y则表示把y的值赋给从地址x开始的第i个地址单元。

(8) 地址和指针赋值语句：

① x=&y表示将y的地址赋给x，y可以是一个名字或一个临时变量，而x是指针名或临时变量；

② x=*y表示将y所指示的地址单元中的内容(值)赋给x，y是一个指针或临时变量；

③ *x=y表示将x所指对象的值置为y的值。

3．三地址代码的具体实现

三地址代码是中间代码的一种抽象形式。在编译程序中，三地址代码语言的具体实现通常有三种表示方法：四元式、三元式和间接三元式。

1) 四元式

四元式是具有四个域的记录(即结构体)结构，这四个域为

(op, arg1, arg2, result)

其中，op为运算符；arg1、arg2及result为指针，它们可指向有关名字在符号表中的登记项或一临时变量(也可空缺)。常用的三地址语句与相应的四元式对应如下：

x=y op z	对应 (op, y, z, x)
x=−y	对应 (uminus, y, _, x)
x=y	对应 (=, y, _, x)
par x1	对应 (par, x1, _, _)
call P	对应 (call, _, _, P)
goto L	对应 (j, _, _, L)
if x rop y goto L	对应 (jrop, x, y, L)

例如，赋值语句a=b*(c+d)相应的四元式代码为

① (+, c, d, t_1)

② (*, b, t_1, t_2)

③ (=, t_2, _, a)

我们约定：凡只需一个运算量的算符一律使用arg1。此外，注意这样一个规则：如果op是一个算术或逻辑运算符，则result总是一个新引进的临时变量，它用来存放运算结果。由上例也可看出，四元式出现的顺序与表达式计值的顺序是一致的，四元式之间的联系是通过临时变量实现的。四元式由于其表示更接近程序设计的习惯而成为一种普遍采用的中间代码形式。

2) 三元式

三元式是具有三个域的记录结构，这三个域为

(op, arg1, arg2)

其中，op为运算符；arg1、arg2既可指向有关名字在符号表中的登记项或临时变量，也可以指向三元式表中的某一个三元式。例如，相应于赋值语句a=(b+c)*(b+c) 的三元式代码为

① (+, b, c)

② (+, b, c)

③ (*, ①, ②)

④ (=, a, ③)

上述三元式 ③ 表示 ① 的结果与 ② 的结果相乘。由上例可知，三元式出现的先后顺序和表达式各部分的计值顺序是一致的。

3) 间接三元式

在三元式代码表的基础上另设一张表，该表按运算的次序列出相应三元式在三元式表中的位置，这张表称为间接码表。三元式表只记录不同的三元式语句，而间接码表则表示由这些语句组成的运算次序。例如，赋值语句

a=(b+c)*(b+c) 对应的三元式表与间接码表为

三元式表：① (+, b, c)
② (*, ①, ①)
③ (=, a, ②)

间接码表：① ① ② ③

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它三元式引用；二是三元式位置顺序即为运算顺序。在代码优化阶段，当需要调整三元式的运算顺序时会遇到困难，这是因为三元式中的arg1、arg2也可以是指向某些三元式位置的指针，当这些三元式的位置顺序发生变化时，含有指向这些三元式位置指针的相关三元式也需随之改变指针值。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，而间接三元式则通过间接码表来描述语句的运算次序。这两种方法都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，四元式与间接三元式都比三元式方便得多。

4.4 表达式及赋值语句的翻译

4.4.1 简单算术表达式和赋值语句的翻译

简单算术表达式是一种仅含简单变量的算术表达式。简单变量是指普通变量和常数，但不含数组元素及结构体成员引用等复合型数据结构。简单算术表达式的计值顺序与四元式出现的顺序相同，因此很容易将其翻译成四元式形式，这些翻译方法稍加修改也可用于产生三元式或间接三元式。

考虑以下文法G[A]：A→i=E

E→E+E｜E*E｜−E｜(E)｜i

在此，非终结符A代表“赋值句”。文法G[A]虽然是一个二义文法，但通过确定运算符的结合性及规定运算符的优先级就可避免二义性的发生。

为了实现由表达式到四元式的翻译，需要给文法加上语义子程序，以便在进行归约的同时执行对应的语义子程序。语义子程序所涉及的语义变量、语义函数说明如下：

(1) 对非终结符E定义语义变量E.place，即用E.place表示存放E值的变量名在符号表中的入口地址或临时变量名的整数码。

(2) 定义语义函数newtemp( )，即每次调用newtemp( ) 时都将回送一个代表新临时变量的整数码；临时变量名按产生的顺序可设为T_1、T_2、…。

(3) 定义语义函数emit(op, arg1, arg2, result)，emit的功能是产生一个四元式并填入四元式表中。

(4) 定义语义函数lookup(i.name)，其功能是审查i.name是否出现在符号表中，是则返回i.name在符号表的入口指针，否则返回NULL。

使用上述语义变量和函数，可写出文法G[A] 中的每一个产生式的语义子程序。

(1) A→i=E
{ p=lookup(i.name);
if(p==NULL) error( );
else emit(=, E.place, _, P); }

(2) $E \rightarrow E^{(1)}+E^{(2)}$
{ E.place=newtemp( );
emit(+, $E^{(1)}$.place, $E^{(2)}$.place, E.place); }

(3) $E \rightarrow E^{(1)}*E^{(2)}$
{ E.place=newtemp( );
emit(*, $E^{(1)}$ .place, $E^{(2)}$.place, E.place); }

(4) $E \rightarrow -E^{(1)}$
{ E.place=newtemp( );
emit(uminus, $E^{(1)}$.place, _, E.place); }

(5) $E \rightarrow (E^{(1)})$
{ E.place= $E^{(1)}$.place; }

(6) E→i
{ p=lookup(i.name);
if(p!=NULL) E.place=p;
//另一种表示为 E.place=entry(i)
else error( ); }

例**4.2** 试分析赋值语句X=−B*(C+D) 的语法制导翻译过程。

[解答] 赋值语句X=−B*(C+D) 的语法制导翻译过程如表4.2所示(加工分析过程参考表4.1)。

表 4.2 赋值语句 X=-B*(C+D) 的翻译过程

输入串	归约产生式	符号栈	语义栈(place)	四元式
X=-B*(C+D)#		#	_	
=-B*(C+D)#	(6)	#i	_X	
-B*(C+D)#		#i=	_X_	
B*(C+D)#		#i=-	_X_ _	
*(C+D)#	(6)	#i=-i	_X_ _B	
*(C+D)#	(4)	#i=-E	_X_ _B	(uminus, B, _, T_1)
*(C+D)#		#i=E	_X_T_1	
(C+D)#		#i=E*	_X_T_1_	
C+D)#		#i=E*(	_X_T_1_ _	
+D)#	(6)	#i=E*(i	_X_T_1_ _C	
+D)#		#i=E*(E	_X_T_1_ _C	

续表

输入串	归约产生式	符号栈	语义栈(place)	四元式
D)#		#i=E*(E+	_X_T_1_ _C_	
)#	(6)	#i=E*(E+i	_X_T_1_ _C_D	
)#	(2)	#i=E*(E+E	_X_T_1_ _C_D	(+, C, D, T_2)
)#		#i=E*(E	_X_T_1_ _T_2	
#	(5)	#i=E*(E)	_X_T_1_ _T_2_	
#	(3)	#i=E*E	_X_T_1_T_2	(*, T_1, T_2, T_3)
#	(1)	#i=E	_X_T_3	(=, T_3, _, X)
#		#A	_X	

4.4.2 布尔表达式的翻译

在程序语言中，布尔表达式一般由运算符与运算对象组成。布尔表达式的运算符为布尔运算符，即┐、∧、∨，或为not、and和or(注：C语言中为！、&&和||)，其运算对象为布尔变量，也可为常量或关系表达式。关系表达式的运算对象为算术表达式，其运算符为关系运算符<、<=、==、!=、>=、>等。关系运算符的优先级相同但不得结合，其运算优先级低于任何算术运算符。布尔运算符的运算顺序一般为┐、∧、∨，且∧和∨服从左结合。布尔算符的运算优先级低于任何关系运算符(注意，此处的运算优先级约定不同于C语言)。

此外，对布尔运算、关系运算、算术运算的运算对象的类型可不区分布尔型或算术型，假定不同类型的变换工作将在需要时强制执行。为简单起见，我们遵循以上运算约定讨论下述文法G[E]生成的布尔表达式：

G[E]：E→E∧E | E∨E | ┐E | (E) | i rop i | i

注意：布尔表达式在程序语言中不仅用作计算布尔值，还作为控制语句(如if-else、while等)的条件表达式，用以确定程序的控制流向。无论布尔表达式的作用如何，按照程序执行的顺序，都必须先计算出布尔表达式的值。此外，rop代表六个关系运算符。

计算布尔表达式的值通常有两种方法。第一种方法是仿照计算算术表达式的方法，按布尔表达式的运算顺序一步步地计算出真假值来。假定逻辑值true用1表示、false用0表示，则布尔表达式$1\vee(\neg 0\wedge 0)\vee 0$值的计算过程为

$$
\begin{aligned}
&1\vee(\neg 0\wedge 0)\vee 0\\
=&1\vee(1\wedge 0)\vee 0\\
=&1\vee 0\vee 0\\
=&1\vee 0\\
=&1
\end{aligned}
$$

另一种方法是根据布尔运算的特点实施某种优化，即不必一步一步地计算布尔表达式中所有运算对象的值，而是省略不影响运算结果的运算。例如，在计算A∨B时，若计算出的A值为1，则B值就无需再计算了；因为不管B的结果是什么，A∨B的值都为1。同理，在计算A∧B时若发现A值为0，则B值也无需计算，A∧B的值一定为0。

注意：上述优化措施在布尔表达式含有布尔函数调用且函数调用引起副作用(指对全局量的赋值)时就会出现问题。如下面的两个等效的C语言程序，因其布尔表达式书写的顺序不同而得到不同的结果：

(1)
```
# include “stdio.h”
int p;
int f( )
{
    p=0;
    return(0);
}
void main( )
{
    p=1;
    if(p || f( )) printf("True! \n");
    else printf("False! \n");
}
```

(2)
```
# include “stdio.h”
int p;
int f( )
{
    p=0;
    return(0);
}
void main( )
{
    p=1;
    if( f( ) || p) printf("True! \n");
    else printf("False! \n");
}
```

如何确定一个表达式的真假出口呢？考虑表达式$E^{(1)} \vee E^{(2)}$，若$E^{(1)}$为真，则立即知道E也为真，因此，$E^{(1)}$的真出口也就是整个E的真出口；若$E^{(1)}$为假，则$E^{(2)}$必须被计值，此时$E^{(2)}$的第一个四元式就是$E^{(1)}$的假出口。当然，$E^{(2)}$的真假出口也就是整个E的真假出口。类似的考虑适用于对$E^{(1)} \wedge E^{(2)}$的翻译。我们将$E^{(1)} \vee E^{(2)}$和$E^{(1)} \wedge E^{(2)}$的翻译用图4-5表示，而对形如$\neg E^{(1)}$的表达式则只需调换$E^{(1)}$的真假出口就可得到该表达式E的真假出口。

(a) $E^{(1)} \vee E^{(2)}$；(b) $E^{(1)} \wedge E^{(2)}$

图4-5 E(1)∨E(2)和E(1)∧E(2)的翻译图

在自底向上的分析过程中，一个布尔式的真假出口往往不能在产生四元式的同时就填上，我们只好把这种未完成的四元式的地址(编号)作为E的语义值暂存起来，待整个表达式的四元式产生完毕之后，再来填写这个未填入的转移目标。

对于每个非终结符E，我们需要为它赋予两个语义值E.tc和E.fc，以分别记录E所对应的四元式需要回填“真”、“假”出口的四元式地址所构成的链。这是因为在翻译过程中，常常会出现若干转移四元式转向同一个目标但目标位置又未确定的情况，此时可用“拉链”的方法将这些四元式链接起来，待获得转移目标的四元式地址时再进行返填。例如，假定E的四元式需要回填“真”出口的有p、q、r这三个四元式，则它们可链接成如图4-6所示的一条真值链(记作tc)。

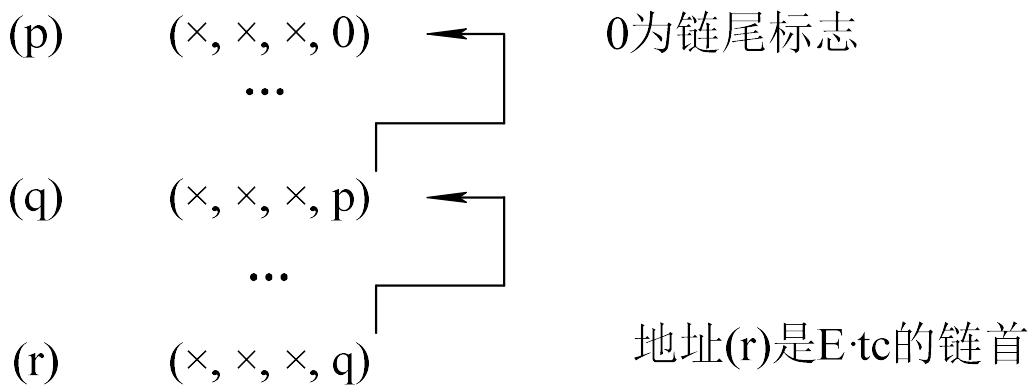


图4-6 拉链法链接四元式示意

为了处理E.tc和E.fc这两项语义值，我们需要引入如下的语义变量和函数：

(1) nxq：始终指向下一条将要产生的四元式的地址(序号)，其初值为1。每执行一次emit语句后，nxq自动增1。

(2) merge(p_1, p_2)：把以p_1和p_2为链首的两条链合并为一条以p2为链首的链(即返回链首值p_2)。

(3) backpatch(p, t)：把链首p所链接的每个四元式的第四区段(即result)都改写为地址t。

merge( )函数如下：

```
int merge( p1, p2 )
{
    if(p2==0)   return( p1 );
    else
    {
        p=p2;
        while( 四元式 p 的第四区段内容不为 0 )
            p=四元式 p 的第四区段内容;
        把 p1 填进四元式 p 的第四区段;
        return( p2 );
    }
}
```

backpatch( )函数如下：

```
void backpatch( p, t )
{
    Q=p;
    while( Q!=0 )
    {
        q=四元式 Q 的第四区段内容;
        把 t 填进四元式 Q 的第四区段;
        Q=q;
    }
}
```

为了便于实现布尔表达式的语法制导翻译，并在扫描到“∧”与“∨”时能及时回填一些已经确定了的待填转移目标，我们将前述文法G[E] 改写为下面的文法G'[E]，以利于编制相应的语义子程序：

G'[E]：E→EAE | EBE | ┐E | (E) | i rop i | i

EA→E∧

EB→E∨

这时，文法G'[E]的每个产生式和相应的语义子程序如下：

(1) $E \rightarrow i$ { E.tc=nxq; E.fc=nxq+1;
emit(jnz, entry(i), _, 0);
emit(j, _, _, 0); }

(2) $E \rightarrow i^{(1)}$ rop $i^{(2)}$ { E.tc=nxq; E.fc=nxq+1;
emit(jrop, entry($i^{(1)}$), entry($i^{(2)}$), 0);
emit(j, _, _, 0); }

(3) $E \rightarrow (E^{(1)})$ { E.tc=$E^{(1)}$.tc; E.fc=$E^{(1)}$.fc; }

(4) $E \rightarrow \neg E^{(1)}$ { E.tc=$E^{(1)}$.fc; E.fc=$E^{(1)}$.tc; }

(5) $E^{A} \rightarrow E^{(1)} \wedge$ { backpatch($E^{(1)}$.tc, nxq);
E^{A}.fc=$E^{(1)}$.fc; }

(6) $E \rightarrow E^{A} E^{(2)}$ { E.tc=$E^{(2)}$.tc;
E.fc=merge(E^{A}.fc, $E^{(2)}$.fc); }

(7) $E^{B} \rightarrow E^{(1)} \vee$ { backpatch($E^{(1)}$.fc, nxq);
E^{B}.tc=$E^{(1)}$.tc; }

(8) $E \rightarrow E^{B} E^{(2)}$ { E.fc=$E^{(2)}$.fc;
E.tc=merge(E^{B}.tc, $E^{(2)}$.tc); }

注意：根据上述语义动作，在整个布尔表达式所对应的四元式全部产生之后，作为整个表达式的真假出口(转移目标)仍尚待回填。此外，由产生式(2)也可看出关系表达式的优先级要高于布尔表达式。

例**4.3** 试给出布尔表达式a∧b∨c≥d作为控制条件的四元式中间代码。

[解答] 设四元式序号从100开始，则布尔表达式a∧b∨c≥d的分析过程如图4-7所示。

即：

```
100 (jnz, a, _, 102)
101 (j, _, _, 104)
102 (jnz, b, _, 106)
103 (j, _, _, 104)
104 (j≥, c, d, 106)
105 (j, _, _, q)
T:  106
    :
F:  q
```

当然，我们也可以通过图 4-8 的分析得到上述四元式序列。

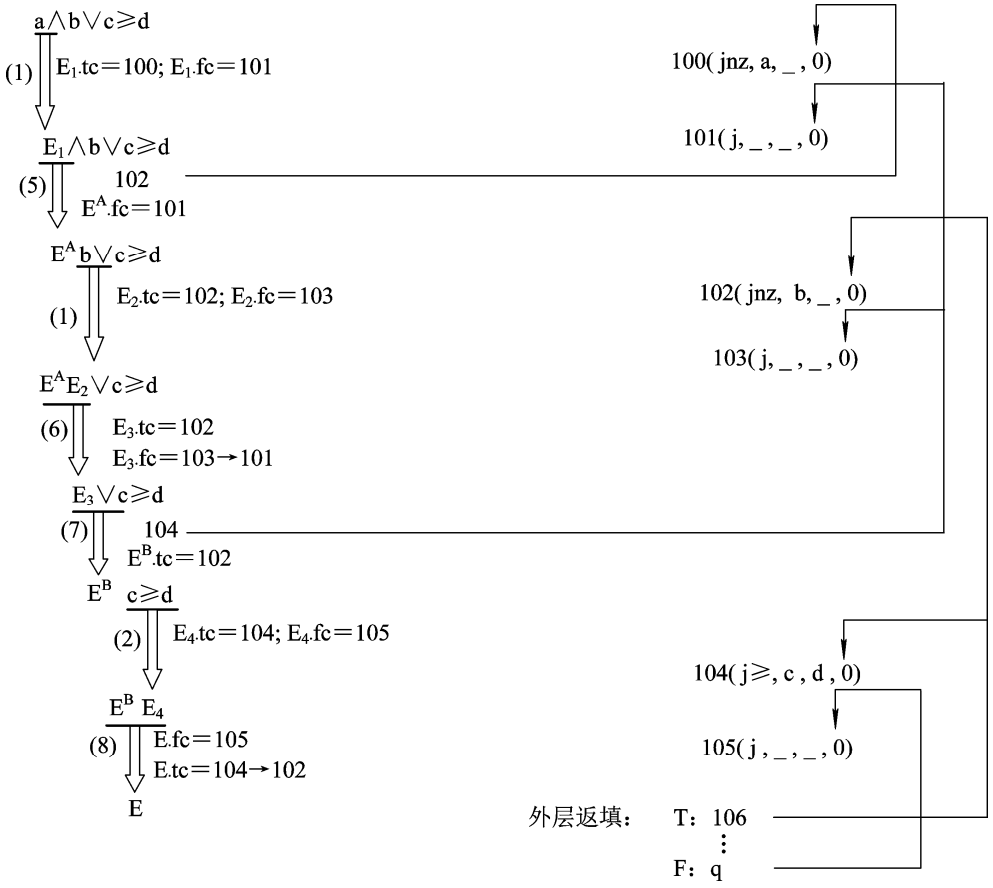


图4-7 表达式a∧b∨c≥d分析示意

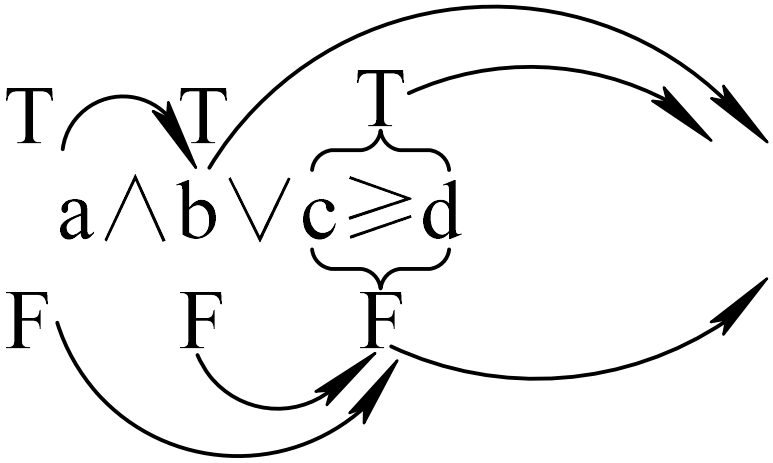


图4-8 a∧b∨c≥d的翻译图

由例4.3可知，每一个布尔变量a都对应一真一假两个四元式，并且格式是固定的，即

(jnz，a，_，0)　　//a为布尔变量

(j，_，_，0)

而每一个关系表达式同样对应一真一假两个四元式，其格式也是固定的，即

(jrop，X，Y，0)　　//X、Y为关系运算符两侧的变量或值

(j，_，_，0)

例4.4 试给出布尔表达式a∨m≠n∨c∧x＞y的四元式代码。

[解答] 该布尔表达式的翻译图如图4-9所示，所对应的四元式代码如下：

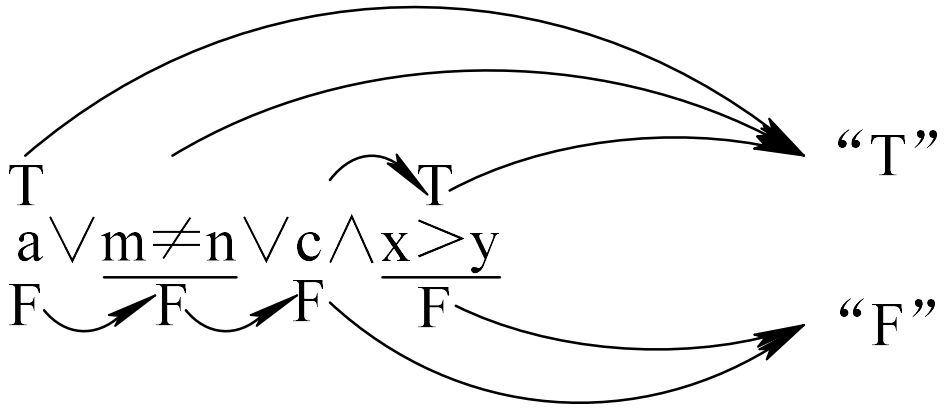


图4-9 例4.4的翻译图

100 (jnz, a, _, 108)
101 (j, _, _, 102)
102 (j≠, m, n, 108)
103 (j, _, _, 104)
104 (jnz, c, _, 106)
105 (j, _, _, q)
106 (j>, x, y, 108)
107 (j, _, _, q)

T: 108
⋮
F: q

4.5 控制语句的翻译

在源程序中，控制语句用于实现程序流程的控制。一般程序流程控制可分为下面三种基本结构：

(1) 顺序结构，一般用复合语句实现。

(2) 选择结构，用if和switch等语句实现。

(3) 循环结构，用for、while、do等语句实现。

4.5.1 条件语句if的翻译

1．条件语句if的代码结构

我们按下面的条件语句if的模式进行讨论：

$$\text{if(E) } S_1\text{; else } S_2\text{;}$$

条件语句if(E) S_1; else S_2; 中布尔表达式E的作用仅在于控制对S_1和S_2的选择，因此可将作为转移条件的布尔式E赋予两种“出口”：一是“真”出口，出向S_1；一是“假”出口，出向S_2。于是，条件语句可以翻译成如图4-10所示的代码结构。

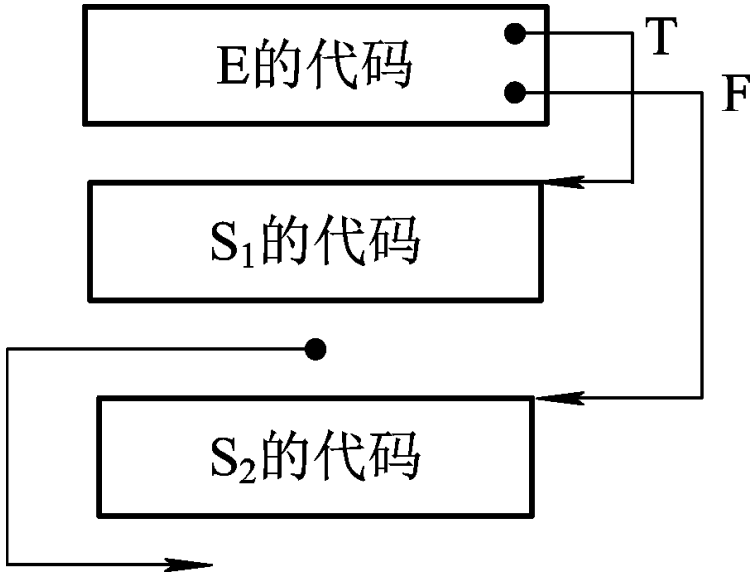


图4-10 条件语句if(E) S1;else S2; 的代码结构

我们知道，非终结符E具有两项语义值E.tc和E.fc，它们分别指出了尚待回填真假出口的四元式串。E的“真”出口只有在扫描完布尔表达式E后的“)”时才能知道，而它的“假”出口则需要处理过S_1之后并且到else时才能明确。这就是说，必须把E.fc的值传下去，以便到达相应的else时才进行回填。S_1语句执行完就意味着整个if-else语句已执行完毕，因此，在S_1的编码之后应产生一条无条件转移指令，这条转移指令将导致程序控制离开整个if-else语句。但是，在完成S_2的翻译之前，这条无条件转移指令的转移目标是不知道的，甚至在翻译完S_2之后仍无法确定，这种情形是由语句的嵌套性所引起的。

例如下面的语句：

if(E1) if(E2) S_1; else S_2; else S_3;

在S1代码之后的那条无条件转移指令不仅应跨越S_2，而且应跨越S_3。这也就是说，转移目标的确定和语句所处的环境密切相关。

不含else的条件语句if(E) S的代码结构图如图4-11所示。

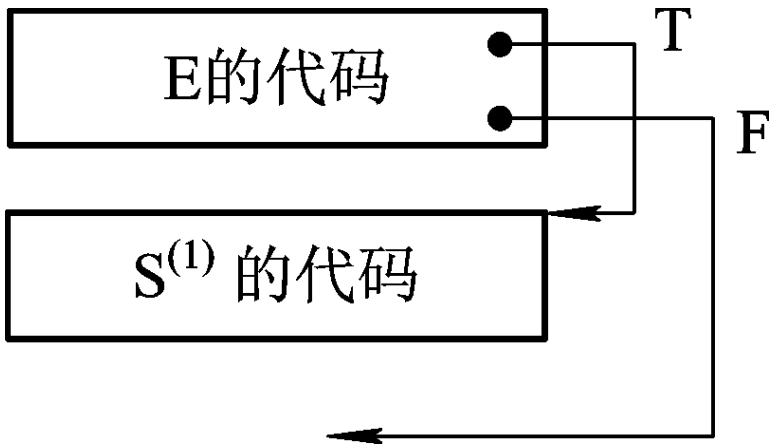


图4-11 if(E) S的代码结构

2．条件语句if的文法和语义子程序的设计

条件语句if的文法G[S] 如下：

G[S]：S→if(E) $S^{(1)}$;

S→if(E) $S^{(1)}$; else $S^{(1)}$;

为了在扫描条件语句过程中不失时机地处理和回填有关信息，可将G[S] 改写为如下的G'[S]：

G'[S]：(1) S→$CS^{(1)}$;

(2) C→if(E)

(3) S→$TPS^{(2)}$;

(4) TP→$CS^{(1)}$; else

根据程序语言的处理顺序，首先用产生式(2)C→if(E) 进行归约，这时E的真出口即为E所生成四元式序列后的下一个地址。因此，将“)”后的第一个四元式地址回填至E的真出口，E的假出口地址则作为待填信息放在C的语义变量C.chain中，即

C→if(E) { backpatch(E.tc, nxq);
C.chain=E.fc; }

接下来用产生式(1)S→CS$^{(1)}$; 继续向上归约。这时已经处理到S→if(E) S$^{(1)}$;，由于归约时E的真出口已经处理，而E的假出口(即语句S的出口)同时是语句S$^{(1)}$的出口，但此时语句S的出口地址未定，故将C.chain和S$^{(1)}$.chain一起作为S的待填信息链用函数merge链在一起保留在S的语义值S.chain中，即有

S→CS$^{(1)}$;　　{S.chain=merge(C.chain, S$^{(1)}$.chain)}

如果此时条件语句为不含else的条件句，则在产生式(1)、(2)归约为S后即可以用下一个将要产生的四元式地址(即S的出口地址)来回填S的出口地址链(即S.chain)；如果此时条件语句为if-else形式，则继续用产生式(4)$T^P \rightarrow CS^{(1)}$; else归约。

用$T^P \rightarrow CS^{(1)}$; else 归约时首先产生$S^{(1)}$语句序列之后的一个无条件转移四元式(以便跳过$S^{(2)}$，见图4-10的结构框图)，该四元式的地址(即标号)保留在q中，以便待获知要转移的地址后再进行回填，也即

(i) （$S^{(1)}$的第一个四元式） //E 的真出口

(q−1) （$S^{(1)}$的最后一个四元式）

(q) (j, _, _, 0) //无条件跳过 $S^{(2)}$，其转移地址有待回填

(q+1) （$S^{(2)}$的第一个四元式） //E 的假出口

此时q的出口也就是整个条件语句的出口，因此应将其与S.chain链接后挂入链头为TP.chain的链中。此外，emit 产生四元式q后nxq自动加1(即为q+1)，其地址即为else后(也即$S^{(2)}$)的第一个四元式地址，它也是E的假出口地址，因此应将此地址回填到E.fc即C.chain中，即有

```
T^P→CS^(1); else { q=nxq;
                   emit( j, _, _, 0 );
                   backpatch( C.chain, nxq );
                   T^P.chain=merge( S.chain, q ); }
```

最后用产生式(3)$S \rightarrow TPS^{(2)}$; 归约。$S^{(2)}$语句序列处理完后继续翻译if语句之后的后继语句，这时就有了后继语句的四元式地址，该地址也是整个if语句的出口地址，它与$S^{(2)}$语句序列的出口一致。由于$S^{(2)}$的出口待填信息在$S^{(2)}$.chain中，故将T^P.chain与$S^{(2)}$.chain链接后挂入链头为S.chain的链中，即

$S \rightarrow TPS^{(2)}$;　　{S.chain=merge(T^P.chain, $S^{(2)}$.chain);}

4.5.2 循环语句的翻译

1．循环语句while的代码结构

循环语句while(E) S(1); 通常被翻译成如图4-12所示的代码结构。布尔表达式E的“真”出口出向$S^{(1)}$代码段的第一个四元式，紧接$S^{(1)}$代码段之后应产生一条转向测试E的无条件转移指令；而E的“假”出口将导致程序控制离开整个while语句而去执行while语句之后的后继语句。

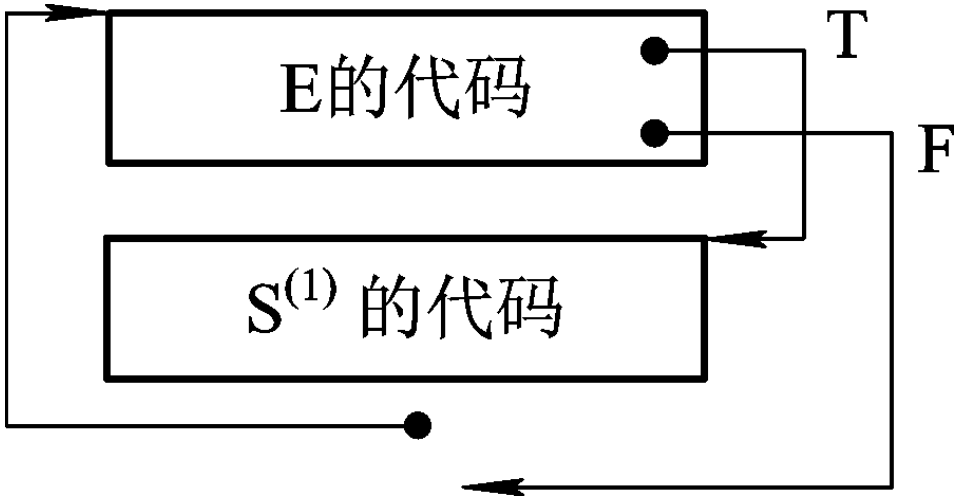


图4-12 循环语句while的代码结构

注意：E的“假”出口目标即使在整个while语句翻译完之后也未必明确。例如：

if (E_1) while (E_2) S_1; else S_2;

这种情况仍是由于语句的嵌套性引起的，所以只好把E的假出口作为循环语句的语义值S.chain保留下来，以便在处理外层语句时再伺机回填。

2．循环语句while的文法和语义子程序设计

同样，我们给出易于及时处理和回填的循环语句while的文法G[S] 如下：

G[S]：(1) $S \rightarrow W^{d}S^{(1)}$

(2) $W^{d} \rightarrow W^{(E)}$

(3) $W \rightarrow$ while

根据while语句的扫描加工顺序，首先用产生式(3)W→while进行归约，这时nxq即为E的第一个四元式地址，我们将其保留在W.quad中。然后继续扫描并用产生式(2)Wd→$W^{(E)}$归约，即扫描完“)”后可以用backpatch(E.tc,nxq)回填E.tc值；而E.fc则要等到$S^{(1)}$语句序列全部产生后才能回填，因此E.fc作为待填信息用Wd.chain=E.fc传下去。

当用产生式(1)S→$W^dS^{(1)}$; 归约时，S(1)语句序列的全部四元式已经产生。根据图4-12 while语句代码结构的特点，此时应无条件返回到E的第一个四元式继续对条件E进行测试，即形成四元式 (j, _, _, Wd.quad)，同时用backpatch(S(1).chain,W^d.quad) 回填E的入口地址到$S^{(1)}$语句序列中所有需要该信息的四元式中。

在无条件转移语句 (j, _, _, Wd.quad) 之后即为while语句的后继语句，而这个后继语句中的第一个四元式地址即为while语句E的假出口，保存在Wd.chain中。考虑到嵌套情况，将Wd.chain信息作为整个while语句的出口保留在S.chain中，以便适当时机回填。因此，文法G[S] 对应的语义加工子程序如下：

(1) W→while　　{ W.quad=nxq; }

(2) W^d→W(E)　　{ backpatch(E.tc, nxq);
W^d.chain=E.fc;
W^d.quad= W.quad; }

(3) S→W^d $S^{(1)}$;　　{ backpatch($S^{(1)}$.chain, W^d.quad);
emit((j, _, _, W^d.quad);
S.chain=W^d.chain; }

3．循环语句do和for的代码结构

循环语句do $S^{(1)}$ while$^{(E)}$; 通常被翻译成如图4-13所示的代码结构。首先翻译$S^{(1)}$的代码，然后翻译布尔表达式E的代码；E的“真”出口出向$S^{(1)}$代码段的第一个四元式，而E的“假”出口则使程序控制离开整个do语句而去执行do语句之后的后继语句。

C语言的for语句不同于其他高级语言中的for语句，其功能更加强大，使用更加灵活。for循环语句的一般形式为：

for(E_1; E_2; E_3) $S^{(1)}$;

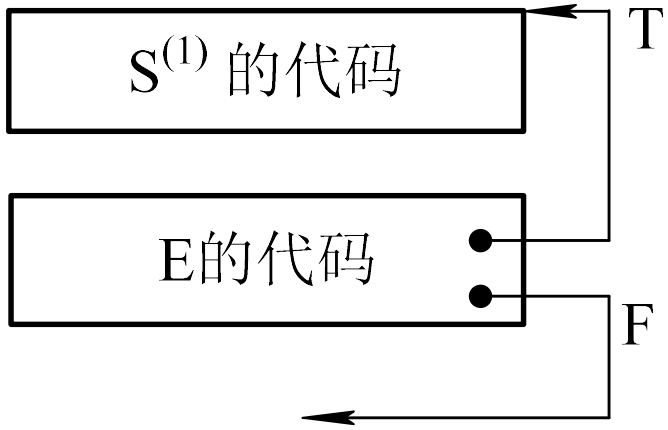


图4-13 循环语句do的代码结构

for语句圆括号中两个分号分隔的三个表达式E_1、E_2和E_3的作用如下：

(1) E_1：给循环控制变量赋初值，只在循环开始时执行一次。

(2) E_2：作为控制循环的条件在每次循环之前进行计算；如果条件成立(非0)则执行循环体语句，如果条件不成立(为0)则结束循环。

(3) E_3：用于改变循环控制变量的值，使得循环条件趋向于不成立(以便结束循环)。

for语句的执行过程分解为如下几步：

(1) 计算E_1；

(2) 求解E_2；若值为真(非0)则执行循环体语句$S^{(1)}$，然后转(3)；若值为假(0)则转(5)；

(3) 计算E_3；

(4) 转(2)继续执行；

(5) 结束循环。

因此，for(E1; E2; E3) S(1); 就相当于：

```
E1;
while(E2)
{
    S(1);
    E3;
}
```

所以，循环语句for(E_1; E_2; E_3) $S^{(1)}$; 可以依照图4-12翻译成如图4-14所示的代码结构。

当然，我们还可按同样方法得到循环语句do $S^{(1)}$ while$^{(E)}$; 和for(E_1; E_2; E_3) $S^{(1)}$; 的文法及语义加工子程序。

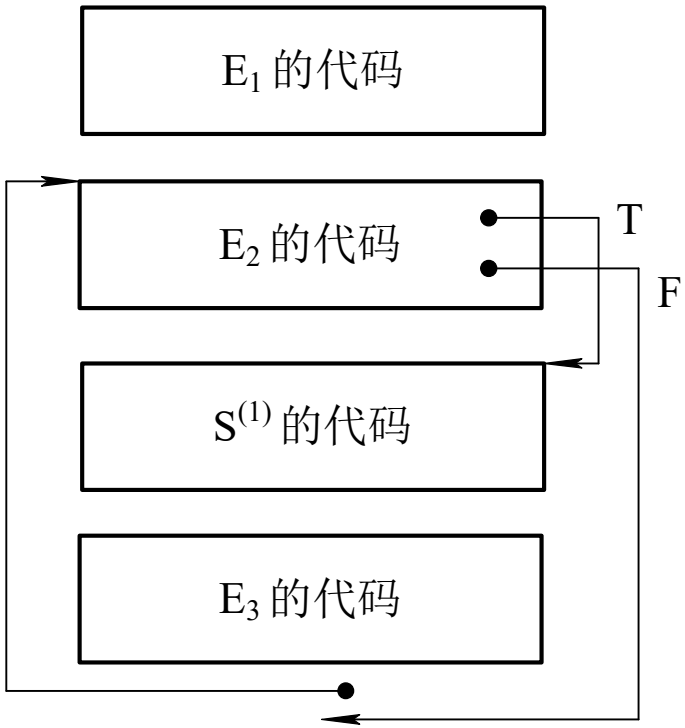


图4-14 循环语句for的代码结构

4.5.3 三种基本控制结构的翻译

1. 三种基本控制结构的文法

我们给出三种基本控制结构的文法G[S] 如下：

G[S]: (1) S→CS
(2) | T^p S
(3) | W^d S
(4) | {L}
(5) | A　　//A 代表赋值语句
(6) L→L^S S;
(7) | S;
(8) C→if(E)
(9) T^p→CS; else
(10) W→while
(11) W^d→W(E)
(12) L^S→L;

G[S] 中各产生式对应的语义子程序如下：

(1) $S \rightarrow C\ S^{(1)}$; { S.chain=merge(C.chain, $S^{(1)}$.chain); }

(2) $S \rightarrow T^{p}\ S^{(2)}$; { S.chain=merge(T^{p}.chain, $S^{(2)}$.chain); }

(3) $S \rightarrow W^{d}\ S^{(1)}$; { backpatch($S^{(1)}$.chain, W^{d}.quad);
emit(j, _, _, W^{d}.quad);
S.chain=W^{d}.chain; }

(4) $S \rightarrow \{L\}$ { S.chain=L.chain; }

(5) $S \rightarrow A$ { S.chain=0; //空链 }

(6) $L \rightarrow L^{s}\ S^{(1)}$; { L.chain=$S^{(1)}$.chain; }

(7) L→S; { L.chain=S.chain;}

(8) $C \rightarrow if(E)$ { backpatch(E.tc, nxq);
C.chain=E.fc; }

(9) $T^p \rightarrow C\ S^{(1)}$; else { q=nxq;
emit(j, _, _, 0);
backpatch(C.chain, nxq);
T^p.chain=merge($S^{(1)}$.chain, q);}

(10) $W \rightarrow$ while { W.quad=nxq;}

(11) $W^d \rightarrow W(E)$ { backpatch(E.tc, nxq);
W^d.chain=E.fc;
W^d.quad=W.quad;}

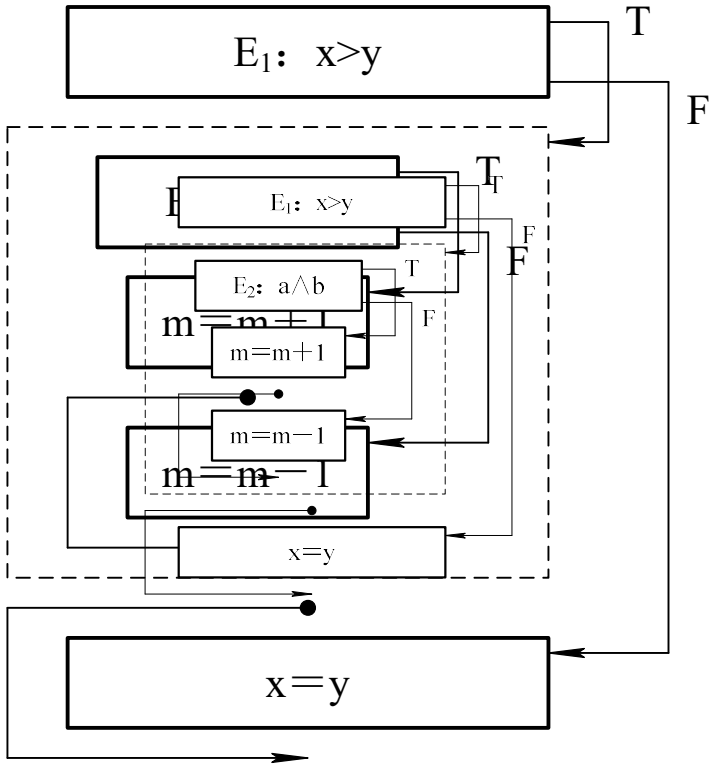
(12) $L^S \rightarrow L$; { backpatch(L.chain, nxq); }

2．翻译示例

例**4.5** 将下面的语句翻译成四元式：

if(x＞y) if(a∧b) m=m+1; else m=m-1; else x=y;

[解答] 该语句对应的代码结构图如图4-15所示，它所对应的四元式序列如下：


E1：x>y
T
F
E1：x>y
T
F
E2：a∧b
T
F
m=m+1
m=m-1
x=y
x=y

100(j>, x, y, 102)
101(j, _, _, 110)
102(jnz, a, _, 104)
103(j, _, _, 108)
104(jnz, b, _, 106)
105(j, _, _, 108)
106(+, m, 1, m)
107(j, _, _, 109)
108(-, m, 1, m)
109(j, _, _, 111)
110(=, y, _, x)
111

如果按图4-16所示的翻译图进行翻译，则第107句四元式为(j，_，_，111)，虽然与上面的107句不同，但仔细分析就会发现：翻译图与代码结构图两者所翻译的结果在功能上是完全相同的。

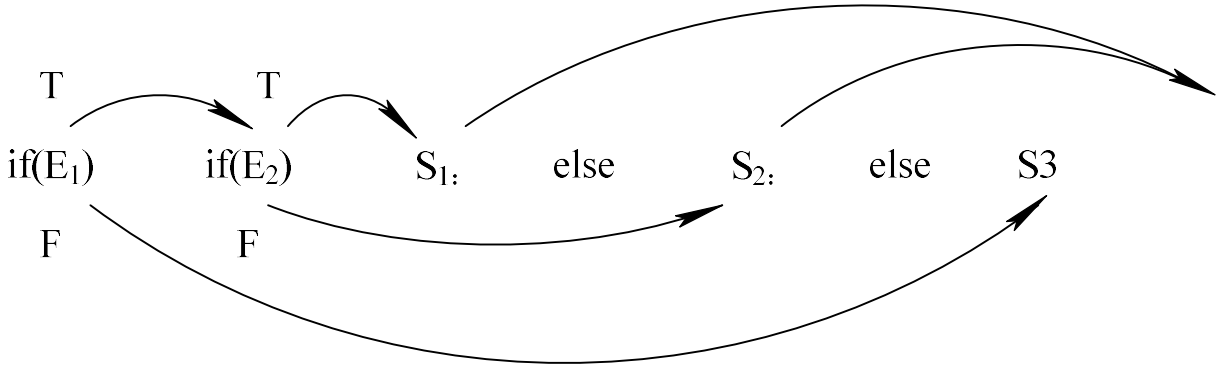


图4-16 例4.5的翻译图

例4.6 将下面的语句翻译成四元式：

```
while( A<B )
    if( C<D ) X=Y+Z
```

[解答] 我们首先画出该语句对应的代码结构图如图4-17所示。

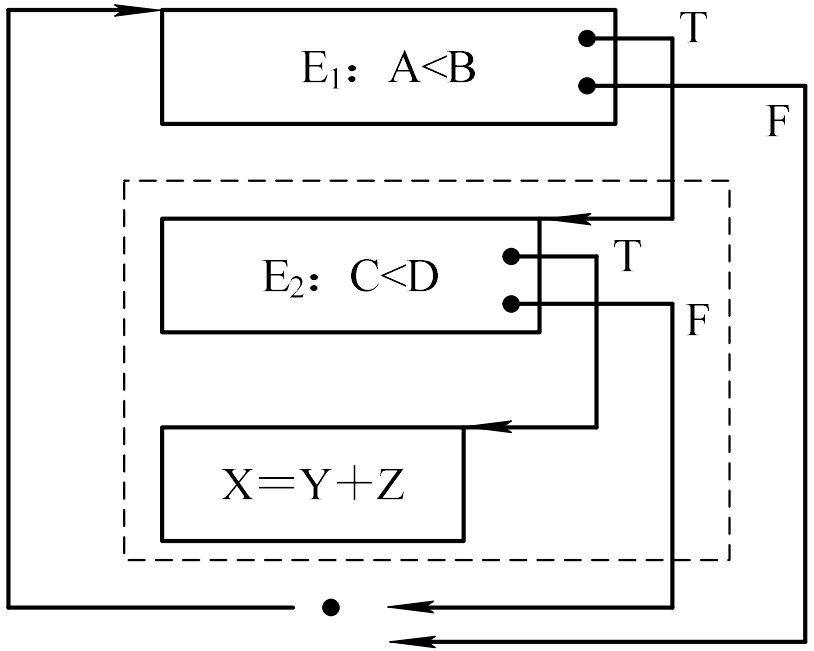


图4-17 例4.6的代码结构图

按照文法及加工子程序(包括前述赋值句和布尔表达式的翻译法)得到该语句对应的四元式序列如下：

```
100(j<, A, B, 102)        //E1为 T
101(j, _, _, 107)         //E1为 F
102(j<, C, D, 104)        //E2为 T
103(j, _, _, 106)         //E2为 F
104(+, Y, Z, T)
105(=, T, _, X)
106(j, _, _, 100)         //转对 E 的测试
107
```

例**4.7** 将下面的语句翻译成四元式：

```
if( a∧b )
   while( x<y )
      if( m≠n )
         m=n;
      else
         m=m+1;
else
   while( m>n )
      x=x+y;
```

[解答] 我们首先画出该语句对应的代码结构图如图4-18所示。

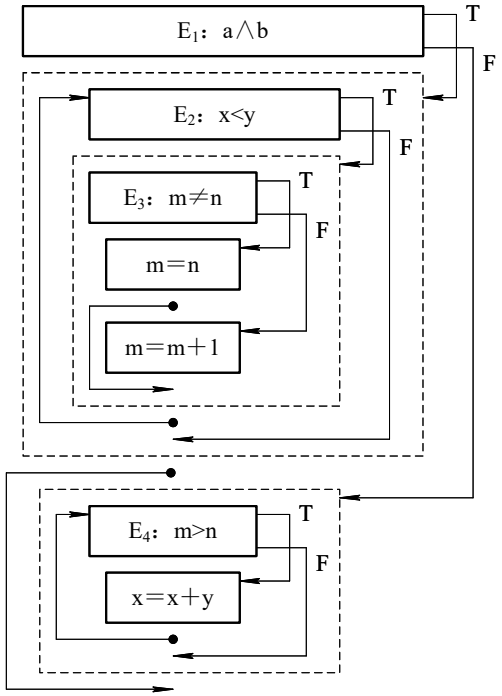


图4-18 例4.7的代码结构图

虽然根据文法及加工子程序可以得到该语句对应的四元式序列，但我们知道每个布尔变量及每个关系表达式都对应一真一假固定格式的两个四元式，且if语句在else之前应有一无条件转移语句跳过else后的语句，而while语句则在结束时有一个向回跳的无条件转移语句。由本题可知，共有两个布尔变量a、b和三个关系表达式，共需10条四元式，而两个if语句和两个while语句共需4条无条件转移四元式，再加上3条赋值四元式，总计为17条四元式。再依据代码结构图来确定各个条件转移和无条件转移四元式的转移地址，就很容易得到该语句的四元式序列如下：

100(jnz, a, _, 102)
101(j, _, _, 113)
102(jnz, b, _, 104)
103(j, _, _, 113)
104(j<, x, y, 106)
105(j, _, _, 112)
106(j≠, m, n, 108)
107(j, _, _, 110)
108(=, n, _, m)
109(j, _, _, 111)
110(+, m, 1, m)
111(j, _, _, 104)
112(j, _, _, 117)
113(j>, m, n, 115)
114(j, _, _, 117)
115(+, x, y, x)
116(j, _, _, 113)
117

例**4.8** 将下面的语句翻译成四元式：

```
for( i=0; i<=10; i++ )
    x=x+1;
```

[解答] 根据图4-14所示的for语句代码结构可以翻译成如下的四元式序列：

```
100 (=, 0, _, i)
101 (j<=, i, 10, 103)
102 (j, _, _, 106)
103 (+, x, 1, x)
104 (+, i, 1, i)
105 (j, _, _, 101)
106
```

例4.9 按已学过的文法及语义加工子程序分析下述语句语义加工的全过程：

while(x<y) x=x+1

[解答] 语句while(x<y) x=x+1的语义加工过程见表4.3。

表 4.3 while(x<y) x=x+1# 的语义加工过程

输入串	符号栈	语义栈（place）	语义动作	四元式
while(x<y) …	#	_	移进	
(x<y) x=x+1#	#while	_ _	归约	
			w.quad=100	
(x<y) x=x+1#	#W	_ _	移进	
x<y) x=x+1#	#W(	_ _ _	移进	
<y) x=x+1#	#W(x	_ _ _ _	归约（参见赋值句文法）	
			i_1.place=entry(x)	
<y) x=x+1#	#W(i_1	_ _ _ x	移进	
y) x=x+1#	#W(i_1<	_ _ _ x_	移进	
) x=x+1#	#W(i_1<y	_ _ _ x_ _	归约	
			i_2.place=entry(y)	

) x=x+1#	#W(i_1<i_2	_ _ _ x_ y	归约（参见布尔表达式文法）	
			E_1.tc=100	
			E_1.fc=101	102
				100 (j<,x,y,0)
				105
				101 (j, _,_,0)
) x=x+1#	#W(E_1	_ _ _ _	移进	
x=x+1#	#W(E_1)	_ _ _ _ _	归约	
			Backpatch(100,102)	
			W^d.chain=101	
x=x+1#	#W^d	_ _	移进	
=x+1#	#W^dx	_ _ _	归约	
			i_3.place=entry(x)	
=x+1#	#$W^d i_3$	_ _ x	移进	
x+1#	#$W^d i_3$=	_ _ x_	移进	
+1#	#$W^d i_3$=x	_ _ x_ _	归约	
			E_2.place=entry(x)	
+1#	#$W^d i_3$=E_2	_ _ x_ x	移进	
1#	#$W^d i_3$=E_2+	_ _ x_ x_	移进	
#	#$W^d i_3$=E_2+1	_ _ x_ x_ _	归约	
			E_3.place=entry(1)	
#	#$W^d i_3$=E_2+E_3	_ _ x_ x_ 1	归约	
			E_4.place=T_1	
				102 (+,x,1,T_1)
#	#$W^d i_3$=E_4	_ _ x_ _	归约	
				103 (=,T_1, _,x)
#	#$W^d S^{(1)}$	_ _ _	归约	

			Backpatch($S^{(1)}$.chain,100) (因无 $S^{(1)}$.chain 故不回填)	
				104 (j, _,_,100)
			S.chain=101	
#	#S	_ _	while 语句分析结束 外层返填；	
			由 $L^S \rightarrow L$ 归约得：	
			Backpatch(S.chain,105)	

4.5.4 多分支控制语句switch的翻译

多分支控制语句具有如下形式的语法结构：

```
switch( E )
{   case c1: S1;
    case c2: S2;
          :
    case ci: Si;
          :
    case cn: Sn;
    default: Sn+1
}
```

其中n≥1。switch语句的语义是：先计算整型表达式E的值，然后将表达式的值依次和case后的常数ci比较，当与某常数c_i相等时就执行语句S_i，并结束多分支控制语句；若与诸常数均不相等，则执行语句S_{n+1}。

多分支控制语句switch常见的中间代码形式如下：

```
                E 计值后存放在临时单元 T 的中间代码;
                goto test;
P1:             S1的中间代码;
                goto next;
P2:             S2的中间代码;
                goto next;
                  ⋮
Pn:             Sn的中间代码;
                goto next;
Pn+1:           Sn+1的中间代码;
                goto next;
test:           if( T==c1 ) goto P1;
                if( T==c2 ) goto P2;
                  ⋮
                if( T==cn ) goto Pn;
                if( T=='default' ) goto Pn+1;
next:
```

进行语义加工处理时应先设置一空队列queue，当遇到ci时，将这个ci连同nxq(指向标号ci后语句Si的入口)送入队列queue，然后按通常的办法产生语句Si的四元式。需要注意的是，在Si的四元式之后要有一个goto next的四元式。当处理完default: Sn+1之后，应产生以test为标号的n个条件转移语句的四元式。这时，逐项读出queue的内容即可形成如下的四元式序列：

(case, c_1, P_1, _)
(case, c_2, P_2, _)
⋮
(case, c_n, P_n, _)
(case, T.place, default, _)

其中，T.place是存放E值的临时变量名，每个四元式(case, c_i, P_i, _)实际上代表一个如下的条件语句：

if(T==c_i) goto P_i

为了便于语法制导翻译，我们给出了switch语句的文法和相应的语义加工子程序如下：

```
(1) A→switch( E )    {  T.place=E.place;
                        F1.quad=nxq;
                        emit( j, _, _, 0 );                    //转向 test }
(2) B→A{case c       {  P=1;
                        queue[P].label=c;
                        queue[P].quad=nxq;  }
(3) D→B:S;           { 生成 S 的四元式序列;
                        backpatch( S.chain, nxq );
                        B.quad=nxq;
                        emit( j, _, _, 0 );                    //转向 next }
(4) D→F:S;           { 生成 S 的四元式序列;
                        backpatch( S.chain, nxq );
                        B.quad=nxq;
                        emit( j, _, _, 0 );                    //转向 next
                        F.quad=merge( B.quad, F.quad );  //转向 next 的语句拉成链 }
```

(5) F→D;case c { P=P+1;
queue[P].label=c;
queue[P].quad=nxq; }

(6) S→D;default:S; }
{ 生成 S 的四元式序列;
backpatch(S.chain, nxq);
B.quad=nxq;
emit(j, _, _, 0);
F.quad=merge(B.quad, F.quad); //形成转向 next 的链首 }
P=P+1;
queue[P].label='default';
queue[P].quad=nxq;
F_3.quad=nxq; //指向标号 test
m=1;
do
{
c_i=queue[m].label;
P_i=queue[m].quad;
m=m+1;
if(c_i!='default') emit(case, c_i, P_i, _);
else emit(case, T.place, default, _);
}while(m<=P+1);
backpatch(F_1.quad, F_3.quad);
backpatch(F.quad, nxq); //填写所有转向 next 语句的转移地址}

4.5.5 语句标号和转移语句的翻译

程序语言中直接改变控制流程的语句是goto L语句，其中L是源程序中的语句标号。标号L在源程序中可以以两种方式出现：

(1) 定义性出现。定义性出现的语句形式为

L：S

此时，带标号的语句S所生成的第一个四元式地址即为标号L的值。

(2) 引用性出现。引用性出现的语句形式为

goto L

它引用L的值作为四元式 (j, _, _, L) 中转向的目标地址。对标号L的处理方法是：当标号L定义性出现时，应将标号此时对应的四元式地址(即标号L的值)登录到符号表中L所对应的项；当标号L引用性出现时，则引用符号表中该标号L的值。

显然，在源程序中，如果标号的定义性出现在前而引用性出现在后，即先定值后引用(称为向后引用)，则填、查符号表及将转移语句翻译成四元式很容易。但是，如果标号引用性出现在前而定义性出现在后(称为向前引用)，则引用时不可能从符号表中获得标号L的值，此时只能生成有待回填的四元式 (j, _, _, 0)，等到向前翻译到标号L定义性出现时，再将标号L的值回填到待填的四元式中。

翻译goto L语句时需要查符号表，看L是否定值，有以下几种情况：

(1) L已经定值，即L.value为符号表中所记录的L值，这时生成 (j, _, _, L.value) 语句。

(2) 在符号表中未出现标号L项，则goto L中的L是首次出现，故生成 (j, _, _, 0) 形式的语句并在符号表中登录L项，给L项标记为“未定值”并将四元式 (j, _, _, 0) 的地址作为L的值记入符号表(作为L引用链的链头)，以待L定值后回填。

(3) 在符号表中已有标号L项但未定值，此时的goto L语句并非首次出现，故生成四元式 (j, _, _, 0) 并将其地址挂入到L的引用链中，待L定值后再进行回填。

翻译语句L：S时，在识别L后也要查符号表。如L为首次出现，则在符号表中建立L项，将此时的nxq值作为L的值登入符号表并置“已定值”标记；如果符号表中已有L项且标记为“未定值”，这意味着是向前引用情况，应将此时的nxq值作为L的值登入符号表并置“已定值”，同时以此值回填L的引用链；若查找符号表发现L已定值，则表示L出现了重复定义的错误。

4.6 数组元素的翻译

4.6.1 数组元素的地址计算及中间代码形式

在表达式或赋值语句中若出现数组元素，则翻译时将牵涉到数组元素的地址计算。数组在存储器中的存放方式决定了数组元素的地址计算法，从而也决定了应该产生什么样的中间代码。数组在存储器中的存放方式通常有按行存放和按列存放两种。在此，我们讨论以行为主序存放方式的数组元素地址计算方法。

数组的一般定义为

$$A[l1:u_1, l_2:u_2, \ldots, l_k:u_k, \ldots, l_n:u_n]$$

其中，A是数组名，l_k是数组A第k维的下界，u_k是第k维的上界。为简单起见，假定数组A中每个元素的存储长度为1，a是数组A的首地址，则数组元素$A[i_1, i_2, \ldots, i_n]$ 的地址D的计算公式如下：

$$D=a+(i_1-l_1)d_2d_3\cdots d_n+(i_2-l_2)d_3d_4\cdots d_n+\cdots+(i_{n-1}-l_{n-1})d_n+(i_n-l_n)$$

其中，$d_i=u_i-l_i+1(i=1, 2, \ldots, n-1)$。整理后得到

$$D=CONSPART+VARPART$$

其中 $CONSPART=a-(\cdots((l_1d_2+l_2)d_3+l_3)d_4+\cdots+l_{n-1})d_n+l_n$

$VARPART=(\cdots((i_1d_2+i_2)d_3+i_3)d_4+\cdots+i_{n-1}d_n)+i_n$

CONSPART中的各项(如li、di(i=1, 2, …, n))在处理说明语句时就可以得到，因此CONSPART值可在编译时计算出来后保存在数组A的相关符号表项里。此后，在计算数组A的元素地址时仅需计算VARPART值，而直接引用CONSPART值。

实现数组元素的地址计算时，将产生两组四元式序列：一组计算CONSPART，其值存放在临时变量T中；另一组计算VARPART，其值存放在临时变量T_1中，即用T_1[T] 表示数组元素的地址。这样，对数组元素的引用和赋值就有如下两种不同的四元式：

(1) 变址存数：若有T_1[T] =X，则可以用四元式([ ]=, X, _, T_1[T]) 表示。

(2) 变址取数：若有X=T_1[T]，则可用四元式(=[ ], T_1[T], _, X) 表示。

4.6.2 赋值语句中数组元素的翻译

为了便于语法制导翻译，我们定义一个含有数组元素的赋值语句文法G[A] 如下：

G[A]： (1) A→V=E

(2) V→i [elist] | i

(3) elist→elist, E | E

(4) E→E+E | (E) | V

其中，A代表赋值语句；V代表变量名；E代表算术表达式；elist代表由逗号分隔的表达式，它表示数组的一维下标；i代表简单变量名或数组名。

在用产生式(2)、(3)进行归约时，为了能够及时计算数组元素的VARPART，我们将产生式(2)、(3)改写为

(2') V→elist] | i

(3') elist→elist, E | i [E

把数组名i和最左的下标式写在一起的目的是在整个下标串elist的翻译过程中随时都能知道数组名i的符号表入口，从而随时能够了解登记在符号表中有关数组i的全部信息。

为产生计算VARPART的四元式序列，还需要设置如下的语义变量和函数：

(1) elist.ARRAY：表示数组名在符号表的入口。

(2) elist.DIM：计数器，用来计算数组的维数。

(3) elist.place：登录已生成VARPART中间结果的单元名字在符号表中的存放位置，或是一个临时变量的整数码。

(4) limit(ARRAY, k)：参数ARRAY表示数组名在符号表的入口，k表示数组当前计算的维数；函数limit( ) 计算数组ARRAY的第k维长度dk。

含有数组元素的赋值语句对应的文法G[A] 及相应的语义子程序如下(省略语义检查，仅给出主要语义动作)：

```
(1) A→V=E        { if( V.offset==null )
                       emit( =, E.place, _, V.place );                  //V 是简单变量
                   else emit( []=, E.place, _, V.place[V.offset] );     //V 是下标变量
                   }
(2) E→E(1) +E(2)  { T=newtemp;  emit( +, E(1).place, E(2).place, T );  E.place=T;  }
(3) E→ (E(1))     { E.place=E(1).place;  }
(4) E→V           { if (V.offset==null)
                       E.place=V.place;                                 //V 是简单变量
                   else {T=newtemp;                                     //V 是下标变量
                         emit( =[ ], V.place[V.offset], _, T );
                         E.place=T;  };  }
```

(5) V→elist] { T=newtemp; emit(−, elist.ARRAY, C, T);
V.place=T; V.offset=elist.place; }

/*假定通过数组名的符号表入口不仅能获得地址 a 而且也能得到常数 C(CONSPART=a−C)*/

(6) V→i { V.place=entry(i); V.offset=null; }

(7) elist→elist$^{(1)}$, E { T=newtemp; k=elist$^{(1)}$.DIM+1;
d_k=limit(elist$^{(1)}$.ARRAY, k); emit(*, elist$^{(1)}$.place, d_k, T);
emit(+, E.place, T, T); elist.ARRAY=elist$^{(1)}$.ARRAY;
elist.place=T; elist.DIM=k; }

(8) elist→i [E { elist.place=E.place; elist.DIM:=1; elist.ARRAY=entry(i); }

4.6.3 数组元素翻译示例

例**4.10** 已知A是一个10×20的数组(每维下界均为1)且按行存放，求：

(1) 赋值语句X= A[I, J] 的四元式序列；

(2) 赋值语句A[I+2, J+1] =M+N的四元式序列。

要求给出语法制导翻译过程。

[解答] 由于A是10 × 20的数组，故d_1=10，d_2=20，C=d_2+1=21。

(1) 根据文法G[A] 及对应的语义加工子程序，赋值语句X=A[I, J] 的语法制导翻译过程如图4-19所示。

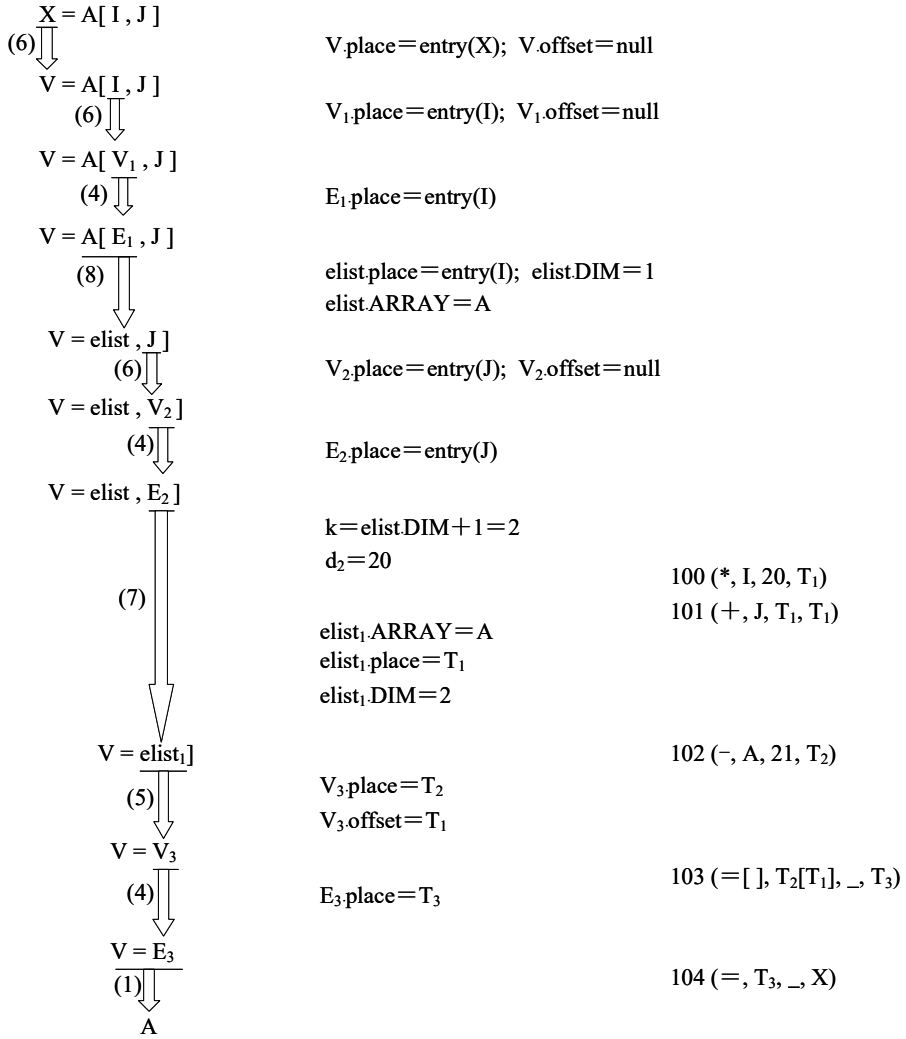


图4-19 X=A[I, J] 的语法制导翻译过程

注意：由(7)计算出的是数组元素A[I, J] 的地址，即为I×列数长度(即d2=20)+J，而此处数组A的行、列下界值为1，即实际上多计算了一行一列，故应减去，则实际A[I, J] 对应的地址是：

$$A+(20-1)*I+J-1=A+20I+J-21$$

最后得到的赋值句X=A[I, J] 的四元式序列为

100 (*, I, 20, T_1)	//d_2=20
101 (+, J, T_1, T_1)	//得到 20I+J
102 (−, A, 21, T_2)	//得到 A−21
103 (=[ ],T_2[T_1],_,T_3)	//T_2[T_1]即为 A[I, J]，即 T_3=T_2[1]
104 (=, T_3, _, X)	

(2) 根据文法G[A] 及对应的语义加工子程序，赋值语句A[I+2, J+1] =M+N的语法制导翻译过程如图4-20所示(为节省篇幅，特将表达式的翻译由顺序进行改为同时进行)。因此得到赋值句A[I+2, J+1] =M+N的四元式序列为

100 (+, I, 2, T_1)
101 (+, J, 1, T_2)
102 (*, T_1, 20, T_3)
103 (+, T_2, T_3, T_3)
104 (−, A, 21, T_4)
105 (+, M, N, T_5)
106 ([ ]=, T_5, _, $T_4[T_3]$)　　//$T_4[T_3] = T_5$

例**4.11** 试给出下列语句的四元式序列：

if(p1==0∧p2＞10) X[1, 1] =1; else X[7, 6] =0;

其中，X是10×20的数组(每维下界为1)且按行存放；一个数组元素占用两个字节，机器按字节编址。

[解答] 拓展数组元素翻译的语义子程序功能，得到该语句对应的四元式序列如下：

100 (j=, P_1, 0, 102)
101 (j, _, _, 110)
102 (j>, P_2, 10, 104)
103 (j, _, _, 110)
104 (*, 1, 40, T_1)
105 (*, 1, 2, T_2)
106 (+, T_1, T_2, T_3)
107 (−, X, 42, T_4)
108 ([]=, 1, _, $T_4[T_3]$)
109 (j, _, _, 115)
110 (*, 7, 40, T_1)
111 (*, 6, 2, T_2)
112 (+, T_1, T_2, T_3)
113 (−, X, 42, T_4)
114 ([]=, 0, _, $T_4[T_3]$)
115

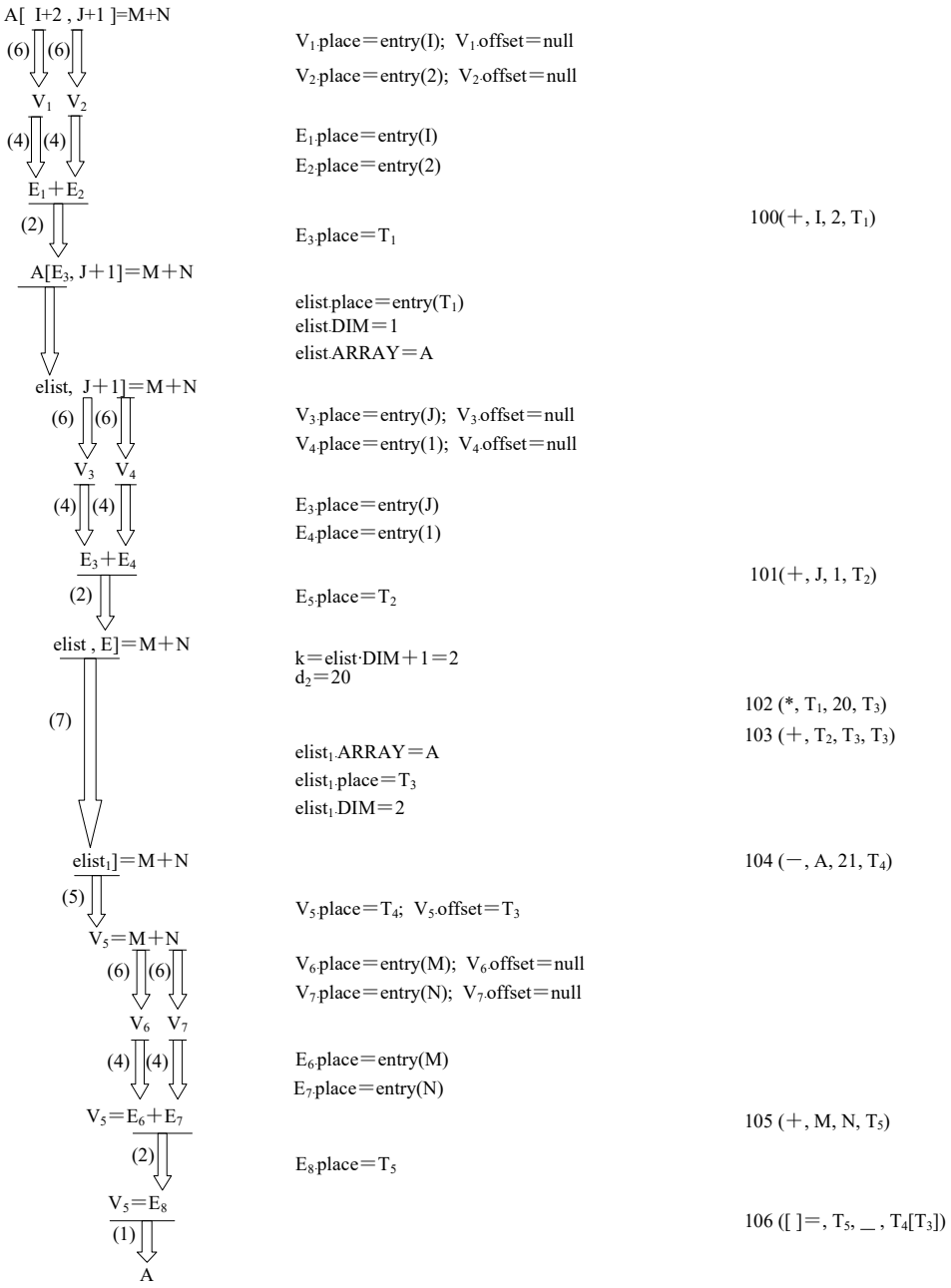


图4-20 A[I+2, J+1] =M+N 的语法制导翻译过程

4.7 过程或函数调用语句的翻译

4.7.1 过程或函数调用的方法

过程或函数(注：C语言将过程与函数统称为函数)是程序设计中最常用的手段之一，也是程序语言中最常用的一种结构。

过程或函数调用语句的翻译是为了产生一个调用序列和返回序列。如果在P过程中有过程调用语句call Q，则当目标程序执行到过程调用语句call Q时所做的过程调用工作如下(参见第6章)：

(1) 为被调用过程Q分配活动记录的存储空间。

(2) 将实在参数传递给被调用过程Q的形式单元。

(3) 建立被调用过程Q的外围嵌套过程的层次显示表，以便存取外围过程的数据。

(4) 保留被调用时刻的环境状态，以便调用返回后能恢复过程P的原运行状态。

(5) 保存返回地址(通常是调用指令的下一条指令地址)。

(6) 在完成上述调用序列动作后，生成一条转子指令转移到被调用过程的代码段开始位置。

在过程调用结束返回时：

(1) 如果是函数调用，则在返回之前将返回值存放到指定的位置。

(2) 恢复过程P的活动记录。

(3) 生成goto返回地址的指令，返回到P过程。

在编译阶段对过程或函数调用语句的翻译所做的工作主要是参数传递。参数传递的方式很多，我们在此只讨论传递实在参数地址(传地址)的处理方式。

如果实在参数是一个变量或数组元素，则直接传递它的地址；如果是其它表达式，如A+B或2，则先把它的值计算出来并存放在某个临时单元T中，然后传送T的地址。

注意：所有实在参数的地址都应存放在被调用过程(如Q)能够取得到的地方。在被调用的过程中，每个形式参数都有一个单元(称为形式单元)用来存放相应的实在参数的地址，对形式参数的任何引用都当作是对形式单元的间接访问。在通过转子指令进入被调用过程后，被调用过程的第一步工作就是把实在参数的地址取到对应的形式单元中，然后再开始执行本过程中的语句。

传递实在参数地址的一个简单办法是把实参的地址逐一放在转子指令的前面。例如，过程调用call Q(A+B, Z) 将被翻译成：

```
计算A+B置于T中的代码          //即生成四元式：( +, A, B, T )
    par T                     //第一个实参地址
    par Z                     //第二个实参地址
    call Q                    //转子指令
```

这样，在目标代码执行过程中，通过执行转子指令call Q而进入过程Q之后，Q就可根据返回地址(假定为K，它是call Q后面的那条指令地址)寻找到存放实在参数地址的单元(在此分别为K−3对应着T和K−2对应着Z)。

4.7.2 过程或函数调用语句的四元式生成

根据上述关于过程或函数调用的目标结构，我们现在来讨论如何产生反映这种结构的四元式序列。

一种描述过程或函数调用语句的文法G[S] 如下：

G[S]：(1) S→call i(elist)

(2) elist→elist, E

(3) elist→E

为了在处理实在参数串的过程中记住每个实参的地址，以便最后把它们排列在转子指令call之前，我们需要把这些地址保存起来。用来存放这些地址的有效办法是使用队列这种数据结构，以便按序记录每个实在参数的地址。我们赋予产生式elist→elist,E的语义的动作是将表达式E的存放地址E.place放入队列queue中；而产生式S→call i(elist) 的语义动作是对队列queue中的每一项P生成一个四元式 (par, _, _, P)，并让这些四元式按顺序排列在对实参表达式求值的那些四元式之后。注意，实参表达式求值的语句已经在把它们归约为E的时候生成。下面是文法G[S] 和与之对应的语义加工子程序：

(1)	S→call i (elist)	{for(队列 queue 中的每一项 P) emit(par, _, _, P); emit(call, _, _, i.place); }
(2)	elist→elist, E	{将 E.place 加入到 queue 的队尾}
(3)	elist→E	{初始化 queue 仅包含 E.place}

注意：(1) 中S的四元式首先包括elist的四元式(即对各实参表达式求值的四元式)，接下来还包括顺序为每一个参数产生的一个四元式 (par, _, _, P)，最后还包括生成 (call, _, _, i.place) 的四元式。

4.8 说明语句的翻译

4.8.1 变量说明的翻译

程序中的每个名字(如变量名)都必须在使用之前进行说明，而说明语句的功能就是为编译程序说明源程序中的每一个名字及其性质。简单说明语句的一般形式是用一个基本字来定义某些名字的性质，如整型变量、实型变量等。

简单说明语句的文法G[D]定义如下：

G[D]: D→int namelist｜float namelist

namelist→namelist, i｜i

其中，int、float为保留字，用来说明名字的性质分别为整型或实型，其相应的翻译工作是将名字及其性质登录在符号表中。

按照自底向上的制导翻译方法用上述文法的产生式进行归约时，首先将所有的名字都归约成namelist后才能把它们的性质登入符号表，这意味着必须用一个队列(或栈)来保存namelist中的所有名字。我们可以把文法G[D] 改为G'[D]：

G'[D]：D→D, i | int i | float i

这样，就能把所说明的性质及时地告诉每个名字i；或者说，每当读进一个标识符时就可以把它的性质登记到符号表中，而无需到最后再集中登记了。

我们给D的语义子程序设置了一个函数和一个语义变量：函数fill(i, A)的功能是把名字i和性质A登录在符号表中；考虑到一个性质说明(如int)后可能有一系列名字，故设置D的语义变量D.att来传递相关名字的性质。这样，文法G'[D] 和相应的语义加工子程序如下：

(1) D→int i　　{ fill(i, int)；D.att=int；}

(2) D→float i　　{ fill(i, float)；D.att=float；}

(3) D→D(1), i　　{ fill(i, $D^{(1)}$.att)；D.att= $D^{(1)}$.att；}

4.8.2 数组说明的翻译

包括变量说明和数组说明的文法G[D] 定义如下：

G[D]：D→int namelist｜float namelist

namelist→namelist, V｜V

V→i [elist]｜i

elist→elist, E｜E

E→E+E｜(E)｜i

当处理数组说明时，需要把数组的有关信息汇集在一个称为“内情向量”的表格中，以供后来计算数组元素的地址时查询。例如，数组int A[l_1:u_1, l_2:u_2, …, l_n:u_n] 相应的内情向量见表4.4。

表 4.4 数组内情向量表

l_1	u_1	d_1
l_2	u_2	d_2
⋮	⋮	⋮
l_n	u_n	d_n
维数：n	CONSPART=a-C 中的 C	
类型：int	数组 A 的首地址 a	

如果不检查数组引用时的下标是否越界，则内情向量的内容还可以进一步压缩，如l、u、d三栏只用l、d栏即可。显然，内情向量的大小是由数组的维数n所确定的。

对静态数组来说，它的每一维上、下界ui和li都是常数，故每维的长度di(从而可求出CONSPART中的C)在编译时就可计算出来，在编译时就能知道数组所需占用存储空间的大小。在这种情况下，内情向量只在编译时有用而无需将其保留到目标程序的运行时刻，因此，可将它安排为符号表的一部分。

如果是可变数组，有些维的上、下界ui、li是变量，则某些维的长度di以及C在运行时才能计算出来，因此，数组所需的存储空间大小在程序运行时才能知道。在这种情况下，编译时应分配数组的内情向量表区。在目标程序运行中，当执行到数组A所在的分程序时，把内情向量的各有关成分填入此表区，然后再动态地申请数组所需的存储空间。这表明可变数组在编译时，一方面要分配它的内情向量表区，另一方面必须产生在运行时动态建立内情向量和分配数组空间的目标指令，而这些指令就是在翻译数组说明时产生的。

4.9 递归下降语法制导翻译方法简介

自底向上分析法适应更多的文法，因此，自底向上分析制导翻译技术也受到普遍重视。但是，自顶向下分析法也有自底向上分析法不可取代的优点，它可以在一个产生式的中间调用语义子程序。例如，假定我们正在为非终结符A寻找匹配，并已确定用A的候选式BCD(即用A→BCD规则)，那么可分别在识别出B、C和D之后直接调用某些语义子程序，而无需等到整个候选式匹配完之后。

为了完整起见，我们简略地讨论一下自顶向下分析制导翻译技术，如递归下降分析制导翻译技术，它的特点是将语义子程序嵌入到每个递归过程中，通过递归子程序内部的局部量和参数来传递语义信息。

作为一个例子，我们考虑下面关于算术表达式的文法G[E]：

G[E]：E→T{+T}
T→F{*F}
F→(E) | i　　(注：在此“{”和“}”为元语言符号)

关于这个文法的递归下降分析程序见3.3.1节，我们很容易将其改造为如下的递归下降分析制导翻译程序：

```
E()                                          //E→T{+T}
{
    E(1).place=T( );                         //调用过程 T
    do{
          scaner( );                         //读进下一个符号
          E(2).place=T( );
          T1=newtemp;
          emit( +, E(1).place, E(2).place, E(1).place );
      }while(lookahead=='+');
    return(E(1).place);
}
T( )                                         //T→F{*F}
{
    T(1).place=F( );
    do{
          scaner( );
          T(2).place=F( );
          T(1)=newtemp;
          emit( *, T(1).place, T(2).place, T(1).place );
       }while(lookahead=='*');
    return(T(1).place);
 }
```

```
F()                                          //F→ (E) | i
{
    if(lookahead=='i')
    {
        scaner( );
        return(i.place);
    }
    else if(lookahead=='(')
        {
            scaner( );
            F.place=E( );
            if(lookahead==')')
            {
                scaner( );
                return(F.place);
            }
            else error( );
        }
        else error( );
}
```

习 题 4

4.1 完成下列选择题：

(1) 中间代码的优点是　　。

A．节省存储空间　　B．编译时间短

C．编译结构在逻辑上更为简单明确　　D．节省内存且编译时间短

(2) 四元式之间的联系是通过　　实现的。

A．指示器　　B．临时变量

C．符号表　　D．程序变量

(3) 间接三元式表示法的优点为　　。

A．采用间接码表，便于优化处理

B．节省存储空间，不便于表的修改

C．便于优化处理，节省存储空间

D．节省存储空间，不便于优化处理

(4) 表达式 (┐A∨B)∧(C∨D) 的逆波兰表示为　　。

A．┐AB∨∧CD∨　　B．A┐B∨CD∨∧

C．AB∨┐CD∨∧　　D．A┐B∨∧CD∨

(5) 后缀式_____对应的中缀表达式是a-(-b)*c(注：@表示求负运算)。

A．a-b@c*　　B．ab@-c*

C．ab-c@*　　D．ab@c*-

(6) 后缀式ab+cd+/ 可用中缀表达式_____来表示。

A．a+b/c+d　　B．(a+b)/(c+d)

C．a+b/(c+d)　　D．a+b+c/d

(7) 表达式 (a+b)*c的后缀表达式为_____。

A．ab*c+　　B．abc*+　　C．ab+c*　　D．abc+*

(8) 中间代码生成时所依据的是_____。

A．语法规则　　B．词法规则

C．语义规则　　D．等价变换规则

(9) 四元式表示法的优点为_____。

A．不便于优化处理但便于表的更动

B．不便于优化处理但节省存储空间

C．便于优化处理也便于表的更动

D．便于表的更动也节省存储空间

(10) 有一语法制导翻译如下所示：

S→bAb　　{ print"1" }

A→ (B　　{ print"2" }

A→a　　{ print"3" }

B→Aa)　　{ print"4" }

若输入序列为b (((aa) a) a) b，且采用自底向上的分析方法，则输出序列为　　。

A．32224441　　B．34242421

C．12424243　　D．34442212

4.2 何谓“语法制导翻译”？试给出用语法制导翻译生成中间代码的要点，并用一简例予以说明。

4.3 令S.val为文法G[S]生成的二进制数的值，例如对输入串101.101，则S.val=5.625。按照语法制导翻译方法的思想，给出计算S.val的相应的语义规则，G[S] 如下：

G[S]：S→L. L｜L

L→LB｜B

B→0｜1

4.4 下面的文法生成变量的类型说明：

D→id L

L→, id L | : T

T→integer | real

试构造一个翻译方案，仅使用综合属性，把每个标识符的类型填入符号表中(对所用到的过程，仅说明功能即可，不必具体写出)。

4.5 写出翻译过程调用语句的语义子程序。在所生成的四元式序列中，要求在转子指令之前的参数四元式par按反序出现(与实现参数的顺序相反)。此时，在翻译过程调用语句时，是否需要语义变量(队列)queue？

4.6 设某语言的while语句的语法形式为

S→while E do S(1)

其语义解释如图4-21所示。

(1) 写出适合语法制导翻译的产生式；

(2) 写出每个产生式对应的语义动作。

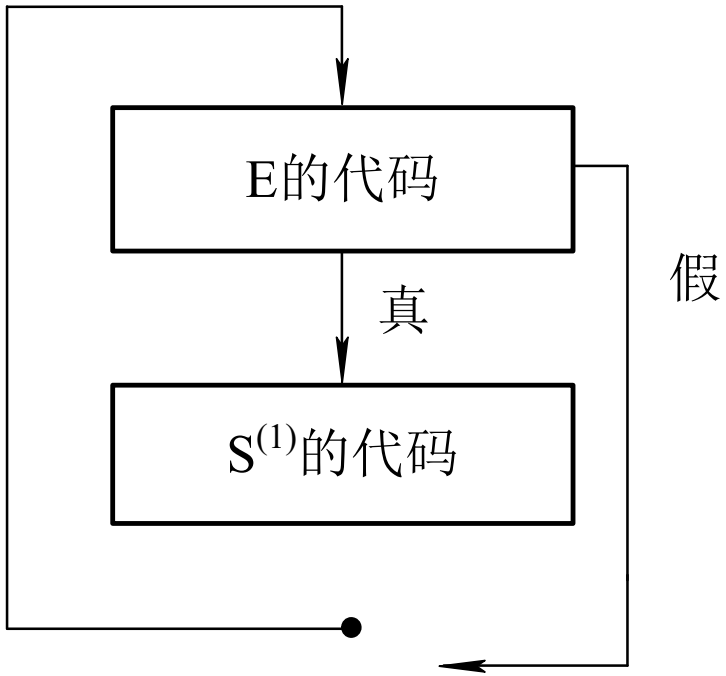


图4-21 习题4.6的语句结构图

4.7 改写4.4.2节中布尔表达式的语义子程序，使得$i^{(1)}$ rop $i^{(2)}$不按通常方式翻译为下面的相继两个四元式：

(jrop, $i^{(1)}$, $i^{(2)}$, 0)

(j, _, _, 0)

而是翻译成如下的一个四元式：

(jnrop, $i^{(1)}$, $i^{(2)}$, 0)

使得当$i^{(1)}$ rop $i^{(2)}$为假时发生转移，而为真时并不发生转移(即顺序执行下一个四元式)，从而产生效率较高的四元式代码。

4.8 按照4.5.3节的三种基本控制结构的文法将下面的语句翻译成四元式序列：

```
while( A<C∧B<D )
{
    if( A≥1 ) C=C+1;
    else while( A≤D )
    A=A+2;
}
```

4.9 按照4.5.3节的三种基本控制结构的文法将下面的语句翻译成四元式序列：

```
while( a∨b )
  if( x<y )
    while( c∧d )
      k=k+1 ;
  else
    if( m<n∧k<q )
      m=k ;
    else
      while( m≠k )
        m=m+1 ;
```

4.10 有一张纸厚0.5 mm，假如它足够大且不断把它对折，问对折多少次后它的厚度可以达到珠穆朗玛峰的高度(8848 m)。设纸厚随对折次数变化为h(初始为0.5 mm)、对折次数为n、循环控制变量为a且8848 m = 8848000 mm，下面是实现求解的程序段，将该程序段翻译成四元式序列。

```
n=0, a=1;
h=0.5;
while( a )
{
    n++;
    h=h*2;
    if( h>=8848000 )
    a=0;
}
```

4.11 求自然对数的底数e的程序段如下，将该程序段翻译成四元式序列。

```
e=1, n=1;
for( i=1; 1/n>=0.000001; i++ )
{
  n=n*i;
  e=e+1/n;
}
```

4.12 已知源程序如下：

```
prod=0;
i=1;
while( i≤20 )
{
  prod=prod+a[i]*b[i];
  i=i+1;
}
```

试按语法制导翻译法将上述源程序翻译成四元式序列(设A是数组a的起始地址，B是数组b的起始地址；机器按字节编址，每个数组元素占四个字节)。

4.13 给出文法G[S]：S→SaA｜A

A→AbB｜B

B→cSd｜e

(1) 请证实AacAbcBaAdbed是文法G[S] 的一个句型；

(2) 请写出该句型的所有短语、素短语以及句柄；

(3) 为文法G[S] 的每个产生式写出相应的翻译子程序，使句型AacAbcBaAdbed经该翻译方案后，输出为131042521430。