编译原理

2020年5月

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

编译原理
第六章

上海交通大学

张冬茉

Email:zhang-dm@cs.sjtu.edu.cn

2020年5月

第6章 中间代码生成

在编译器的分析-综合模型中，前端对源程序进行分析并产生中间表示，后端在此基础上生成目标代码。理想情况下，和源语言相关的细节在前端分析中处理，而关于目标机器的细节则在后端处理。

将源程序翻译成一种中间表示具有如下优点：

- **编译逻辑结构简单明确，与机器相关的工作集中到目标代码生成阶段，难度和工作量下降。**
- **便于移植和维护。**
- **利于优化。代码优化将分成与机器无关的中间代码优化及与机器相关的目标代码优化两个阶段，是优化更有效。**

如果有m种语言需要n种机型的编译器，只要写出m种前端和n种后端即可

与中间代码相关的内容包括：

- 中间代码表示
- 静态类型检查
- 中间代码生成

理论上说，编译器中除了源程序和目标程序外的所有中间结果都可以说是中间代码表示，主要包括：

- 语法树（高层中间表示形式）
- 三地址代码（低层中间形式）

6.1 语法树的变体——有向无环图

表达式的有向无环图（Directed Acyclic Graph，简称DAG）与语法分析树类似，一个DAG的叶子结点对应于原子运算分量，内部结点对应于运算符。与语法分析树不同的是，如果DAG中一个结点N表示一个公共子表达式，那么N可能有多个父结点。

例如：表达式a+a*(b-c)+(b-c)*d的DAG如下：

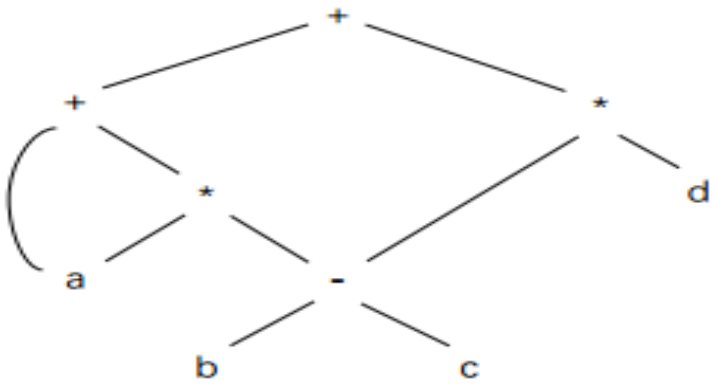


图 1 表达式 *a+a*(b-c)+(b-c)*d* 的 DAG

有向无环图DAG的值的编码

DAG的每个结点通常作为一条记录被存放在数组中，一个记录包括的结点信息有：

- 结点标号：如果记录表示叶子结点，那么结点标号是该结点的文法符号；如果记录表示内部结点，那么结点标号是运算符；
- 词法值：如果记录表示叶子结点，那么记录还包括该结点的词法值，通常是一个指向符号表的指针或者一个常量；
- 左右子结点：如果记录表示内部结点，那么记录还包括该结点的左右子结点。

由于一个DAG的结点都会保存在一个记录数组中，因此我们可以通过数组下标引用某个记录从而获取结点信息，这个数组下标称为相应结点的值编码。对图1的DAG，它的记录数组为表1：

其中，左边的下标是每个结点的值编码，并且在某些记录中可以包括值编码。例如：对于第5条记录，它表示内部结点"-"，该结点左边的子结点是叶子结点"b"，右边的子结点是叶子结点"c"，由于代表b和c的记录已经存在并且它们的值编码分别为2和3，因此内部结点"-"的记录为(-, 2, 3)。

下标	记录		
1	num	指向符号 a 的指针	
2	num	指向符号 b 的指针	
3	num	指向符号 c 的指针	
4	num	指向符号 d 的指针	
5	-	2	3
6	*	1	5
7	+	1	6
8	*	5	4
9	+	7	8

表 1 图 1 中的 DAG 对应的记录数组

6.2三地址代码

三地址代码是形如x = y op z的指令集合，之所以名为“三地址代码”，是因为指令x = y op z具有三个地址：两个运算分量y和z，一个结果变量x。由于三地址代码会对多运算符算术表达式和控制流语句的嵌套结构进行拆分，因此适用于目标代码的生成和优化。三地址代码可以是DAG的线性表示，例如：与图1表达式a+a*(b-c)+(b-c)*d的DAG对应的三地址代码如下：

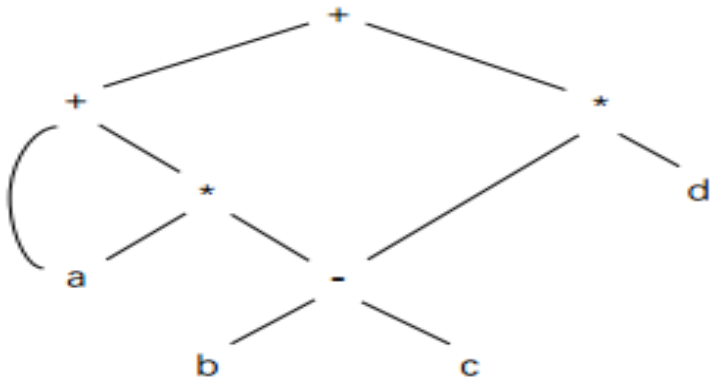


图 1 表达式 *a+a*(b-c)+(b-c)*d* 的 DAG

t1:=b-c
t2:=a*t1
t3:=a+t2
t4:=t1*d
t5:=t3+t4

a+a*(b-c)+(b-c)*d三地址代码

6.2.1 地址和指令

三地址代码基于两个基本的概念：地址和指令。

简单地说，

- 地址就是运算分量；

一个地址的表现形式可以是变量名、常量或者编译器生成的临时变量。

- 指令就是运算符。

表2是几种常见的三地址指令形式：

指令	形式	描述
赋值指令	x = y op z	op 是双目运算符或逻辑运算符，x、y、z 是地址
	x = op y	op 是单目运算符，y 是地址
复制指令	x = y	把 y 的值赋给 x
无条件转移指令	goto L	下一步要执行的指令是带有标号 L 的三地址指令
条件转移指令	if x goto L	当 x 为真时，下一步执行带有标号 L 的指令，否则下一步执行序列中的后一条指令
	if FALSE x goto L	当 x 为假时，下一步执行带有标号 L 的指令，否则下一步执行序列中的后一条指令
	if x relop y goto L	当 x 和 y 满足 relop 关系时，下一步执行带有标号 L 的指令，否则下一步执行序列中的后一条指令
参数传递和过程调用指令	param x_1 … param x_n call p, n	“param x”进行参数传递，“call p, n”进行过程调用。“call p, n”中的 p 表示过程，n 表示参数个数。这个 n 是必须的，因为前面的一些 param 指令可能是 p 返回之后才执行的某个函数调用的参数
带下标的复制指令	x = y[i]	把距离位置 y 处 i 个内存单元的位置中存放的值赋给 x
	x[i] = y	把距离位置 x 处 i 个内存单元的位置中的内容设置为 y
地址及指针赋值指令	x = &y	把 x 的值设置为 y 的地址
	x = *y	把位置 y 中存放的值赋给 x
	*x = y	把位置 x 中的内容设置为 y

表 2 几种常见的三地址指令形式

6.2.2三地址代码的数据结构——四元式

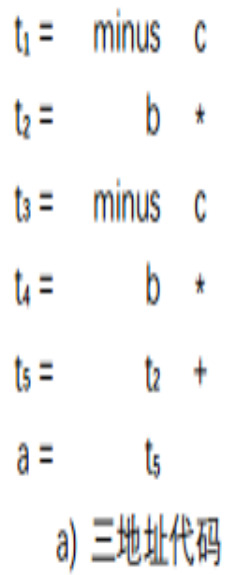
一个四元式是一条表示三地址指令的记录，它有4个字段：

- op：表示一个运算符；
- arg1：表示第一个运算分量；
- arg2：表示第二个运算分量；
- result：表示结果变量。

一个四元式中可能用到了所有4个字段，也可能只用到了其中几个字段，它的几个特例如下：

- 形如x = minus y的单目运算符指令不使用arg2字段，”minus”表示单目减运算符；
- 形如x = y的赋值指令不使用arg2字段，并且它的op字段是”=”；
- 形如param x的参数传递指令不使arg2和result字段；条件和非条件转移指令将目标标号放入result字段。

图2是一个例子，左边给出了三地址代码，右边是对应的四元式表示：



a) 三地址代码

	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

b) 四元式

图 2 三地址代码及其四元式表示

//6.2.3三地址代码的三元式表示

如果我们仿照DAG的记录数组用值编码表示临时变量的地址，那么就可以省略四元式中的result字段，三元式就是由此而来的。

一个三元式只有3个字段：

op、arg1和arg2：

- op:表示一个运算符；
- arg1：表示第一个运算分量；
- arg2：表示第二个运算分量；

它们的含义和在四元式中相同。为了取代四元式中的result字段，三元式用值编码表示结果变量的地址。

图3中三地址代码的三元式表示

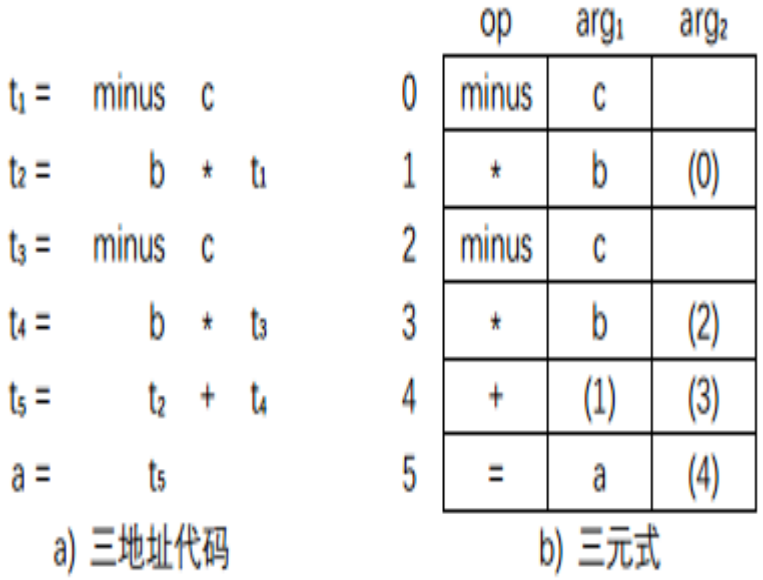


图 3 三地址代码及其三元式表示

一个四元式中可能用到了所有4个字段，也可能只用到了其中几个字段，它的几个特例如下：

- 形如x = minus y的单目运算符指令不使用arg2字段，”minus”表示单目减运算符；
- 形如x = y的赋值指令不使用arg2字段，并且它的op字段是”=”；
- 形如param x的参数传递指令不使arg2和result字段；条件和非条件转移指令将目标标号放入result字段。

图2是一个例子，左边给出了三地址代码，右边是对应的四元式表示：

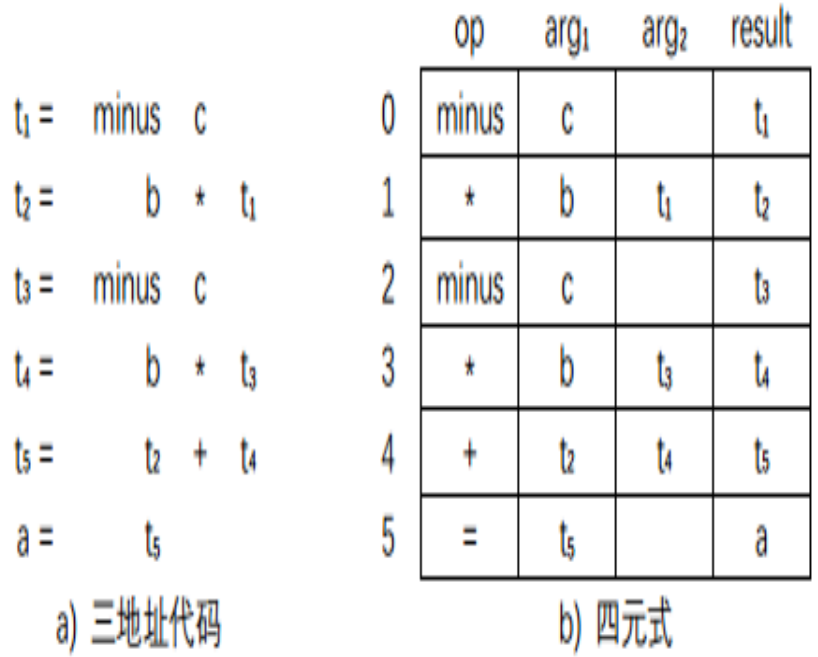


图 2 三地址代码及其四元式表示

//三地址代码的间接三元式表示

三地址代码的三元式表示存在一个问题，如果记录数组中的某条记录R的位置发生改变，那么所有使用到记录R的值编码的记录都需要更新。举个例子，在图3(b)中，如果把第1和第2条记录的位置互换，那么第3和第4条记录的内容都会发生改变。为了解决这个问题，提出了用间接三元式来表示三地址代码。

一个间接三元式在三元式的基础上增加了一个列表，这个列表包含了指向三元式的指针。仍以例子说明，图4是图3中三元式的间接三元式：

图4(a)是图3(b)的间接三元式，它使用了一个instruction数组保存要执行的指令，每条指令是一个指向某个三元式的指针。这样的话，当我们改变指令顺序时，就不用再更新三元式了，如图4(b)所示。

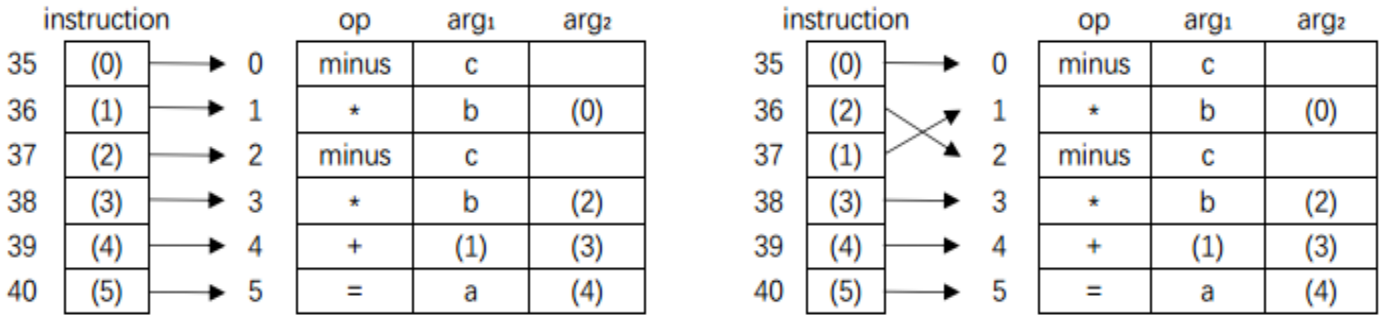


图 4 间接三元式

//6.2.4静态单赋值形式

- 静态单赋值（Static Single Assignment，简称SSA）形式是另一种中间表示形式，它和三地址代码的主要区别在于：
- 第一，SSA中的所有赋值都是针对具有不同名字的变量的，这也是“静态单赋值”名字的由来。这一特性如下图5中表示：
- 第二，由于同一个变量可能在两个不同的控制流路径中被定值，因此SSA使用一种被称为φ函数的表示规则将这个变量的两处定值合并起来。这一特性如图6所示：X3的值取决于X1和X2的值

p	=	a	+	b
q	=	p	-	c
p	=	q	*	d
p	=	e	-	p
q	=	p	+	q

a) 三地址代码

p_1	=	a	+	b
q_1	=	p_1	-	c
p_2	=	q_1	*	d
p_3	=	e	-	p_2
q_2	=	p_3	+	q_1

b) 静态单赋值形式

图 5 三地址代码和 SAA 的比较

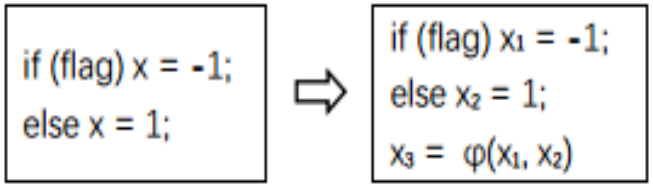


图 6 SAA 的φ函数

//6.3 类型和声明

类型的应用分为类型检查和翻译：

- 类型检查具有发现程序中错误的作用。如果一个语言能够在编译期间进行类型检查，从而保证不会在运行期间发生类型错误，那么这个语言是强类型的。类型检查有两种方式，综合和推导：
- 类型综合：根据子表达式的类型构造出父表达式的类型，它要求名字先声明再使用。例如，表达式A+B的类型是根据A和B的类型定义的；
- 类型推导：根据一个语言中结构的使用方式来确定该结构的类型，它不要求名字在使用前先进行声明。例如，如果empty是一个测试列表是否为空的函数，那么表达式empty(x)中的x必须是一个列表类型。
- 类型翻译时的应用：根据一个名字的类型，编译器可以确定这个名字在运行时需要多大的存储空间，类型信息还会在其他很多地方的翻译时被用到。

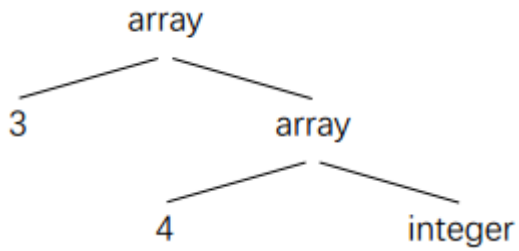
//6.3.1 类型表达式

类型表达式可以是基本类型，也可以是通过把称为类型构造算子的运算符作用于类型表达式得到的。不同语言的基本类型和类型构造算子可能不同。具体地，一个类型表达式可以是：

- 基本类型：例如C语言的基本类型包括int、float、double、char等；
- 数组：将类型构造算子array作用于一个数字和一个类型表达式可以得到一个类型表达式；
- 记录：一个记录是包含一个或多个字段的数据结构，这些字段都有一个唯一的名字。将类型构造算子record(struct)作用于字段的名字和类型可以得到一个类型表达式；
- 函数。使用类型构造算子→可以构造得到函数类型的类型表达式，“从类型s到类型t的函数”记作“s→t”；
- 类。在面向对象的编程语言中用来封装数据和方法的抽象数据类型（ADT）。

例6.1

- 可以使用与DAG相似的方式表示一个类型表达式：叶子结点可以是基本类型、类和类型变量，内部结点是类型构造算子。例如，类型 int[3][4] 可以表示成：



//6.3.2 类型等价

两种类型的结构等价，当且仅当下面的某个条件为真：

- 它们是相同的基本类型；
- 它们是相同的类型构造算子应用于结构等价的类型而构造得到的；
- 一个类型是另一个类型表达式的名字；

其中：如果一个类型的名字仅仅代表它自身，则前两个条件定义了类型表达式的名字等价。

注意：类型等价的原则在翻译是必须是确定的。

//6.3.3 声明

我们考虑一次只声明一个变量的情况（多个变量的情况用第五章的方法）。这种情况下，一次声明包括一个变量类型和一个变量名并以一个";"结尾，如int x;。这种声明方式可以扩展成序列的形式，如int x; float y; char z;，这种形式通常用于声明记录中的字段。

- 变量的声明可以用下面的文法表示：
- $D \rightarrow T\ id;D \mid \varepsilon$

在这个文法中：

- 符号T表示基本类型、数组类型和记录类型；
- 符号D表示声明序列，这个序列中至少声明了一个变量；
- 符号id表示变量名，严格地说，在声明序列中的任何两个变量的名字都不相同。

//6.3.4局部变量名的存储布局

- 变量的类型告诉我们它在运行时刻需要多大的内存空间，在编译时刻，我们可以使用变量的内存大小信息为每个变量分配一个相对地址。
- 一个变量的相对地址需要用该变量的起始地址和该变量的类型宽度来刻画。其中，起始地址是用于存储变量的字节块的第一个字节的地址，类型宽度是用于存储变量的字节块包含的字节数。
- 例如，假设x是一个整型变量并且它的起始地址为100，那么x的相对地址是从100开始，到103结束的4字节的字节块。
- 假设存储变量的方式是连续的（即对变量x和y，x的相对地址是从d1到di的字节块，那么y的起始地址是di+1），则存储变量的关键问题变成了确定每个变量需要多大的存储空间（说白了就是确定每个类型的宽度）。

//6.3.4局部变量名的存储布局

下面我们对计算基本类型、数组类型和记录类型的宽度分别进行说明:

- 基本类型:

  基本类型的宽度是由语言事先定义好的，比如Java的int类型的宽度是4个字节;

- 数组类型:

  数组类型的宽度是由元素的类型宽度和元素的数量共同决定的，比如数组类型int[3]的宽度是4*3=12个字节;

- 记录类型:

  记录类型的宽度是由该记录中所有字段的类型宽度共同决定的，比如记录类型record{ int x; int y; }的宽度是4+4=8个字节。

//例6.2

包括基本类型、数组类型和记录类型在内的声明语句的一个可行的SDT如下：

产生式	语义动作
D → T id; D₁	{ top.push(id.lexeme, T.type, offset); offset = offset + T.width; }
D → ε	
T → B C	{ C.t = B.type; C.w = B.width; } { T.type = C.type; T.width = C.width; }
T → record '{' D '}'	{ Stack.push(top); Stack.push(offset); top = new Env(); offset = 0; } { T.type = record(top); T.width = offset; offset = Stack.pop(); top = Stack.pop(); }
B → int	{ B.type = integer; B.width = 4; }
B → float	{ B.type = float; B.width = 4; }
C → [num]C₁	{ C₁.t = C.t; C₁.w = C.w; C.type = array(num.value, C₁.type); C.width = num.value * C₁.width; }
C → ε	{ C.type = C.t; C.width = C.w; }

表 0 声明语句的翻译方案

在表0中，每个符号的含义是：

- 每个非终结符号都有两个综合属性type和width，前者表示类型，后者表示类型宽度；
- 非终结符号C还有两个继承属性t和w，它们的作用是将类型和宽度信息从语法分析树的B结点传递到对应于产生式C→ε的结点；
- 变量Env表示符号表，top表示指向符号表的指针，offset表示记录中的字段相对记录首地址的偏移量，Stack表示用于存放top和offset的栈。

对记录类型的宽度进行计算是一个比较复杂的过程，这种情况下，需要把记录中的字段保存到一个新的符号表中（相近的概念是变量的作用域）：

- （1）首先，在产生式T→record'{'P'}'中，第一个动作对当前的上下文环境进行保存，该动作把指向当前符号表的指针top和记录字段偏移量的变量offset保存到Stack中，并把top指向一个新符号表，把offset重置为0;
- （2）然后，在产生式D→T id;D1中，动作把名为id.lexeme的变量加入到符号表中，并将offset加上该变量的类型宽度;
- （3）接着，重复第2步直到没有新的变量被声明，即到达产生式D→ε;
- （4）最后，在产生式T→record'{'P'}'中，第二个动作计算记录类型的宽度并对之前的上下文环境进行还原，该动作把记录的类型设为record(top)，把记录的宽度设为offset，并把top指向第1步保存在Stack中的符号表，把offset设为第1步保存在Stack中的值。

6.4 表达式的翻译

包括赋值运算、加法运算和取负运算在内的表达式的一个可行的SDT如下：

在表1中，每个符号的含义是：

- 非终结符号S表示一个表达式；非终结符号E表示一个子表达式，它的addr属性表示对应变量的代数值；终结符号id表示一个运算分量，它的lexeme属性是由词法分析器返回的值；
- top是一个指向当前符号表的指针，top.get(x)表示从符号表中取得标号为x的记录；
- gen是一个负责生成三地址代码的函数，传递给它的参数就是需要生成的三地址代码。参数中包括变量和字面常量，字面常量需要在左右加上单引号；
- Temp是一个负责生成临时地址的函数，这个临时地址通常用于编译器产生的临时变量。

产生式	语义动作
$S \rightarrow id = E$	{ gen(top.get(id.lexeme) '=' E.addr); }
$E \rightarrow E_1 + E_2$	{ E.addr = new Temp(); gen(E.addr '=' E_1.addr '+' E_2.addr); }
$E \rightarrow -E_1$	{ E.addr = new Temp(); gen(E.addr '=' 'minus' E_1.addr); }
$E \rightarrow (E_1)$	{ E.addr = E_1.addr; }
$E \rightarrow id$	{ E.addr = top.get(id.lexeme); }

表 1 表达式的翻译方案

6.4.1表达式的计算

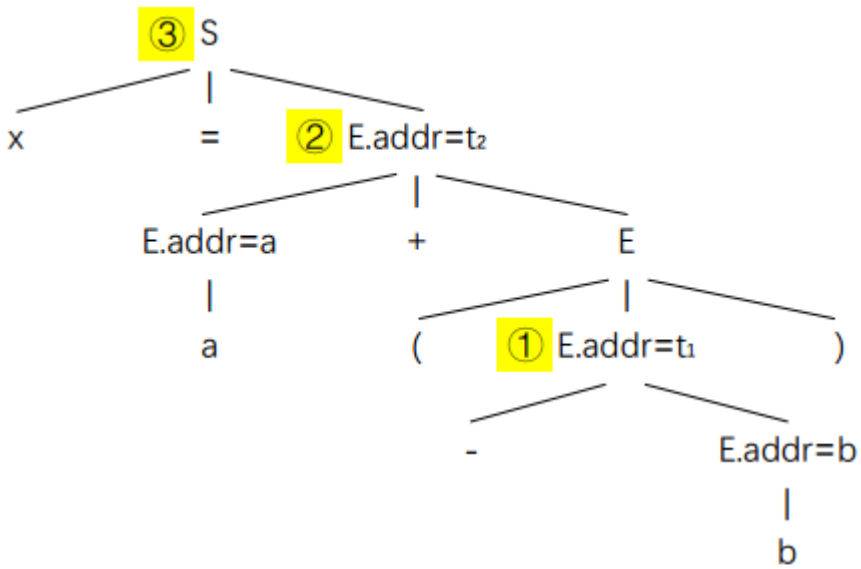
以表达式x=a+(-b)为例，它的注释语法分析树（图1a)和三地址代码(图1b):

(1) 由于每个语义动作都在产生式的最右端，因此这个SDT可以在自底向上的语法分析过程中实现;

(2) 第一次归约发生在图1(a)中的标记1处，这里使用了产生式E→-E，相应的语义动作生成了图1(b)中的第1条指令;

(3) 第二次归约发生在图1(a)中的标记2处，这里使用了产生式E→E+E，相应的语义动作生成了图1(b)中的第2条指令;

(4) 第三次归约发生在图1(a)中的标记3处，这里使用了产生式S→id=E，相应的语义动作生成了图1(b)中的第3条指令。



a) 注释语法分析树

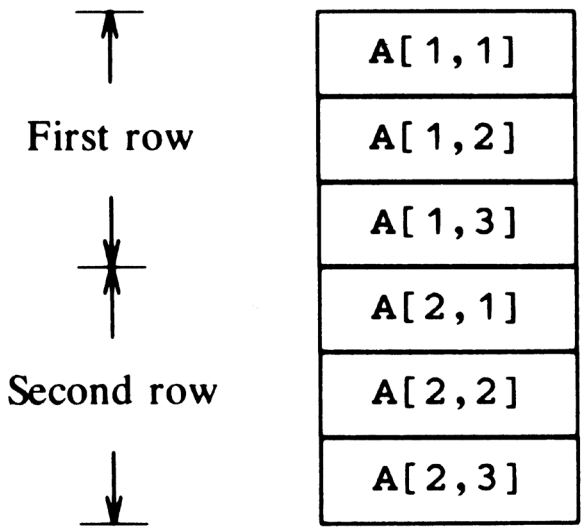
t_1 = minus b
t_2 = a + t_1
x = t_2

b) 三地址代码

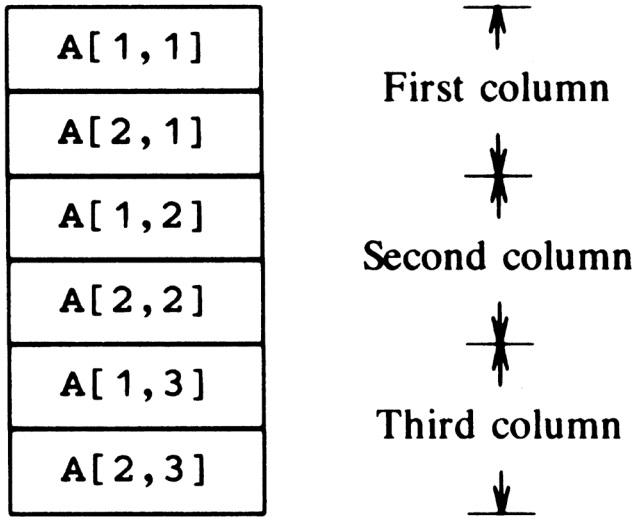
图 1 *x=a+(-b)*的翻译过程

6.4.2数组元素的寻址

相同类型的一组数据形成数组。数组总是存放在一片连续单元里。一维数组的元素是线性序的，所以与所分配的连续空间极易建立一一对应关系。二维数组形式上可以用两种形式之一来保存，或者行优先（一行接一行），或者列优先（一列接一列）。下图给出了一个2×3数组的(a)行优先形式和(b)列优先形式。



(a) ROW-MAJOR



(a) COLUMN-MAJOR

如果二维数组采用行优先形式存储，

可用如下公式计算A[i1][i2]的相对地址：

- $$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

其中:low1 和 low2 分别是 i1，i2 的下界，n2 是 i2 可取值的个数。也就是说，如果 high2 是 i2 的上界，那么 n2 = high2-low2 + 1。因为一般 i1，i2 是编译时惟一尚未知道的值，可把上述表达式重写成：

- $$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2 + low_2) \times w)$$

其中：该表达式的后一项W可以在编译时确定。

将行优先形式推广到多维数组。

表达式可推广成如下表达式以计算 A[i1] [i2] …[ik] 的相对地址：

$$((...((i_1 n_2)+i_2)n_3+i_3)...)n_k+i_k)\times w$$
$$+\text{base}-((...((low_1 n_2+low_2)n_2+low_3)...)n_k+low_k)\times w$$

因为对任何 j，nj =highj -lowj +1 是固定的，第二行的项可以由编译器计算出来。

对于C语言系列的数组，low=0, 所以第二行为base, 而nj为定义时给出。

6.4.3数组引用的翻译

数组引用的一个可行的SDT如表2:

产生式	语义动作
S → id = E	{ gen(top.get(id.lexeme) '=' E.addr); }
S → L = E	{ gen(L.array.base '[' L.addr ']' '=' E.addr); }
E → E_1 + E_2	{ E.addr = new Temp(); gen(E.addr '=' E_1.addr '+' E_2.addr); }
E → id	{ E.addr = top.get(id.lexeme); }
E → L	{ E.addr = new Temp(); gen(E.addr '=' L.array.base '[' L.addr ']'); }
L → id[E]	{ L.array = top.get(id.lexeme); L.type = L.array.type.elem; L.addr = new Temp(); gen(L.addr '=' E.addr '*' L.type.width); }

L → L_1[E]	{ L.array = L_1.array;
	L.type = L_1.type.elem;
	t = new Temp();
	L.addr = new Temp();
	gen(t '=' E.addr '*' L.type.width);
	gen(L.addr '=' L_1.addr '+' t); }

表 2 数组引用的翻译方案

在表2中，每个符号的含义是：

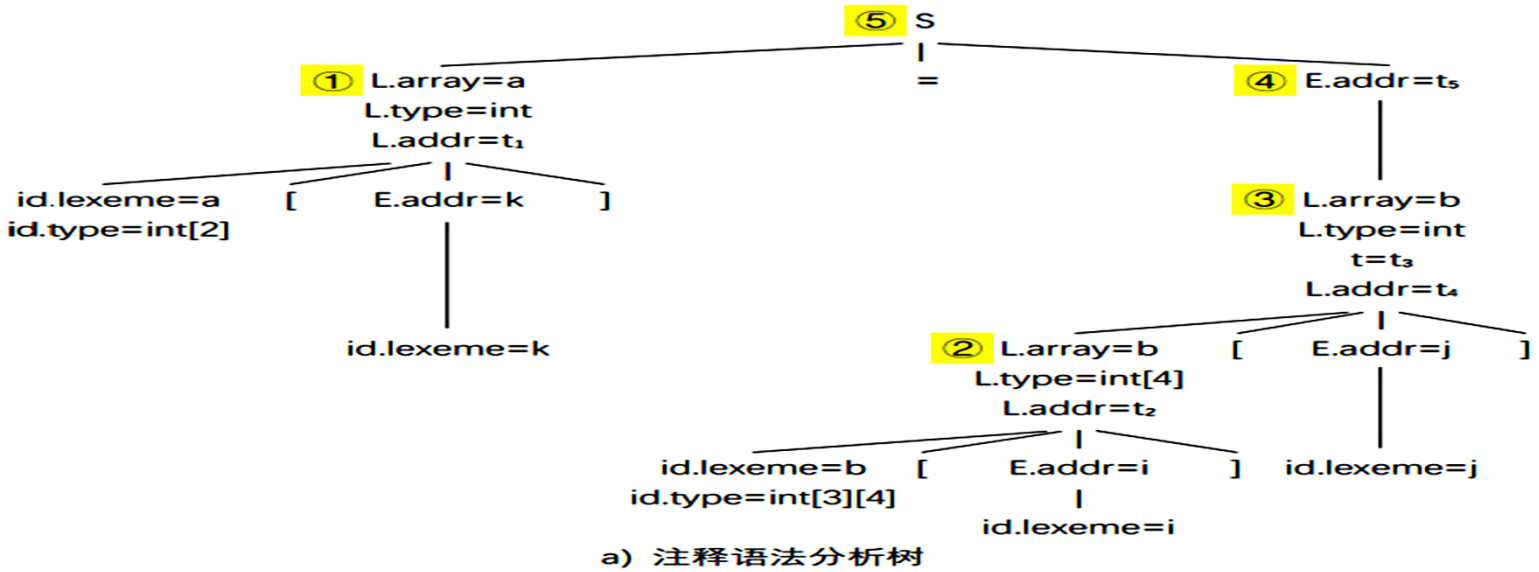
- 非终结符号S和E、终结符号id、变量top、函数gen和Temp的含义与表1中的相同，非终结符号L表示一个数组变量；
- L.array是指向数组名字对应的符号表条目的指针，假设L.array指向条目a，a.base是数组的基地址，a.type是数组的类型，a.type.elem是数组中元素的类型。例如：

  对类型为int[2]的数组a，有L.array=a，L.array.base=a[0]，
  L.array.type=int[2]，L.array.type.elem=int；
- L.addr是相对数组基地址偏移的字节数，对距离数组基地址L.addr个字节的元素的引用是L.array.base[L.addr]；
- L.type是数组中元素的类型，等同于L.array.type.elem，L.type.width表示数组中元素的类型宽度。例如：

  对类型为int[2]的数组a，有L.type=int，L.type.width=4；
  对类型为int[3][4]的数组b，有L.type=int[4]，L.type.width=16。

例6.3

- 以表达式 *a[k]=b[i][j]* 为例，它的注释语法分析树和三地址代码如下：



a) 注释语法分析树

$t_1 = k * 4$
$t_2 = i * 16$
$t_3 = j * 4$
$t_4 = t_2 + t_3$
$t_5 = b[t_4]$
$a[t_1] = t_5$

b) 三地址代码

图 2 *a[k]=b[i][j]*的翻译过程

- 在图2中，详细的翻译过程如下：
- 由于每个语义动作都在产生式的最右端，因此这个SDT可以在自底向上的语法分析过程中实现；
- 第一次归约发生在图2(a)中的标记1处，这里使用了产生式L→id[E]，相应的语义动作生成了图2(b)中的第1条指令；
- 第二次归约发生在图2(a)中的标记2处，这里使用了产生式L→id[E]，相应的语义动作生成了图2(b)中的第2条指令；
- 第三次归约发生在图2(a)中的标记3处，这里使用了产生式L→L[E]，相应的语义动作生成了图2(b)中的第3和第4条指令；
- 第四次归约发生在图2(a)中的标记4处，这里使用了产生式E→L，相应的语义动作生成了图2(b)中的第5条指令；
- 第五次归约发生在图2(a)中的标记5处，这里使用了产生式S→L=E，相应的语义动作生成了图2(b)中的第6条指令。

//6.5 类型检查

- 类型检查具有发现程序中错误的作用。如果一个语言能够在编译期间进行类型检查，从而保证不会在运行期间发生类型错误，那么这个语言是强类型的。
- 类型检查的思想还可用于提高系统的安全性，可安全地导入和执行软件模块。

6.5.1 类型检查规则

类型检查有两种方式，综合和推导：

- 类型综合：根据子表达式的类型构造出父表达式的类型，它要求名字先声明再使用。例如，表达式A+B的类型是根据A和B的类型定义的；
- 类型推导：根据一个语言中结构的使用方式来确定该结构的类型，它不要求名字在使用前先进行声明。例如，如果empty是一个测试列表是否为空的函数，那么表达式empty(x)中的x必须是一个列表类型。

6.5.2 类型转换

类型转换有两种方式，拓宽和窄化：

- 拓宽转换：把范围较小的类型拓宽为范围较大的类型，这种转换能够保留所有原有的信息，通常由编译器自动完成，因此也称为自动类型转换。例如，把int类型转换为float类型；
- 窄化转换：把范围较大的类型拓宽为范围较小的类型，这种转换可能丢失一些原有的信息，通常由开发者手动完成，因此也称为强制类型转换。例如，把float类型转换为int类型。

6.6 控制流

控制流语句和布尔表达式的翻译是结合在一起的，布尔表达式的值被用于：

- 改变控制流：即用程序到达的位置来表示布尔表达式的值。用这种方法实现控制流语句中的布尔表达式尤其方便。

  例如：对于表达式 E1 or E2，如果确定了 E1 是真，那么可以肯整个表达式为真，而不必再去计算 E2。

- 计算逻辑值：将真和假按数值编码，这样的布尔表达式的计算类似于对算术表达式的计算。

  例如：可以用非0表示真，0表示假；或者非负数表示真，负数表示假。

上述两种方法中，不能绝对地说某一种优于另一种。本节将讨论把布尔表达式翻译成三地址码的控制流方法。

6.6 .1布尔表达式

布尔表达式的一个可行的SDT如下:

产生式	语义动作
$B \rightarrow B_1 \mid\mid MB_2$	{ backpatch(B_1.falselist, M.instr); B.truelist = merge(B_1.truelist, B_2.truelist); B.falselist = B_2.falselist; }
$B \rightarrow B_1 \&\& MB_2$	{ backpatch(B_1.truelist, M.instr); B.truelist = B_2.truelist; B.falselist = merge(B_1.falselist, B_2.falselist); }
$B \rightarrow !B_1$	{ B.truelist = B_1.falselist; B.falselist = B_1.truelist; }
$B \rightarrow (B_1)$	{ B.truelist = B_1.truelist; B.falselist = B_1.falselist; }
$B \rightarrow E_1$ **rel** E_2	{ B.truelist = makelist(nextinstr); B.falselist = makelist(nextinstr + 1); gen('if' E_1.addr **rel**.op E_2.addr 'goto _'); gen('goto _'); }
B → true	{ B.truelist = makelist(nextinstr); gen('goto _'); }
B → false	{ B.falselist = makelist(nextinstr); gen('goto _'); }
M → ε	{ M.instr = nextinstr; }

表 3 布尔表达式的翻译方案

在表3中，每个符号的含义是：

- 非终结符号B表示一个布尔表达式，它的truelist属性是一个包含指令地址的列表，这些地址是当B为真时控制流应该跳转到的指令地址，它的falselist属性也是一个包含指令地址的列表，这些地址是当B为假时控制流应该跳转到的指令地址；非终结符号E表示一个表达式，它的addr属性表示对应变量的代数值；
- 符号M是一个标记非终结符号，它的instr属性负责记录下一条指令的地址；
- 变量nextinstr表示下一条指令的地址，即下一次生成的三地址代码会被放在nextinstr所指向的地址上；

- 函数makelist(i)负责创建一个只包含指令地址i的列表，并返回一个指向新创建列表的指针；
- 函数merge(p, q)负责将p和q指向的列表进行合并，并返回一个指向合并的列表的指针；
- 函数backpatch(p, i)的功能比较复杂，首先，p是一个指向列表的指针，对p指向的列表中的每个指令地址j，地址j上的指令是一个未填写目标跳转地址的转移指令（如goto _）；其次，i是一个地址，这个地址是一个目标跳转地址；最后，函数backpatch用i填写每个j上的转移指令的标跳转地址。

6.6.2 短路代码

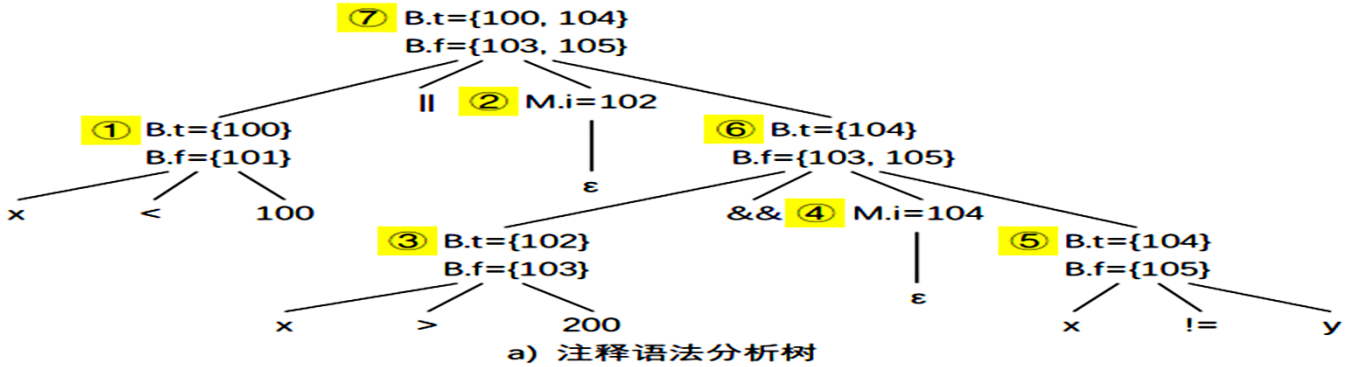
即使不对布尔运算符生成代码并且不必生成整个表达式的代码，我们也可以把布尔表达式翻译成的三地址码。这种风格的计算有时叫做“短路”或“跳转”代码。如果用代码序列中的位置来表示表达式的值，那么不用为布尔运算符 ||,&&和！生成代码就可以计算表达式的值。

以语句 *if (x<100 || x >200 && x!=y) x=0* **为例：逻辑跳转代码为：**

- *if x<100 goto L2*
- *If False x>200 goto L1*
- *If False x!=y goto L1*
- *L2: x=0*
- *L1:*

例6.4

以表达式：*x<100||x>200&&x!=y*为例，它的注释语法分析树和三地址代码如下：



a) 注释语法分析树

```
1)   99:   code
(ni)100:
```

```
2) 100:   if x<100 goto _
   101:   goto _
(ni)102:
```

```
3) 102:   if x>200 goto _
   103:   goto _
(ni)104:
```

```
4) 104:   if x!=y goto _
   105:   goto _
(ni)106:
```

```
5) 101:   …
   102:   if x>200 goto 104
   103:   …
```

```
6) 100:   …
   101:   goto 102
   102:   …
```

```
7) 100:   if x<100 goto _
   101:   goto 102
   102:   if x>200 goto 104
   103:   goto _
   104:   if x!=y goto _
   105:   goto _
(ni)106:
```

b) 三地址代码

图 3 *x<100 || x>200 && x!=y* 的翻译过程

在图3中，详细的翻译过程如下：

- 由于每个语义动作都在产生式的最右端，因此这个SDT可以在自底向上的语法分析过程中实现并假设初始时指令地址从100开始；
- 第一次归约发生在图3(a)中的标记1处，此时nextinstr指向地址100。这里使用了产生式B→E rel E，相应的语义动作把地址100放入B.truelist中，把地址101放入B.falselist中，并生成了图3(b)(2)中的两条转移指令，这两条转移指令的目标跳转地址都未被填写；
- 第二次归约发生在图3(a)中的标记2处，此时nextinstr指向地址102。这里使用了产生式M→ε，相应的语义动作把M.instr设为102；
- 第三次归约发生在图3(a)中的标记3处，此时nextinstr指向地址102。这里使用了产生式B→E rel E，相应的语义动作把地址102放入B.truelist中，把地址103放入B.falselist中，并生成了图3(b)(3)中的两条转移指令，这两条转移指令的目标跳转地址都未被填写；

- 第四次归约发生在图3(a)中的标记4处，此时nextinstr指向地址104。这里使用了产生式M→ε，相应的语义动作把M.instr设为104；
- 第五次归约发生在图3(a)中的标记5处，此时nextinstr指向地址104。这里使用了产生式B→E rel E，相应的语义动作把地址104放入B.truelist中，把地址105放入B.falselist中，并生成了图3(b)(4)中的两条转移指令，这两条转移指令的目标跳转地址都未被填写；
- 第六次归约发生在图3(a)中的标记6处，此时nextinstr指向地址106。这里使用了产生式B→B1 && MB2，相应的语义动作设置了B.truelist和B.falselist，并用地址104填充地址102上的转移指令的目标跳转地址，如图3(b)(5)所示；

- 第七次归约发生在图3(a)中的标记7处，此时nextinstr指向地址106。这里使用了产生式B→B1 || MB2，相应的语义动作设置了B.truelist和B.falselist，并用地址102填充地址101上的转移指令的目标跳转地址，如图3(b)(6)所示；
- 最终的三地址代码如图3(b)(7)所示，在第六和第七次归约中填充转移指令的目标跳转地址的技术称为回填，回填技术用来在一趟扫描中完成对布尔表达式或控制流语句的目标代码生成。

6.6.3 控制流语句

控制流语句的一个可行的SDT如下：

产生式	语义动作
S → if (B) M S_1	{ backpatch(B.truelist, M.instr); S.nextlist = merge(B.falselist, S_1.nextlist); }

S → if (B) M_1 S_1 N else M_2 S_2	{ backpatch(B.truelist, M_1.instr);
	backpatch(B.falselist, M_2.instr);
	tcmp = merge(S_1.nextlist, N.nextlist);
	S.nextlist = merge(temp, S_2.nextlist); }

S → while M_1 (B) M_2 S_1	{ backpatch(S_1.nextlist, M_1.instr); backpatch(B.truelist, M_2.instr); S.nextlist = B.falselist; gen('goto' M_1.instr); }
S → { L }	{ S.nextlist = L.nextlist; }
S → A	{ S.nextlist = **null**; }
M → ε	{ M.instr = nextinstr; }
N → ε	{ N.nextlist = makelist(nextlist); gen('goto _'); }
L → L_1 M S	{ backpatch(L_1.nextlist, M.instr); L.nextlist = S.nextlist; }
L → S	{ L.nextlist = S.nextlist; }

表 4 控制流语句的翻译方案

6.6.3 控制流语句

控制流语句的一个可行的SDT如下：

产生式	语义动作
S → if (B) M S_1	{ backpatch(B.truelist, M.instr); S.nextlist = merge(B.falselist, S_1.nextlist); }

S → if (B) M_1 S_1 N else M_2 S_2	{ backpatch(B.truelist, M_1.instr);
	backpatch(B.falselist, M_2.instr);
	tcmp = merge(S_1.nextlist, N.nextlist);
	S.nextlist = merge(temp, S_2.nextlist); }

S → while M_1 (B) M_2 S_1	{ backpatch(S_1.nextlist, M_1.instr); backpatch(B.truelist, M_2.instr); S.nextlist = B.falselist; gen('goto' M_1.instr); }
S → { L }	{ S.nextlist = L.nextlist; }
S → A	{ S.nextlist = **null**; }
M → ε	{ M.instr = nextinstr; }
N → ε	{ N.nextlist = makelist(nextlist); gen('goto _'); }
L → L_1 M S	{ backpatch(L_1.nextlist, M.instr); L.nextlist = S.nextlist; }
L → S	{ L.nextlist = S.nextlist; }

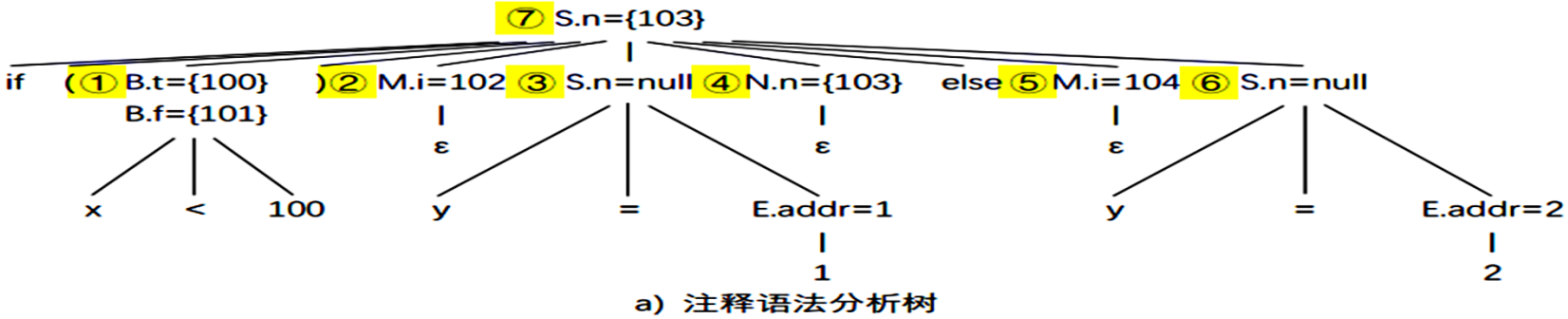
表 4 控制流语句的翻译方案

在表4中，每个符号的含义是：

- 非终结符号S表示一个表达式，L表示一个语句列表，A表示一个赋值语句，B表示一个布尔表达式；
- S的nextlist属性是一个包含指令地址的列表，这些地址是紧跟在S代码之后的转移指令的地址；L的nextlist属性与此类似；
- 符号M是一个标记非终结符号，它的instr属性负责记录下一条指令的地址；
- 符号N是一个标记非终结符号，它的nextlist属性是一个包含指令地址的列表，这些地址上的指令是无条件转移指令。

例6.5

以语句：if(*x<100) y=1;else y=2;*为例，它的注释语法分析树和三地址代码如下：



a) 注释语法分析树

```
1)   99:   code
(ni)100:
```

```
2) 100:   if x<100 goto _
     101:   goto _
(ni)102:
```

```
3) 102:   y = 1
(ni)103:
```

```
4) 103:   goto _
(ni)104:
```

```
5) 104:   y = 2
(ni)105:
```

```
6) 100:   if x<100 goto 102
     101:   goto 104
     102:   ···
```

```
7) 100:   if x<100 goto 102
     101:   goto 104
     102:   y = 1
     103:   goto _
     104:   y = 2
(ni)105:
```

b) 三地址代码

图 4 *if (x<100) y=1 else y=2* 的翻译过程

- 在图4中，详细的翻译过程如下：
- 由于每个语义动作都在产生式的最右端，因此这个SDT可以在自底向上的语法分析过程中实现；
- 为了美观，truelist、falselist、nextlist和instr都用它们的首字母表示，nextinstr用ni表示。假设初始时指令地址从100开始，即nextinstr指向地址100，如图4(b)(1)所示；
- 第一次归约发生在图4(a)中的标记1处，此时nextinstr指向地址100。这里使用了产生式B→E rel E，相应的语义动作把地址100放入B.truelist中，把地址101放入B.falselist中，并生成了图4(b)(2)中的两条转移指令，这两条转移指令的目标跳转地址都未被填写；
- 第二次归约发生在图4(a)中的标记2处，此时nextinstr指向地址102。这里使用了产生式M→ε，相应的语义动作把M.instr设为102；

- 第三次归约发生在图4(a)中的标记3处，此时nextinstr指向地址102。这里使用了产生式S→A，相应的语义动作把S.nextlist设为空，并生成了图4(b)(3)中的一条赋值指令；
- 第四次归约发生在图4(a)中的标记4处，此时nextinstr指向地址103。这里使用了产生式N→ε，相应的语义动作把地址103放入N.nextlist中，并生成了图4(b)(4)中的一条转移指令，这条转移指令的目标跳转地址未被填写；
- 第五次归约发生在图4(a)中的标记5处，此时nextinstr指向地址104。这里使用了产生式M→ε，相应的语义动作把M.instr设为104；

- 第六次归约发生在图4(a)中的标记6处，此时nextinstr指向地址104。这里使用了产生式S→A，相应的语义动作把S.nextlist设为空，并生成了图4(b)(5)中的一条赋值指令；
- 第七次归约发生在图4(a)中的标记7处，此时nextinstr指向地址105。这里使用了产生式S→if (B) MS1N else MS2，相应的语义动作设置了S.nextlist，并用地址102填充地址100上的转移指令的目标跳转地址，用地址104填充地址101上的转移指令的目标跳转地址，如图4(b)(6)所示；
- 最终的三地址代码如图4(b)(7)所示，在第六次归约中用到了回填技术。

例 6.6

混合模式布尔表达式的翻译示例

例如：4+a>b-c && d

t_1=4+a

t_2=b-c

if t_1>t_2 goto L_1

goto Lfalse

L_1: if d goto Ltrue

goto Lfalse

例6.7

布尔表达式代码的例子

- if (x<100 || x > 200 && x!= y) x = 0;
- 相应的代码

```
      if x < 100 goto L2
      goto L3
L3:   if x > 200 goto L4
      goto L1
L4:   if x != y goto L2
      goto L1
L2:   x = 0
L1:
```

例6.8 While (a<b) if (c<5) while (x>y) z=x+1;else x=y;

100: if a<b goto 102

101: goto 112

102: if c<5 goto 104

103: goto 110

104: if x>y goto 106

105: goto 100

106: t1=x+1

107: z=t1

108: goto 104

109: goto 100

110: x=y

111: goto 100

112:

第六章 总结

主要内容:

- 选择一种中间语言形式：有向无环图（DAG），三地址代码
- 表达式的翻译：只含简单变量的表达式，含数组元素的表达式，布尔表达式
- 控制流语句的翻译：if 语句和 while语句的翻译

第六章 作业

- 构造下列表达式的无环有向图（dag），假设+是左结合的：

  a+a+(a+a+a+(a+a+a+a))

- 分别将下面的C++语句翻译为三地址代码，设开始代码为100：

  1.if (a<b && e<d || !e<f+1) while(s>0){s--;i++;} else i--;

  2.X=a[b[i][j]]+c [b[i][j]];

  其中：a,b,c数组均为整数数组（W=4), b数组为2行3列的二维数组。

谢 谢!