第六章　语义分析与中间代码生成

词法分析和语法分析负责判断源程序在“形式”上是否合法，语义分析负责判断源程序在“语义”上是否合法。如果一个源程序在形式上和语义上都是合法的，那么这个源程序就是源语言的一个正确的程序，接下来就可以把它转换为中间代码。本章主要讨论如何利用第五章介绍的语法制导翻译技术实现语义分析和中间代码生成中的各项任务。首先介绍基于 S-属性定义的针对表达式和普通语句的类型检查的实现，然后介绍如何处理源程序中的说明语句以获取名字的属性信息。接着介绍作为源程序中间表示的中间语言的常用三地址指令，最后详细讨论高级语言常见可执行语句的翻译方法，包括赋值语句、布尔表达式、条件语句和循环语句等。

6.1 类型检查

从某种意义上来说，程序是作用在一个数据集合上的运算序列，而数据可以划分成不同的类型。每个程序设计语言都有自己的类型机制，包括数据类型的种类，及可对数据进行的运算和运算的规则。例如 C 语言，数据类型分为基本类型、指针类型、构造类型等，其中基本类型又有整型、实型、字符型等，构造类型包括数组类型、结构体类型、共同体类型等。根据数据的类型，编译器可以确定数据在运行时刻需要占用多大的存储空间。一般来说，源程序总是通过类型说明语句来定义变量和变量的类型。编译器在分析类型说明语句时，将变量及其类型等属性信息存放到符号表（变量表）中，如何采用语法制导翻译技术处理源程序的说明部分见 6.2 节。

类型检查是编译过程中语义分析的重要组成部分，主要工作是判断程序中每一个运算的运算分量的类型是否和预期的一致或者相容。例如，Java 语言要求&&运算符的两个运算分量必须是 boolean 类型，如果不满足这个条件编译器要报告类型错误，如果满足这个条件，类型检查通过，同时记录计算结果也是 boolean 类型的。

根据程序设计语言的类型机制，在编译时刻完成的类型检查称为**静态类型检查**，大部分程序设计语言采用静态类型检查，如 Pascal 和 C。经过静态类型检查之后，目标程序运行时将不会发生因数据类型不匹配而导致的错误。具体来说，静态类型检查主要完成以下几项工作：

（1）运算类型检查。例如对于算术表达式“X+Y”，需根据变量 X 和 Y 的类型信息（通过查询变量表获得），检查 X 和 Y 在类型上是否匹配。如果 X 和 Y 同为整型，则“+”运算是整数加；如果 X 和 Y 同为实型，则“+”运算是实数加。

（2）强制类型转换。例如在“X+Y”中，如果 X 为整型，Y 为实型，则应该首先用一个类型转换函数将 X 转换为实型，再与 Y 做实数加运算。

（3）语句类型检查。程序中的语句如果包含运算或者数据，也需要做类型的检查。

类型检查的实现一般采用语法制导翻译技术。下面讨论如何用语法制导翻译技术来实现表达式的类型检查和语句的类型检查。

表达式的类型检查

表 6-1 给出了一个实现表达式类型检查的语法制导定义。其中 E 代表算术表达式，每个表达式都有一个“type”（类型）属性，根据属性计算方法可知“type”属性是综合属性。“MOD”是运算的占位符，代表某个算术运算。“E_1 [E_2]”是数组引用结构，array(s,t)是数组类型，其中 s 是数组中元素的个数，t 是数组元素的类型；“E_1↑”是指针引用结构，pointer(t)是指针类型，t 是该指针指向的元素的类型。“num”代表整型常数，“type-error”是错误类型。函数 lookup 是一个语义动作，其参数 id.entry 是变量 id 在变量表中的入口指针，lookup（id.entry）可查询变量表获取 id 的类型信息。

表 6-1 表达式类型检查的语法制导定义

序号	产生式	语义规则
1	E → num	E.type := integer
2	E → id	E.type := lookup（id.entry）
3	E → E_1 MOD E_2	E.type := if E_1.type=integer and E_2.type=integer then integer else type-error
4	E → E_1 [E_2]	E.type := if E_2.type=integer and E_1.type=array(s,t) then t else type-error
5	E → E_1↑	E.type := if E_1.type=pointer(t) then t else type-error

产生式“E → num”的语义规则用于给 E 的类型属性赋值为整型；产生式“E → id”的语义动作查询变量表，取 id 的类型信息赋给 E.type；产生式“E → E_1 MOD E_2”的语义规则作类型一致性检查，这里假设 MOD 运算要求其两个运算分量必须都为整型，否则类型错误，两个整型数据做 MOD 运算，结果也为整型。若 MOD 运算允许其运算分量既可以为整型也可以为实型，则可以改写语义规则，作相容性检查，一个可能的语义规则如下：

{ E.type := if E_1.type=integer and E_2.type=integer

then integer

else if E_1.type=integer and E_2.type=real

then real

else if E_1.type=real and E_2.type=integer

then real

else if E_1.type=real and E_2.type=real

then real

else type-error }

产生式“E → E_1 [E_2]”生成数组引用结构，其语义规则检查 E_2 和 E_1 是否为整数类型和数组类型，是的话将数组元素的类型赋给 E.type，否则类型错误。产生式“E → E_1↑”生成指针引用结构，其语义规则检查 E_1 是否为指针类型，是的话将该指针指向的元素的类型赋给 E.type，否则类型错误。

表 6-1 的语法制导定义只有综合属性的计算，因此是 S-属性定义，可采用自底向上的属性计算方法来翻译句子。

【例 6-1】：使用 6-1 中的语法制导定义翻译句子 id_1*id_2+num，令 id_1 和 id_2 均为整型。

解：句子的分析树见图 6-1。对句子作自底向上的语法分析，输出最左归约序列。应用产生式归约的同时可执行产生式对应的语义规则，完成翻译工作。

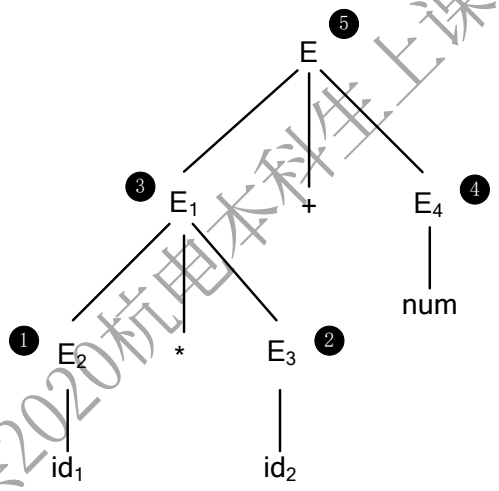


图 6-1 类型检查的例子

翻译过程如下：

（1）用产生式“E → id”归约，计算 E_2.type=integer；

（2）用产生式“E → id”归约，计算 E_3.type=integer；

（3）用产生式“E → E_1 MOD E_2”归约，计算 E_1.type=integer；

（4）用产生式“E → num”归约，计算 E_4.type=integer；

（5）用产生式“E → E_1 MOD E_2”归约，计算 E.type=integer。

计算结果 E.type 为 integer，不是“type-error”，类型检查通过。

语句的类型检查

表 6-2 给出了一个实现语句类型检查的语法制导定义。

表 6-2 语句类型检查的语法制导定义

序号	产生式	语义规则
1	S → id:=E	S.type := if id.type=E.type then void else type-error
2	S → if E then S_1	S.type := if E.type=boolean then S_1.type else type-error
3	S → while E do S_1	S.type := if E.type=boolean then S_1.type else type-error
4	S → S_1; S_2	S.type := if S_1.type=void and S_2.type=void then void else type-error

产生式“S → id:=E”生成赋值语句，其语义规则判断赋值语句左边的 id 的类型是否和右边的表达式 E 的类型一致，如果是则类型检查通过，并给 S.type 赋值 void（代表无类型），否则类型错误；产生式“S → if E then S_1”生成条件语句，其语义规则判断 E 的类型是否为 boolean 类型的，如果是则将 S_1.type 赋给 S.type，否则类型错误；产生式“S → while E do S_1”生成循环语句，其语义规则同样判断 E 的类型是否为 boolean 类型的，如果是则将 S_1.type 赋给 S.type，否则类型错误；产生式“S → S_1; S_2”生成复合语句，其语义规则判断 S_1 和 S_2 是否通过了类型检查（type 属性为 void），如果是则将 void 赋给 S.type，否则类型错误。

表 6-2 的语法制导定义同样也只有综合属性的计算，是 S-属性定义，类似地可采用自底向上的属性计算方法来实现语句的类型检查。

6.2 说明语句的处理

源程序由过程（或者函数）构成，当分析某个过程的说明语句时，可以获取该过程的局部名字（变量名、常量名等）及其属性信息，如类型、相对地址等，然后在对应的符号表中创建表项，并将名字的相关属性信息填入这些表项。相对地址是指目标代码运行时对静态数据区基址或活动记录局部数据区基址的一个偏移值，也称为偏移地址。在这里获取名字的相对地址，将为目标代码的生成做好准备。

大部分高级语言的同一个过程中的说明语句定义的名字一般来说具有相同的作用域，如 C、Pascal 和 Fortran 等面向过程的程序设计语言。通常把同一个过程中定义的名字组织在一张符号表中，在分析过程中需要一个全局变量 offset 来记录下一个名字的偏移地址，offset 的初始值为 0，标记了第一个名字的偏移地址。

表 6-3 给出的是翻译一般说明语句的语法制导定义。P 是开始符号，通过推导生成变量说明语句，其唯一的一条产生式“P → MD”中的 M 称为“标记非终结符号”，M 的产生式“M → ε”似乎对于句子的分析没有实际意义，其实不然。引入 M 是为了在自底向上翻译句子时通过执行其对应的语义规则“offset:=0”来对 offset 赋初值。注意，对于任何句子的自底向上分析，都是首先采用“M → ε”来进行归约。语义动作 enter(name, type, offset)用来为名字 name 建立一个符号表表项，并填入此名字的类型 type 及其相对地址 offset。T 代表数据类型，T.type 表示名字的类型，在表 6-3 给出的语法制导定义中，类型有实型、整型、数组类型（指定了数组的长度和数组元素的类型）、指针类型（指定了指针指向的数据的类型）；T.width 表示名字的域宽，即存储一个该类型数据所需的存储单元的个数，这里假设整型的域宽为 4，实型的域宽为 8，指针类型的域宽为 4，数组的域宽可以通过把数组的长度乘上数组元素的域宽来计算。

表 6-3 一般说明语句的处理

序号	产生式	语义规则
1	P → MD	
2	M → ε	offset:=0
3	D → D;D	
4	D → id:T	enter(id.name, T.type, offset); offset:=offset+T.width
5	T → integer	T.type:=integer; T.width:=4
6	T → real	T.type:=real; T.width:=8
7	T → array [num] of T_1	T.type:=array(num.val, T_1.type); T.width:=num.val*T_1.width
8	T → ↑T_1	T.type:=pointer(T_1.type); T.width:=4

【例 6-2】：使用表 6-3 中的语法制导定义翻译句子 id_1: real ; id_2: ↑integer。

解：表 6-3 语法制导定义是 S-属性定义，因为所有的属性都可以看作是综合属性。翻译句子可以采用自底向上的属性计算方法。

该句子的分析树见图 6-2。采用自底向上的语法分析方法分析句子构建分析树，需要 8 步最左归约操作。在归约的同时，执行产生式对应的语义规则可以完成翻译工作。

具体步骤如下：

（1）用产生式“M → ε”归约，将 offset 置为初值 0；

（2）用产生式“T → real”归约，计算 T_1 的类型属性和域宽，T_1.type:=real，T_1.width:=8；

（3）用产生式“D → id:T”归约，完成 id_1 的类型定义分析，用语义动作 enter(id_1, real, 0)将 id_1 及其类型信息 real 和偏移地址 0 填入符号表，同时 offset 递增为 8；

（4）用产生式“T → integer”归约，计算 T_3 的类型属性和域宽，T_3.type:=integer，T_3.width:=4；

（5）用产生式“T → ↑T_1”归约，计算 T_2 的类型属性和域宽，T_2.type:=pointer(integer)，T_2.width:=4；

（6）用产生式“D → id:T”归约，完成 id_2 的类型定义分析，用语义动作 enter(id_2, pointer(integer), 8)将 id_2 及其类型信息 pointer(integer)和偏移地址 8 填入符号表，同时 offset 递增为 12；

（7）用产生式“D → D;D”归约，无语义动作；

（8）用产生式“P → MD”归约，无语义动作。

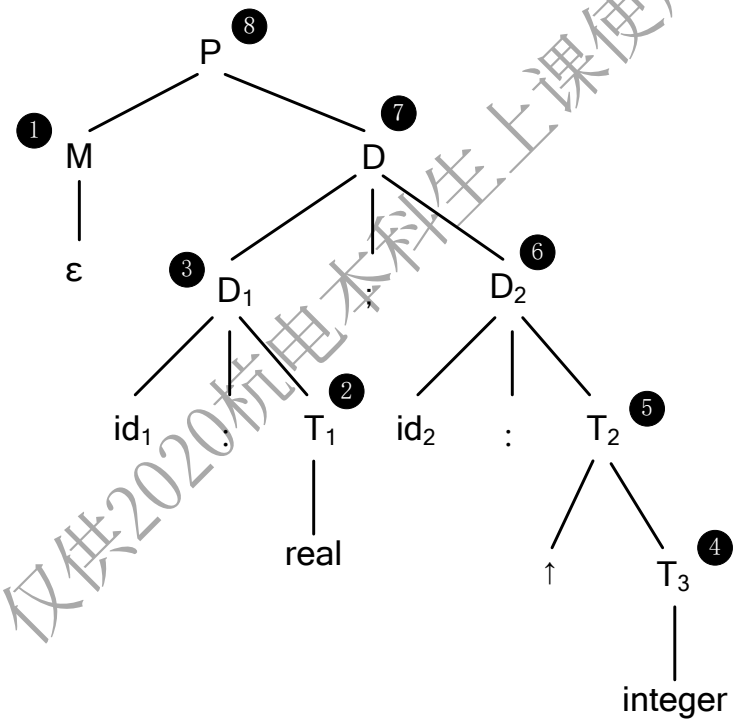


图 6-2 自底向上翻译句子 id_1: real ; id_2: ↑integer

翻译完成后，在符号表填入了说明语句定义的两个变量及这两个变量的类型信息和偏移地址，见表 6-4。

表 6-4 例 6-2 的符号表

名字	类型	偏移地址
id_1	real	0
id_2	pointer(integer)	8

某些语言允许嵌套过程，即在说明部分可以嵌套说明另外的过程或者函数，见图 6-3 所示的 Pascal 语言的例子。

```
（1） program sort（input，output）;
（2）   var a：  array[0..10］ of integer；
（3）         x : integer；
（4）   procedure  readarray；
（5）     var i ：  integer；
（6）     begin  … a…    end { readarray};
（7）   procedure  exchange（i，j：  integer）；
（8）     begin
（9）       x：= a[i] ；a[i]:=a[j]; a[j]:＝x
（10）    end { exchange};
（11）  procedure quicksort（m，n：  integer）；
（12）    var k，v：  integer；
（13）    function partition（y，z：  integer）  ：  integer；
（14）      var i，j ：  integer；
（15）      begin …a…
（16）          …v…
（17）          …exchange（i，j）；…
（18）      end { partition};
（19）    begin  …  end { quicksort};
（20）  begin  …  end. { sort }
```

图 6-3 一个 Pascal 语言例子

在这个例子中，（2）—（19）行都是主过程的说明部分。主过程是 sort，下面嵌套定义了 3 个过程 readarray、exchange 和 quicksort，其中 quicksort 下面又嵌套定义了一个函数 partition。整个程序的嵌套关系如图 6-4 所示。

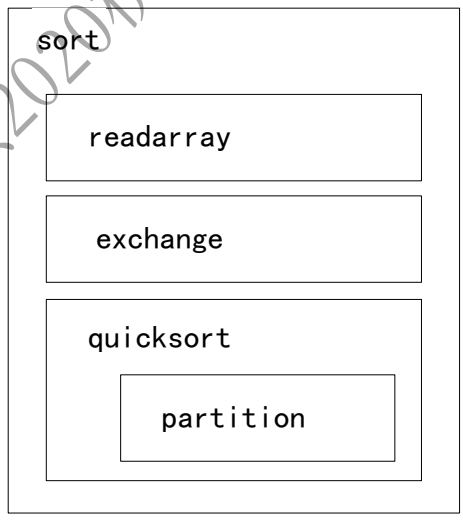


图 6-4 图 6-3 程序中过程的嵌套关系

对于允许嵌套过程的语言，每当遇到一个嵌入的过程说明时，应当暂停包围此过程的外围过程说明语句的处理，而进入嵌套过程说明语句的处理。每个过程（或者函数）都可以定义自己的名字，一个可行的处理方法是为每个过程创建一张单独的符号表，用来存放该过程定义的名字。名字有两类，一类是变量，一类是过程（包括函数）。对于变量名，表项信息

同样是类型、偏移地址等；对于过程名，表项信息是一个指针，指针指向该过程对应的符号表。同时给符号表增加一个表头，表头存放一个指针，指向该符号表对应过程的外围过程的符号表，该指针反映了过程之间的嵌套关系。表头同时存放符号表的总域宽，即该符号表中记录的所有变量的域宽之和，也就是该符号表对应过程中定义的所有变量占用的总的存储空间。图 6-3 程序例子的符号表及符号表之间的关系如图 6-5 所示。

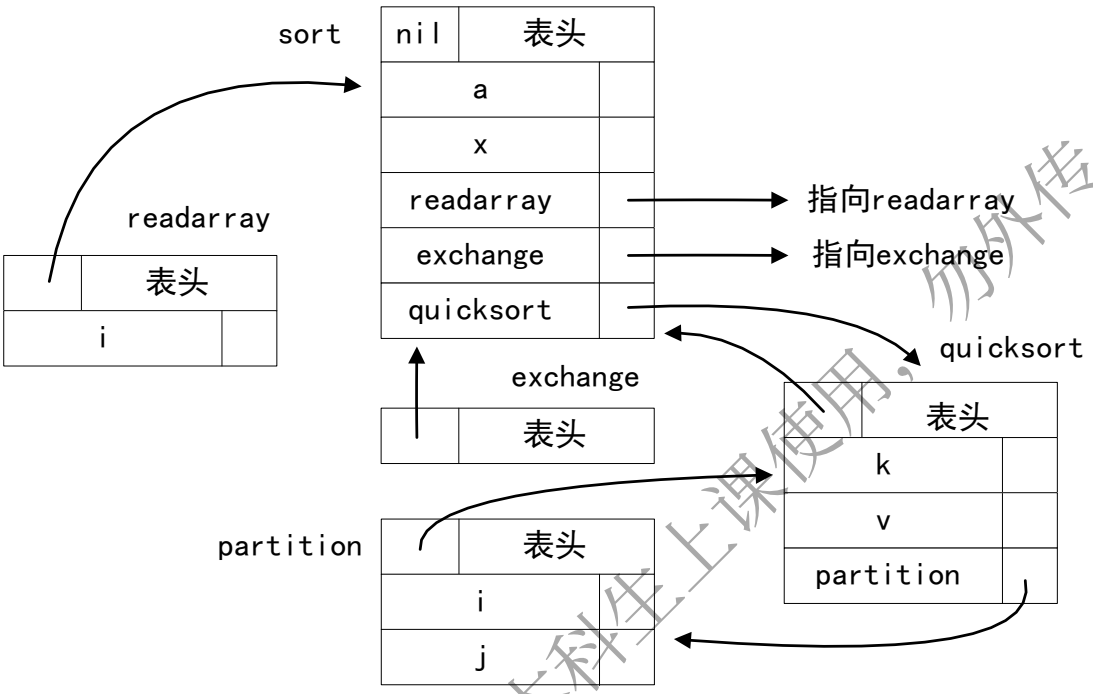


图 6-5 例子程序的符号表

表 6-5 给出了一个翻译允许嵌套过程的说明语句的解决方案。其中包含几个语义动作：

（1）mktable(previous)：创建一张新的符号表，并返回指向新表的一个指针。参数 previous 是一个指向其它符号表的指针，该符号表正好是新创建符号表对应过程的外围过程的符号表。previous 被存入新符号表的表头；

（2）enter(table, name, type, offset)：在符号表指针 table 指向的符号表中，插入一个名字为 name，类型为 type，偏移地址为 offset 的变量表项；

（3）addwidth(table, width)：在 table 指向的符号表的表头填入该符号表所有名字占用的总域宽；

（4）enterproc(table, name, newtable)：在 table 指向的符号表中插入一个名字为 name 的过程表项，newtable 为该过程的符号表的指针。

在表 6-5 给出的语法制导定义中，用到了两个数据结构：

（1）tblptr：是一个栈，用于存放符号表指针，指针指向各外围过程对应的符号表。例如，对于图 6-3 的例子，当在处理过程 partition 中的说明语句时，由于 partiton 嵌套在

quicksort 中，而 quicksort 嵌套在 sort 中，栈 tblptr 从栈顶到栈底分别存放了 partiton、quicksort、sort 的符号表指针；

（2）offset：是一个栈，用于存放变量的偏移地址。在当前过程的说明部分分析结束时，offset 里记录的是该过程所有名字占用的总域宽。在分析过程中，offset 中的值和 tblptr 中的值是一一对应的。例如，当分析图 6-3 例子的 partition 时，栈 offset 从栈顶到栈底分别存放了 partiton、quicksort、sort 的偏移地址。

表 6-5 允许嵌套过程的说明语句的处理

序号	产生式	语义规则
1	P → M D	addwidth (top (tblptr), top (offset)); pop(tblptr); pop (offset)

2	M → ε	t := mktable (nil);
		push(t, tblprt);
		push (0, offset)

3	D → D_1 ; D_2	

4	D → proc id ; N D_1; S	t := top(tblptr);
		addwidth(t, top(offset));
		pop(tblptr);
		pop(offset);
		enterproc(top(tblptr), id.name, t)

5	D → id : T	enter(top(tblptr), id.name, T.type, top(offset)); top(offset) := top(offset) + T.width

6	N → ε	t := mktable(top(tblptr));
		push(t, tblptr);
		push(0, offset)

利用表 6-5 的语法制导定义自底向上翻译句子，第一步归约总是用到产生式“M → ε”。M 是一个标记非终结符号，其产生式“M → ε”对应的语义规则首先为主过程创建一张符号表，符号表的表头指针为 nil（空），因为主过程没有外围过程；然后用压栈操作 push 分别为栈 tblptr 和栈 offset 赋初值。嵌套过程的说明由产生式“D → proc id ; N D_1; S”生成，其中 N 也是一个标记非终结符号。当进入一个嵌套过程的分析时，首先用 N 的产生式“N → ε”进行归约，其对应的语义规则为嵌套过程创建符号表，该符号表的表头指针（当前 tblptr 的栈顶指针）指向其外围过程的符号表，同时将嵌套过程符号表的指针压入栈 tblptr，对应的为栈 offset 压入嵌套过程偏移地址的初值 0。S 代表嵌套过程除说明部分之外的其它语句。

当用产生式“D → id : T”归约时，完成变量 id 说明语句的分析，这时候使用语义动作 enter(top(tblptr), id.name, T.type, top(offset))将 id 的属性信息填入 tblptr 的栈顶指针指向的符号表中，id 的名字为 id.name，类型为 T.type，偏移地址为 offset 栈的栈顶值。

当用产生式“D → proc id ; N D_1; S”归约时，完成一个嵌套过程说明部分的处理。首先使用 addwidth 在当前过程的符号表的表头添加总域宽，然后将栈 tblptr 和栈 offset 的栈顶

弹出，最后使用语义动作 enterproc(top(tblptr), id.name, t)在当前过程的外围过程的符号表中添加当前过程的过程表项。

用产生式“P → M D”归约是分析句子的最后一步，首先在主过程符号表的表头添加总域宽，然后对栈 tblptr 和栈 offset 执行弹栈操作，此时两个栈均为空栈。

【例 6-3】：使用 6-5 中的语法制导定义翻译如下句子（句型）：

$id_1 : T_1$;

proc id_2 ;

$id_3 : T_2$;

S

假设 id_1 的类型为 real，域宽为 8，id_3 的类型为 integer，域宽为 4。

解：句子的分析树如图 6-6 所示。

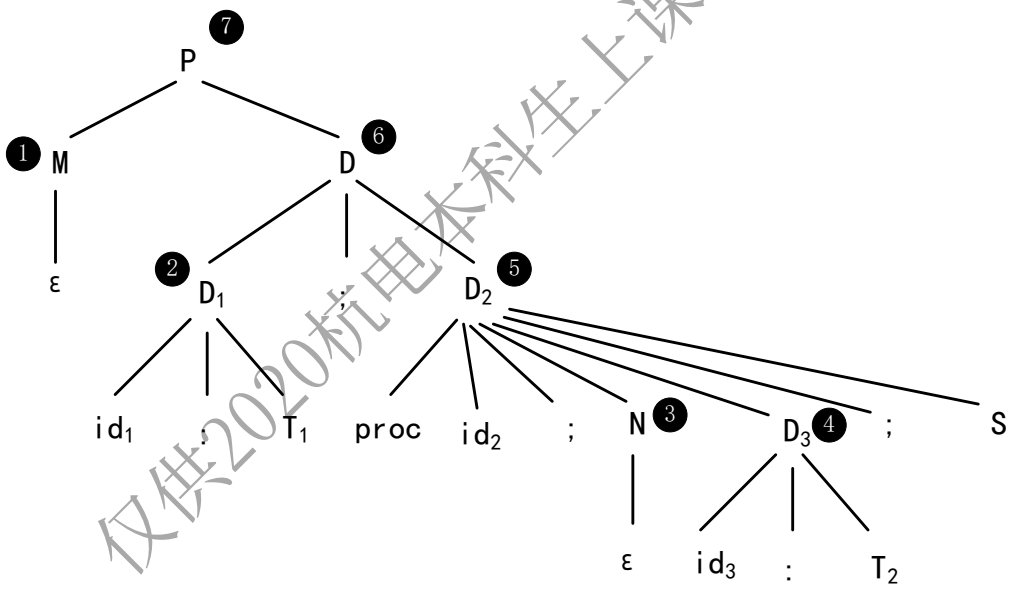


图 6-6 例子 6-3 句子的自底向上翻译

翻译过程如下：

（1）用产生式“M → ε”归约，执行 t_1 := mktable (nil)，为主过程创建符号表；push (t_1, tblptr)，给栈 tblptr 赋初值；push (0, offset) ，给栈 offset 赋初值；

（2）用产生式“D → id : T”归约，执行 enter(t_1, id_1, real, 0)，在主过程符号表中添加变量 id_1 的表项；top(offset) := 0 + 8 := 8，递增主过程的偏移地址；

（3）用产生式“N → ε”归约，执行 t_2 := mktable (t_1)，为嵌套过程创建符号表；push (t_2, tblptr)，将嵌套过程符号表指针 t_2 压入栈 tblptr；push (0, offset)，将嵌套过程偏移地址初值 0 压入栈 offset；

（4）用产生式“D → id : T”归约，执行 enter (t_2, id_3, integer, 0)，在嵌套过程符号表中添加变量 id_3 的表项；top(offset) := 0 + 4 := 4，递增嵌套过程的偏移地址；

（5）用产生式“D → proc id ; N D_1; S”归约，t := top (tblptr) := t_2，取嵌套过程的符号表指针赋给 t；执行 addwidth (t, 4)，在嵌套过程符号表表头添加总域宽；pop (tblptr) 和 pop (offset)执行弹栈操作，退出嵌套过程的分析；执行 enterproc (t_1, id_2, t)，在主过程符号表中为过程 id_2 添加过程表项；

（6）用产生式“D → D_1 ; D_2”归约，无语义动作；

（7）用产生式“P → M D”归约，执行 addwidth (t_1, 8)，在主过程符号表表头添加总域宽；pop (tblptr) 和 pop (offset)执行弹栈操作，最后两个栈都变成空栈。

翻译完成之后，符号表情况如图 6-7 所示。

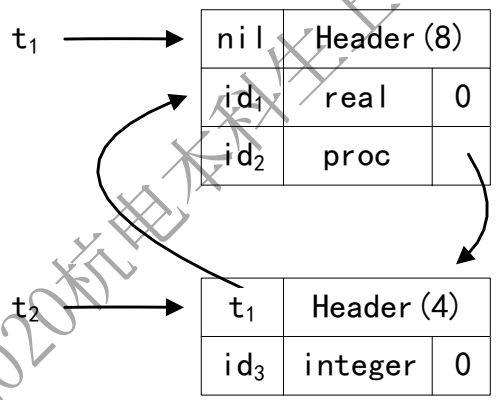


图 6-7 例子 6-3 句子的符号表

6.3 中间语言

为生成高质量的目标代码，大部分编译器先把源程序翻译为中间语言表示。针对中间语言表示，可以执行多种代码优化操作。不同的编译器对中间语言表示的选择和设计各有不同。本节介绍一种常见的中间语言表示——三地址代码。

在三地址代码中，不允许一条指令的右边出现多个运算符，也就是说不允许出现组合的表达式。因此，形如“x+y*z”这样的算术表达式要被翻译成如下的三地址指令序列：

t_1=y*z

t_2=x+t_1

其中 t_1、t_2 是编译器产生的临时变量。因为三地址代码不含多运算符表达式，控制流语句的嵌套结构也被转化为包含转移指令的线性指令序列，所以有利于目标代码的生成和优化。因为引入临时变量来记录程序计算得到的中间结果，所以三地址代码可以方便地进行重组。

最常见的三地址代码形如 x = y op z，其中 x、y、z 是三个地址，op 是运算符。地址可以是如下形式之一：

（1）变量：变量是程序员在源程序中使用说明语句定义的。可以直接采用源程序中的变量名字作为三地址代码中的地址。在实现中，变量名字被替换为指向符号表中该变量表项的指针，关于变量的所有信息均存放在该表项中。

（2）常量：源程序中经常直接把常量作为运算对象加入运算。

（3）临时变量：在每次需要临时变量时产生一个新名字是必要的，这对后续的代码优化也是有利的。

下面介绍几种常用的三地址指令：

（1）x = y op z：赋值指令，其中 op 是一个双目算术运算符或逻辑运算符。

（2）x = op y：赋值指令，其中 op 是一个单目运算符。基本的单目运算符包括单目减、逻辑非和转换运算等。将整数转化成浮点数的运算就是转换运算的一个例子。

（3）x = y：复制指令，把 y 的值赋给 x。

（4）goto L：无条件转移指令，下一步要执行的指令是带有标号 L 的三地址指令。

（5）if x goto L 或 if False x goto L：条件转移指令，分别当 x 为真或为假时，这两个指令的下一步将执行带有标号 L 的指令，否则下一步将执行指令序列中的下一条指令。

（6）if x relop y goto L：条件转移指令，当 x 和 y 之间满足 relop 关系（含<、>、<=、>=、=、≠等）时，下一步将执行带有标号 L 的指令，否则将执行指令序列中的下一条指令。

（7）param 和 call：参数传递和过程调用指令，param x 进行参数传递，call p, n 和 y=call p, n 分别进行过程调用和函数调用，p 为被调用过程（或函数）的名字，n 为参数的个数。常见的用法如下：

```
param x1
param x2
  ...
param xn
call p, n
```

（8）return：过程返回指令，执行 return y 将从被调用过程返回到调用过程，其中 y 表示返回值。

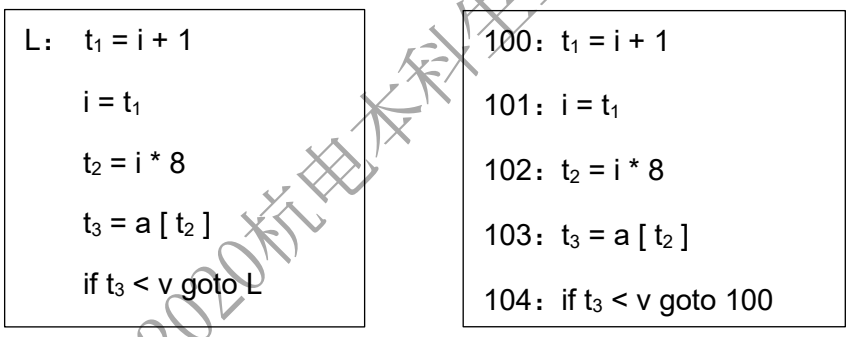
（9）x = y [i] 和 x [i] = y：索引赋值指令。x = y [i]指令将把距离位置 y 处 i 个内存单元的位置中存放的值赋给 x 。指令 x [i] = y 将距离位置 x 处 i 个内存单元的位置中的内容设置为 y 的值。

（10）x = &y、x = *y 或*x = y：地址及指针赋值指令。指令 x=&y 将 x 的值设置为 y 的地址。这个 y 通常是一个变量名字，也可能是一个临时变量，x 是一个指针名字或临时变量。在指令 x=*y 中，y 通常是一个指针，这个指令使得 x 的值等于存储在这个指针指向的位置中的值。*x=y 则把 y 的当前值赋给由指针 x 指向的位置中。

【例 6-4】：考虑如下语句：

do i = i+1; while (a[i] < v);

两种可能的翻译见图 6-8。第一种翻译使用了符号化标号，如第一条指令上的 L。第二种翻译为每条指令附加了一个位置号，在这里选择以 100 作为开始位置。在这两种翻译中，最后一条指令都是目标为第一条指令的条件转移指令。这里假设每个数组元素占 8 个字节，故用 i*8 来计算 a 的下标。



(a) 符号化标号　　(b) 位置标号

图 6-8 例 6-4 语句的两种翻译结果

上面对三地址指令的描述说明了各类指令的组成部分，但是并没有描述这些指令如何实现。在编译器中这些指令可以实现为对象，或者实现为带有运算符字段和运算分量字段的记录。四元式、三元式和间接三元式是三种常见的实现方式。

四元式有四个字段，分别是 op、arg_1、arg_2、result。op 是运算符或者是该运算符的内部编码。例如，对于三地址指令“x = y op z”，其四元式的 op 字段存放“+”、arg_1 字段存放 y、arg_2 字段存放 z、result 字段存放 x。以下是一些特例：

（1）形如“x = uminus y”的单目运算符指令和复制指令“x = y”不使用 arg_2。需要注意的是，对于复制指令“x = y”，op 是“=”，而对其它运算来说赋值运算符是隐含表示的。

（2）像 param 这样的运算既没有 arg_2，也没有 result。

（3）条件指令或无条件转移指令将目标标号放入 result 字段。

【例 6-5】：考虑语句 a=b*-c+b*-c，将它翻译为三地址代码，并表示为四元式。

解：图 6-9 中，(a)是上述语句的三地址代码，(b)是三地址代码的四元式表示形式。

```
t1 = uminus c
t2 = b * t1
t3 = uminus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

(a)三地址代码

	op	arg_1	arg_2	result
0	uminus	c		t_1
1	*	b	t_1	t_2
2	uminus	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 6-9 例 6-5 语句的三地址代码和四元式

其中，uminus 表示单目减运算。为了提高可读性，在代码中直接使用实际的标识符，比如用 a、b、c 作为 arg_1、arg_2 以及 result 字段的内容，而没有使用指向符号表相应表项的指针。临时变量，如 t_1—t_5，可以像程序员定义的变量名字一样被加入到代码中。

三元式只有三个字段，分别是 op、arg_1、arg_2。在四元式中通常是通过引入临时变量来记录中间计算结果，并将临时变量置于 result 字段。而三元式使用运算的位置来表示中间计算结果，因此不需要 result 字段。

图 6-10 中的(a)是【例 6-5】语句的三元式表示。

	op	arg_1	arg_2
0	uminus	c	
1	*	b	(0)
2	uminus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

(a)三元式

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	op	arg_1	arg_2
0	uminus	c	
1	*	b	(0)
2	uminus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

(b)间接三元式

图 6-10 例 6-5 语句的三元式和间接三元式

三元式相对于四元式来说，更节省存储空间，但是在优化代码时，经常需要调整指令的执行顺序，这时候四元式的优势就体现出来了。使用四元式时，如果移动了一个计算临时变量 t 的指令，那些使用 t 的指令不需要做任何改变。而使用三元式时，对于运算结果的引用是

通过指令的位置来实现的，因此如果改变一条指令的位置，则引用该指令的结果的所有指令都要做相应的修改。间接三元式克服了三元式的这个缺点。

间接三元式在三元式的基础上，另外设置了一个包含指向三元式的指针的列表，称为间接码表，间接码表中指针的顺序代表三元式的执行顺序。实践中可以使用数组 instruction 来实现间接码表，图 6-10(b)就是【例 6-5】语句的间接三元式表示。如果使用间接三元式，代码优化时可以通过对 instruction 中元素的重新排序来调整指令的执行顺序，但不会影响三元式本身。

6.4 赋值语句的翻译

赋值语句是高级程序设计语言的基本语句，本节讨论赋值语句的翻译方法。

简单赋值语句的翻译

赋值语句一般由左部终结符号、赋值运算符、右部表达式构成，形如“id := E”。表 6-6 是翻译简单赋值语句的语法制导定义，其中用到了如下几个语义动作：

（1）lookup：参数为 id.name，检查名为 name 的 id 是否在符号表里面已存在，如果存在返回 id 的偏移地址，如果不存在返回 nil。如果返回的是 nil，说明 id 是未定义就被使用，编译器要报错；

（2）emit：生成一条三地址代码并输出，翻译过程中把 emit 输出的三地址代码按输出的顺序排列，就构成翻译结果；

（3）newtemp：返回一个临时变量。当需要临时变量时，调用 newtemp，第一次调用返回 t_1，第二次调用返回 t_2，以此类推。

E 代表表达式，E.place 表示存放 E 的值的地址，方便起见可以用名字本身代替。为区分减运算和负运算，把负运算记为“uminus”。显然表 6-6 给出的语法制导定义是 S-属性定义，可以按自底向上的顺序翻译句子。

表 6-6 翻译简单赋值语句的语法制导定义

序号	产生式	语义规则

1	S → id := E	p := lookup(id.name);
		if p ≠ nil then
		　　emit (p ‘:=’ E.place)
		else error

2	E → E_1 + E_2	E.place := newtemp; emit (E.place ‘:=’ E_1.place ‘+’ E_2.place)
3	E → E_1 * E_2	E.place := newtemp; emit (E.place ‘:=’ E_1.place ‘*’ E_2.place)
4	E → - E_1	E.place := newtemp; emit (E.place ‘:=’ ‘uminus’ E_1.place)
5	E → (E_1)	E.place := E_1.place

6	E → id	p := lookup(id.name); if p ≠ nil then E.place := p else error

表 6-6 中的文法是二义的，但是可以通过确定运算符的结合性及规定运算符的优先级，避免二义性的发生，对任何一个合法的句子确定唯一的一棵分析树。

【例 6-6】：分析句子　$id_1 := id_2 * (- id_3)$。

句子的分析树如图 6-11 所示，自底向上构建该分析树，需 6 步最左归约，归约的同时执行产生式对应的语义规则，可完成翻译工作。

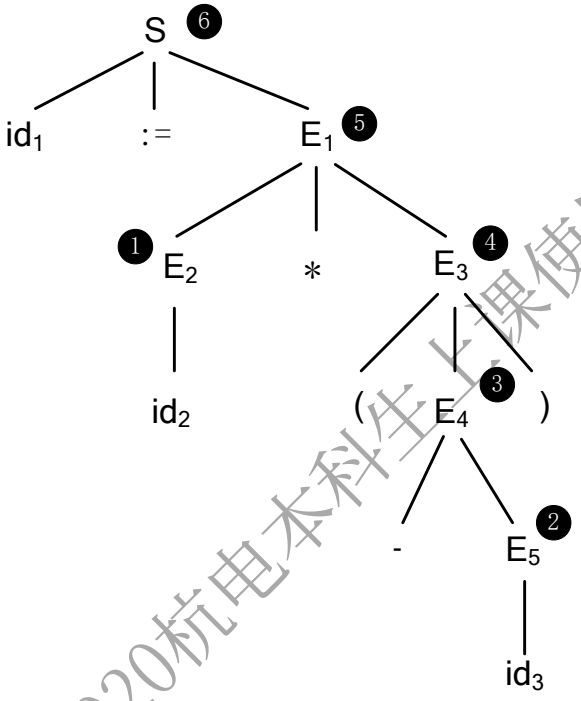


图 6-11 例 6-6 语句的自底向上翻译

翻译过程如下：

（1）用产生式“E → id”归约，首先调用 lookup，检查符号表中有没有 id_2 的表项，这里假设 id_2 在说明部分已定义，并且已登记在符号表里。获取 id_2 的偏移地址后赋给 p，这里直接用 id_2 代表偏移地址。然后计算 $E_2.place := id_2$；

（2）用产生式“E → id”归约，调用 lookup 检查符号表中有没有 id_3 的表项，这里假设 id_3 在说明部分也已定义，并且已登记在符号表里。获取 id_3 的偏移地址后赋给 p，然后计算 $E_5.place := id_3$；

（3）用产生式“E → - E_1”归约，调用 newtemp，返回临时变量 t_1 并赋给 $E_4.place$，然后调用 emit，输出一条三地址代码“t_1 := uminus id_3”；

（4）用产生式“E → (E_1)”归约，计算 $E_3.place := E_4.place := t_1$；

（5）用产生式“E → E_1 * E_2”归约，调用 newtemp，返回临时变量 t_2 并赋给 E_1.place，然后调用 emit，输出一条三地址代码“t_2 := id_2 * t_1”；

（6）用产生式“S → id := E”归约，首先调用 lookup，检查符号表中有没有 id_1 的表项，获取 id_1 的偏移地址后赋给 p，然后调用 emit，输出一条三地址代码“id_1 := t_2”。

在翻译过程中，3 次调用 emit，输出了 3 条三地址代码：

t_1 := uminus id_3

t_2 := id_2 * t_1

id_1 := t_2

这也是最终的翻译结果。

对数组的引用

在赋值语句及其它语句中经常出现对数组的引用。如何翻译包含数组引用的语句呢？

数组元素通常存储在一个连续的存储区中，在 C 和 Java 中一个具有 n 个元素的一维数组的元素是按照 0,1,...,n-1 编号的，也可以在声明时指定上下界（low、high）。一维数组的存储方式见图 6-12。

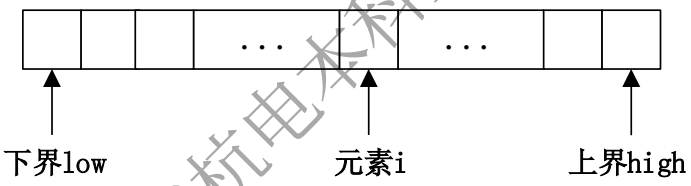


图 6-12 一维数组示例

下界的地址称为基址（base），元素的个数为：high-low+1，假设数组元素的域宽为 w（存储一个数组元素需要 w 个字节），则数组元素 A[i]的地址可由（6.1）计算。

$$base + (i-low)\times w = i\times w + (base - low\times w) \qquad (6.1)$$

其中子表达式 c = base - low×w 的值可以在编译时刻计算，通常将 c 的值存放在数组名字 A 的符号表表项里，这样 A[i]的地址也可由 i×w + c 计算。

图 6-13(a)是一个二维数组的结构，数组名字为 A，由 n_1 行、n_2 列组成。一般地，令二维数组第一维的上下界为：low_1、$high_1$，第二维的上下界为：low_2、$high_2$，则第一维的长度 $n_1=high_1-low_1+1$、第二维的长度 $n_2=high_2-low_2+1$。对于二维数组，不同语言采用的存储方式不同，其中 Pascal、C 语言是按行存储，Fortran 语言是按列存储。对于一个 2×3 的二维数组 A，按行存储和按列存储的示意图分别见 6-13(b)、6-13(c)。

类似地，令二维数组的基址为 base，元素域宽为 w，存储方式为按行存储，则数组元素 $A[i_1, i_2]$的位置可由（6.2）计算：

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

$$= (i_1 \times n_2+i_2) \times w + (base - (low_1\times n_2 + low_2) \times w) \qquad (6.2)$$

同样，子表达式 base - (low1×n2 + low2) ×w 可在编译时刻计算并被存放在符号表中。

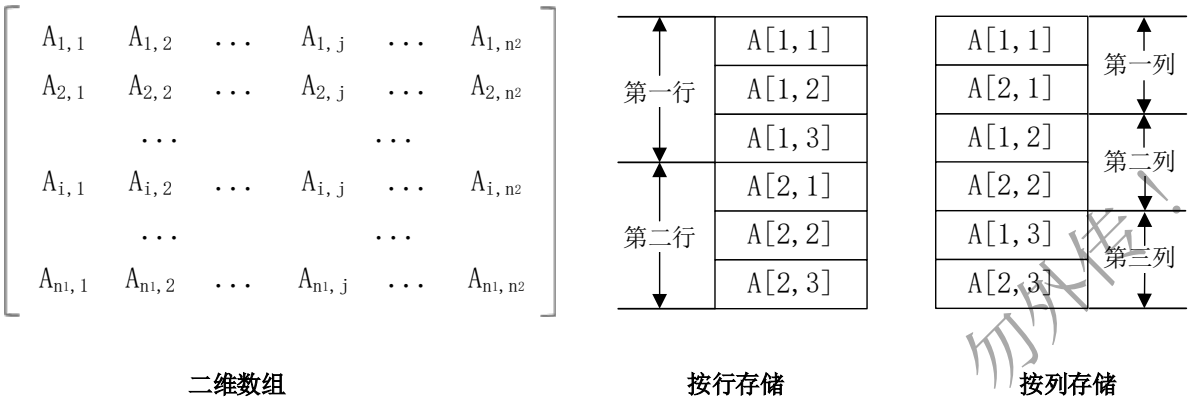


图 6-13 二维数组及其存储方式

可将以上讨论推广到 k 维数组。令 k 维数组每维的下界为：low1、low2、...、lowk，每维的长度为：n1、n2、...、nk。若采用按行存储，则数组元素 A[i1,i2,...,ik]的位置可由（6.3）计算。

$$((\ldots((i_1\times n_2+i_2)\times n_3+i_3)\ldots)\times n_k+i_k)\times w$$

$$+ (base - ((\ldots((low_1\times n_2+low_2)\times n_3+low_3)\ldots)\times n_k+low_k)\times w) \qquad (6.3)$$

同理（6.3）中第二行的子表达式可在编译时刻计算并被存放在符号表中。

下面讨论如何翻译包含数组引用的赋值语句。假设含数组引用的结构由如下产生式生成。

L→id | id [Elist]

Elist→Elist, E | E

非终结符号 L 的第一个候选式中的 id 是简单变量，第二个候选式中的 id 是数组名字，Elist 是数组的下标列表。非终结符号 Elist 的产生式生成下标列表，为了将下标列表 Elist 与数组名字联系起来，我们将产生式改写为：

L→id | Elist]

Elist→Elist, E | id [E

这样对于每个 Elist 可以引进一个综合属性 array，用来记录指向数组 id 在符号表中表项的指针。另外设置属性 Elist.ndim 来记录 Elist 中的下标表达式的个数，即数组的维度；函数 limit(array,j)返回 nj，即由 array 指向的数组的第 j 维的长度；属性 Elist.place 用来存放 Elist

中的下标表达式计算出来的值。L 有两个属性 L.place 和 L.offset。如果 L.offset 为 null，则说明 L 是一个简单变量引用，否则说明 L 是一个数组变量引用，这时候 L.offset 记录的是数组元素的偏移地址。

翻译含数组元素引用的赋值语句的语法制导定义如表 6-7 所示。

表 6-7 翻译简单赋值语句的语法制导定义？

序号	产生式	语义规则
1	S→L:=E	if L.offset=null then emit(L.place ':=' E.place); else emit(L.place '[' L.offset ']' ':=' E.place)
2	E→E_1+E_2	E.place:=newtemp; emit(E.place ':=' E_1.place '+' E_2.place)
3	E→(E_1)	E.place:=E_1.place
4	E→L	if L.offset = null then E.place:=L.place else begin E.place:=newtemp; emit(E.place ':=' L.place '[' L.offset ']') end
5	L→id	L.place:=id.place; L.offset:=null
6	L→Elist]	L.place:=newtemp; emit(L.place ':=' Elist.array '-' ((…((low_1×n_2+low_2)×n_3+low_3)…)×n_k+low_k)×w); L.offset:=newtemp; emit(L.offset ':=' w '*' Elist.place)

7	Elist→$Elist_1$, E	t:=newtemp;
		m:=$Elist_1$.ndim+1;
		emit(t ':=' $Elist_1$.place '*' limit($Elist_1$.array,m));
		emit(t ':=' t '+' E.place);
		Elist.array:=$Elist_1$.array;
		Elist.place:=t;
		Elist.ndim:=m

8	Elist→id[E	Elist.place:=E.place; Elist.ndim:=1; Elist.array:=id.place

所有语义规则中的的属性均为综合属性，因此表 6.7 为 S-属性定义，适合自底向上的属性计算。

【例 6-7】：设 A 为一个 10×20 的数组，即 n_1=10，n_2=20；并设域宽 w=4；数组的第一个元素为 A[1,1]，则有 low_1=1，low_2=1，所以(low_1×n_2+low_2)×w=(1×20+1)×4=84。要求将赋值语句 x:=A[y,z]翻译为三地址代码。

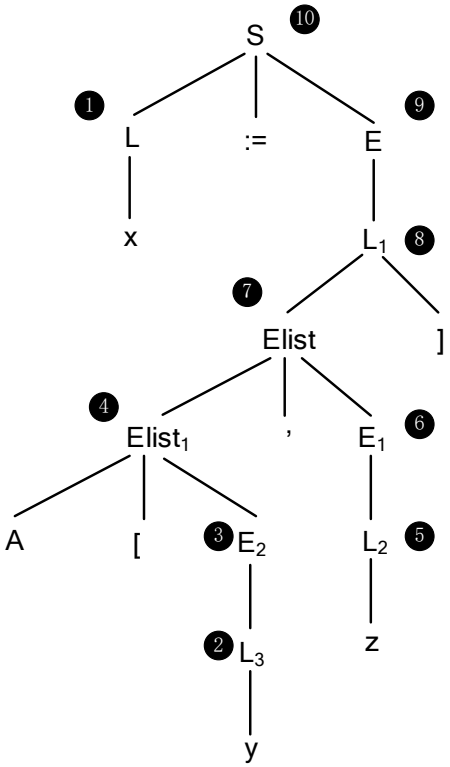


图 6-14 x:=A[y,z]的语法分析树

解：对句子 x:=A[y,z]做自底向上语法分析，构造最左归约序列，同时构造语法分析树（如图 6-14 所示）。在每步归约的同时执行产生式对应的语义规则，完成句子的翻译。翻译过程如下：

（1）用产生式 L→id 归约。执行语义规则：L.place := x；L.offset := null；

（2）用产生式 L→id 归约。执行语义规则：L_3.place := y；L_3.offset := null；

（3）用产生式 E→L 归约。执行语义规则：E_2.place := L_3.place := y；

（4）用产生式 Elist→id[E 归约。执行语义规则：

$Elist_1$.place := E_2.place := y；

$Elist_1$.ndim := 1；

$Elist_1$.array := A；

（5）用产生式 L→id 归约。执行语义规则：L_2.place := z；L_2.offset := null；

（6）用产生式 E→L 归约。执行语义规则：E_1.place := L_2.place := z；

（7）用产生式 Elist→$Elist_1$, E 归约。执行语义规则：

t := t_1；

m := $Elist_1$.ndim + 1 := 2；

emit(t_1 := y * 20)；// limit(A, 2) =20，即数组 A 的第二维长度为 20。

emit(t_1 := t_1 + z)；

Elist.array := Elist$_1$.array := A；

Elist.place := t_1；

Elist.ndim := 2；

（8）用产生式 L→Elist]归约。执行语义规则：

L_1.place := t_2；

emit(t_2 := A - 84)；//($low_1 \times n_2 + low_2) \times w = (1 \times 20 + 1) \times 4 = 84$ 已计算。

L_1.offset := t_3；

emit(t_3 := 4 * t_1)；

（9）用产生式 E→L 归约。执行语义规则：E.place := t_4；emit(t_4 := t_2[t_3])；

（10）用产生式 S→L:=E 归约。执行语义规则：emit(x := t_4)。

最后输出的三地址代码序列为：

t_1:=y*20

t_1:=t_1+z

t_2:=A-84

t_3:=4*t_1

t_4:=t_2[t_3]

x:=t_4

6.5 布尔表达式和控制流语句的翻译

布尔表达式是程序设计语言中常见的语言结构，它有两个基本作用，一个是用来计算逻辑值，一个是用作控制流语句中的条件表达式。控制流语句包括条件语句和循环语句，本节重点讨论“if-then”语句、“if-then-else”语句和“while-do”语句。

布尔表达式的结构与一般的算术表达式类似，只不过运算符和运算对象不同。布尔表达式的运算符包括 or、and、not 等，运算对象是布尔常量、布尔变量或关系表达式。布尔常量有两个：true 和 false。关系表达式形如 E_1 relop E_2，其中 E_1 和 E_2 是算术表达式，relop 是关系运算符，包括：=、<、>、<=、>=、≠ 等。关系表达式成立其值为 true，不成立其值为 false。

本节考察的布尔表达式由如下文法生成：

E → E or E | E and E | not E | (E) | id relop id | true | false （6.4）

逻辑运算的优先级关系和结合性规则如下：or 运算优先级低于 and 运算，and 运算优先级低于 not 运算，or 和 and 是左结合的，not 是右结合的。规定了以上优先级关系和结合性，可为文法（6.4）的任一合法句子确定唯一的一棵分析树。

翻译布尔表达式需要考虑它在源程序中的作用。如果是为了计算逻辑值，则可以采用类似翻译算术表达式的方法来翻译布尔表达式。对于逻辑值的表示，可以直接用布尔常量 true 和 false，也可以把逻辑值数值化，如用 1 表示 true，用 0 表示 false。

例如，对于布尔表达式：a or b and not c，根据运算优先级和结合性，可翻译为如下三地址代码序列：

t_1 := not c

t_2 := b and t_1

t_3 := a or t_2

又如，对于形如 a<b 的关系表达式，可等价地看作：

If a<b then 1 else 0

可进一步翻译为如下三地址代码序列（假设代码序号从 100 开始）：

100：if a<b goto 103

101：t:=0

102：goto 104

103：t:=1

104：...

数值表示法翻译布尔表达式的语法制导定义如表 6-8 所示。

表 6-8 翻译布尔表达式的语法制导定义

序号	产生式	语义规则
1	E → E_1 or E_2	E.place := newtemp; emit (E.place ':=' E_1.place 'or' E_2.place)
2	E → E_1 and E_2	E.place := newtemp; emit (E.place ':=' E_1.place 'and' E_2.place)
3	E → not E_1	E.place := newtemp; emit (E.place ':=' 'not' E_1.place)

4	E → (E_1)	E.place := E_1.place
5	E → id_1 relop id_2	E.place := newtemp; emit ('if' id_1.place relop.op id_2.place 'goto' nextstat + 3); emit (E.place ':=' '0'); emit ('goto' nextstat + 2); emit (E.place ':=' '1')
6	E → true	E.place := newtemp; emit (E.place ':=' '1')
7	E → false	E.place := newtemp; emit (E.place ':=' '0')

类似表 6-6 给出的翻译算术表达式的语法制导定义，这里 E.place 属性表示存放布尔表达式 E 的值的地址，可通过调用 newtemp 返回一个临时变量来代替。语义动作 emit 产生并输出三地址语句。nextstat 是一个全局变量，是输出序列中下一条三地址代码的序号， emit 每执行一次，nextstat 自动加 1。

【例 6-8】：翻译如下布尔表达式：

a<b or c<d and e<f

解：基于表 6-8 中的文法，构建最左归约序列，自底向上分析该布尔表达式，同时可构建如图 6-15 所示的分析树。在构建分析树的同时，执行语义规则翻译句子。

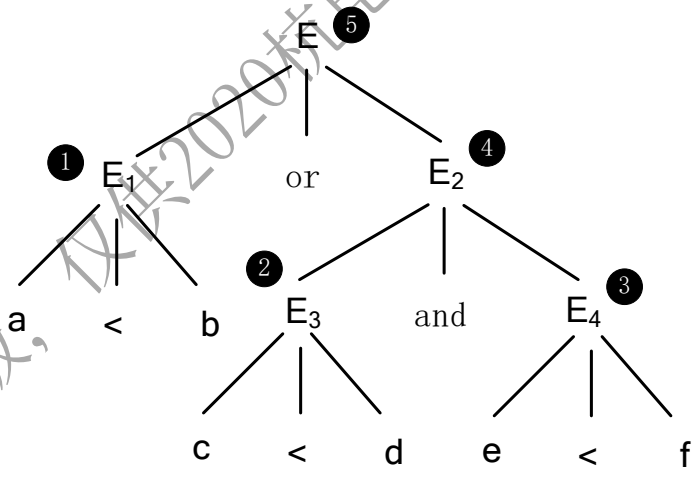


图 6-15 例 6-8 语句的翻译

假设在翻译该布尔表达式之前，已经输出了 99 条三地址代码，当前 nextstat 等于 100。翻译过程如下：

（1）用产生式 "E → id_1 relop id_2" 归约，调用 newtemp，返回临时变量 t_1 并赋给 E_1.place，然后调用 emit 连续输出 4 条三地址代码：

100　if a<b goto 103

101　t_1 := 0

102　goto 104

103　t_1 := 1

（2）用产生式“E → id_1 relop id_2”归约，调用 newtemp，返回临时变量 t_2 并赋给 E_3.place，然后调用 emit 连续输出 4 条三地址代码：

104　if c<d goto 107

105　t_2 := 0

106　goto 108

107　t_2 := 1

（3）用产生式“E → id_1 relop id_2”归约，调用 newtemp，返回临时变量 t_3 并赋给 E_4.place，然后调用 emit 连续输出 4 条三地址代码：

108　if e<f　goto 111

109　t_3 := 0

110　goto 112

111　t_3 := 1

（4）用产生式“E → E_1 and E_2”归约，调用 newtemp，返回临时变量 t_4 并赋给 E_2.place，然后调用 emit 输出 1 条三地址代码：

112　t_4 := t_2 and t_3

（5）用产生式“E → E_1 or E_2”归约，调用 newtemp，返回临时变量 t_5 并赋给 E.place，然后调用 emit 输出 1 条三地址代码：

113　t_5 := t_1 or t_4

以上翻译过程输出 100—113 共 14 条三地址代码，这就是布尔表达式 a<b or c<d and e<f 的翻译结果。只要各变量的值确定，执行这个代码序列，将计算出该布尔表达式的逻辑值，最终结果存放在临时变量 t_5 中。

如果布尔表达式是作为控制流语句中的条件表达式，只计算出它的值为 true 还是为 false 是不够的，我们实际需要计算的是布尔表达式为 true 时控制流程跳转到哪里，为 false 时又跳转到哪里。

为翻译控制流语句中的布尔表达式，需要先考察控制流语句的控制流程和代码结构。本节重点讨论条件语句和循环语句两类控制流语句，生成控制流语句的文法如下：

$$
\begin{aligned}
S \rightarrow\ & \text{if } E \text{ then } S_1 \\
& \mid \text{if } E \text{ then } S_1 \text{ else } S_2 \\
& \mid \text{while } E \text{ do } S_1
\end{aligned}
\qquad (6.5)
$$

其中 E 表示待翻译的布尔表达式，为 E 引入两个属性：E.true 和 E.false，分别称为 E 的真出口（E 为 true 时要跳转到的位置）和 E 的假出口（E 为 false 时要跳转到的位置）。E 的另一个属性是 E.code，表示 E 翻译后得到的三地址代码的序列。S 是语句（不仅指控制流语句），S.code 属性表示语句 S 翻译后得到的三地址代码的序列，S.begin 属性是三地址代码的标号，标记 S 翻译后第一条三地址代码的位置，S.next 属性是三地址代码的标号，指向紧跟在 S 的三地址代码序列后面的第一条三地址代码的位置。在翻译过程中引入一个语义动作 newlabel，当需要时调用它返回一个新的符号标号，符号标号用于标记三地址代码的位置。

文法（6.5）生成的 3 种控制流语句的代码结构和控制流程分别见图 6-16(a)、6-16(b)、6-16(c)。

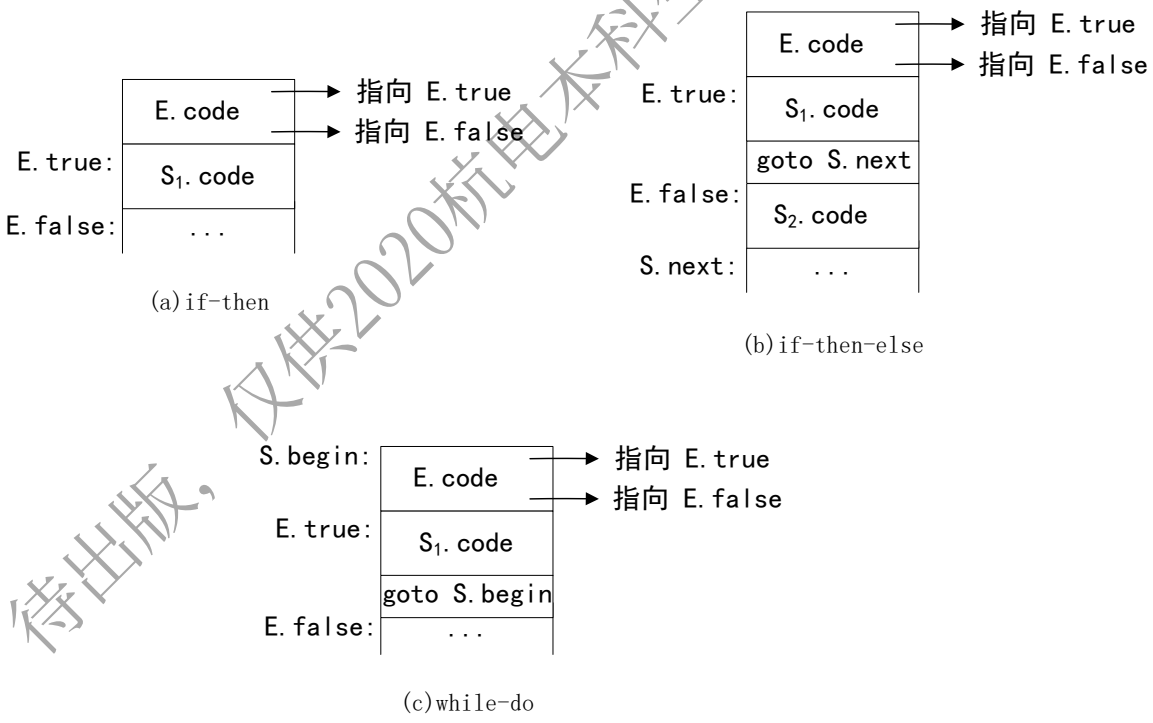


图 6-16 3 种常见控制流语句的代码结构和控制流程

分析控制流语句的语法制导定义见表 6-9。

表 6-9 翻译控制流语句的语法制导定义

序号	产生式	语义规则
1	S → if E then S_1	E.true := newlabel; E.false := S.next; S_1.next := S.next; S.code := E.code || gen(E.true, ':') || S_1.code
2	S → if E then S_1 else S_2	E.true := newlabel; E.false := newlabel; S_1.next := S.next; S_2.next := S.next; S.code := E.code || gen(E.true, ':') || S_1.code ||gen('goto', S.next) || gen(E.false, ':') || S_2.code
3	S → while E do S_1	S.begin:= newlabel; E.true := newlabel; E.false := S.next; S_1.next := S.begin; S.code := gen(S.begin, ':') || E.code || gen(E.true, ':') || S_1.code || gen('goto', S.begin)

第 1 条产生式“S → if E then S_1”生成“if-then”语句，翻译时首先调用 newlabel 产生一个新的符号标号作为 E.true。由于 E 为真时跳转执行 S_1，故 S_1 的第一条三地址代码标记为 E.true，E 为假时跳转到 S.code 的下一条代码（S.next），这个位置同时也是 S_1.next，需标记为 E.false。S 的三地址代码 S.code 由 E 的三地址代码 E.code 并上 S_1 的三地址代码 S_1.code 构成。语义动作 gen 用来输出符号串，语义规则中的符号“||”表示符号串的并运算。

第 2 条产生式“S → if E then S_1 else S_2”生成“if-then-else”语句，翻译时首先调用 newlabel 为 E.true 和 E.false 产生新的标号。由于 E 为真时跳转执行 S_1，故 S_1 的第一条三地址代码标记为 E.true，E 为假时跳转执行 S_2，故 S_2 的第一条三地址代码标记为 E.false。执行完 S_1 后，不能继续执行 S_2，故在 S_1.code 的后面需跟上一条无条件跳转指令跳过 S_2.code，去执行 S.next 指向的代码。从另一个角度讲，S_1 执行完之后 S 也就执行完了，同样 S_2 执行完之后 S 也执行完了，故 S_1 的下一条代码（S_1.next）和 S_2 的下一条代码（S_2.next）就是 S 的下一条代码（S.next）。S 的三地址代码 S.code 由 E 的三地址代码 E.code 并上 S_1 的三地址代码 S_1.code，再并上跳转语句“goto S.next”，最后并上 S_2 的三地址代码 S_2.code 构成。

第 3 条产生式“S → while E do S_1”生成“while-do”语句，翻译时首先调用 newlabel 为 S.begin 和 E.true 产生新的标号。由于 E 为真时跳转执行 S_1，故 S_1 的第一条三地址代码标记为 E.true，E 为假时跳转到 S.code 的下一条代码（S.next），这个位置需标记为 E.false。执行完 S_1.code 后，需要再次执行 E 的三地址代码判断下一步的跳转方向，故在 S_1.code 的后面需跟上一条无条件跳转指令跳转到 S.begin。换个角度来说，S_1 的下一条代码

（S_1.next）就是 S.begin 标记的这条代码。S 的三地址代码 S.code 由 E 的三地址代码 E.code 并上 S_1 的三地址代码 S_1.code，再并上跳转语句“goto S.begin”构成。

根据属性计算方法，考察表 6-9 中语法制导定义各属性的性质，可知 E.true、E.false、S.next 和 S.begin？均为继承属性，E.code 和 S.code 为综合属性。其中继承属性的计算均满足 L-属性定义的要求，故表 6-9 中语法制导定义是 L-属性定义。为翻译句子，可将该语法制导定义改写为一个适合以深度优先顺序计算属性的翻译模式，如图 6-17 所示。

```
S → if {E.true := newlabel; E.false := S.next;}
        E
      then {S1.next := S.next;}
      S1 {S.code := E.code || gen(E.true, ':') || S1.code}
S → if {E.true := newlabel; E.false := newlabel;}
      E
      then {S1.next := S.next;}
      S1
      else {S2.next := S.next;}
      S2 {S.code := E.code || gen(E.true, ':')
          || S1.code || gen('goto', S.next)
          || gen(E.false, ':') || S2.code}
S → while {S.begin:= newlabel; E.true := newlabel;
            E.false := S.next;}
      E
      do {S1.next := S.begin;}
      S1 {S.code := gen(S.begin, ':') || E.code
                || gen(E.true, ':') || S1.code || gen('goto', S.begin)}
```

图 6-17 控制流语句的翻译模式

下面讨论控制流语句中布尔表达式的翻译。控制流语句中的布尔表达式 E 可翻译为一系列的条件转移和无条件转移三地址代码语句，这些转移语句的目标要么是其真出口（E.true），要么是其假出口（E.false）。首先分析一下 or、and 、not 的运算规则。

（1）对于布尔表达式“E = E_1 or E_2”，若 E_1 为 true，则整个 E 为 true，若 E_1 为 false，则还需考察 E_2；E_1 为 false 情况下若 E_2 为 true，则 E 还是 true，若 E_2 为 false，E 才为 false。E 的代码结构及 E 与 E_1 、 E_2 真假出口之间的关系见图 6-18(a)。

（2）对于布尔表达式“E = E_1 and E_2”，若 E_1 为 true，则还需考察 E_2，若 E_1 为 false，则整个 E 为 false；E_1 为 true 情况下若 E_2 同时为 true，则整个 E 为 true，若 E_2 为 false，E 也为 false。E 的代码结构及 E 与 E_1 、 E_2 真假出口之间的关系见图 6-18(b)。

（3）对于布尔表达式“E = not E_1”，若 E_1 为 true，则 E 为 false；若 E_1 为 false，则 E 为 true。E 的代码与 E_1 的代码相同，只不过它们的真假出口需要互换，见图 6-18(c)。

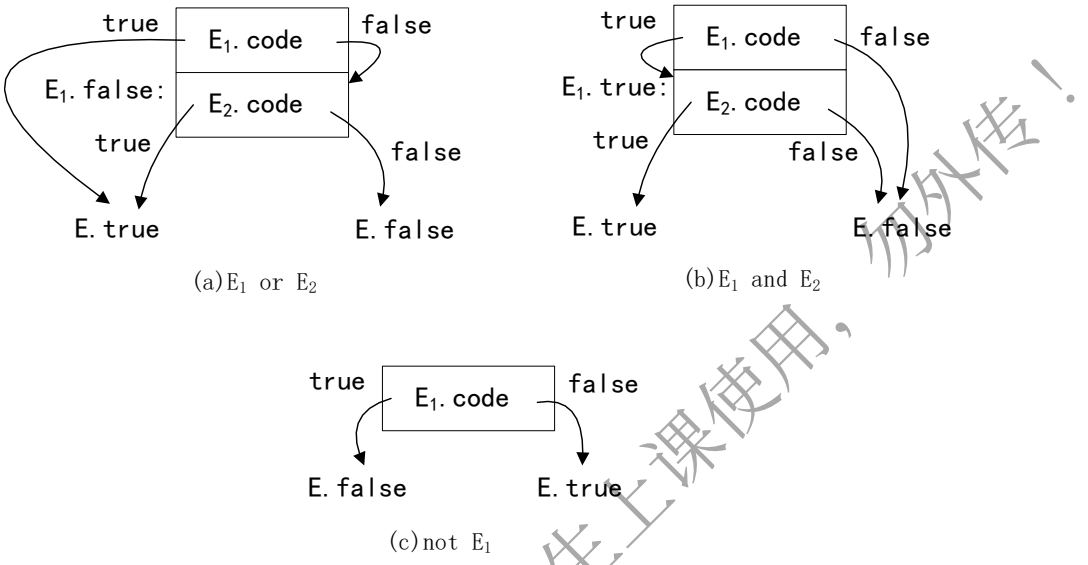


图 6-18 含 or、and 、not 运算的布尔表达式分析

翻译控制流语句中布尔表达式的语法制导定义如表 6-10 所示。

表 6-10 翻译布尔表达式的语法制导定义

序号	产生式	语义规则
1	E → E_1 or E_2	E_1.true:=E.true; E_1.false:=newlable; E_2.true:=E.true; E_2.false:=E.false; E.code:=E_1.code || gen(E_1.false ‘:’) || E_2.code
2	E → E_1 and E_2	E_1.true:=newlable; E_1.false:=E.false; E_2.true:=E.true; E_2.false:=E.false; E.code:=E_1.code || gen(E_1.true ‘:’) || E_2.code
3	E → not E_1	E_1.true:=E.false; E_1.false:=E.true E.code:=E_1.code
4	E → (E_1)	E_1.true:=E.true; E_1.false:=E.false; E.code:=E_1.code
5	E → id_1 relop id_2	E.code:=gen(‘if’ id_1.place relop.op id_2.place ‘goto’ E.true) || gen(‘goto’ E.false)
6	E → true	E.code:=gen(‘goto’ E.true)

7	E → false	E.code:=gen(‘goto’ E.false)

表 6-10 中的语法制导定义也是 L-属性定义，可转换为一个适合深度优先计算属性值的翻译模式，见图 6-19。

E → {E_1.true:=E.true; E_1.false:=newlable;}
E_1
or {E_2.true:=E.true; E_2.false:=E.false;}
E_2 {E.code:=E_1.code || gen(E_1.false ‘:’) || E_2.code}

E → {E_1.true:=newlable; E_1.false:=E.false;}
E_1
and {E_2.true:=E.true; E_2.false:=E.false;}
E_2 {E.code:=E_1.code || gen(E_1.true ‘:’) || E_2.code}

E → not {E_1.true:=E.false; E_1.false:=E.true;}
E_1 {E.code:=E_1.code}

E → ({E_1.true:=E.true; E_1.false:=E.false;}
E_1
) {E.code:=E_1.code}

E → id_1 relop id_2 {E.code:=gen(‘if’ id_1.place relop.op id_2.place ‘goto’ E.true) || gen(‘goto’ E.false)}

E → true {E.code:=gen(‘goto’ E.true)}

E → false {E.code:=gen(‘goto’ E.false)}

图 6-19 控制流语句中布尔表达式的翻译模式

【例 6-9】：用图 6-19 中的翻译模式翻译如下布尔表达式：

a<b or c<d and e<f

解：首先对句子作语法分析，将{、}括起来的语义规则作为终结符号加入到语法分析中，并构建带语义结点的分析树，见图 6-20。

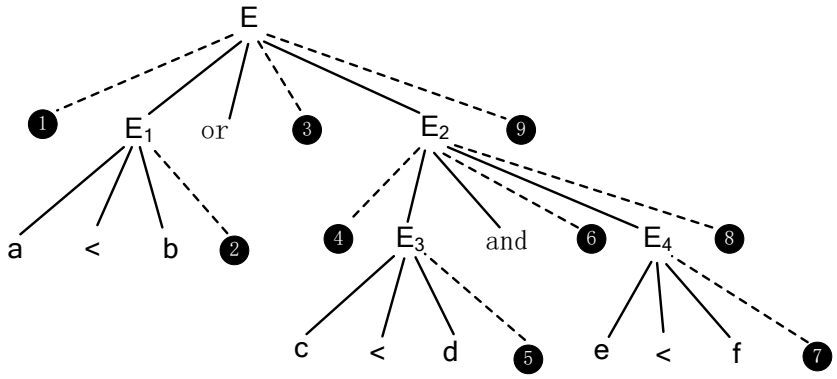


图 6-20 例 6-9 中句子带语义结点的分析树

分析树中共有 9 个语义结点，按深度优先的顺序对语义结点进行排序，并按这个顺序执行语义结点中的语义规则（假设 E 的真、假出口分别为 Ltrue 和 Lfalse）：

（1）E_1.true := E.true := Ltrue；newlable 第一次调用返回 L_1，故 E_1.false := L_1。

（2）E_1.code := { if a<b goto Ltrue

goto L_1 }

（3）E_2.true := E.true := Ltrue；E_2.false := E.false := Lfalse。

（4）newlable 第二次调用返回 L_2，故 E_3.true := L_2；E_3.false := E_2.false := Lfalse。

（5）E_3.code := { if c<d goto L_2

goto Lfalse }

（6）E_4.true := E_2.true := Ltrue；E_4.false := E_2.false := Lfalse。

（7）E_4.code := { if e<f goto Ltrue

goto Lfalse }

（8）E_2.code := { (E_3.code)

L_2:(E_4.code) }

（9）E.code := { (E_1.code)

L_1:(E_2.code) }

将 E.code 展开得如下翻译结果：

if a<b goto Ltrue

goto L_1

L_1: if c<d goto L_2

goto Lfalse

L_2: if e<f goto Ltrue

goto Lfalse

【例 6-10】：用图 6-17 和图 6-19 中的翻译模式翻译如下控制流语句：

while a<b do

if c<d then x:=y+z

else x:=y-z

解：对句子作语法分析，得到带语义结点的分析树，见图 6-21。

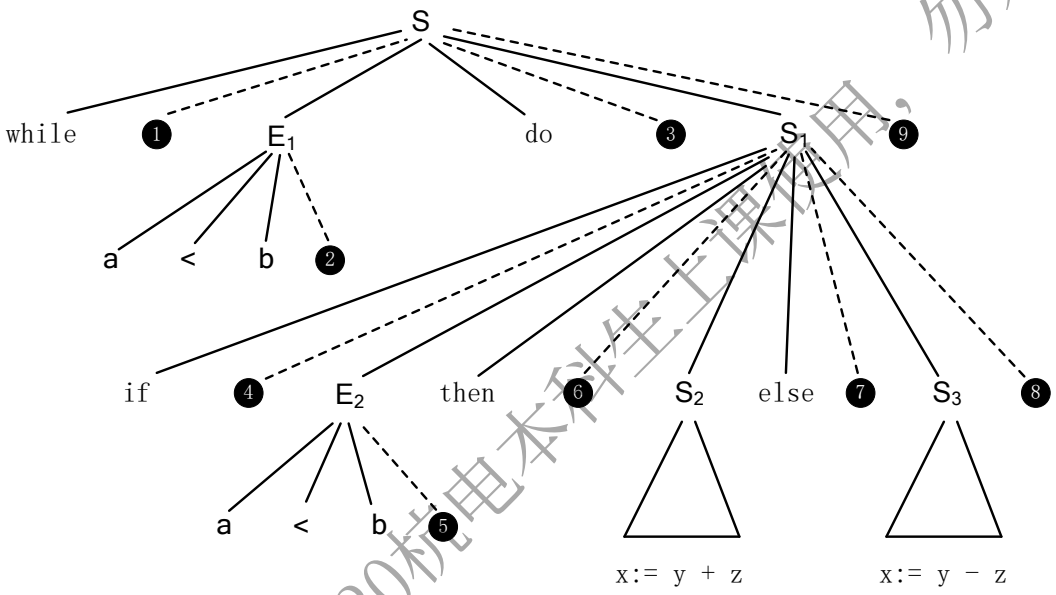


图 6-21 例 6-10 中句子带语义结点的分析树

这里只关注控制流语句及作为条件表达式的布尔表达式的翻译，对赋值语句 x:=y+z 和 x:=y-z 可采用上一节介绍的语法制导定义来翻译，假设翻译结果已记录在 S_2.code 和 S_3.code 中。按深度优先顺序计算各结点属性值，过程如下：

（1）S.begin := L_1；E_1.true := L_2；E_1.false := S.next。

（2）E_1.code := { if a<b goto L_2

goto S.next }

（3）S_1.next := L_1；

（4）E_2.true := L_3；E_2.false := L_4。

（5）E_2.code := { if c<d goto L_3

goto L_4 }

（6）S_2.next := S_1.next。

（7）S_3.next := S_1.next。

（8）S_1.code := { (E_2.code)

L_3 : (S_2.code)

goto L_1

L_4 : (S_3.code) }

（9）S.code := { L_1 : (E_1.code)

L_2 : (S_1.code)

goto L_1

}

将 S.code 展开得到翻译结果如下：

L_1: if a<b goto L_2

goto Lnext

L_2: if c<d goto L_3

goto L_4

L_3: t_1:=y+z

x:=t_1

goto L_1

L_4: t_2:=y-z

x:=t_2

goto L_1

Lnext:

6.6 回填技术

上一节翻译控制流语句及控制流语句中的布尔表达式是基于 L-属性定义的。要翻译句子，L-属性定义首先需被改写为一个适合以深度优先顺序计算属性值的翻译模式，然后通过两遍扫描来完成句子的翻译。第一遍扫描是将用{、}括起来的语义规则作为产生式中的终结符号来对句子作语法分析，获得带语义结点的分析树；第二遍扫描是对带语义结点的分析树做

深度优先的遍历，遍历过程中访问到语义结点时，执行语义结点对应的语义规则来计算属性完成句子的翻译。对于只包含综合属性的 S-属性定义，语法分析和语义分析是在同一遍实现的，翻译句子只需要一遍扫描就可以完成，显然效率更高。

能否基于 S-属性定义通过一遍扫描来翻译控制流语句及控制流语句中的布尔表达式呢？一遍扫描完成翻译的难点在哪里呢？通过一遍扫描来获得布尔表达式和控制流语句的三地址代码的主要问题在于，当生成某些转移语句时还无法确定该语句将要转移到的位置，即布尔表达式（或子表达式）的真、假出口可能还不知道在哪里。要解决这个问题可以考虑先暂时生成不带转移标号的转移语句。对每条这样的转移语句先记录下来，一旦转移目标确定，再根据记录将转移标号“**回填**”到相应的转移语句中。

表 6-11 是一个翻译控制流语句中布尔表达式的 S-属性定义，其中为每个表达式 E 引入了两个属性：

（1）E.truelist（E 的真链）：一个指针，指向一个语句标号的链表，这些语句标号指向的都是需要填上表达式 E 的真出口的转移语句；

（2）E.falselist（E 的假链）：一个指针，指向一个语句标号的链表，这些语句标号指向的都是需要填上表达式 E 的假出口的转移语句。

在翻译过程中，当 E 的真、假出口确定之后，可以根据其真链和假链进行回填。nextquad 是一个全局变量，标记的是下一条三地址代码的地址（假设三地址代码用四元式来实现）。M 是一个标记非终结符号，引入 M 的目的是为了引进语义动作为 M.quad 赋值。M.quad 标识 E_2 的第一条三地址语句的位置。用到了几个语义动作：

（1）makelist(i)：建立新链表，然后返回新链表的指针。新建链表只包含 i 这一个语句标号；

（2）merge(p_1, p_2)：合并由指针 p_1 和 p_2 所指向的两个链表，返回结果链表的指针；

（3）backpatch(p, i)：用目标标号 i 回填 p 所指链表中记录的每一个语句标号指向的转移语句；

（4）emit(S)：产生一条三地址语句 S，变量 nextquad 递增 1。

表 6-11 一遍扫描翻译布尔表达式的语法制导定义

序号	产生式	语义规则
1	E → E_1 or M E_2	backpatch (E_1.falselist , M.quad) ; E.truelist := merge (E_1.truelist, E_2.truelist) ; E.falselist := E_2.falselist
2	E → E_1 and M E_2	backpatch (E_1.truelist , M.quad) ; E.truelist := E_2.truelist ; E.falselist := merge (E_1.falselist, E_2.falselist)

3	E → not E_1	E.truelist := E_1.falselist ; E.falselist := E_1.truelist
4	E → (E_1)	E.truelist := E_1.falselist ; E.falselist := E_1.truelist
5	E → id_1 relop id_2	E.truelist := makelist (nextquad) ; E.falselist := makelist (nextquad + 1) ; emit (‘if’ id_1.place relop.op id_2.place ‘goto_’) ; emit (‘goto_’)
6	M → ε	M.quad := nextquad
7	E → true	E.truelist := makelist (nextquad) ; emit (‘goto_’)
8	E → false	E.falselist := makelist (nextquad) ; emit (‘goto ’)

【例 6-11】：用表 6-11 翻译以下句子：

a<b or c<d and e<f

假设变量 nextquad 的当前值是 100，已知 E 的真、假出口分别是 L_1 和 L_2。

解：采用自底向上语法分析方法对句子作语法分析，构建如图 6-22 所示的分析树。在每一步最左归约的同时，执行产生式对应的语义规则。

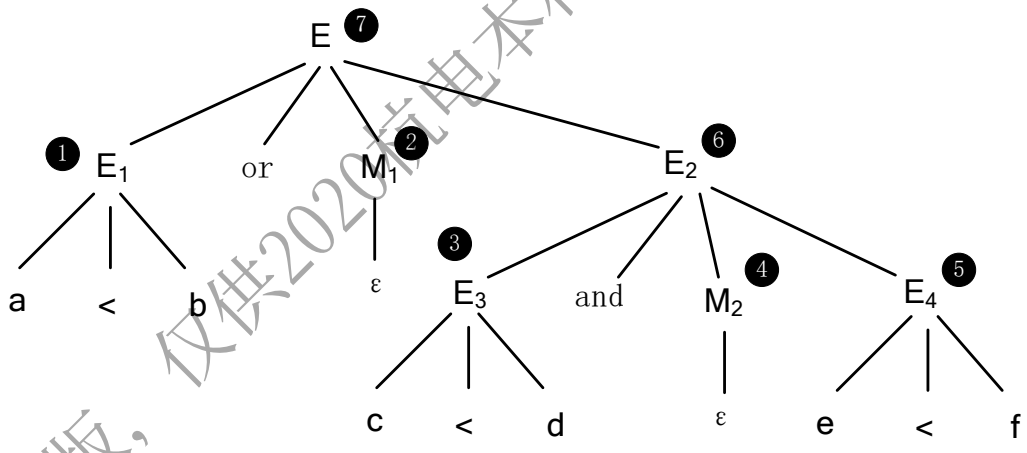


图 6-22 例 6-10 句子的分析树

（1）用产生式“E → id_1 relop id_2”归约，计算：

E_1.truelist := makelist (nextquad) := { 100 } ;

E_1.falselist := makelist (nextquad + 1) := { 101 } ;

emit (‘if a<b goto_’) ;

emit (‘goto_’)

（2）用产生式“M → ε”归约，计算：

M_1.quad := 102

（3）用产生式“E → id_1 relop id_2”归约，计算：

E_3.truelist := makelist (nextquad) := { 102 } ;

E_3.falselist := makelist (nextquad + 1) := { 103 } ;

emit (‘if c<d goto_’) ;

emit (‘goto_’)

（4）用产生式“M → ε”归约，计算：

M_2.quad := 104

（5）用产生式“E → id_1 relop id_2”归约，计算：

E_4.truelist := makelist (nextquad) := { 104 } ;

E_4.falselist := makelist (nextquad + 1) := { 105 } ;

emit (‘if e<f goto_’) ;

emit (‘goto_’)

（6）用产生式“E → E_1 and M E_2”归约，计算：

backpatch ({ 102 } , 104) ;

E_2.truelist := E_4.truelist := { 104 } ;

E_2.falselist := merge (E_3.falselist, E_4.falselist) := { 103, 105 }

（7）用产生式“E → E_1 or M E_2”归约，计算：

backpatch ({ 101 } , 102) ;

E.truelist := merge (E_1.truelist, E_2.truelist) := { 100, 104 } ;

E.falselist := E_2.falselist := { 103, 105 }

在翻译过程中，（6）、（7）两步分别将 104 回填到 102 这条转移语句，将 102 回填到 101 这条转移语句。在第（7）步计算出 E 的真链包含 100 和 104 两个语句标号，因此一旦 E 的真出口（当前假设为 L_1）确定将回填到这两条语句中，计算出 E 的假链包含 103 和 105 两个语句标号，因此一旦 E 的假出口（当前假设为 L_2）确定将回填到这两条语句中。

翻译得到的三地址代码如下：

100：if a<b goto L1

101：goto 102

102：if c<d goto 104

103：goto L2

104：if e<f goto L1

105：goto L2

类似地可以采用“回填”技术通过一遍扫描翻译控制流语句，语法制导定义见表 6-12。其中 L 表示语句列表（复合语句），A 表示赋值语句。M 是一个标记非终结符号，引入 M 的目的是为了用 M.quad 标识它后面的 E 或 S 的第一条三地址代码的位置。N 也是一个引入的标记非终结符号，为了在 S_1 的最后产生一条转移语句跳过 S_2 的代码。

为 S、L、N 分别引入 nextlist 属性，S.nextlist 指向一个语句标号的链表，相应的语句将控制流程转移到紧接着语句 S 之后要执行的代码处，L.nextlist 的意义也类似。N.nextlist 指向一个语句标号链表，链表中只包含一个 N 的语义规则所生成的转移语句的标号。

表 6-12 一遍扫描翻译控制流语句的语法制导定义

序号	产生式	语义规则
1	S → if E then M_1 S_1 N else M_2 S_2	backpatch (E.truelist , M_1.quad) ; backpatch (E.falselist , M_2.quad) ; S.nextlist := merge (S_1.nextlist , merge (N.nextlist , S_2.nextlist))
2	N → ε	N.nextlist := makelist (nextquad) ; emit (‘goto_’)
3	M → ε	M.quad := nextquad
4	S → if E then M S_1	backpatch (E.truelist , M.quad) ; S.nextlist := merge (E.falselist, S_1.nextlist)
5	S → while M_1 E do M_2 S_1	backpatch (S_1.nextlist , M_1.quad) ; backpatch (E.truelist , M_2.quad) ; S.nextlist := E.falselist ; emit (‘goto’ M_1.quad)
6	S → begin L end	S.nextlist := L.nextlist
7	S → A	S.nextlist := makelist ( )
8	L → L_1 ; M S	backpatch (L_1.nextlist , M.quad) ; L.nextlist := S.nextlist

【例 6-12】：用表 6-12 翻译以下句子：

if (a<b or c<d and e<f) then A_1 else A_2

假设全局变量 nextquad 的当前值是 100。

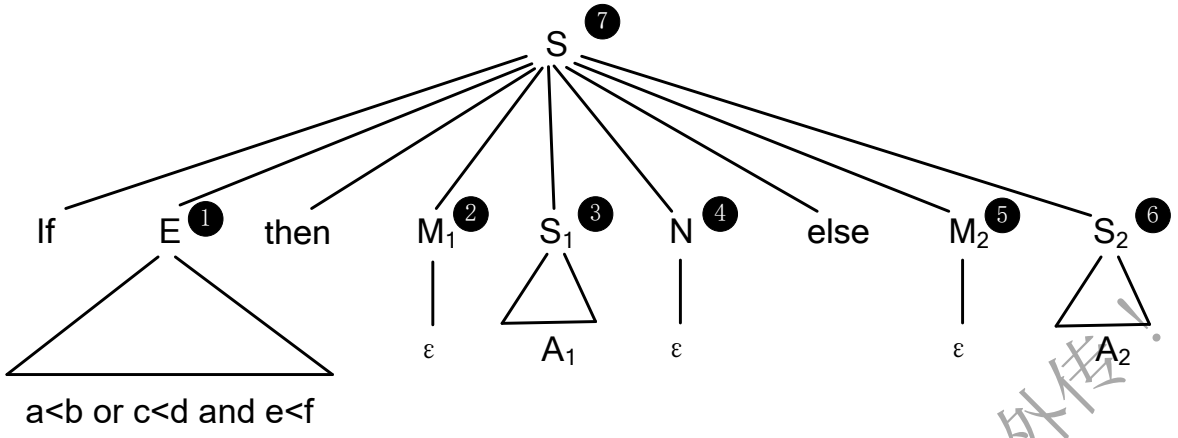


图 6-23 例 6-12 句子的分析树

解：分析树的框架如图 6-23 所示。翻译过程如下：

（1）如例 6-11 翻译 a<b or c<d and e<f，得到如下代码：

100： if a<b goto _

101： goto 102

102： if c<d goto 104

103： goto _

104： if e<f goto _

105： goto _

同时有：

E.truelist := { 100, 104 }

E.falselist := { 103, 105 }

（2）用产生式“M → ε”归约，计算：

M_1.quad := 106

M_1.quad 标记了 S_1 即赋值语句 A_1 的第一条三地址代码的位置。

（3）分析 A_1。赋值语句的翻译在 6.4 节作了介绍，假设分析 A_1 得到 10 条三地址代码：

106： …

…　　(A_1 的三地址代码)

115：…

同时计算：S_1.nextlist := makelist ( )　:= {　}

（4）用产生式“N → ε”归约，计算：

N.nextlist := makelist (nextquad) := { 116 } ;

同时输出一条三地址代码：

116：goto _

（5）用产生式“M → ε”归约，计算：

M_2.quad := 117

M_2.quad 标记了 S_2 即赋值语句 A_2 的第一条三地址代码的位置。

（6）分析 A_2，假设得到 10 条三地址代码：

117：…

…　　(A_2 的三地址代码)

126：…

同时计算：S_2.nextlist := makelist ( )　:= {　}

（7）用产生式“S → if E then M_1 S_1 N else M_2 S_2”归约，执行 backpatch（{ 100, 104 }, 106），将布尔表达式的真出口回填到代码 100、104 中；backpatch（{ 103, 105 }, 117），将布尔表达式的假出口回填到代码 103、105 中。同时计算：

S.nextlist := merge (S_1.nextlist , merge (N.nextlist , S_2.nextlist))

:= { 116 }

全部的翻译结果如下：

100：if a<b goto 106

101：goto 102

102：if c<d goto 104

103： goto 117

104： if e<f goto 106

105： goto 117

106： …

… (A_1 的三地址代码)

115： …

116： goto _

117： …

… (A_2 的三地址代码)

126： …

在翻译 S 后面的语句时，若确定了下一条三地址代码的标号，则需将该标号回填到 S.nextlist 中语句标号指向的代码，即在第 116 条代码中回填上语句标号 127。

6.7 小结

语义分析和中间代码生成在逻辑上是两个相互独立的编译步骤，但是由于它们在实现的时候基本上都是采用同一种技术——语法制导翻译技术，于是在构造编译器过程中往往将语义分析和中间代码生成作为一“遍”来实现。静态语义检查是语义分析的重要内容，包括类型检查、控制流检查、唯一性检查等。只要源程序语句中存在包含运算对象的结构，就需要进行类型检查，以判断数据对象在当前位置上是否是类型合法的。类型检查及其它的静态语义检查均可以基于语法制导翻译技术来实现。源程序中的语句分为可执行语句和非可执行语句，赋值语句、条件语句、循环语句等是可执行语句，而说明语句是不可执行语句，因为说明语句不是用来完成各类运算和数据处理的，在生成的中间代码和目标代码中也没有与说明语句对应的代码段。说明语句的作用是声明在程序中要用到的变量名、常量名和过程名等。处理说明语句的时候需要获取这些名字的属性值并把它们存入符号表中。对于各种可执行语句，都可以通过精心设计语法制导定义来实现到中间代码的等价变换，而且每种语句都可以有多个翻译方案。基于 S-属性定义翻译句子时可以结合句子的自底向上语法分析，通过一遍扫描完成句子的语法分析和语义分析。而基于 L-属性定义翻译句子时需要将语法分析和语义分析分开进行，第一遍做语法分析获得带语义结点的分析树，第二遍对分析树做深度优先的

遍历按顺序执行语义结点中的语义规则完成句子的翻译。本章的重点是理解掌握语义分析和中间代码生成中的各项任务及其基于语法制导翻译技术的实现方法，包括类型检查、说明语句处理、赋值语句翻译和控制流语句翻译等。尤其是理解控制流语句及作为控制流语句的控制条件的布尔表达式的两种不同翻译方法，一种是基于 L-属性定义的两遍扫描方法，另一种是基于回填技术的一遍扫描方法。本章只是给出了几种典型语句的有限的几个翻译方案，要求读者通过学习本章示例，融会贯通、举一反三，能自行设计处理高级语言其它常见语句的翻译模式。

习题

6.1 翻译算术表达式 - (a+b) * (c +d) +(a +b + c)为：

(1) 四元式；

(2) 三元式；

(3) 间接三元式。

6.2 参考教材中 while 语句的翻译方法，给出 repeat S until E 的翻译模式。