编译原理

Compiler Construction Principles

朱 青

信息学院计算机系，

中国人民大学，

zqruc2012@aliyun.com

第6章 语义分析和中间代码生成

- 6.1 中间语言
- 6.2 一些语法成分的翻译
  - 6.2.1 说明语句
  - 6.2.2 赋值语句
  - 6.2.3 布尔表达式
  - 6.2.4 控制语句
  - 6.2.5 过程调用

6.1 中间语言

- 静态语义检查
  - 类型检查：相容
  - 控制流检查： 合法
  - 一致性检查：定义一致
  - 名字的匹配检查：
- 翻译
  - 生成中间语言：逻辑清楚，易于优化，移植性好，
  - 中间语言的形式：后缀式，三元式，四元式，间接三元式，图

后缀式

- 后缀式（逆波兰式）
  - 表达式E的后缀形式E’的写法：
    - 若E是变量或常量，则E’ = E
    - 若E=E_1 op E_2,则E’ =E_1’ E_2’ op
    - 若E=(E_1)，则E’ = E_1

    后缀形式的语法制导翻译

    E→E1 op E2　{E.CODE := E1.CODE||E2.CODE||op }

    E→(E1)　　{E.CODE := E1.CODE}

    E→i　　{E.CODE := i}
  - 例如： a*(b+c)后缀形式：abc+*

后缀式

- 特点：
  - 运算对象(操作数）的出现顺序与中缀表示相同
  - 运算符按实际计算的顺序从左到右排序，且每一运算符总是紧跟在它的运算对象之后，无须括号
  - 计值易于在计算机上实现
- 例如，后缀式ab＋c*的计值过程
  - 1，把a压入栈
  - 2，把b压入栈
  - 3，弹出栈顶两项，相加之和压入栈
  - 4，把c压入栈
  - 5，弹出栈顶两项，相乘之积压入栈
- 推广到表达式外的范围
  - 例如 a:=b*c+b*d 后缀形式：abc*bd*+:=

图

- 抽象语法树
  - 内部结点表示运算符，后代表示运算对象
- 无循环有向图（DAG）p168
  - 与抽象语法树
    - 相同之处：内部结点表示运算符，后代表示运算对象
    - 不同之处：考虑到公共子表达式（不只一个父结点），更加紧凑高效
- 实现途径：
  - 记录指针

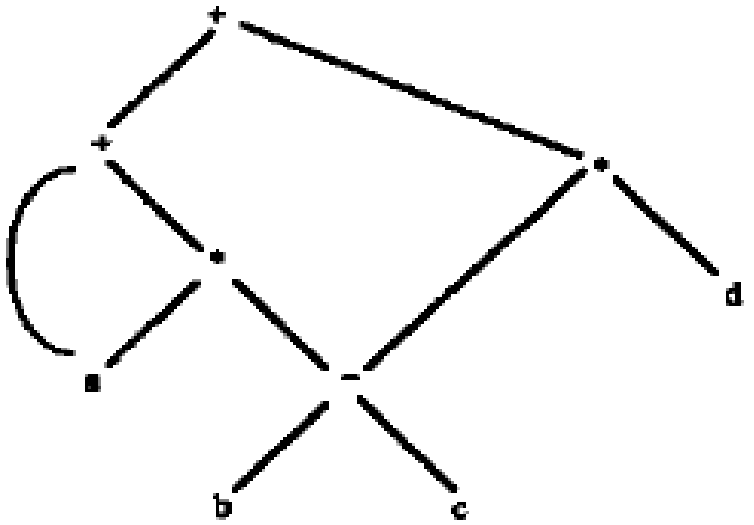


图 7.2　a+a*（b−c)+(b−c)*d 的 DAG

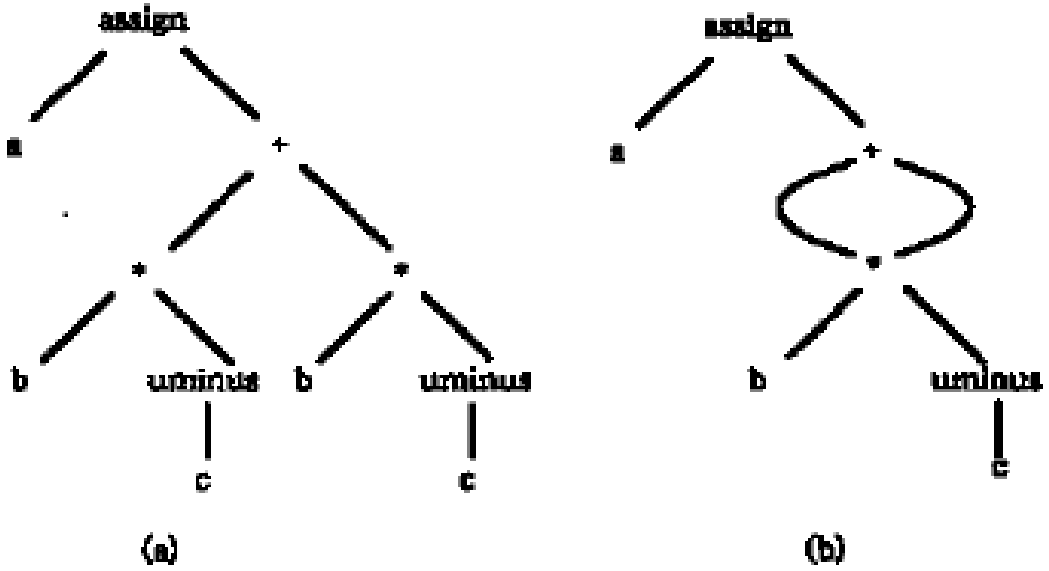


图 7.3　a:=b*−c+b*−c 的图表示法

(a)语法树；(b)DAG。

三地址代码

- 三地址代码的一般形式：
  - x:= y op z
  - y , z :操作数　x：结果　op：操作符
  - 例：x＋y*z 的三地址形式序列：

    T_1 := y*z

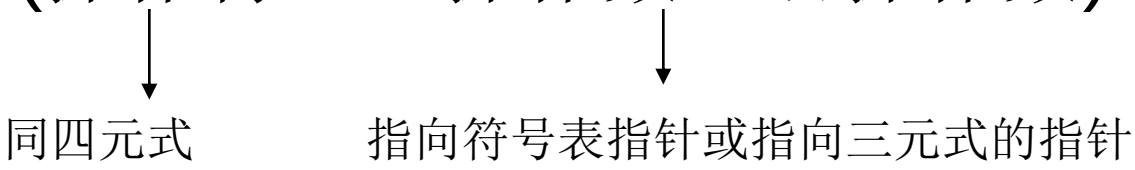
    T_2 := x+T_1
- 三地址代码的具体实现
  - 四元式
  - 三元式
  - 间接三元式

四元式

- 是一个带有四个域的记录结构
  - （操作符，左操作数，右操作数，结果）
  - 当是一元运算时，无右操作数；
  - 当是零元运算时无左、右操作数
  - 在按语法制导翻译产生的四元式中，操作符用一整数码表示，用于表示运算符的种属，及其他语义信息，如数据类型，运算方式及运算精度等。另三个量通常是指向符号表有关名字的入口
  - 例如：对于赋值语句A:=－B*(C+D), 四元式表示为
    - (－ , B, ,T1)
    - (+, C, D ,T2)
    - (*, T1, T2, T3)
    - (:=, T3 , , A)

三元式

- 三个域：(操作符，左操作数，右操作数)



- 三元式编号代表运算结果
- 例如：赋值语句A:=－B*(C+D)，写成三元式
  - ①(－，B，)
  - ②(+，C，D)
  - ③(*，①，②)
  - ④(:=，A，③)

间接三元式

- 三元式序列所需存储空间少（不需临时变量），但不便于优化处理
- 解决方法：建立两个表：
  - 三元式表：存放各三元式本身
  - 间接码表：按运算先后顺序列出三元式在三元式表中的位置
- 例如， 对以下赋值语句

  X:=(A+B)*C

  B:=A+B

  Y:=C*(A+B)

  若用三元式表示，可写成

  (1) (+, A, B)
  (2) (*,(1),C)
  (3) (:=,X,(2))
  (4) (:=,B,(1))
  (5) (+, A, B)
  (6) (*,C,(5))
  (7) (:=,Y,(6))

其中，三元式（1）和（5）形式完全一致，但却不能将（5）省去；对于（2）、（6）来说，也应当说是一致的（因为乘法满足交换律），同样不能将（6）省去。然而，若按间接三元式表示，则可写成：

间接码表	三元式表	
（1）	（1）	（+，A，B）
（2）	（2）	（*，（1），C）
（3）	（3）	（:＝，X，（2））
（4）	（4）	（:＝，B，（1））
（1）	（5）	（:＝，Y，（2））
（2）		
（5）		

三种三地址表示形式的比较

- 三元式
  - 优点：节省空间
  - 缺点：不利于优化
- 四元式与间接三元式
  - 缺点：占用存储空间相对较大
  - 优点：代码调整时改动少

课堂练习

- 求下列各式的逆波兰表示？
  - 1， -a-(b*c/(c-d) + (-b)*a)
  - 2，-A+B*C↑(D/E)/F
- 3，写出A+B*(C-D)-E/F↑G的三元组表示和四元组表示

课堂练习

- 1，-a-(b*c/(c-d) + (-b)*a)的逆波兰表示：
  a-bc*cd-/b-a*+-
- 2， -A+B*C↑(D/E)/F的逆波兰表示:
  A-BCDE/↑*F/+
- 3，A+B*(C-D)-E/F↑G
  的三元组表示：
  (1) （－，C, D)
  (2) (*, B, (1))
  (3) (+, A, (2))
  (4) (↑ , F, G)
  (5) (/, E, (4))
  (6) (-, (3), (5))

四元组表示
(1) （－，C, D , T1)
(2) (*, B, T1, T2)
(3) (+, A, T2 , T3)
(4) (↑, F, G , T4)
(5) (/, E, T4, T5)
(6) (-, T3, T5 , T6)

第6章 语义分析和中间代码生成

- 6.1 中间语言
- 6.2 一些语法成分的翻译
  - 6.2.1 说明语句
  - 6.2.2 赋值语句
  - 6.2.3 布尔表达式
  - 6.2.4 控制语句
  - 6.2.5 过程调用

6.2 一些语法成分的翻译

- 6.2.1 说明语句
- 6.2.2 赋值语句
- 6.2.3 布尔表达式
- 6.2.4 控制语句
- 6.2.5 过程调用

6.2.1 说明语句的翻译

- 说明语句
  - 定义局部于该过程的数据对象（以标识符标识）
  - 为数据对象分配空间，在符号表中登记数据对象的名字，类型，分配的存储地址
  - 存储地址：　　基址 + 偏移量
  - 过程enter (name, type, offset):将名字name填入到符号表中

说明语句的翻译

- 简单的说明语句的翻译模式(p174)

P→MD	
M→ε	{offset:=0}
D→D;D	
D→i:T	{enter(i.name,T.type,offset); offfset:= offset+T.width}
T→integer	{T.type:=integer; T.width:=4}
T→real	{T.type:=real; T.width:=8}
T→array[num] of T1	{T.type:=array(num.val, T1.type); T.width:=num.val*T1.width}
T→ ↑ T1	{T.type := pointer(T1.type); T,width := 4}

说明语句的翻译

- 嵌套的过程声明中声明语句翻译模式
  - 增加一个嵌套过程的文法：D→proc i:D1;S
    - i：过程名，该过程局部于它的外围过程
    - 创建过程i的符号表，登记过程i中说明的名字
    - 过程i的外围过程的名字在过程i中可用，过程i有指针指向它的外围过程符号表
  - 符号表的组织
    - 例：见书P176

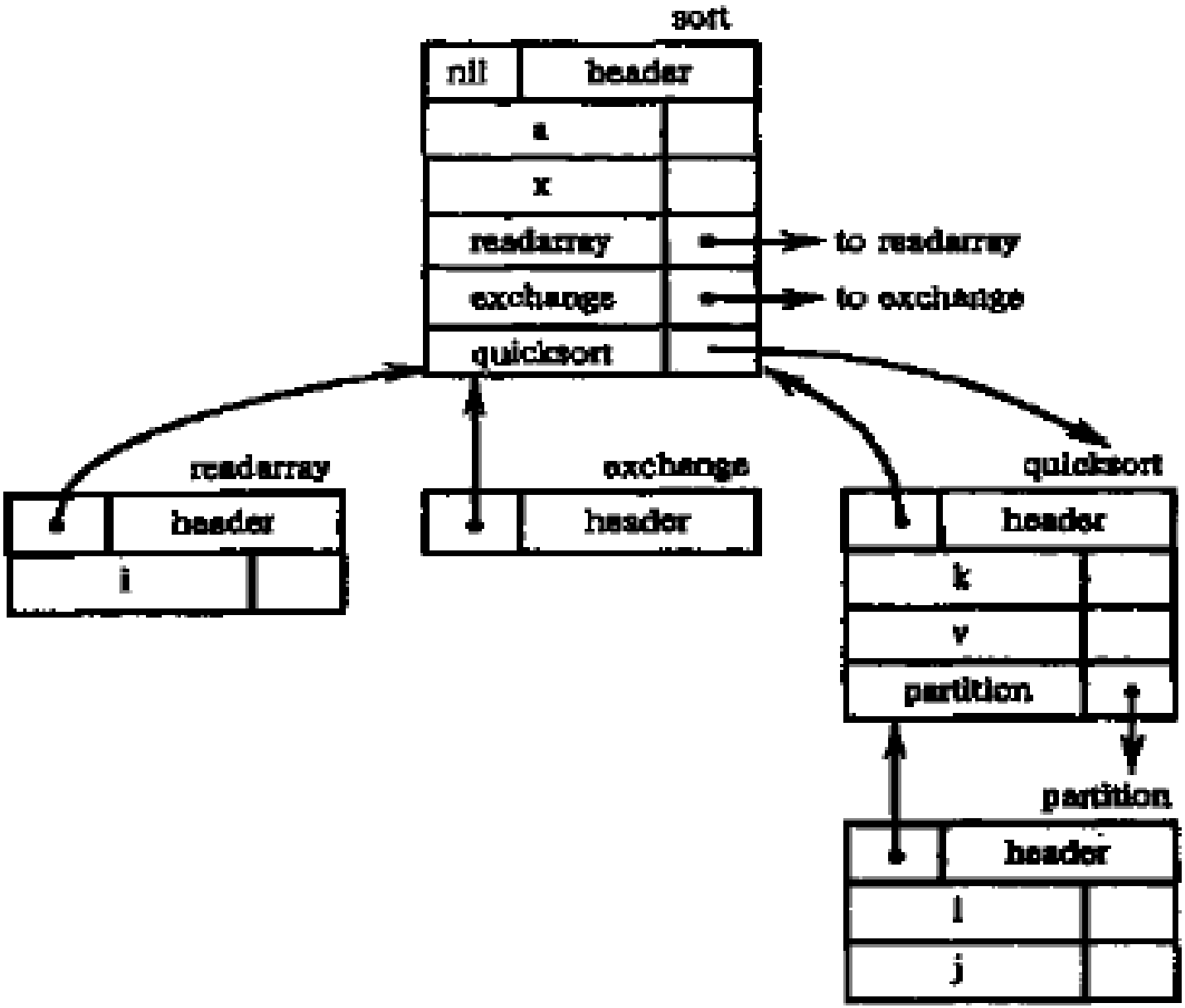


图 7.7　嵌套过程的符号表

说明语句的翻译

- 嵌套过程声明的翻译模式

P→MD	{addwidth(top(tblptr),top(offset)); // 在过程符号表中记录 pop(tblptr); pop(offset) } 所有名字总长度
M→ε	{t:=mktable(nil); push(t,tblptr);push(0,offset)} // 初始化过程符号表
D→D;D	
D→i:T	{enter(top(tblptr),i.name,T.type,top(offset); top(offset) := top(offset)+T.width} // 为数据对象分配存储空间
D→proc i: N D1;S	{t:=top(tblptr); // 记录嵌套过程符号表指针 Addwidth(t,top(offset); pop(tblptr); pop(offset) // 弹出嵌套过程的符号表指针 enterproc(top(tblptr),i.name,t)} //嵌套过程指向外围过程
N→ε	{t:=mktable(top(tblptr)); push(t,tblptr);push(0,offset)} // 初始化嵌套过程符号表
.....	.....

说明语句的翻译

- 记录类型的声明
  - 文法：T→record LD end
  - 为记录创建一个符号表记录域名 （由D表示）
  - 翻译模式

T→record LD end	{T.type:=record(top(tblptr)); T.width:=top(offset); // 记录的长度 pop(tblptr); pop(offset)}
L→ε	{t:=mktable(top(tblptr)); //为记录创建符号表 push(t,tblptr);push(0,offset)} .....

6.2.2 赋值语句的翻译

- 只含简单算术表达式的赋值语句
  - 约定算符优先性，结合性
  - 翻译模式

S→i:=E	{P:=lookup(i.name); //查找符号表，看变量i是否说明 if P<>nil then emit(P, ':=',E.place) //生成三地址语句 else error}
E→E1 op E2	{E.place :=newtemp ; //结果指向临时变量 emit(E.place , ':=', E1.place, 'op', E2.place) }
E→-E1	{E.place :=newtemp ; emit(E.place , ':=', 'unimus', E1.place) }
E →(E1)	{E.place := E1.place}
E→i	{P:=lookup(i.name); if P<>nil then E.place:=P else error}

赋值语句的翻译

- 例：a:=-b*c+d 按上述方案的语法制导翻译过程。

按归约次序依次为

1）E→b，i.name为b，设E.place为b；

2）E→-E1,设newtemp为t1，emit生成三地址语句t1:= uminus b;

3) E→c，i.name为c，设E.place为c；

4) E→E1*E2,设newtemp为t2，此时E1.place为t1， E2.place为c，emit生成三地址语句t2:= t1*c;

5) E→d，i.name为d，设E.place为d；

6) E→E1+E2,设newtemp为t3，此时E1.place为t2， E2.place为d，emit生成三地址语句t3:= t2+d;

7) S→a:=E, i.name为a，此时E.place为t3 ，emit生成三地址语句a:= t3；

赋值语句的翻译

- 类型转换
  - 许多语言允许混合类型运算，但在运算前须转换为相同类型
  - 引入类型转换功能的三地址语句t:=itr x，将整型变量x转换为实型变量t
  - 对于产生式E->E1 op E2, 语义子程序可写成

```
t:=newtemp;
 if E1 .TYPE=integer and E2.TYPE=integer then
        begin emit(t, ':=',E1.place,'op', E2.PLACE);
                        E.TYPE:=integer
        end
 else if E1 .TYPE=real and E2.TYPE=real then
        begin emit(t, ':=',E1.place,'op', E2.PLACE);
                        E.TYPE:=real
        end
 else if E1 .TYPE=interger then
        begin  t1:=newtemp;
               emit(t1, ':=',itr, E1.PLACE);
               emit(t, ':=',t1,'op', E2.PLACE);
                        E.TYPE:=real
        end
else    begin   t1:=newtemp;
               emit(t1, ':=',itr, E2.PLACE);
               emit(t, ':=', E1.PLACE,'op', t1);
                        E.TYPE:=real
        end
E.place := t
```

6.2.3 布尔表达式的翻译

- 布尔表达式：一般由布尔运算符作用于布尔变量或关系表达式构成
  - 文法：E->E∧E|E∨E|┐E|(E)| i|i rop i|true|false
  - 组成：
    - 逻辑运算符：∧，∨，┐(and , or ,not)
    - 布尔运算量：逻辑值，布尔变量，关系表达式
    - 关系式：E1 rop E2
- 布尔表达式的作用：
  - 作为控制语句的条件式：如if E then S, while E do S
  - 计算逻辑值：E ∧ E
- 布尔表达式运算符优先关系
  - ①括号：由里至外
  - ②算术运算符：*、/
    +、-
  - ③关系运算符：<、<=、>、>=、<>
  - ④逻辑运算符：┐
    ∧
    ∨

布尔表达式的翻译

- 布尔表达式的计值或翻译
  - ① 数值法： A∨B∧C ： （∧，B，C，T1）
    （∨，A，T1，T2）
  - ②解释法: A∨B∧C : if A then true else
    if B then C else false
    - 三种逻辑运算的等价解释
      - A∨B: if A then true else B
      - A∧B: if A then B else false
      - ¬A: if A then false else true

布尔表达式的翻译

- 数值表示法的翻译方案

E→E1 or E2	{E.place:=newtemp; emit(E.place ':=' E1.place 'or' E2.place) }
E→E1 and E2	{E.place:=newtemp; emit(E.place ':=' E1.place 'and' E2.place) }
E→not E1	{E.place :=newtemp ; emit(E.place ':=' 'not' E1.place) }
E →(E1)	{E.place := E.place}
E→id1 relop id2	{E.place:=newtemp; emit('if' id1.place relop id2.place 'goto' nextstat+3); emit(E.place ':=' '0'); emit('goto' nextstate+2); emit(E.place ':=' '1')}

布尔表达式的翻译

- 数值表示法的翻译方案（续）

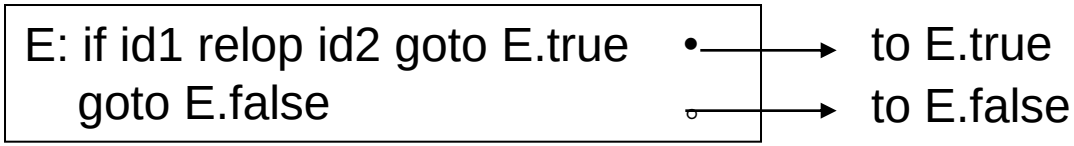
E→id	{E.place:=newtemp; emit(‘if’ id.place ‘goto’ nextstate+3); emit(E.place ‘:=’ ‘0’); emit(‘goto’ nextcode+2); emit(E.place ‘:=’ ‘1’)}
E→true	{E.place:=newtemp; emit(E.place ‘:=’ ‘1’)}
E→false	{E.place:=newtemp; emit(E.place ‘:=’ ‘0’);

布尔表达式的翻译

- 例：对布尔表达式a<b or c=d and not e>f 生成三地址代码
  - 100：if a<b goto 103
  - 101: t1:=0
  - 102: goto 104
  - 103: t1:=1
  - 104: if c=d goto 107
  - 105: t2:=0
  - 106: goto 108
  - 107: t2:=1
  - 108: if e>f goto 111
  - 109: t3:=0
  - 110: goto 112
  - 111: t3:=1
  - 112: t4:= not t3
  - 113: t5:= t2 and t3
  - 114: t6:= t1 or t5

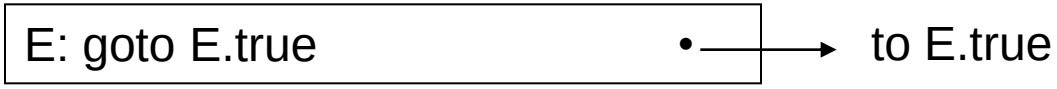
布尔表达式的翻译

- 控制流语句中布尔表达式的翻译
  - 用解释法翻译为一系列条件转移和无条件转移的三地址代码
  - 布尔表达式E有两个出口：真出口“E.true”和假出口“E.false”
  - E→id1 relop id2（id1和id2均为算术量，relop为关系运算符）



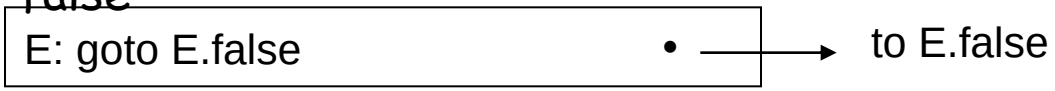
  - E→id

  - E→true

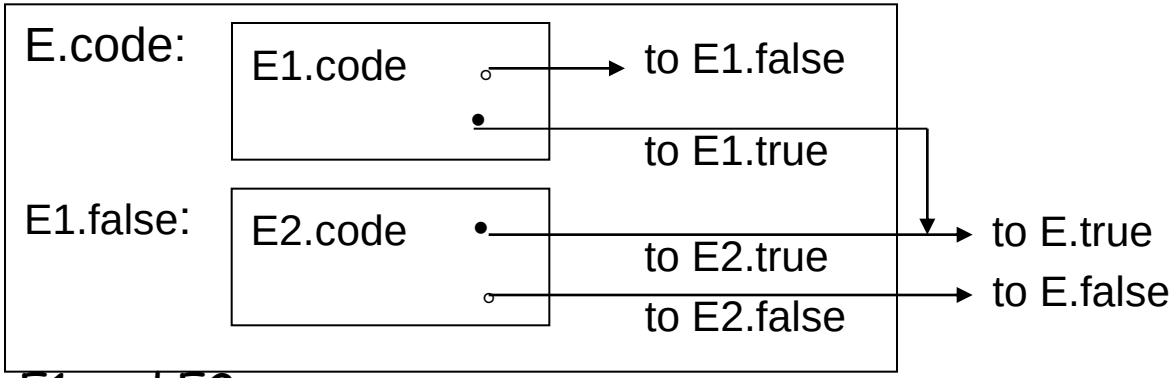


布尔表达式的翻译

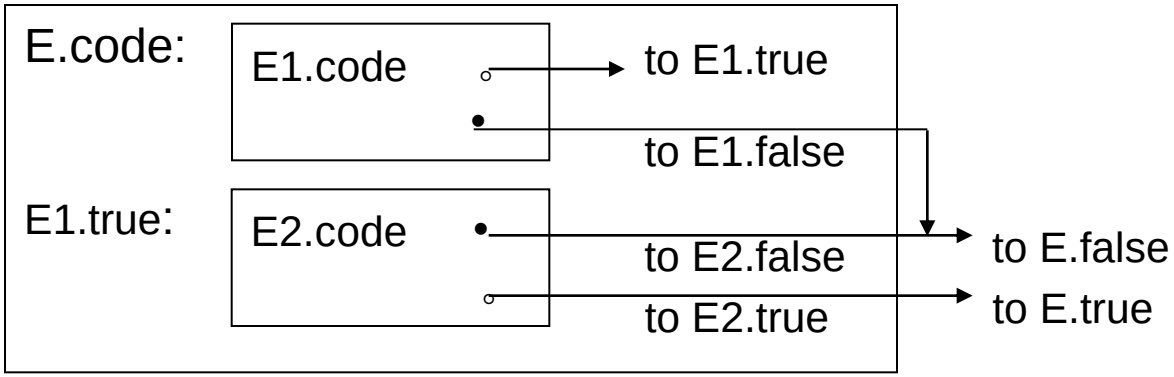
- E→false



- E→E1 or E2

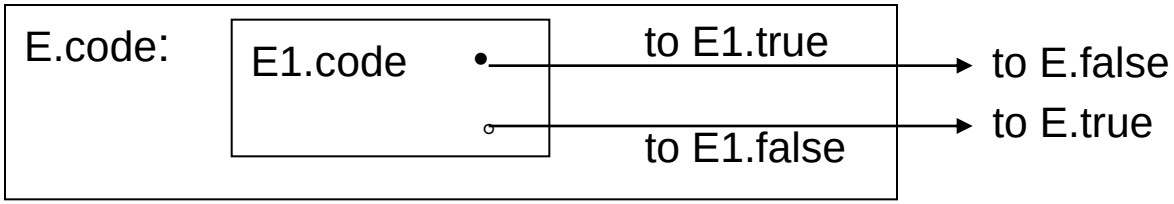


- E→E1 and E2

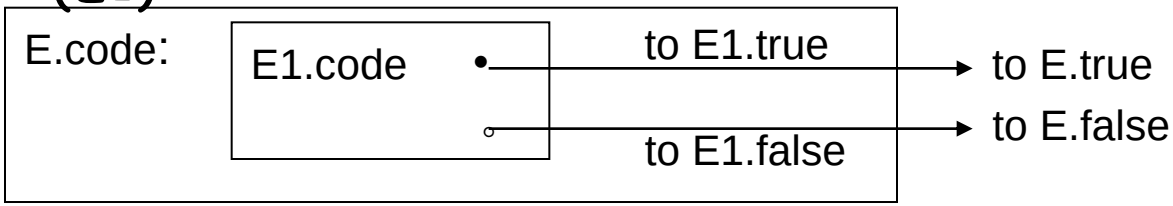


布尔表达式的翻译

- E→not E1



- E→(E1)



布尔表达式的翻译

- 一遍扫描存在的问题：转向时无法确定转向地址
  - 例如　语句　if a<b or c<d and e<f then S1 else S2的四元式

(1) if a<b goto (7) //转移至(E.true)
(2) goto (3)
(3) if c<d goto (5)
(4) goto (p+1) //转移至(E.false)
(5) if e<f goto (7) //转移至(E.true)
(6) goto (p+1) //转移至(E.false)
(7) S1的四元式 // (E.true) 入口
……

(p-1) ……

(p) goto (q)

(p+1) S2的四元式 //(E.false)入口

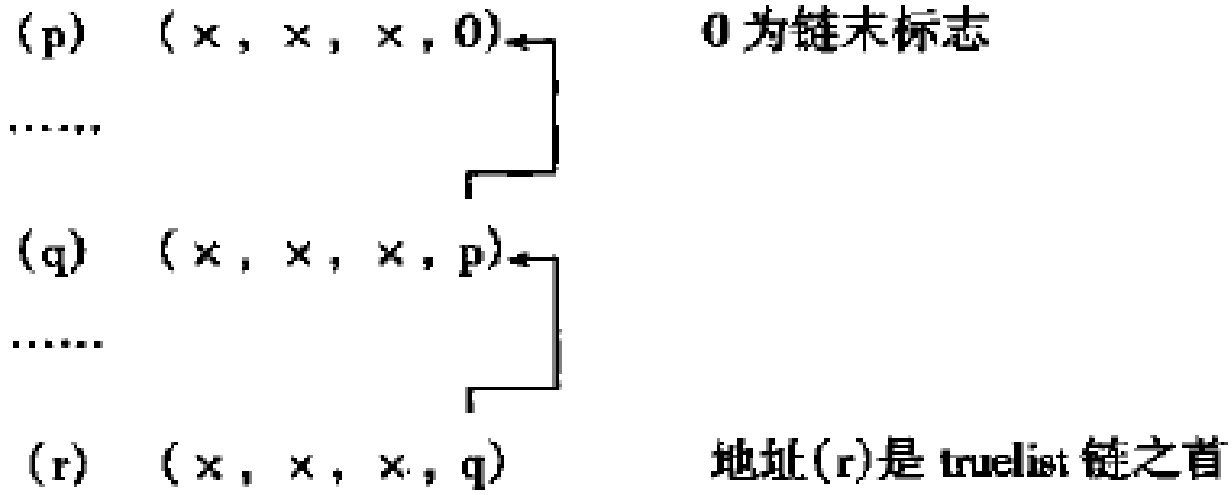
……

(q-1) ……

(q)

布尔表达式的翻译

- 解决方案：回填技术——拉链回填技术
  - 建立链表，记录跳转指令的标号
  - 转向目标确定后，回填入对应的跳转指令



- 翻译模式：见书P190

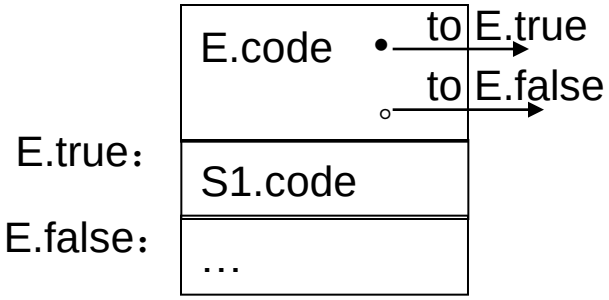
布尔表达式的翻译

- 例：用回填技术完成a<b or c<d and e<f的翻译，设nextquad为100。
  - 生成的三地址代码为：

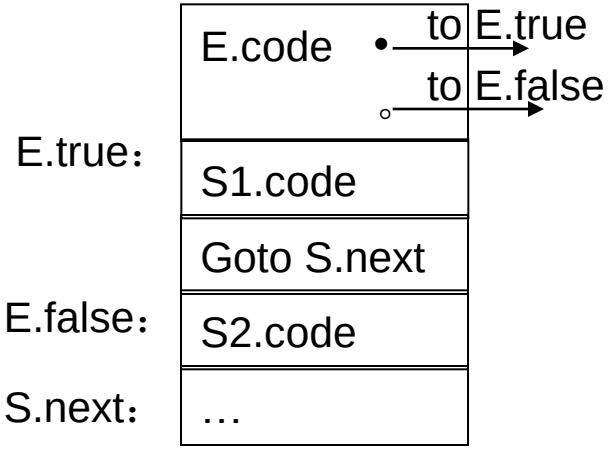
100: (j<, a, b, 0)　　//“真”出口
101: (j, -, -, 102)
102: (j<, c, d, 104)
103: (j, -, -, 0)　　//“假”出口
104: (j<, e, f, 0)　　//“真”出口
105: (j, -, -, 0)　　//“假”出口

6.2.4 控制流语句的翻译

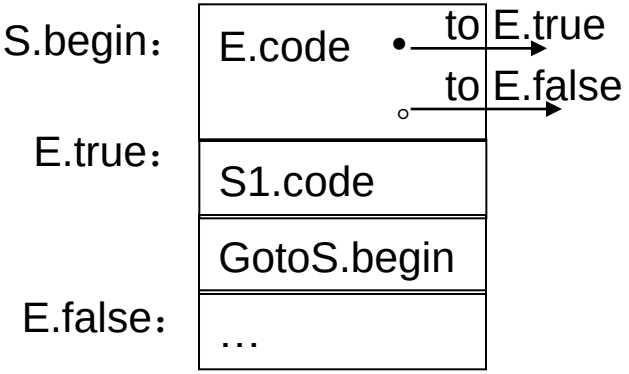
- 常见控制结构：序列、条件分支、while循环
- 控制语句在源程序中的可以并列也可嵌套
- 控制转移



If-then代码结构



If-then-else代码结构



while-do代码结构

控制流语句的翻译

- 相关文法

S->if E then S1　　S 为语句

| if E then S1 else S2　　E 布尔表达式

| While E do S1

| begin L end

| A　　A 为赋值语句

L->L；S　　L 为语句串

| S

- 翻译模式：书P195
  - 使代码正确的跳转，确定E.truelist， E.falselist，S.nextlist的入口

标号与转向语句的翻译

- 定义标号 L: S
  - 在符号表登记L，地址为S的三地址代码开始位置
- 引用标号 goto L
  - 产生四元式（j, -, -, p), p为L的地址
- 问题：
  - 先定义，后引用（向后引用）：易处理
  - 先引用，后定义（向前引用）：转向地址待定，须回填

标号与转向语句的翻译

- goto L的翻译
  - 对于向前引用，建立转向地址待回填的四元式链
  - 翻译思想
    - L已定义，L.value为符号表中L的地址，则生成（j, -, -, L.value)
    - 若L尚未在符号表中出现，则把L填入表中，置L的“定义否”标志为“未”，把nextquad填进L的地址栏,作为新链首，然后，产生四元式(j, -, -, 0),其中 0 为链尾标志
    - 若L已在符号表中出现（但“定义否”标志为“未”），则把它的地址栏中的编号（记为q）取出，把nextstat填进该栏作新链首，然后，产生四元式(j, -, -, q)。

标号与转向语句的翻译

- 标号语句文法：S→<label>S

<label>→ i ：

用<label>→ i ：进行归约时，应做如下的语义动作：

1.若i所指的标识符（L）不在符号表中，则把它填入，“类型”为“标号”，“定义否”为“已”，“地址”为nextquad。

2.若L已在符号表中但“类型”不为“标号”或“定义否”为“已”，则报告出错。

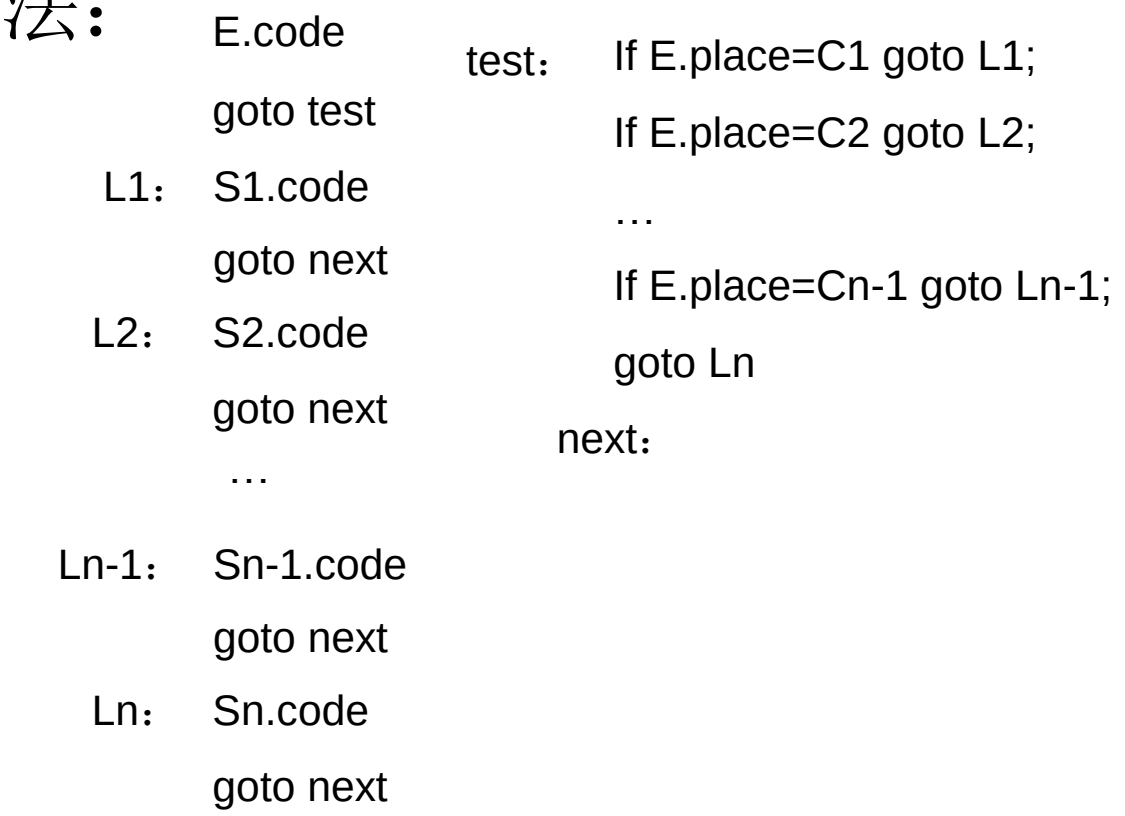
3.若L已在符号表中，则把标志“未”改为“已”，然后，把地址栏中的链首（设为q）取出，同时把nextquad填在其中，最后，执行backpatch（q，nextquad）。

CASE语句的翻译

- case语句的文法：

S→case E of
C1: S1;
C2: S2;
…
C_{n-1}: S_{n-1};
otherwise: Sn
end

- 目标代码结构



6.2.5 过程调用

- 实质：把程序控制转移到子程序(过程段)
- 语义动作：
  - 传递参数信息（实在参数的地址）给被调用的子程序
  - 告诉子程序在它工作完毕后返回到什么地方。
- 例如：过程调用ＣＡＬＬ　Ｓ（Ａ＋Ｂ，　Ｚ）

  将被翻译成：

```
T:=A+B
par T        //第一个实参地址
par Z        //第二个实参地址
call S       //转子指令
```

过程调用

- 翻译模式

（1）S→call id（Elist）

{For　队列QUEUE 的每一项p　Do
emit('par',p);
emit('call',id.place)}

（2） Elist → Elist , E

{把E.PLACE 排在QUEUE 的末端 }

（3） Elist →E

{建立一个QUEUE，只包含一项E.PLACE}