第8章　静态语义分析和中间代码生成

8.1 符号表

8.2 静态语义分析

8.3 中间代码生成

8.1 符号表

8.1.1 符号表的作用和地位

8.1.2 符号的主要属性及作用

8.1.3 符号表的实现

8.1.4 符号的组织

8.1.1 符号表的作用和地位

- 在编译程序中，符号表用来存放语言程序中出现的有关标识符的属性信息，这些信息集中反映了标识符的语义特征属性。
- 在词法分析及语法分析过程中不断积累和更新表中的信息，并在词法分析到代码生成的各阶段，按各自的需要从表中获取不同的属性信息。
- 不论编译策略是否分趟，符号表的作用和地位是一致的。

符号表的功能

- 收集符号属性：在分析语言程序中标识符说明部分时，编译程序根据说明信息收集有关标识符的属性，并在符号表中建立符号的相应属性信息。
- 上下文语义的合法性检查的依据：通过符号表中的属性记录可进行相应的上下文语义检查。
- 作为目标代码生成阶段地址分配的依据：每个符号变量在目标代码生成时都需要确定存储分配的位置，根据定义的存储类别或被定义的位置来确定。

8.1.2 符号的主要属性及作用

- 语言符号可分为关键字符号，操作符符号和标识符符号。
- 它们之间的主要属性有较大差别。

1、符号名

- 包括变量名，函数名和过程名。
- 每个标识符由若干个字符(非空格字符)组成的符号串表达。
- 一般不允许重名。
- 一般可以用一个标识符在符号表中的位置的整数值(称为内部代码)来代替该符号名。
- 在一些允许操作重载的语言中，函数名，过程名是可以重名的，对于这类重载的标识符要通过它们的参数个数和类型以及函数的返回值类型来区别。

2、符号的类型

- 除过程标识符之外，函数和变量标识符都具有数据类型属性，对于函数的数据类型指的是该函数值的数据类型。
- 基本数据类型有整型、实型、字符型、逻辑型等。
- 变量符号的类型属性决定了该变量的数据在存储空间的存储格式，还决定了在该变量上可以施加的运算操作。
- 还有一些在基本数据类型基础上扩充的符号数据类型，如数组类型、记录结构类型、指针类型等。

3、符号的存储类别

- 大多数程序设计语言对变量的存储类别定义采用两种方式：
  - 关键字指定
  - 根据定义变量说明在程序中的位置来决定
- 区别符号存储类型的属性是编译过程语义处理、检查和存储分配的重要依据。

4、符号的作用域及可视性

- 一个符号变量在程序中起作用的范围，称为它的作用域。
- 定义该符号的位置及存储类关键字决定了该符号的作用域。
- 一般说来，一个变量的作用域就是该变量可以出现的场合，也就是说在某个变量作用域内该变量是可引用的(可视的)，这就是变量可视性的作用域规则。
- 但是变量可视性不仅仅取决于它的作用域，还有两种情况影响到一个变量的可视性。

第一种情况：函数的形式参数

举例：

```
int a;
int func(a, b)
    float a;
    int b;
    {
    …
    …a…
    …
}
```

此处引用的是float a，而int a是不可见的。如果要引用int a，需要使用语法记号::a。

第二种情况：复合语句分程序结构

举例：

```
…
{int a;
    …
    {char a;
    …
        {
        …
                {float a;
                …
                }
        …a…
        }
    }
}
```

为了确立符号的作用域和可视性，符号表属性中除了需要符号的存储类别之外，还定义该符号作程序结构上被定义的层次。

此处引用的是char a

5、符号变量的存储分配信息

- 根据符号变量的存储类别定义及它们出现的位置和次序来确定每一个变量应分配的存储区及在该区中的具体位置。
- 通常一个编译程序有两类存储区，静态存储区和动态存储区。
- 对变量存储分配的属性除了存储类别之外还要确定其所在存储区的具体位置的属性信息；
- 位置信息是按照变量的存储区类依出现先后的次序排列，相对该存储区表头的相对位移量来表示的。

(1) 静态存储区

- 该存储区单元经定义分配后成为静态单元，即在整个语言程序运行过程中是不可改变的。作静态分配的符号变量是具有整个程序运行过程的生命周期。
- 根据变量存储类别及作用域规则，这类静态存储区又可分为公共静态区和若干个局部静态区。
  - 一个语言程序中的公共变量或称外部变量是被指定分配到该公共静态存储区中。在公共静态区中的变量具有的生命周期是该程序运行的全过程且其作用域是整个语言程序。
  - 编译程序为局部静态量设立局部静态区。

(2) 动态存储区

- 根据变量的局部定义和分程序结构，编译程序设置动态存储区来适应这些局部变量的生存和消亡。
- 局部动态变量的生存期是定义该变量的局部范围，即在该定义范围之外次变量已经没有存在的必要。及时撤销时，这些单元的分配可以回收，从而提高程序运行时的空间效率。

```
…
int a;
…
float b;
…
struct cc{
    int d;
    float e;
    …
    } c;
…
```

标识符		位置属性	
a		0	
b		4	
d		0	
e		4	
c		8	

6、符号的其他属性

- **数组内情变量**：编译程序需要将描述数组属性信息的内情变量(包括数组类型，维数，各维的上下界、数组首地址)登录在符号表中，这些属性信息是确定存储分配时数组所占空间的大小和数组元素位置的依据。
- **记录结构型的程序信息**：用来确定结构型变量存储分配时所占空间的尺寸及确定该结构成员的位置。
- **函数及过程的形参**：每个函数或者过程的形参个数，形参的排列次序及每个形参的类型都体现了调用该函数或过程时的属性，它们都应该反映在符号表的函数或过程的标识符表项中。

8.1.3 符号表的实现

- 针对符号表的常见操作
  - 创建符号表　在编译开始，或进入一个作用域
  - 插入表项　在遇到新的标识符声明时进行
  - 查询表项　在引用标识符时进行
  - 修改表项　在获得新的语义值信息时进行
  - 删除表项　在标识符成为不可见或不再需要它的任何信息时进行
  - 释放符号表空间　在编译结束前或退出一个作用域

– 实现符号表的常用数据结构

- 一般的线性表

  如：数组，链表，等

- 有序表

  查询较无序表快，如可以采用折半查找

- 二叉搜索树

- Hash表

8.1.4 符号表的组织

- 作用域与可见性
- 作用域与符号表组织
  - 所有作用域共用一个全局符号表
  - 每个作用域都有各自的符号表

✧ 作用域与可见性

- 嵌套的作用域（*nested scopes*）
- 开作用域与闭作用域（相应于程序中特殊点）
  - 该点所在的作用域为当前作用域
  - 当前作用域与包含它的程序单元所构成的作用域称为开作用域（*open scopes*）
  - 不属于开作用域的作用域称为闭作用域（*close scopes*）

✧ 作用域与可见性

- 常用的可见性规则（*visibility rules*）
  - 在程序的任何一点，只有在该点的开作用域中声明的名字才是可访问的
  - 若一个名字在多个开作用域中被声明，则把离该名字的某个引用最近的声明作为该引用的解释
  - 新的声明只能出现在当前作用域

✧ 作用域与符号表组织

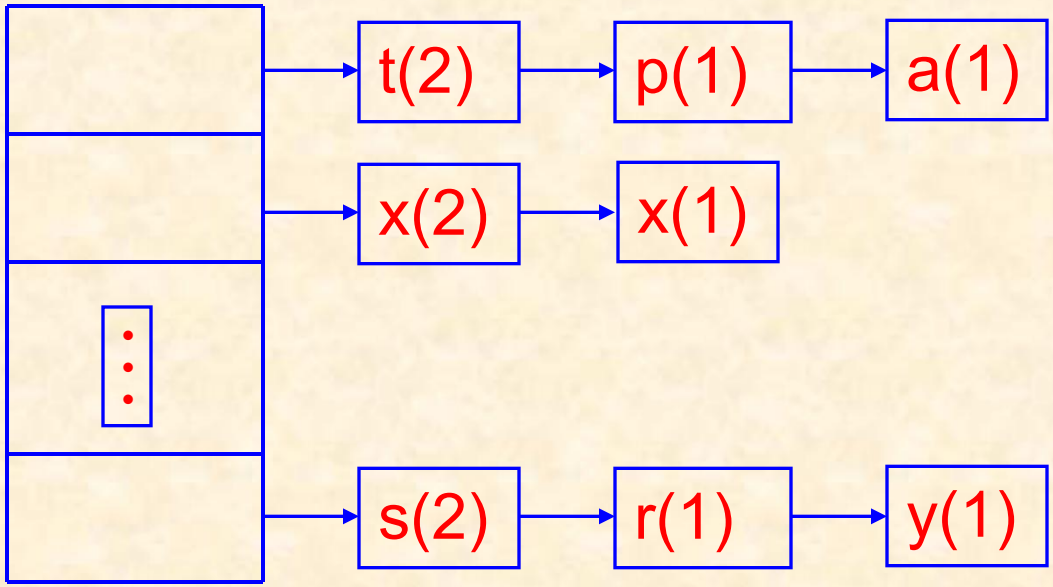
– 作用域与单符号表组织

- 所有嵌套的作用域共用一个全局符号表
- 每个作用域有一个作用域号
- 仅记录开作用域中的符号
- 当某个作用域成为闭作用域时，从符号表中删除该作用域中所声明的名字

✧ 所有嵌套的作用域共用一个

全局符号表

例：右边某语言程序在处理到/*here*/时的符号表（以哈希表为例）



Hash Table （表中数字代表层号）

```
const a=25;
var x,y;
procedure  p;
   var  z;
   begin
        ……
   end;
procedure  r;
   var x, s;
   procedure t;
        var  v;
        begin
           ……
        end;
   begin       /*here*/
        ……
   end;
begin
   ……
end.
```

✧ 所有嵌套的作用域共用一个

全局符号表

例：右边某语言程序在处理到/*here*/时的符号表（以线性表为例）

NAME	KIND	VAL / LEVEL	ADDR	SIZE
a	CONSTANT	25		
x	VARIABLE	LEV	DX	
y	VARIABLE	LEV	DX+1	
p	PROCEDUR	LEV		CX+1
r	PROCEDUR	LEV		CX+2
x	VARIABLE	LEV+1	DX	
s	VARIABLE	LEV+1	DX+1	
t	PROCEDUR	LEV+1		CX+3

Dx: 基地址　　**Cx:** 栈帧中控制单元数目

LEV: 层号

```
const a=25;
var x,y;
procedure  p;
   var  z;
   begin
        ……
   end;
procedure  r;
   var x, s;
   procedure t;
        var  v;
        begin
          ……
        end;
   begin        /*here*/
        ……
   end;
begin
   ……
end.
```

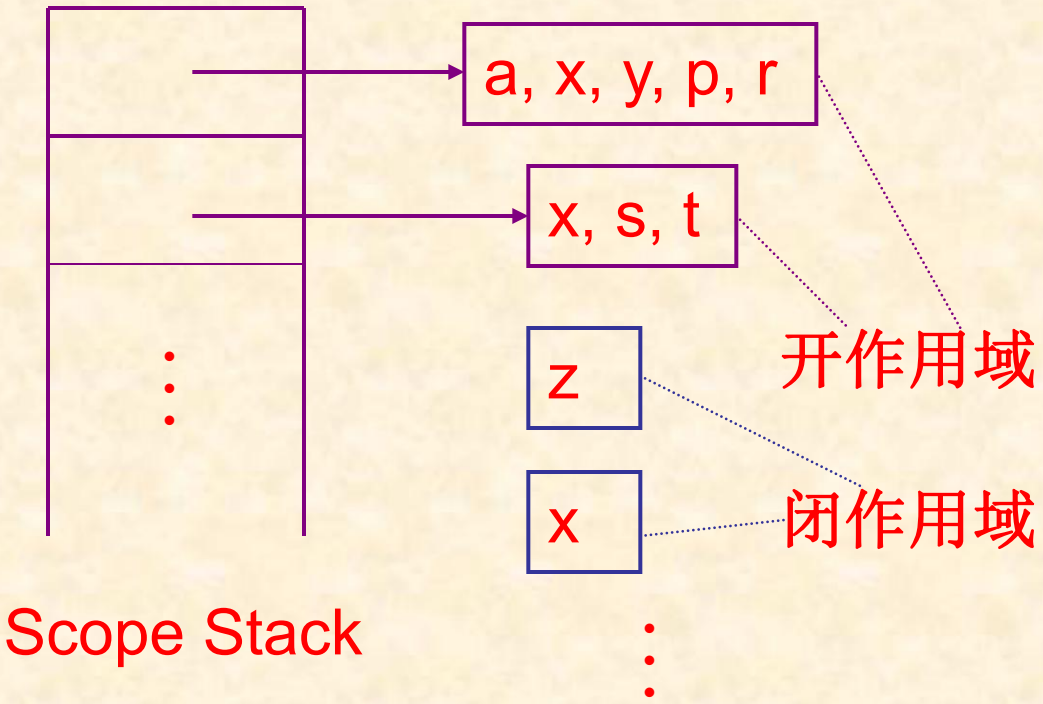
✧ 作用域与符号表组织

– 作用域与多符号表组织

- 每个作用域都有各自的符号表
- 维护一个符号表的作用域栈，每个开作用域对应栈中的一个入口，当前的开作用域出现在该栈的栈顶
- 当一个新的作用域开放时，新符号表将被创建，并将其入栈
- 在当前作用域成为闭作用域时，从栈顶弹出相应的符号表

✧ 每个作用域都有各自的符号表

例：右边程序在处理到/*here*/时的作用域栈如下所示




```
const a=25;
var x,y;
procedure  p;
   var  z;
   begin
        ……
   end;
procedure  r;
   var x, s;
   procedure t;
        var  x;
        begin
          ……
        end;
   begin        /*here*/
        ……
   end;
begin
   ……
end.
```

复　习

第七章：语法制导的语义计算

基于属性文法的语义计算

基于翻译模式的语义计算

第八章：静态语义分析和中间代码

符号表：

符号表作用

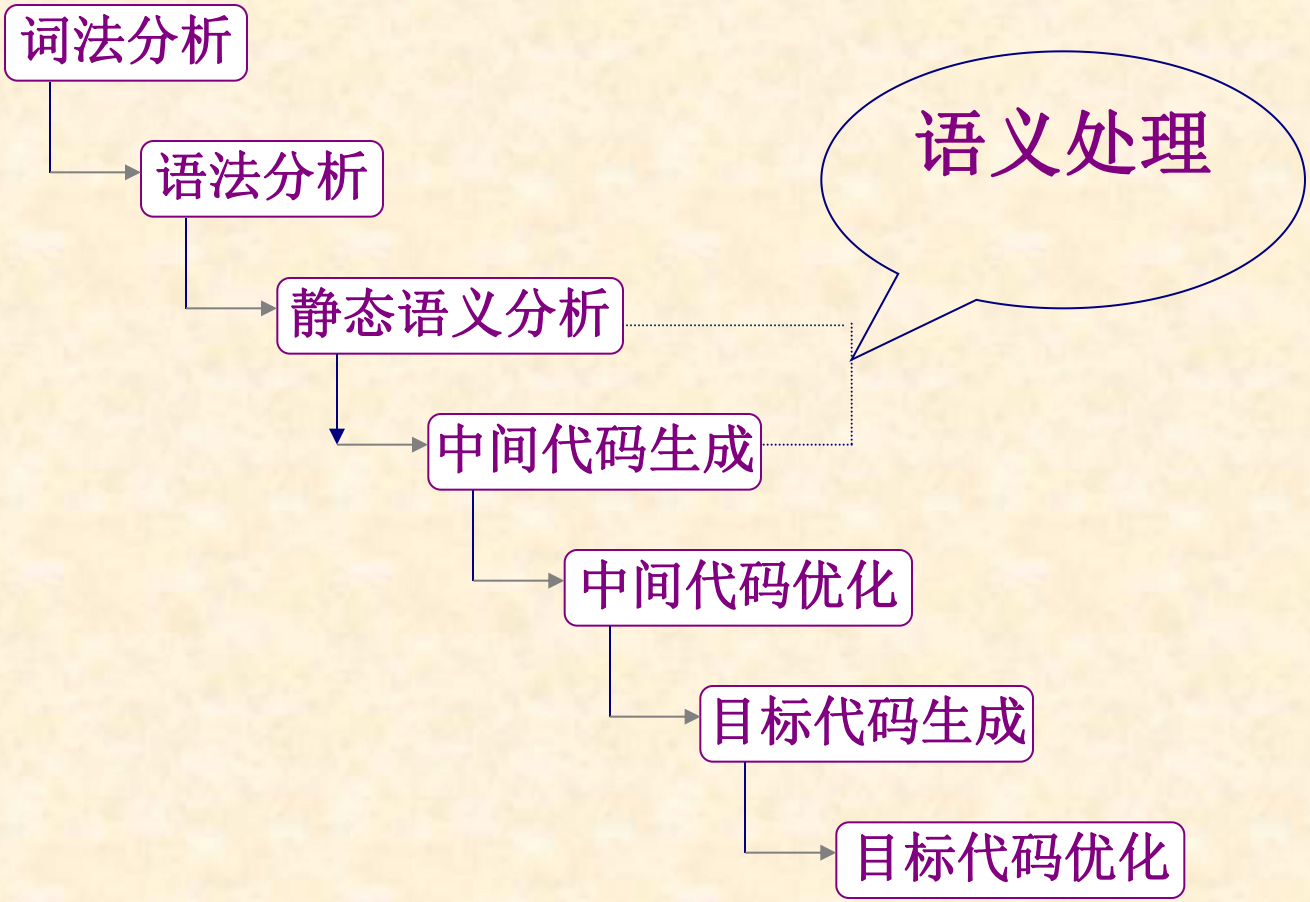
符号表的属性

符号表的实现

符号表的组织

8.2 语义分析与中间代码生成

- 语义分析和中间代码生成在编译程序中的逻辑位置



8.3 中间代码

- 何谓中间代码 (Intermediate code)
  (Intermediate representation)
  (Intermediate language)

  源程序的一种内部表示，不依赖目标机的结构，易于机械生成目 标代码的中间表示。

- 为什么要此阶段

  逻辑结构清楚；利于不同目标机上实现同一种语言；

  利于进行与机器无关的优化；这些内部形式也能用于解释。

- 中间代码的几种形式

  逆波兰　四元式　三元式

逆波兰记号

- 逆波兰记号：最简单的一种中间代码表示。最早用于表示算术表达式，现在也可以扩充到表达式以外的范围。
- 特点：把运算对象写在前面，运算符号写在后面。

逆波兰表示

程序设计语言中的表示	逆波兰表示
a+b	ab+
a+b*c	abc*+
(a+b)*c	ab+c*
a:=b*c+b*d	abc*bd*+:=

后缀表示法表示表达式，其最大的特点是易于计算机处理表达式。

利用一个栈，自左向右扫描算术表达式（后缀表示）。每遇到运算对象，就把它推进栈；碰到运算符，若该运算符是二目的，则对栈顶的两个运算对象实施该运算，并将运算结果代替这两个运算对象而进栈。若是一目运算符，则对栈顶元素执行该运算，并以运算结果代替该元素进栈。最后栈顶元素留在栈顶。

三元式

- 把表达式及各种语句表示成一组三元式。每个三元式由三部分组成：算符、第一运算对象、第二运算对象
- 例：a:=b*c+b*d

  (1) (*, b, c)

  (2) (*, b, d)

  (3) (+, (1), (2))

  (4) (:=, (3), a)

- 与后缀式不同，三元式中含有对中间计算结果的显式引用。
- 对于一目运算符op，只需选用一个运算对象。至于多目运算符，可用若干个相继的三元式表示。

树形表示

- 树形表示是三元式的翻版。
- 表达式的树形表示很容易实现：简单变量或常数的树就是该变量或常数自身。
- 二目运算符对应二叉子树，多目运算对应多叉子树，不过为了便于安排存储空间，一棵多叉子树可以通过引进新结点而表示成一棵二叉子树。

四元式

- 是一种普遍采用的中间代码形式。
- 有四个组成成分：算符op、第一运算对象ARG1、第二运算对象ARG2、运算结果RESULT
- 例：a:=b*c+b*d

  (1) (*, b, c, t_1)

  (2) (*, b, d, t_2)

  (3) (+, t_1, t2, t_3)

  (4) (:=, t_3, -, a)

- **四元式和三元式的主要不同**

  四元式对中间结果的引用必须通过给定的名字而三元式是通过产生中间结果的三元式编号。也就是说，四元式之间的联系是通过临时变量实现的。
- 四元式表示很类似于三地址指令，因为这种表示可以看作是一种虚拟三地址机的通用汇编代码，即这种虚拟机的每条指令包含操作符和三个地址，两个为运算对象的，一个是为结果的。

- 有时，为了更直观，可以把四元式写成简单赋值形式，如：

  (1) t1:=b*c

  (2) t2:=b*d

  (3) t3:=t1+t2

  (4) a:=t3

例：A + B * (C - D) + E / (C - D) ^N
分别用逆波兰、三元式、四元式表示。

例：A + B * (C - D) + E / (C - D) ^N

逆波兰： A B C D - * + E C D – N ^ / +

四元式： (1) (- C D T1)

(2) (* B T1 T2)

(3) (+ A T2 T3)

(4) (- C D T4)

(5) (^ T4 N T5)

(6) (/ E T5 T6)

(7) (+ T3 T6 T7)

例：A + B * (C - D) + E / (C - D) ^N

三元式:

(1) (- C D)

(2) (* B (1))

(3) (+ A (2))

(4) (- C D)

(5) (^ (4) N)

(6) (/ E (5))

(7) (+ (3) (6))

8.4 简单赋值语句的翻译

四元式：(op ,arg1,arg2,result) 或result:=arg1 op arg2

在实际实现中，运算对象和结果或者是一个指针，指向符号表的某一登录项，或者是一个临时变量的整数码。

语义属性：id.name表示id所表示的单词的名称

语义函数：Lookup(id.name)审查id.name是否出现在符号表中，如在，则返回指向该登录项的指针，否则返回nil。

语义过程：emit(gen)表示输出四元式到输出文件上。

newtemp表示生成一个临时变量，每调用一次，生成一新的临时变量。

语义变量：E.place表示存放E的值的存储位置。

((1))S →id:=E

{p:=lookup(id.name);

if p≠nil then

emit (p':='E.place)

else error}

(2) $E \rightarrow E^1+E^2$

{E.place:= newtemp;

emit(E.place ':=' E^1.place '+' E^2.place)}

(3) $E \rightarrow E^1*E^2$

{E.place:= newtemp;

emit(E.place ':=' E^1.place '*' E^2.place)}

(4) E→ - E1

{ E.place:=newtemp;

emit(E.place':=' 'uminus' E1.place)}

(5) E→(E1)

{ E.place:= E1.place}

(6) E→id

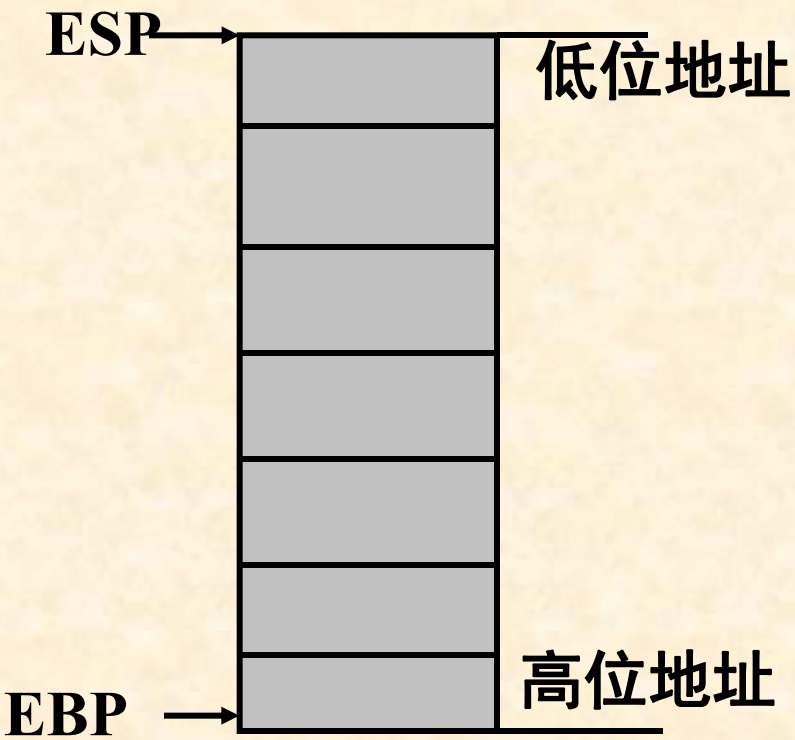
{E.place:=newtemp;

P:=lookup(id.name);

if P≠nil then E.place:=P

else error}

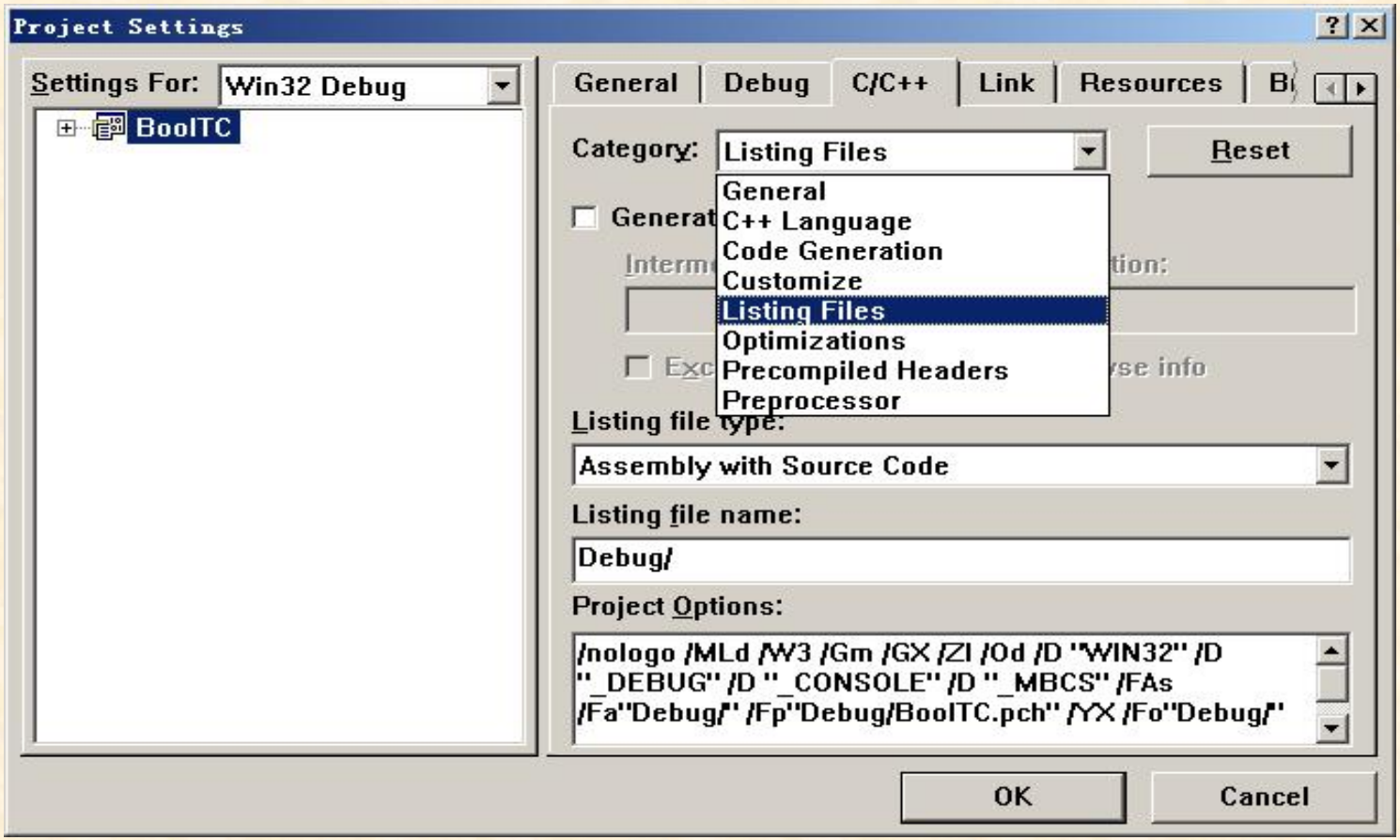
VC++ 中栈的处理



- 栈是程序运行必需的一个存储区域，主要用于存储局部变量。
- 每当系统进入某一函数时，都要为局部变量在栈顶开辟存储空间。
- VC++的debug模式下，除了为局部变量开辟空间外，还会额外开辟64个字节的空间备用）

VC++生成汇编程序的方法

进入Project->settings菜单，如下设置Listing file type：



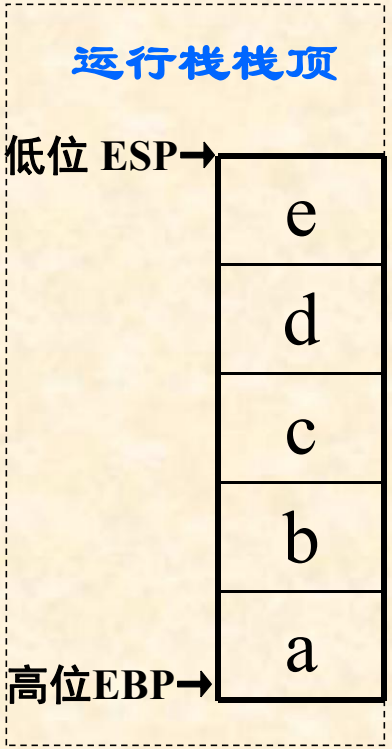
表达式的处理举例

- prt是强制转换指令。dword ptr就是强制将后面的内容转化为一个dword(8086)。
- 意思:根据[ebp]寻找到局部变量b的地址，然后取出其值，放入eax。
- 等于 mov eax, DWORD PTR [ebx+_b$]

```
%d", &a, &b, &c, &d);
```

```
; 5   :   e =a + b / c * d;

 00035   8b 45 f8   mov    eax, DWORD PTR _b$[ebp]
 00038   99         cdq
 00039   f7 7d f4   idiv   DWORD PTR _c$[ebp]
 0003c0f af 45 f0   imul   eax, DWORD PTR _d$[ebp]
 00040   8b 4d fc   mov    ecx, DWORD PTR _a$[ebp]
 00043   03 c8      add    ecx, eax
 00045   89 4d ec   mov    DWORD PTR _e$[ebp], ecx
```




```
_a$ = -4
_b$ = -8
_c$ = -12
_d$ = -16
_e$ = -20
```

- 实际上，在一个表达式中可能会出现不同类型的变量或常数，而不是同一类型。
- 也就是说，编译程序还应该执行这样的语义动作：对表达式中的运算对象应进行类型检查；
- 如不能接受不同类型的运算对象的混合运算，则应指出错误；如果能接受混合运算，则应进行类型转换的语义处理。

8.5 布尔表达式的翻译

- 程序设计语言中的布尔表达式有两个作用：一个是计算逻辑值，更多的情况是二，用做改变控制语句中的条件表达式。
- 布尔表达式是由布尔运算符（and, or和not）施于布尔变量或关系表达式而成。即布尔表达式的形式是E_1 rop E_2，其中E_1和E_2都是算术表达式，rop是关系符，如<=, <, =, >, >=, ≠等等。

8.5.1 布尔表达式的翻译方法

法1. 如同计算算术表达式一样，步步计算出各部分的真假值，最后计算出整个表达式的值：

例：1 or (not 0 and 0) or 0

= 1 or (1 and 0) or 0

= 1 or 0 or 0

= 1 or 0

= 1

法2. 采取某种优化措施，只计算出部分表达式。

这两种方法对于不包含布尔函数调用的表达式是没有什么差别的。

具体翻译方法

- **对于布尔表达式，**

  例：a or b and not c翻译为：

  (1) t_1:=not c

  (2) t_2:=b and t_1

  (3) t_3:= a or t_2

- **对于关系表达式，**

  例：a<b看成if a<b then 1 else 0，翻译为：

  (1) if a<b goto (4)

  (2) t := 0

  (3) goto (5)

  (4) t := 1

  (5) …

用数值表示布尔值的翻译方案

(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→id[1] rop id[2]　　{E.place := newtemp;

emit(‘if’ id[1].place ‘rop’ id[2].place ‘ goto’ nextstat+3);

emit(E.place ‘:=’ ‘0’);

emit(‘goto’ nextstat+2);

emit(E.place ‘:=’ ‘1’}

(5)E→true　　{E.place := newtemp;
emit(E.place ‘:=’ ‘1’)}

(6)E→false　　{E.place := newtemp;
emit(E.place ‘:=’ ‘0’)}

nextstat给出在输出序列中下一四元式序号。emit过程每被调用一次，nextstat增加1。

8.5.2 控制语句中布尔表达式的翻译

- **现在讨论出现在if-then, if-then-else和while-do等语句中的布尔表达式E的翻译。**

  这三种语句的语法为：

  $S \rightarrow$ if E then S^1

  | if E then S^1 else S^2

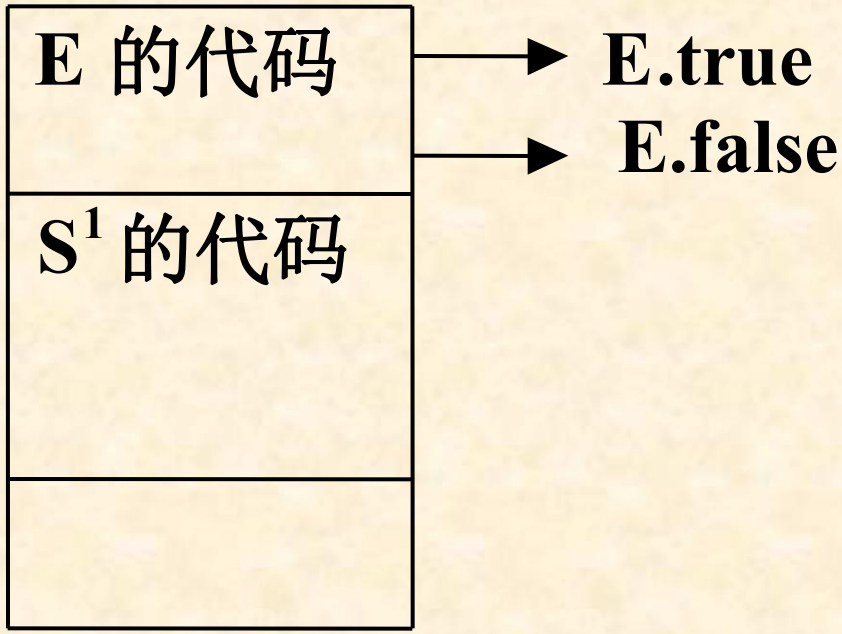
  | while E do S^1

  E.true表示E的“真”出口转移目标

  E.false表示E的“假”出口转移目标

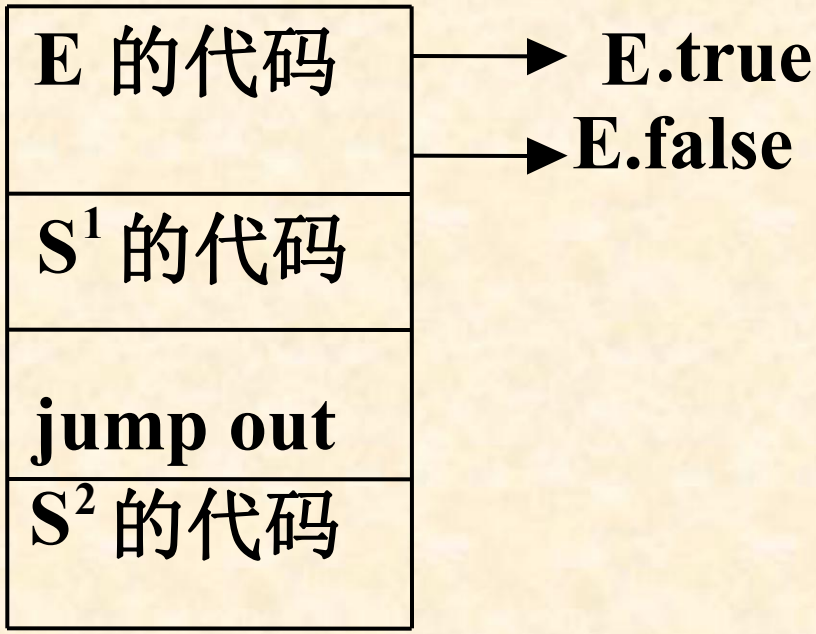
控制语句的代码结构

控制语句　S→if E then S^1



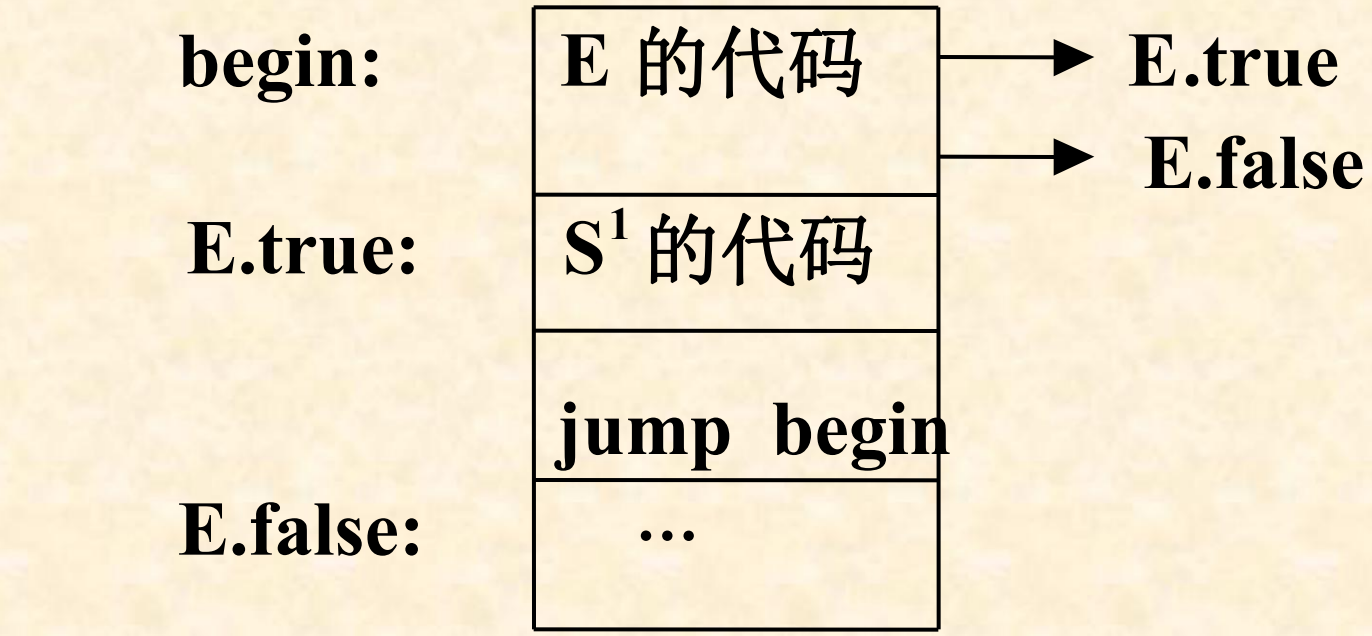
控制语句的代码结构

控制语句 $S \rightarrow$ if E then S^1 else S^2



控制语句的代码结构

控制语句 S→while E do S^1



如：a<b or c<d and e<f

翻译成如下四元式：

(1) if a<b goto E.true
(2) goto (3)
(3) if c<d goto (5)
(4) goto E.false
(5) if e<f goto E.true
(6) goto E.false

（翻译不是最优，（2）是不需要的。）

E.True和E.false的值并不能在产生四元式的同时就知道。

翻译的基本思路

- 作为条件转移的E，仅把E翻译成代码是一串条件转移和无条件转移四元式。
- 对于E为a rop b的形式生成代码为：

  if a rop b goto E.true

  goto E.false
- 对于E为E^1 or E^2的形式，若E^1为真，则可知道E^1的真出口和E的真出口一样。如果E^1为假，那么必须计算E^2，E^1的假出口应是E^2代码的第一个四元式标号，这时E^2的真出口和假出口分别与E的真出口和假出口一样。
- 类似的考虑适于E^1 and E^2和not E^1的情况。

语句 if a<b or c<d and e<f then S^1 else S^2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)(S^1的四元式) // (E.true) 入口

……

(p-1) ……）

(p) goto (q)

(p+1) (S^2的四元式） //(E.false)入口

……

(q-1) ……）

(q)

转移地址在整个布尔表达式的四元式产生完毕之后才得知。

转移地址S^1的四元式产生完毕之后才得知。

➢**拉链技术**—记录需回填地址的四元式。

把需回填E.true的四元式拉成一链，称做“真”链。

把需回填E.false的四元式拉成一链，称为“假”链。

（10） … goto E.true

鏈 …

（20） … goto E.true

鏈 …

（30） … goto E.true

则链成

（10） … goto （0）

鏈 …

（20） … goto （10）

鏈 …

（30） … goto （20）

➢ 把地址（30）称作“真”链链首，0为链尾标志，即地址（10）为“真”链链尾。

➢语义描述使用的变量和过程：

E.true： “真”链，　**E.false：** “假”链

E.codebegin： E 的第一个四元式

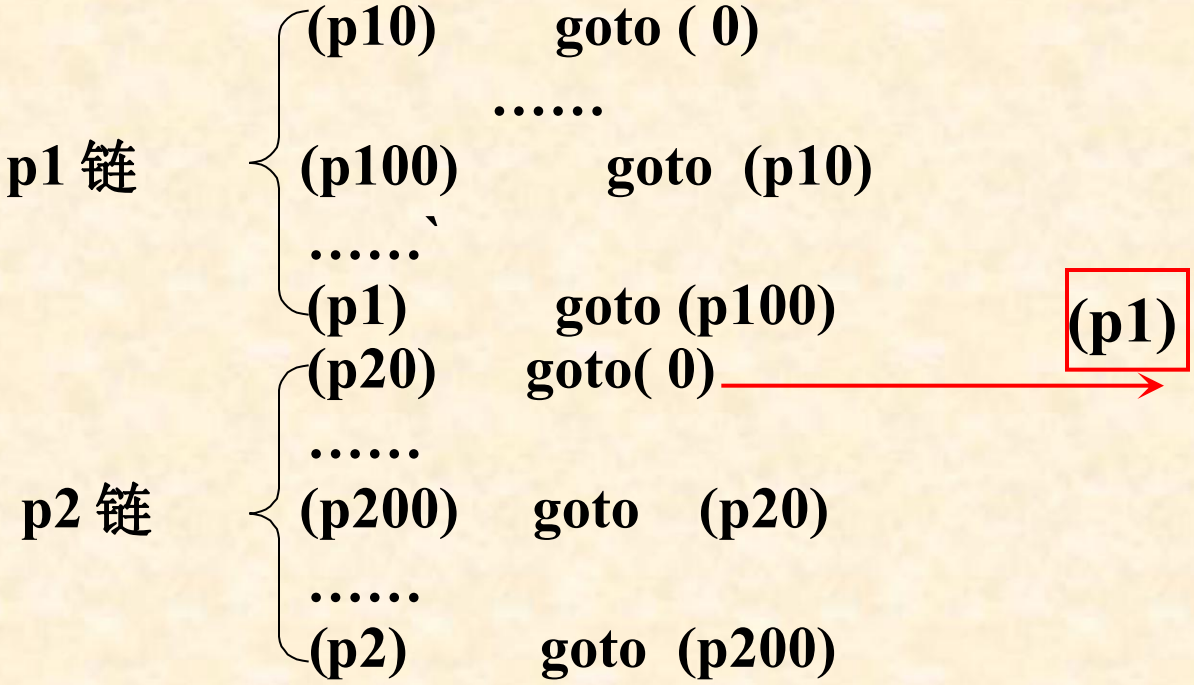
Nextstat： 下一四元式地址

emit( )：输出一条四元式

merge(p1, p2)：把p1的链首填在p2的的尾

backpatch(p, t):把链首p所链接的每个四元式的第四区段都填为t

例：　**merge(p1, p2)**



- 例：对于布尔表达式a<b or c<d and e>f生成的四元式为：

  100：if a<b goto ———

  101：goto 102

  102：if c<d goto 104

  103：goto ———

  104：if e>f goto 100

  105：goto 103

  真链首E.true为104

  假链首E.false为105

8.6 控制语句的翻译

- 条件转移
- 开关语句
- for循环语句
- 出口语句
- goto语句
- 过程调用的四元式产生

8.6.1 条件转移

翻译举例： While (A<B) do if (C<D) then X:=Y+Z

翻译为如下的一串四元式：

100　　if A<B　　goto　102

101　　goto 107

102　　if C<D goto 104

103　　goto 100

104　　T:=Y+Z

105　　X:=T

106　　goto 100

107

8.6.2 开关语句

假定要讨论的开关语句的形式为：

switch E

case V_1: S_1

case V_2: S_2

…

case V_{n-1}: S_{n-1}

default: S_n

end

直观上看，case语句翻译成如下的一连串条件转移语句：

t := E;

L_1: if t≠V_1 goto L_2;

　　S_1;

goto next;

L_2: if t≠V_2 goto L_3;

　　S_2;

goto next;

…

L_{n-1}: if t≠V_{n-1} goto L_n;

　　S_{n-1};

goto next;

L_n: S_n

next:

switch E

case V_1: S_1

case V_2: S_2

…

case V_{n-1}: S_{n-1}

default: S_n

end

开关语句的翻译结果

计算E值，并把结果放到临时变量t 的中间代码

goto test

L_1: S_1的中间代码

goto next

…

L_{n-1}: S_{n-1}的中间代码

goto next

L_n: S_n的中间代码

goto next

Test: if t=V_1 goto L_1

…

if t=V_{n-1} goto L_{n-1}

goto L_n

next:

switch E

case V_1: S_1

case V_2: S_2

…

case V_{n-1}: S_{n-1}

default: S_n

end

8.6.3 for循环语句

- **形式：for i:=E^1 step E^2 until E^3 do S^1**
- **假定E^2总是正的，上述循环语句的ALGOL**
  (Algorithmic Language Algol算法语言 **)意义等价于：**

```
          i:=E1;
          goto OVER;
AGAIN:    i:=i+E2;
OVER:     if i≤E3 then
          begin
                S1;
                goto AGAIN;
          end
```

- 使用如下的产生式来定义文法：

$F_1 \rightarrow$ for i:=E^1

$F_2 \rightarrow F_1$ step E^2

$F_3 \rightarrow F_2$ until E^3

$S \rightarrow F_3$ do S^1

翻译举例

- **循环语句：**

  for I:=1 step 1 until N do M:=M+1

- **翻译为：**

  100　I:=1

  101　goto 103

  102　I:=I+1

  103　if I≤N goto 105

  104　 goto 108

  105　T:=M+1

  106　M:=T

  107　goto 102

  108

8.6.4 出口语句

- 出口语句是指exit或break，是一种结构化的方式退出循环而设置的语句，它们的作用是引起外层循环的终止。
- 无论是哪种语言的出口语句，都要翻译成一转移语句，转移的目标是在break(exit)语句之后循环外层能确定，因此一遍扫描的编译中需要使用回填技术才能给出转移目标。

8.6.5 goto语句

- 多数程序语言中的转移是通过标号和goto语句实现的。

  带标号的语句的形式是L:S;

  goto语句的形式是goto L。
- 当L:S;语句被处理之后，标号L是“定义了”的。即在符号表中，将登记L的项的“地址”栏中登上语句S的第1个四元式的地址。
- 如果goto语句是一个向上转移的语句，那么，当编译程序碰到这个语句时，L必是已定义了的。通过对L查找符号表获得它的定义地址p，编译程序可立即产生出相应于这个goto L的四元式如(j, -, -, p)。

- 如果goto L是一个向下转移的语句，也就是说，标号L尚未定义，那么，若L是第1次出现，则把它填进符号表中并标志上“未定义”。由于L尚未定义，对goto L只能产生一个不完全的四元式(goto, -)，它的转移目标须待L定义时在回填回去。在这种情况下，必须把所有那些以L为转移目标的四元式的地址全都记录下来，可以采用拉链的方法，把所有以L为转移目标的四元式串在一起。

8.6.6 过程调用的四元式产生

- 过程调用的实质是把程序控制转移到子程序(过程段)。
  - 传参：在转子程序之前必须用某种办法把实在参数的信息传给被调用的子程序
  - 记录返回地址：应该告诉子程序在它工作完毕后返回到什么地方。
- 返回地址问题：转子程序指令大多在实现转移的同时就把返回地址放在某个寄存器或内存单元之中
- 传参问题：将实在参数的值或者地址传给形式参数，由子程序进行引用操作。

- 在被调用的子程序(过程)中，每个形式参数都有一个内存单元(称为**形式单元**)，其地位相当于局部变量。
- 传递实在参数的值(传值)的处理方式：
  - 将实在参数的值赋给对应的形式参数。
- 传递实在参数地址(传地址)的处理方式：
  - **变量或数组元素：**直接传递它的地址。
  - **无地址的表达式：**如A+B或2，那么就先把它的值计算出来并存放到某个临时单元T中，然后传送T的地址，所有实在参数的地址应存放在被调用的子程序能够取得到的地方。

8.7 说明语句的翻译

- 程序设计语言中的说明语句旨在定义各种形式的有名实体，如变量、常量、数组、记录(结构)、过程、子程序等等。
- 编译程序把说明语句中定义的名字和性质登记在符号表中，用以检查名字的引用和说明是否一致。许多说明语句的翻译并不生成目标代码。

简单说明语句的翻译

文法：

D→integer<namelist>｜real<namelist>

<namelist>→<namelist>，id｜id

用自下而上翻译存在着一个问题，只能把所有的名字都归约为namelist之后才能把它们的性质登记进符号表。

文法改写：

D→D[1]，id

｜integer id

｜real id填

属性文法

(1) D → integer id { enter(id,int); D.att:=int}

(2) D → real id { enter(id,real); D.att:=real}

(3) D → D^1 ,id { enter(id, D^1.att) ;

D.att:= D^1.att }

8.7.2 过程中的说明语句

- 处理到过程的说明部分时，要为过程的局部名字安排存储。对这些名字建立符号表项时，要记录名字和存储的相对地址。
- 用全程变量offset表示变量在本过程的数据区的相对位置,增加的量为数据对象的宽度,用属性width表示.

P→D　　{ offset:=0}

D→….

D→real id｛enter(id,real,offset）;D.att:=real;

D.width:=8; offset:=offset+D.width}

…

复 习

- 中间代码的形式
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制结构（条件、开关、循环、goto、过程调用）的翻译
- 说明语句的翻译

课堂练习：将下列语句翻译成四元式（重点）

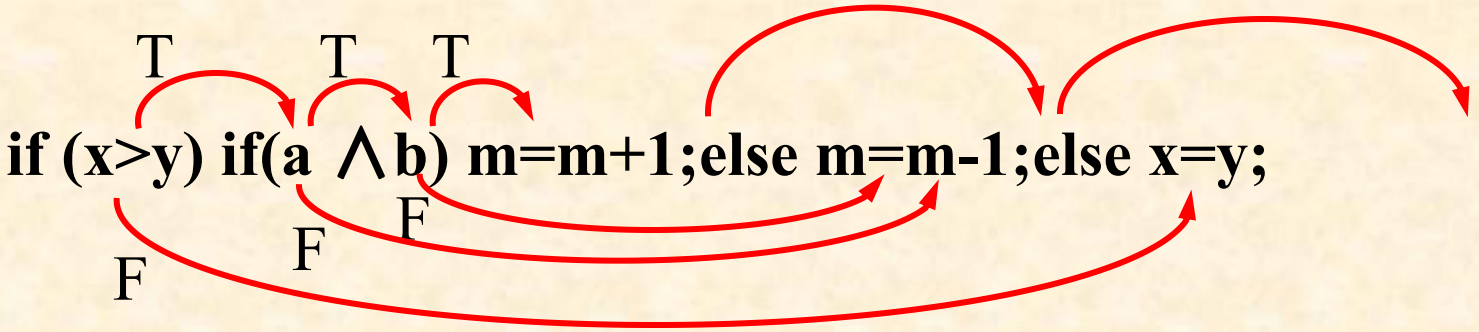
if (x>y) if(a ∧b) m=m+1;else m=m-1;else x=y;

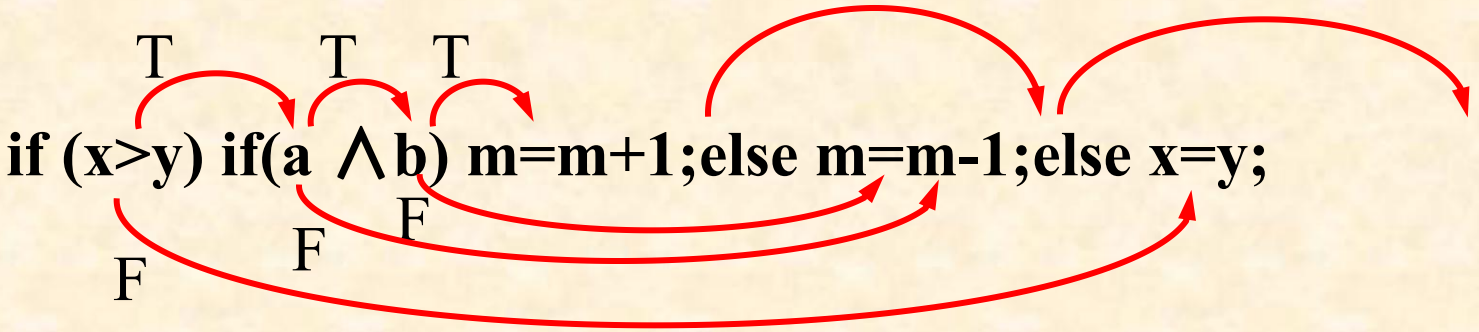
[解答]：解题技巧：不需要根据语义子程序，只需要画出代码结构图，根据转换关系来翻译四元式：

一个布尔分量对应 2 个四元式；

else对应 1 个无条件跳转四元式；

注意真假出口的预留与回填。





100: If x>y goto 102

101: Goto 109

103: If a^b goto 105

104: Goto 107

105: m=m+1;

106: Goto 110

107: m=m-1;

108: goto 110

109: x=y;

110:

课堂练习：将下列语句翻译成四元式（重点）

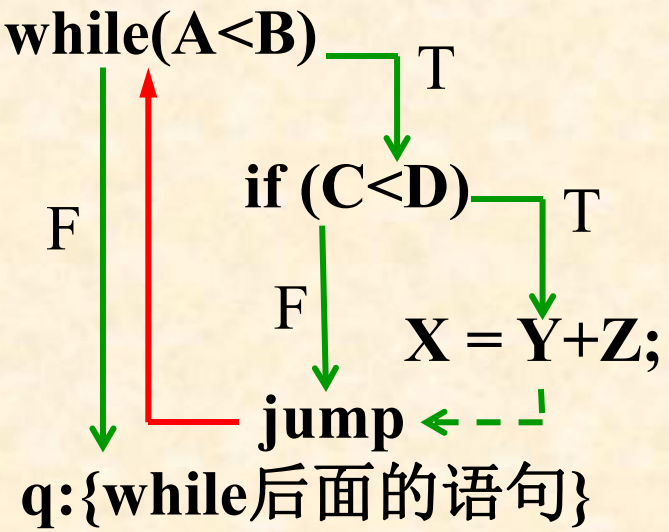
while(A<B) if (C<D) X = Y+Z;

[解答]：只需要画出代码结构图，根据转换关系来翻译四元式：

一个布尔分量对应　　2 个四元式；

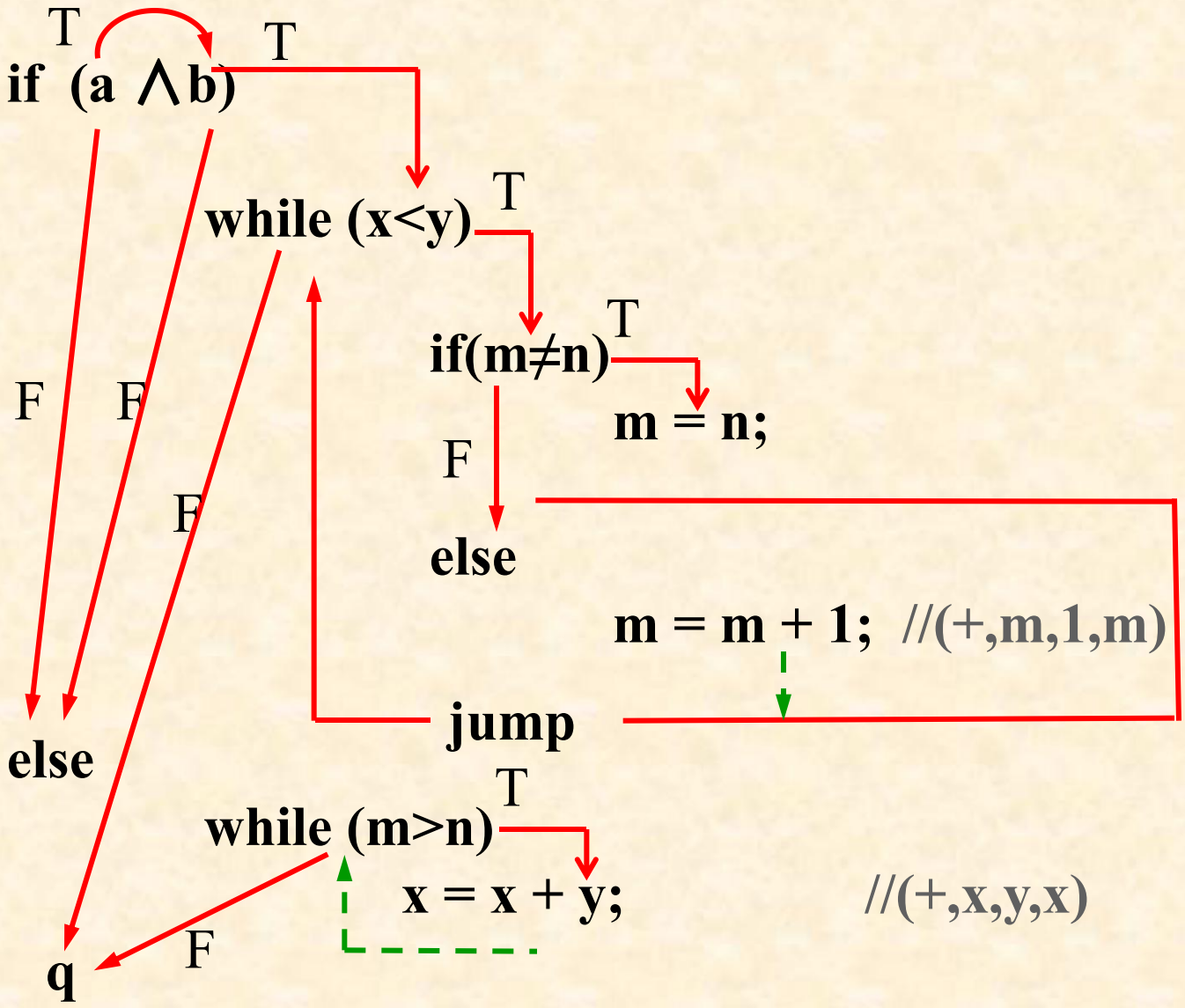
else对应　　1 个无条件跳转四元式；

while的循环体S最后必须有　　1个强制跳转四元式。




```
100:  if A<B  goto 102
101:  goto  106
102:  if C<D  goto 104
103:  goto 100
104:   x=y+z;
105:  goto  100;
106:
```

课堂练习：将下列语句翻译成四元式（重点）



100: if a^b goto 102
101: goto 110
102: if x<y goto 104
103: goto 114
104: if m≠n goto 106
105: goto 108
106: m=n
107: goto 102
108: m=m+1
109: goto 102
110: If m>n goto 112
111: goto 114
112: x=x+y
113: goto 102
114:

8.8 数组和结构的翻译

- 数组说明和数组元素的引用
- 结构（记录）说明和引用的翻译

8.8.1 数组说明和数组元素的引用

- 所谓数组，就是相同数据类型的元素按一定顺序排列的集合。
- 沿着某一维的距离称为一个下标，每维的下标只能在该维的上、下限之内变动。
- 数组的每个元素在数组中的位置可以通过每维的下标来确定
- 如果一个数组所需的存储空间的大小在编译时就已经知道，此数组是一个确定数组或静态数组，否则，称为可变数组或动态数组。

- 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。

8.8.1 数组说明和数组元素的引用

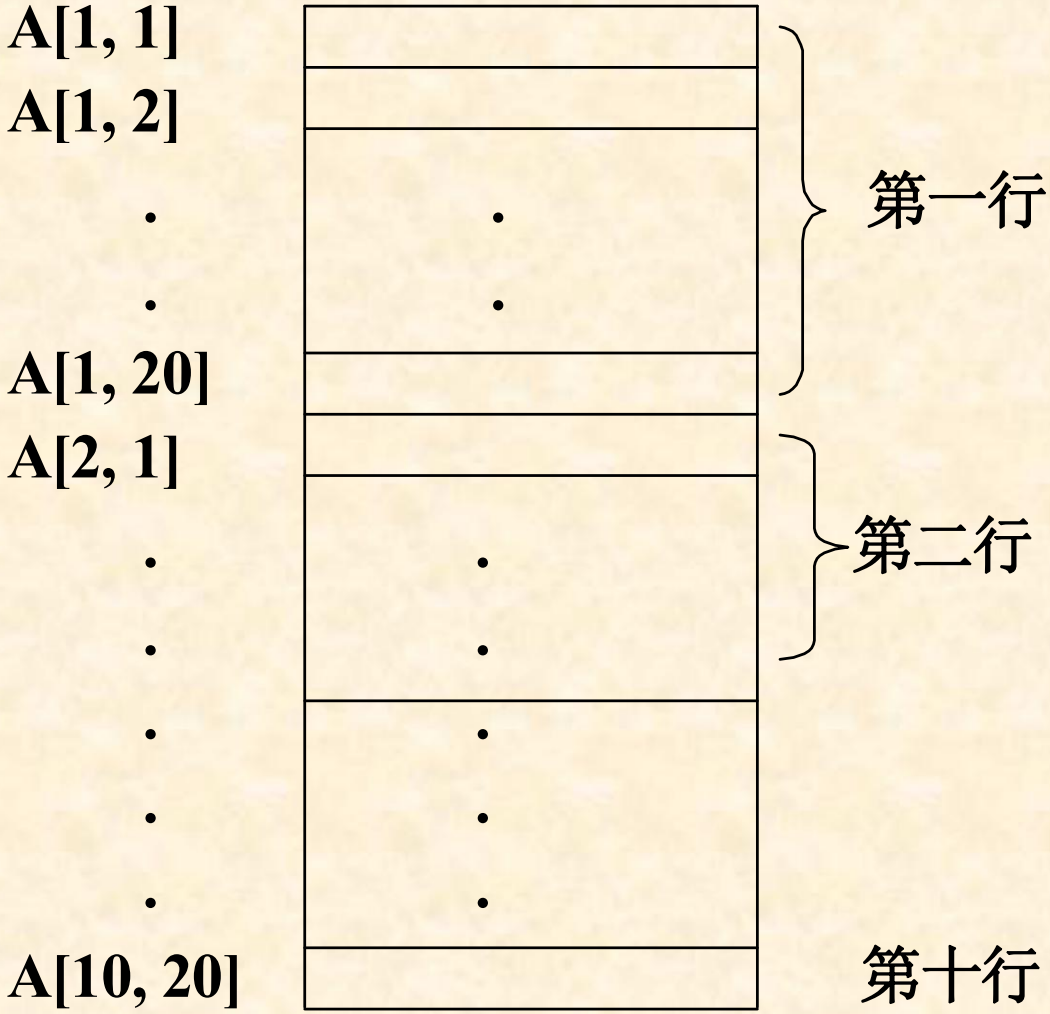
. 数组内情向量:

编译中，将数组的有关信息记录在一些单元中，称为“内情向量”，确定数组，放在符号表中；可变数组，运行时建立相应的内情向量。

每个数组有一个内情向量，其中的信息包括：

数据的类型、维数、各维的上下界、数据的首地址。

例：按行　　A 是 10×20 的二维数组



数组元素的地址计算

设 A[1，1]的地址为 a，每个元素占一个字

A[i，j]的地址： a+（i-1）×20+（j-1）

=（a-21）+（20i+j）

一般：arrayA[l_1：u_1，l_2：u_2，……，l_n：u_n]

令 $d_i=u_i-l_i+1$

元素 A[i_1，i_2，……，i_n]的地址 D

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

经因子分解后得 D=CONSPART+VARPART

其中 CONSPART=a-C

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

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

将产生两组计算数组元素地址的四元式

一组计算ＶＡＲＰＡＲＴ，将它放在某个临时单元Ｔ中；

一组计算ＣＯＮＳＰＡＲＴ，放在另一个临时单元T1中。

同时用表示数组元素的地址。

对应“数组元素引用”（引用其值）和“对数组元素赋值”有两个相应的四元式：“变址取数”和“变址存数”。

“变址取数”的四元式是：

（＝［　］，　T1[T]，－，Ｘ）

珧相当于Ｘ：＝T1[T] 珧

“变址存数”的四元式是：

（［　］＝，　Ｘ，疒－，　T1[T]）

珧相当于T1[T] :=x珧

例如，令A是一个10*20的数组，即d_1=10, d_2=20。那么，赋值语句X:=A[I, J]的四元式序列为：

(*, I, 20, T_1)　　20指d2

(+, J, T_1, T_1)　　T1为VARPART

(-, A, 21, T_2)　　T2=a-C

(=[], T_2[T_1], -, T_3)　　T3:=T2[T1]

(:=, T_3, -, X)

8.8.2 结构(记录)说明和引用的翻译

- 结构记录是由已知类型的数据组和起来的一种数据类型。
- 结构说明的文法：

<type>→struct {f_1};

| int

| char

| pointer

f_1→f_1;f | f

f→<type> i | <type> i[n]

- 编译时，必须记录所有分量的信息，假定带有n个分量的结构按以下形式登记在一张单独的表中：

分量1	分量2	……	分量n

每个分量记有名、类型、长度，(所需的存储单元数)，以及前面各分量的长度总和(用offset表示)

- 由于记录了每个分量的类型和区距，因此每个分量的地址很容易计算。

总结

- 属性文法
- 语法制导翻译
- 中间代码的形式
- 各种语句的翻译：

赋值语句、布尔表达式、

控制结构、说明语句、数组和结构