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

8.1.2 符号的常见属性

8.1.3 符号表的实现

8.1.4 符号表体现作用域与可见性

8.1.1 符号表的作用

- 在语义分析阶段，编译器计算语义信息，并根据这些信息完成静态语义分析，进而生成后续的中间表示形式。
- 符号表是存储语义信息的重要数据结构，几乎在编译的每个阶段每一步都涉及符号表。

符号表的功能

- 自创建后便开始收集符号属性信息：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。
- 在语义分析中，符号表所登记的内容是进行上下文语义的合法性检查的依据。
- 在目标代码生成阶段，符号表是对符号名进行地址分配的依据：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 符号的常见属性

- 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、符号的类别

- 符号的类别代表符号属于哪一种。
- 主要分为：常量符号，变量符号，过程/函数符号，类名符号等不同的类别。

3、符号的类型

- **符号的类型指该符号的数据类型。**
- **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。**

4、符号的存储类别和存储分配信息

- **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。**

- 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，动态存储区又分为栈区和堆区。
- 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；
- 偏移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	

5、符号的作用域及可见性

- **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(可见的)，这就是变量可见性的作用域规则。**
- **但是变量可见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(floate 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

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 符号表体现作用域与可见性

- 作用域：每一个符号在程序中都有一个确定的有效范围。拥有共同有效范围的符号所在的程序单元就构成了一个作用域。
- 嵌套的作用域：作用域之间可以嵌套，但不会交错
- 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
  - 不属于开作用域的作用域称为闭作用域

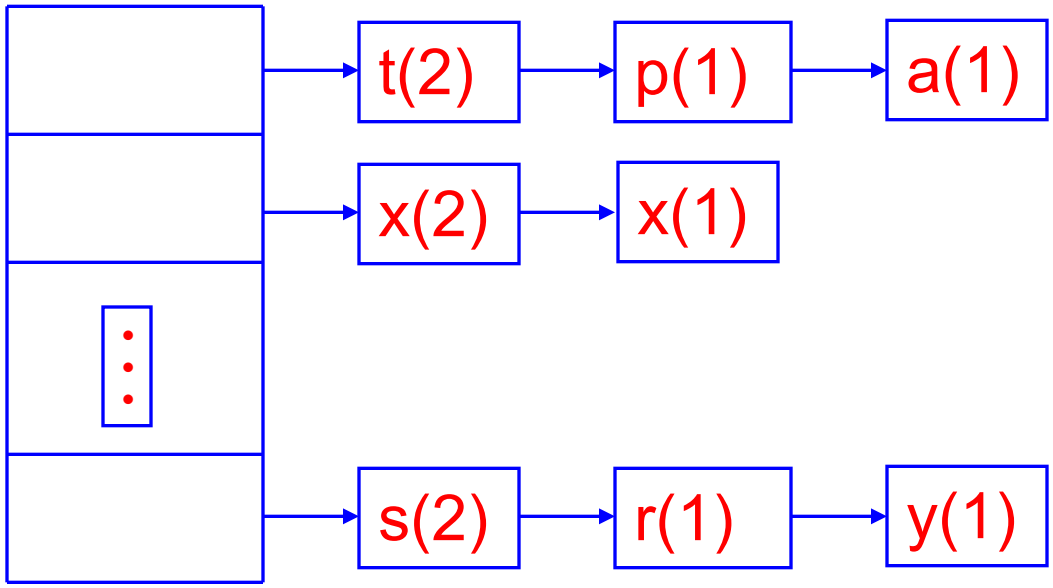
- 可见性：指在程序的某一特定点哪些符号是可以访问的。
- 程序语言中常见的常用的可见性规则（*visibility rules*）
  - 在程序的任何一点，只有在该点的开作用域中声明的名字才是可访问的
  - 若一个名字在多个开作用域中被声明，则把离该名字的某个引用最近的声明作为该引用的解释
  - 新的声明只能出现在当前作用域

8.1.4.1 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.
```

作用域（p）　作用域（t）　作用域（r）　作用域（整个程序）

✧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.
```

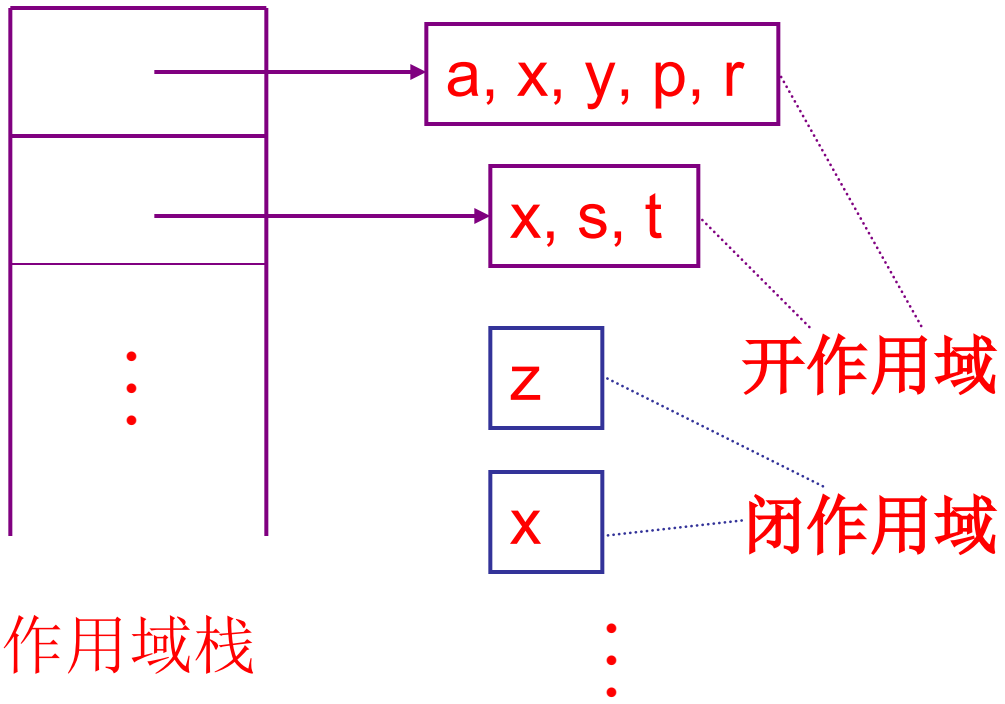
作用域　作用域　作用域　作用域

8.1.4.2 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  v;
      begin
         ……
      end;
   begin /*here*/
      ……
   end;
begin
   ……
end.
```

作用域 作用域 作用域 作用域

8.2 静态语义分析

与语义分析相关的工作

- **静态语义检查**
  - 编译期间所进行的语义检查
- **动态语义检查**
  - 所生成的代码在运行期间进行的语义检查
- **收集语义信息**
  - 为语义检查收集程序的语义信息
  - 为代码生成等后续阶段收集程序的语义信息

8.2.1 静态语义分析的主要任务

- **代码生成前程序合法性检查的最后阶段**

  - **控制流检查**（*flow-of-control checks*）
    控制流语句必须使控制转移到合法的地方（如 *break* 语句必须有合法的语句包围它）
  - **唯一性检查**（*uniqueness checks*） 很多场合要求对象只能被定义一次（如枚举类型的元素不能重复出现）
  - **名字的上下文相关性检查**（*name-related checks*） 某些名字的多次出现之间应该满足一定的上下文相关性
  - **类型检查**（*type checks*）
    检查每个操作是否遵守语言类型系统的定义

8.2.2 类型检查

- **类型检查程序**（*type checker*）负责类型检查工作，主要包括一下内容：
  - 验证语言结构是否匹配上下文所期望的类型
  - 为相关阶段搜集及建立必要的类型信息
  - **实现**某个**类型系统**（*type system*）

一个简单语言

- 以下是定义某个简单语言的上下文无关文法（将用于本讲的设计示例）$G[P]$：

$P \rightarrow D\ ;\ S$

$D \rightarrow V\ ;\ F$

$V \rightarrow V\ ;\ T\ L \mid \varepsilon$

$T \rightarrow$ boolean $\mid$ integer $\mid$ real $\mid$ array [num] of T $\mid$ ^T

$L \rightarrow L\ ,\ \underline{\text{id}} \mid \underline{\text{id}}$

$S \rightarrow \underline{\text{id}} := E \mid$ if E then $S \mid$ if E then S else $S \mid$ while E then S

$\mid S\ ;\ S \mid$ break $\mid$ call $\underline{\text{id}}\ (\ A\ )$

$E \rightarrow$ true $\mid$ false $\mid \underline{\text{int}} \mid \underline{\text{real}} \mid \underline{\text{id}} \mid E\ \underline{\text{op}}\ E \mid E\ \underline{\text{rop}}\ E \mid E[E] \mid E\text{^}$

$F \rightarrow F\ ;\ \underline{\text{id}}\ (\ V\ )\ S \mid \varepsilon$

$A \rightarrow A\ ,\ E \mid \varepsilon$

8.2.2.1 类型表达式和类型系统

- **类型表达式（*type expressions*）**
  - 由**基本类型**，**类型名字**，**类型变量**，及**类型构造子**（*type constructor*）归纳定义的表达式
- **类型系统（*type systems*）**
  - 将类型表达式赋给程序各个部分的**规则集合**

类型表达式 — 举例

- **类型表达式**
  - **分四类定义**

    基本数据类型表达式，积类型表达式，过程类型表达式，专用类型表达式

- **基本数据类型表达式**
  - **纯量类型表达式：*bool*, *int*, *real***
  - **有界数组类型表达式：*array(I,T)***

    T* ∈ { *bool, int, real* }；*I* 代表一个整数区间，如 *1..10
  - **指针数据类型表达式：*pointer(T)***

    ***T* ∈ { *bool, int, real* }**

类型表达式 — 举例（续）

- **积类型表达式**
  - $<T_1, T_2, \ldots, T_n>$

    $T_1, T_2, \ldots, T_n$ 为上述数据类型表达式；若n=0，则表示为 < >
- **过程类型表达式**
  - *fun*（*T*）

    T 是上述积类型表达式
- **专用类型表达式**
  - *type_error*　专用于有类型错误的程序单元
  - *ok*　专用于没有类型错误的程序单元

8.2.2.2 语法制导的类型检查

举例

- 处理声明的翻译模式

$V \rightarrow V_1 ; T \quad \{ L.in := T.type \}$

$\quad L \quad \{ V.type := make\_product\_3\ (V_1.type, T.type, L.num) \}$

$V \rightarrow \varepsilon \quad \{ V.type := <> \}$

$T \rightarrow$ boolean $\quad \{ T.type := bool \}$

$T \rightarrow$ integer $\quad \{ T.type := int \}$

$T \rightarrow$ real $\quad \{ T.type := real \}$

$T \rightarrow$ array [num] of $T_1 \quad \{ T.type := array(1..\ num.lexval, T1.type) \}$

$T \rightarrow \uparrow T_1 \quad \{ T.type := pointer(T1.type) \}$

$L \rightarrow \{ L_1.in := L.in \}\ L_1$, id $\{ addtype(id.entry, L.in) ; L.num := L_1.num + 1 \}$

$L \rightarrow$ id $\quad \{ addtype(id.entry, L.in); L.num := 1 \}$

举例

- 处理表达式的翻译模式

$E \rightarrow true$ { $E.type := bool$ }

$E \rightarrow false$ { $E.type := bool$ }

$E \rightarrow \underline{int}$ { $E.type := int$ }

$E \rightarrow \underline{real}$ { $E.type := real$ }

$E \rightarrow \underline{id}$ { $E.type :=$ *if* $lookup\_type(\underline{id}.name) = nil$
then type_error
else $lookup\_type(\underline{id}.name)$ }

$E \rightarrow E_1$ **op** E_2 { *E.type* := **if** E_1.*type*=*real* **and** E_2.*type*=*real* **then** *real*
else if E_1.*type*=*int* **and** E_2.*type*=*int* **then** *int*
else *type_error* }

$E \rightarrow E_1$ **rop** E_2 { *E.type* := **if** E_1.*type*=*real* **and** E_2.*type*=*real* **then** *bool*
else if E_1.*type*=*int* **and** E_2.*type*=*int* **then** *bool*
else *type_error* }

$E \rightarrow E_1 [E_2]$ { *E.type* := **if** E_2.*type*= *int* **and** E_1.*type*=*array*(*s*, *t*) **then** *t*
else *type_error* }

$E \rightarrow E_1$ ^ { *E.type* := *if* E_1.*type*= *pointer*(*t*) **then** *t*
else *type_error* }

➢ 处理语句、过程声明及程序的翻译模式

$S \rightarrow \underline{\text{id}} := E$ { *S.type* := if *lookup_type* (id.*entry*) = *E.type* then *ok* else *type_error* }

$S \rightarrow$ if E then S_1 { *S.type* := if *E.type=bool* then S_1.*type* else *type_error* }

$S \rightarrow$ if E then S_1 else S_2
{ *S.type* := if *E.type=bool* and S_1.*type* = *ok* and S_2.*type* = *ok* then *ok* else *type_error* }

$S \rightarrow$ while E then S_1 { *S.type* := if *E.type=bool* then S_1.*type* else *type_error* }

$S \rightarrow S_1 ; S_2$ { *S.type* := if S_1.*type* = *ok* and S_2.*type* = *ok* then *ok* else *type_error* }

$S \rightarrow$ break { *S.type* := *ok* }

$S \rightarrow$ *call* $\underline{id}$ (A)
{ $S.type$:= if *match* (*lookup_type* ($\underline{id}.name$), $A.type$)
then *ok else type_error* }

$F \rightarrow F_1$; $\underline{id}$ (V) S { *addtype*($\underline{id}.entry$, *fun* ($V.type$));
$F.type$:= if $F_1.type = ok$ and $S.type = ok$
then *ok else type_error* }

$F \rightarrow \varepsilon$ { $F.type := ok$ }

$A \rightarrow A_1$, E { $A.type$:= *make_product_2* ($A_1.type$, $E.type$) }

$A \rightarrow \varepsilon$ { $A.type$:= <> }

$P \rightarrow D$; S { $P.type$:= if $D.type = ok$ and $S.type = ok$
then *ok* else *type_error* }

$D \rightarrow V$; F { $D.type := F.type$ }

- **增加处理：*break* 只能在某个循环语句内部**

$P \rightarrow D$; { $S.break := 0$ }
S { $P.type :=$ if $D.type = ok$ and $S.type = ok$
then ok else $type\_error$ }

$S \rightarrow$ if E then { $S_1.break := S.break$ }
S_1 { $S.type :=$ if $E.type = bool$ then $S_1.type$ else $type\_error$ }

$S \rightarrow$ if E then { $S_1.break := S.break$ } S_1
else { $S_2.break := S.break$ } S_2
{ $S.type :=$ if $E.type = bool$ and $S_1.type = ok$ and $S_2.type = ok$
then ok else $type\_error$ }

$S \rightarrow$ **while** E **then** { S_1.*break* := 1 } S_1
{ *S.type* := **if** *E.type*=*bool* **then** S_1.*type* **else** *type_error* }

$S \rightarrow$ { S_1 .*break* := *S.break* } S_1 ; { S_2 .*break* := *S.break* } S_2
{ *S.type* := **if** S_1.*type* = *ok* **and** S_2.*type* = *ok*
then *ok* **else** *type_error* }

$S \rightarrow$ **break** { *S.type* := **if** *S.break* = 1
then *ok* **else** *type_error* }

$F \rightarrow F_1$; <u>id</u> (V) { *S.break* := 0 }
S { *addtype*(<u>id</u>.*entry*, *fun* (*V.type*));
F.type := **if** F_1.*type* = *ok* **and** *S.type* = *ok*
then *ok* **else** *type_error* }

8.3中间代码生成

➢ 中间代码

- 源程序的不同表示形式
- 作用
  - 源语言和目标语言之间的桥梁，避开二者之间较大的语义跨度，使编译程序的逻辑结构更加简单明确
  - 利于编译程序的重定向
  - 利于进行与目标机无关的优化

8.3.1 常见的中间表示形式

- 有不同层次不同目的之分
- 中间代码举例
  - *AST*（*Abstract syntax tree*，抽象语法树）
  - *TAC*（*Three-address code*, 三地址码，四元式）
  - *P-code* （特别用于 *Pasal* 语言实现）
  - *Bytecode*（*Java* 编译器的输出, *Java* 虚拟机的输入）
  - *SSA*（*Static single assignment form*，静态单赋值形式）

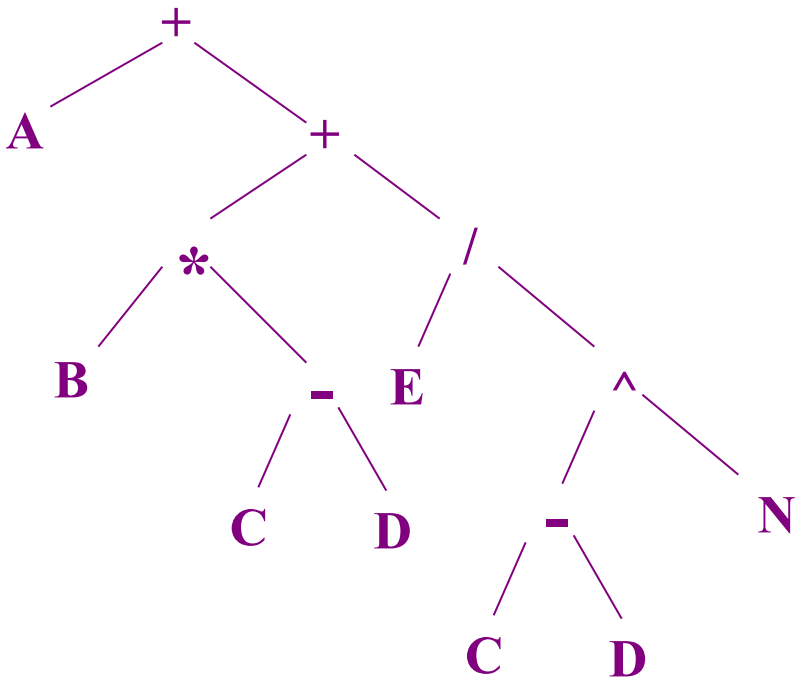
中间代码举例

- **算术表达式 A + B * (C - D) + E / (C - D) ^N**
  - ***TAC*（三地址码）表示**

(1)	(- C D T_1)		T_1 := C - D
(2)	(* B T_1 T_2)		T_2 := B * T_1
(3)	(+ A T_2 T_3)		T_3 := A + T_2
(4)	(- C D T_4)	或	T_4 := C - D
(5)	(^ T_4 N T_5)		T_5 := T_4 ^ N
(6)	(/ E T_5 T_6)		T_6 := E / T_5
(7)	(+ T_3 T_6 T_7)		T_7 := T_3 + T6

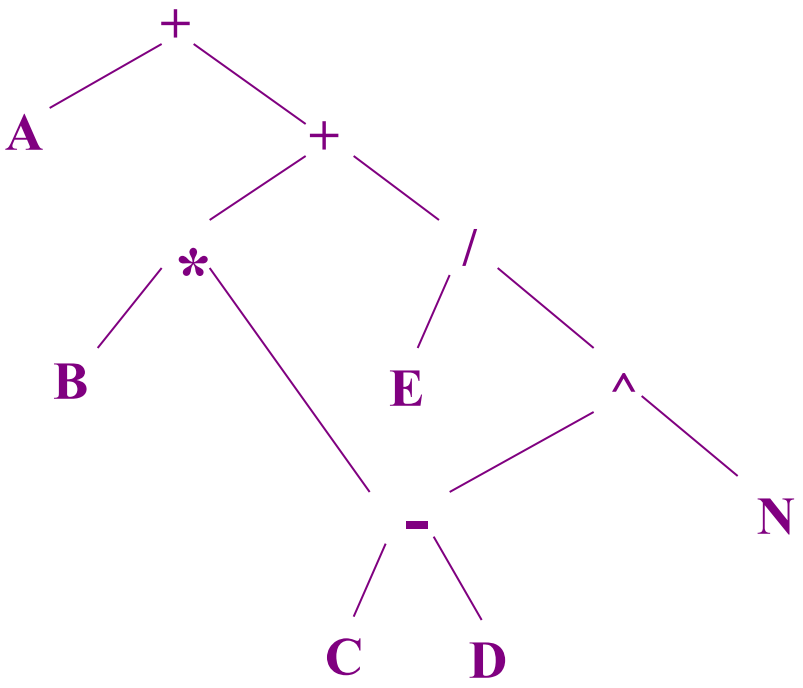
中间代码举例

- **算术表达式 A + B * (C - D) + E / (C - D) ^N**
  - ***AST*（抽象语法树）表示**



中间代码举例

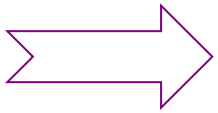
- **算术表达式 A + B * (C - D) + E / (C - D) ^N**
  - ***DAG*（*Directed Acyclic Graph*，有向无圈图，改进型 *AST*）**



中间代码举例

- **静态单赋值形式：程序中的名字仅有一次赋值**

```
x ← 5
x ← x - 3
if x < 3
then
    y ← x * 2
    w ← y
else
    y ← x - 3
w ← x - y
z ← x + y
```



```
x1 ← 5
x2 ← x1 - 3
if x2 < 3
then
    y1 ← x2 * 2
    w1 ← y1
else
    y2 ← x2 - 3
y3 ← φ(y1 , y2)
w2 ← x2 - y3
z ← x2 + y3
```

8.3.2 生成抽象语法树

mknode: 构造内部结点
Mkleaf : 构造叶子结点

举例：

$S \rightarrow \underline{\text{id}} := E$	{ *S.ptr* := *mknode*('assign', *mkleaf*(<u>id</u>.*entry*), *E.ptr*) }
$S \rightarrow \text{if } E \text{ then } S_1$	{ *S.ptr* := *mknode*('if_then', *E.ptr*, S_1.*ptr*) }
$S \rightarrow \text{while } E \text{ do } S_1$	{ *S.ptr* := *mknode*('while_do', *E.ptr*, S_1.*ptr*) }
$S \rightarrow S_1 ; S_2$	{ *S.ptr* := *mknode*('seq' , S_1.*ptr* , S_2.*ptr*) }
$E \rightarrow \underline{\text{id}}$	{ *E.ptr* := *mkleaf*(<u>id</u>.*entry*) }
$E \rightarrow E_1 + E_2$	{ *E.ptr* := *mknode*('+' , E_1.*ptr* , E_2.*ptr*)}
$E \rightarrow E_1 * E_2$	{ *E.ptr* := *mknode*('*' , E_1.*ptr* , E_2.*ptr*)}
$E \rightarrow (E_1)$	{ *E.ptr* := E_1.*ptr* }
……	……

8.3.3 生成三地址码

三地址码*TAC*

– 顺序的语句序列 其语句一般具有如下形式

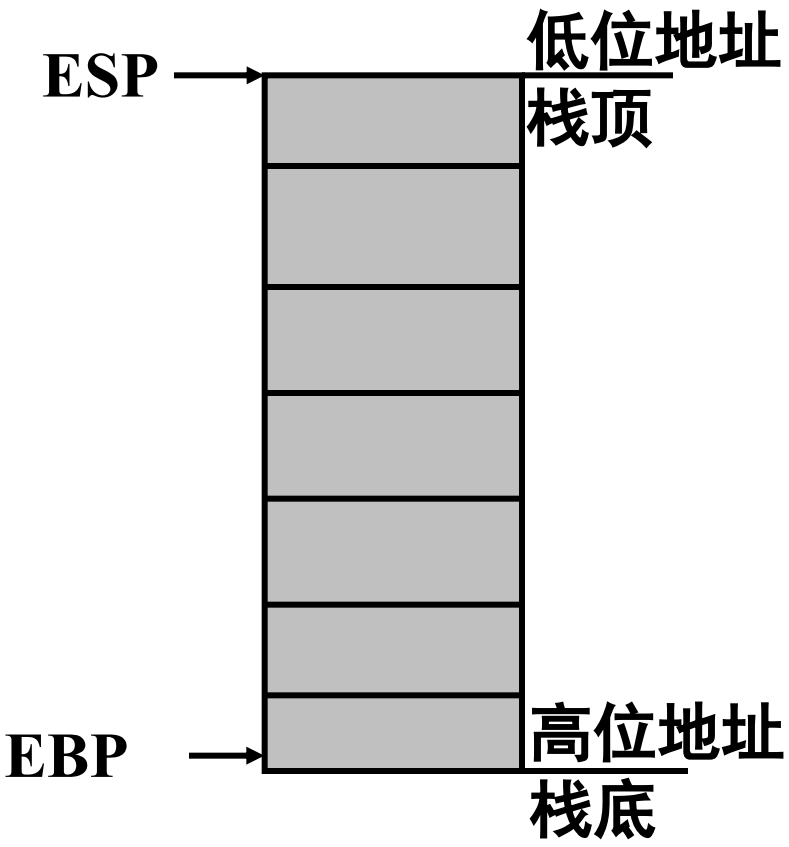
$x := y\ op\ z$

(*op* 为操作符，*y* 和 *z* 为操作数， *x* 为结果)

课程后续部分用到的 *TAC* 语句类型

- 赋值语句 $x := y\ op\ z$ （*op* 代表二元算术/逻辑运算）
- 赋值语句 $x := op\ y$ （*op* 代表一元运算）
- 复写语句 $x := y$ （*y* 的值赋值给 *x*）
- 无条件跳转语句 *goto L* （无条件跳转至标号 *L*）
- 条件跳转语句 *if x rop y goto L* （*rop* 代表关系运算）
- 标号语句 *L*：（定义标号 *L*）
- 过程调用语句序列 $param\ x_1 \ldots param\ x_n\ call\ p,n$
- 过程返回语句 *return y* （*y* 可选，存放返回值）
- 下标赋值语句 $x := y[i]$ 和 $x[i] := y$ （前者表示将地址 *y* 起第*i*个存储单元的值赋给 *x*，后者类似）
- 指针赋值语句 $x := *y$ 和 $*x := y$

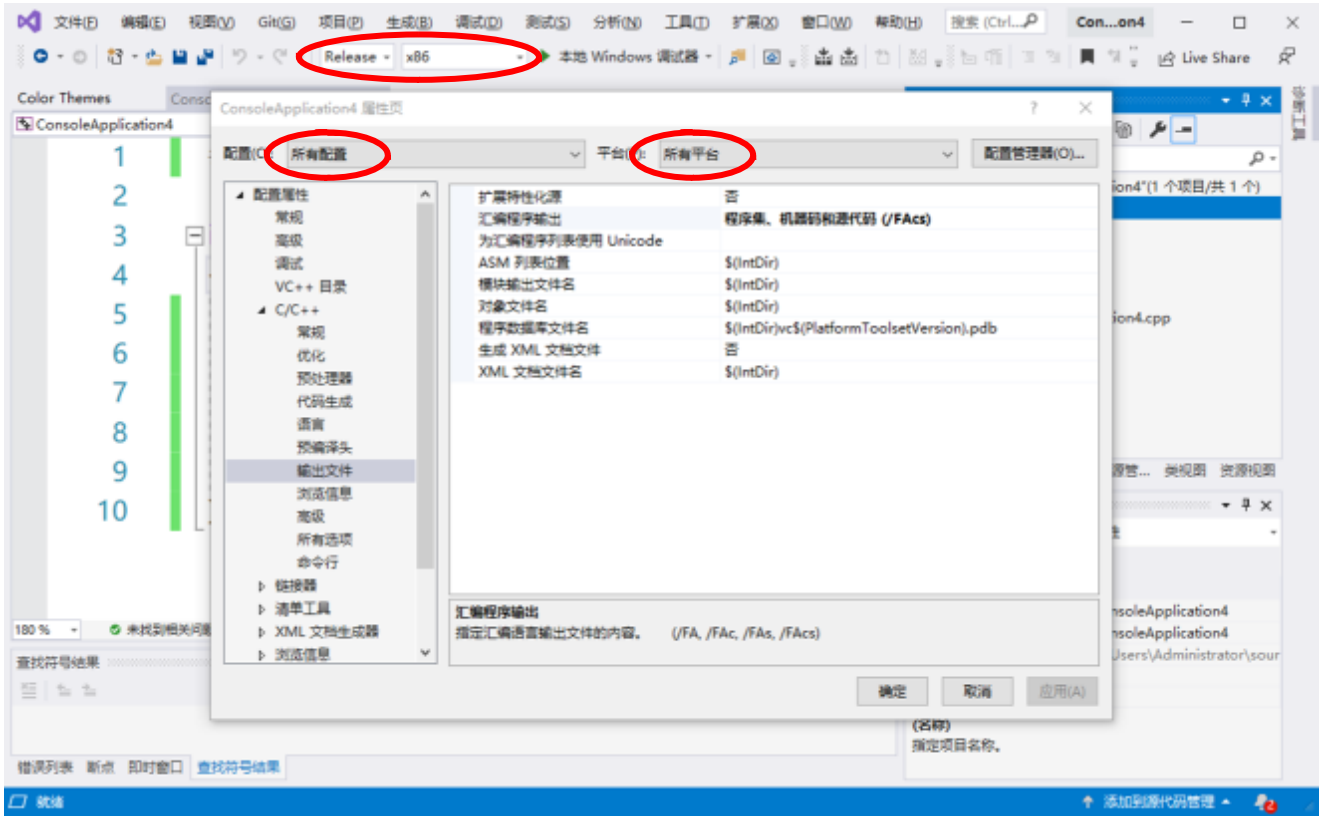
Windows栈结构



- 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。
- ESP：栈指针寄存器(extended stack pointer)，内部存放着一个指针，该指针永远指向系统栈最上面一个栈的栈顶。
- EBP：基址指针寄存器(extended base pointer)，内部存放着一个指针，该指针永远指向系统栈最上面一个栈的栈底。
- 在Windows中，栈底为高位地址，栈顶为地位地址，入栈为高位地址向低位地址进行扩展，出栈反之。

Visual Studio系列生成汇编代码程序的方法

1. 在项目名称上单击鼠标右键->属性；
2. 如下图选择汇编程序输出下拉框为“程序集、机器码和源代码（/FAcs）”



3. “生成”或“重新生成”可执行文件后，可以在项目的Release或者Debug文件夹下，找到cod后缀的汇编文件。

8.3.3.1 赋值语句及算术表达式的翻译

- **语义属性**

  id.*place*：id 对应的存储位置

  ***E.place*：用来存放 *E* 的值的存储位置**

  ***E.code*： *E* 求值的 *TAC* 语句序列**

  ***S.code*： 对应于 *S* 的 *TAC* 语句序列**

- **语义函数/过程**

  ***gen*：生成一条 *TAC* 语句**

  ***newtemp*：在符号表中新建一个从未使用过的名字，并返回该名字的存储位置**

  || 是*TAC* 语句序列之间的链接运算

➢ 翻译模式

$S \rightarrow \underline{\text{id}} := E$ { $S.code := E.code \,||\, gen(\underline{\text{id}}\ .place\ \text{':='}\ E.place)$ }

$E \rightarrow \underline{\text{id}}$ { $E.place := \underline{\text{id}}\ .place$ }

$E \rightarrow \underline{\text{int}}$ { $E.place := newtemp$; $E.code := gen\ (E.place\ \text{':='}\ \underline{\text{int}}\ .val)$ }

$E \rightarrow \underline{\text{real}}$ { $E.place := newtemp$; $E.code := gen\ (E.place\ \text{':='}\ \underline{\text{real}}\ .val)$ }

$E \rightarrow E_1 + E_2$ { $E.place := newtemp$; $E.code := E_1.code \,||\, E_2.code \,||$ $gen\ (E.place\ \text{':='}\ E_1.place\ \text{'+'}\ E_2.place)$ }

$E \rightarrow E_1 * E_2$ { $E.place := newtemp$; $E.code := E_1.code \,||\, E_2.code \,||$ $gen\ (E.place\ \text{':='}\ E_1.place\ \text{'*'}\ E_2.place)$ }

$E \rightarrow -E_1$ { $E.place := newtemp$; $E.code := E_1.code \,||\, gen\ (E.place\ \text{':='}\ \text{'}uminus\text{'}\ E_1.place)$ }

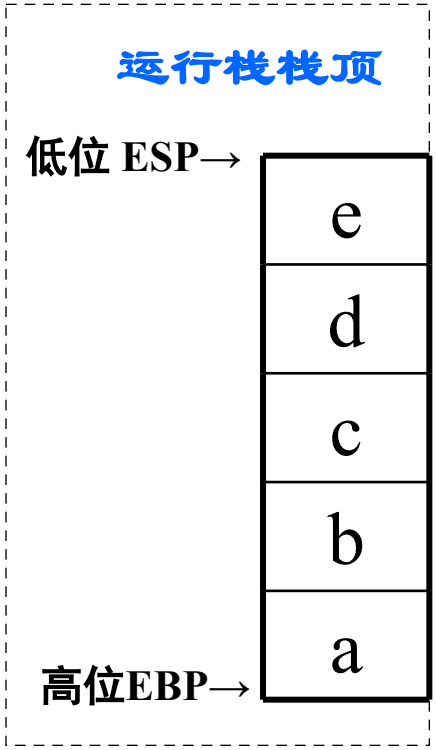
$E \rightarrow (E_1)$ { $E.place := E_1.place$; $E.code := E_1.code$ }

VS对表达式的处理举例

```
#include <stdio.h>
int main(){
    int a, b, c, d, e;
    scanf("%d, %d, %d, %d", &a, &b, &c, &d);
    e =a + b / c * d;
    printf("%d", e);
    return 0;
}
```

```
; 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

8.3.3.2 说明语句的翻译

- **语义属性**

  id.*name*：id 的词法名字（符号表中的名字）

  T.type： 类型属性 （综合属性）

  T.width，*V.width*：数据宽度（字节数）

  L.offset：列表中第一个变量的偏移地址

  L.type： 变量列表被申明的类型 （继承属性）

  L.num： 变量列表中变量的个数

- **语义函数/过程**

  enter (id.*name*, *t*, *o*)：将符号表中 id.*name* 所对应表项的 *type* 域置为 *t*，*offset* 域置为 *o*

➢ 翻译模式

$V \rightarrow V_1 ; T$ { *L.type* := *T.type*; *L.offset* := V_1.*width* ; *L.width* := *T.width* }
L { *V.type* := *make_product_3* (V_1.*type*, *T.type*, *L.num*);
V.width := V_1 .*width* + *L.num* × *T.width* }

$V \rightarrow \varepsilon$ { *V.type* := <>; *V.width* := 0 }

$T \rightarrow$ boolean { *T.type* := *bool* ; *T.width* := 1 }

$T \rightarrow$ integer { *T.type* := *int* ; *T.width* := 4 }

$T \rightarrow$ real { *T.type* := *real* ; *T.width* := 8 }

$T \rightarrow$ array [num] of T_1 { *T.type* := *array*(1.. *num*.*lexval*, T_1.*type*) ;
T.width := num.*val* × T_1.*width* }

$T \rightarrow ^T_1$ { *T.type* := *pointer*(T_1.*type*) ; *T.width* := 4 }

$L \rightarrow$ { L_1. *type* := *L. type* ; L_1. *offset* := *L. offset* ; L_1. *width* := *L. width* ; }
L_1 , id { *enter* (id.*name*, *L. type*, *L. offset* + L_1.*num* × *L. width*) ;
L.num := L_1.*num* +1 }

$L \rightarrow$ id { *enter* (id.*name*, *L. type*, *L. offset*) ; *L.num* := 1}

8.3.3.3 数组说明和数组元素引用的翻译

- **数组引用**

$S \rightarrow E_1[E_2] := E_3$ { $S.code := E_2.code \,||\, E_3.code \,||$ $gen(E_1.place$ '[' $E_2.place$ ']' ':=' $E_3.place)$ }

$E \rightarrow E_1[E_2]$ { $E.place := newtemp;$
$E.code := E_2.code \,||$
$gen(E.place$ ':=' $E_1.place$ '[' $E_2.place$ ']') }

➢ 数组的内情向量

在处理数组时，通常会将数组的有关信息记录在一些单元中，称为“内情向量”. 对于静态数组，内情向量可放在符号表中；对于可变数组，运行时建立相应的内情向量

例： 对于静态数组说明 $A[l_1{:}u_1,l_2{:}u_2,\ldots,l_n{:}u_n]$ ，可以在符号表中建立如下形式的内情向量:

l_1	u_1
l_2	u_2
…	…
l_n	u_n
type	*a*
n	*C*

l_i: 第 *i* 维的下界
u_i: 第 *i* 维的上界
type: 数组元素的类型
a: 数组首元素的地址
n: 数组维数
C: 随后解释

➢ 数组元素的地址计算

例：对于静态数组 $A[l_1:u_1,l_2:u_2,\ldots,l_n:u_n]$，若数组布局采用行优先的连续布局，数组首元素的地址为 a，则数组元素 $A[i_1,i_2,\ldots,i_n]$ 的地址 D 可以如下计算:

$$\begin{aligned} D = a &+ (i_1-l_1)(u_2-l_2)(u_3-l_3)\ldots(u_n-l_n) \\ &+ (i_2-l_2)(u_3-l_3)(u_4-l_4)\ldots(u_n-l_n) \\ &+\ldots+ (i_{n-1}-l_{n-1})(u_n-l_n) + (i_n-l_n) \end{aligned}$$

重新整理后得：$D = a - C + V$，其中

$$C = (\ldots(l_1(u_2-l_2)+l_2)(u_3-l_3)+l_3)(u_4-l_4)+\ldots+l_{n-1})(u_n-l_n)+l_n$$

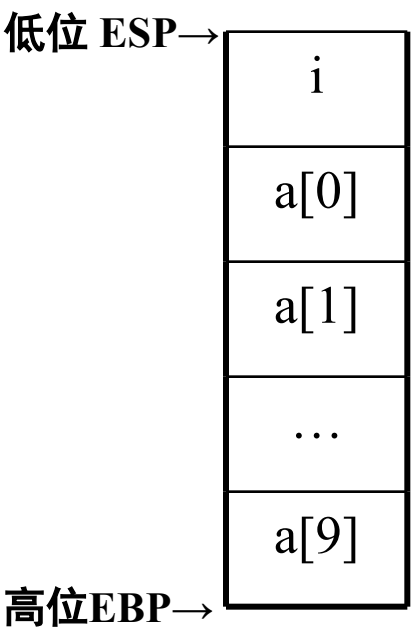
$$V = (\ldots((i_1(u_2-l_2)+i_2)(u_3-l_3)+i_3)(u_4-l_4)+\ldots\ldots+i_{n-1})(u_n-l_n)+i_n$$

（这里的 C 即为前页内情向量中的 C）

VS一维数组举例

```
#include <stdio.h>
void main(){
  int a[10], i;
  for (i=0; i<10; i++)
    a[i]=1;
}
```

```
; 5   :   for (i=0; i<10; i++)
    mov DWORD PTR _i$[ebp], 0
    jmp  SHORT $L339
$L340:
    mov eax, DWORD PTR _i$[ebp]
    add  eax, 1
    mov DWORD PTR _i$[ebp], eax
$L339:
    cmp DWORD PTR _i$[ebp], 10
    jge   SHORT $L341
; 6   :    a[i]=1;
    mov ecx, DWORD PTR _i$[ebp]
    mov DWORD PTR _a$[ebp+ecx*4], 1
    jmp  SHORT $L340
$L341:
; 7   :}
```



_a$ = -40

_i$ = -44

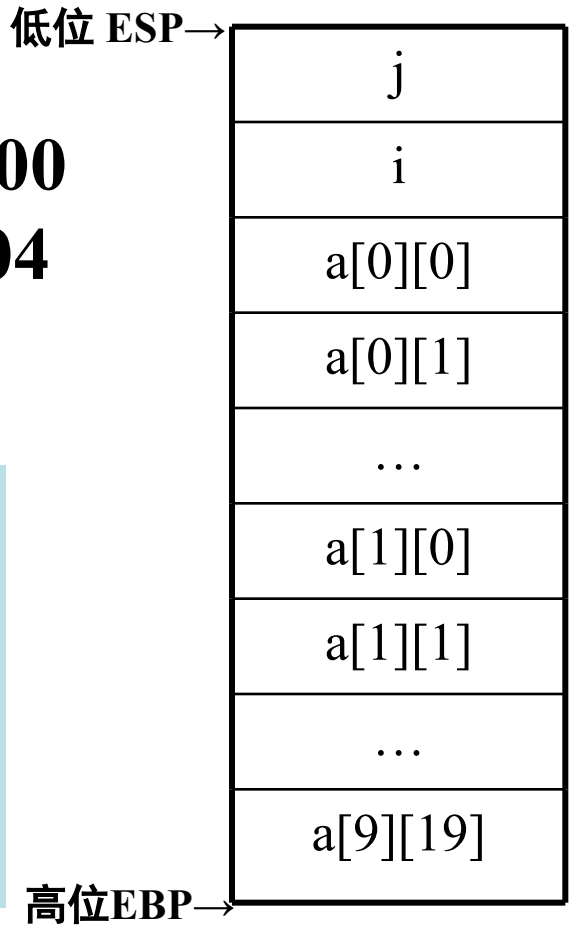
a[i]元素的地址=数组基地址a+i*4

VS二维数组举例

```
void main(){
  int a[10][20], i, j;
  for (i=0; i<10; i++)
    for (j=0; j<20; j++)
    a[i][j]=1;
}
```

_a$ = -800

_i$ = -804




```
; 7    :    a[i][j]=1;
    mov edx, DWORD PTR _i$[ebp]
    imul edx, 80                ; 00000050H
    lea  eax, DWORD PTR _a$[ebp+edx]
    mov ecx, DWORD PTR _j$[ebp]
    mov DWORD PTR [eax+ecx*4], 1
```

a[i][j]元素的地址=数组基地址+i*20*4+j*4

8.3.3.4 布尔表达式的语法制导翻译

1. 直接对布尔表达式求值

例如：可以用数值“1” 表示 true; 用数值“0” 表示 false; 采用与算术表达式类似的方法对布尔表达式进行求值

直接对布尔表达式求值的*S*-翻译模式片段：

$E \to E_1 \vee E_2$ { *E.place := newtemp; E.code := E_1.code || E_2.code || gen (E.place ':=' E_1.place 'or' E_2.place)* }

$E \to E_1 \wedge E_2$ { *E.place := newtemp; E.code := E_1.code || E_2.code || gen (E.place ':=' E_1.place 'and' E_2.place)* }

$E \to \neg E_1$ { *E.place := newtemp; E.code := E_1.code || gen (E.place ':=' 'not' E_1.palce)* }

$E \to (E_1)$ { *E.place := E_1.place ; E.code := E_1.code* }

$E \to \underline{id}_1\ \underline{rop}\ \underline{id}_2$ { *E.place := newtemp; E.code := gen ('if' $\underline{id}_1$.place* **rop**.op *$\underline{id}_2$.place 'goto' nextstat+3) || gen (E.place ':=' '0') || gen ('goto' nextstat+2) || gen (E.place ':=' '1')* }

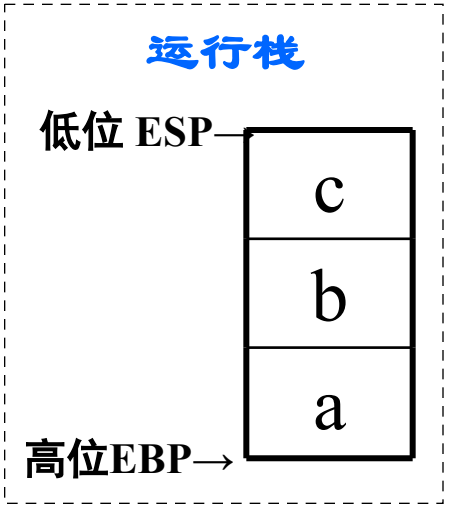
$E \to$ true { *E.place := newtemp; E.code := gen(E.place ':=' '1')* }

$E \to$ false { *E.place := newtemp; E.code := gen(E.place ':=' '0')* }

nextstat 返回输出代码序列中下一条 *TAC* 语句的下标

VS中对于a>b的处理

```
#include <stdio.h>
main(){
  int a, b, c;
  scanf("%d, %d", &a, &b);
  c =a > b;
  printf("%d", c);
}
```




```
; 5    :   c =a > b;

    mov edx, DWORD PTR _a$[ebp]
    xor   eax, eax
    cmp edx, DWORD PTR _b$[ebp]
    setg al
    mov DWORD PTR _c$[ebp], eax
```

_a$ = -4
_b$ = -8
_c$ = -12

2. 通过控制流体现布尔表达式的语义

方法：通过转移到程序中的某个位置来表示布尔表达式的求值结果

优点：方便实现控制流语句中布尔表达式的翻译

常可以得到短路（*short-circuit*）代码，而避免不必要的求值，如：在已知 E_1 为真时，不必再对$E_1 \vee E_2$ 中的 E_2 进行求值；同样，在已知 E_1 为假时，不必再对 $E_1 \wedge E_2$ 2.中的E_2 进行求值

翻译布尔表达式至短路代码（*L*-翻译模式）

$E \rightarrow \{ E_1.true := E.true;\ E_1.false := newlabel \}\ E_1 \vee$
$\{ E_2.true := E.true;\ E_2.false := E.false \}\ E_2$
$\{ E.code := E_1.code \,||\, gen(E_1.false\ ':') \,||\, E_2.code \}$

$E \rightarrow \{ E_1.false := E.false;\ E_1.true := newlabel \}\ E_1 \wedge$
$\{ E_2.false := E.false;\ E_2.true := E.true \}\ E_2$
$\{ E.code := E_1.code \,||\, gen(E_1.true\ ':') \,||\, E_2.code \}$

$E \rightarrow \neg \{ E_1.true := E.false;\ E_1.false := E.true \}\ E_1 \{ E.code := E_1.code \}$

$E \rightarrow (\{ E_1.true := E.true;\ E_1.false := E.false \}\ E_1\ ) \{ E.code := E_1.code \}$

$\mathbf{E} \rightarrow \underline{id}_1\ \underline{rop}\ \underline{id}_2$ $\{ E.code := gen(\text{'if'}\ \underline{id}_1.place\ \underline{rop}.op\ \underline{id}_2.place\ \text{'goto'}$
$E.true) \,||\, gen(\text{'goto'}\ E.false) \}$

$E \rightarrow true$ $\{ E.code := gen(\text{'goto'}\ E.true) \}$

$E \rightarrow false$ $\{ E.code := gen(\text{'goto'}\ E.false) \}$

通过控制流体现布尔表达式的语义

例：布尔表达式 $E = a<b$ or $c<d$ and $e<f$ 可能翻译为如下*TAC* 语句序列（采用短路代码，*E.true* 和*E.false* 分别代表 *E* 为真和假时对应于程序中的位置，可用标号体现）：

```
    if a<b goto E.true
    goto  label1
label1:
    if c<d goto label2
    goto E.false
label2:
    if e<f goto E.true
    goto E.false
```

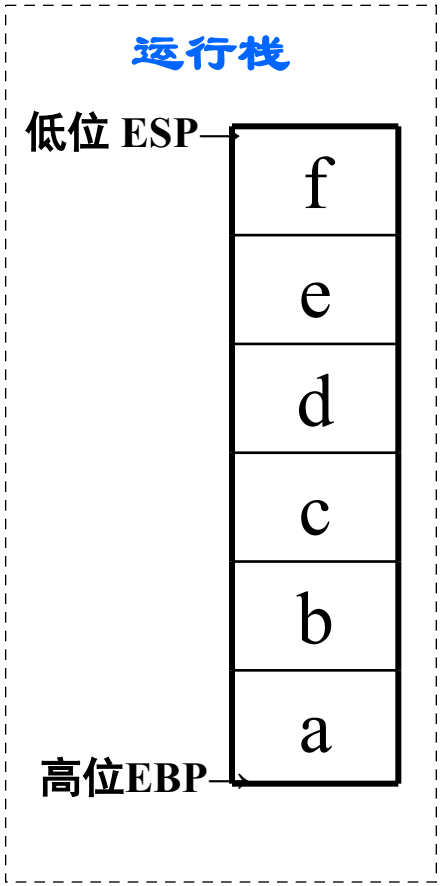
VS中的翻译方法

```
void main(){
  int a, b, c, d, e, f, g;
  scanf("%d, %d, %d, %d, %d, %d", &a, &b, &c, &d, &e, &f);
  if ((a<b)||(c<d)&&(e>f))
    g=1;
  else
    g=2;
  printf("%d", g);
}
```

VS的翻译结果

```
; 6    :   if ((a<b)||(c<d)&&(e>f))
    mov eax, DWORD PTR _a$[ebp]
    cmp eax, DWORD PTR _b$[ebp]
    jl  SHORT $L346
    mov ecx, DWORD PTR _c$[ebp]
    cmp ecx, DWORD PTR _d$[ebp]
    jge   SHORT $L345
    mov edx, DWORD PTR _e$[ebp]
    cmp edx, DWORD PTR _f$[ebp]
    jle    SHORT $L345
$L346:
; 7    :     g=1;
    mov DWORD PTR _g$[ebp], 1
; 8    :   else
    jmp  SHORT $L347
$L345:
; 9    :     g=2;
    mov DWORD PTR _g$[ebp], 2
$L347:
; 10   :   printf("%d", g);
```

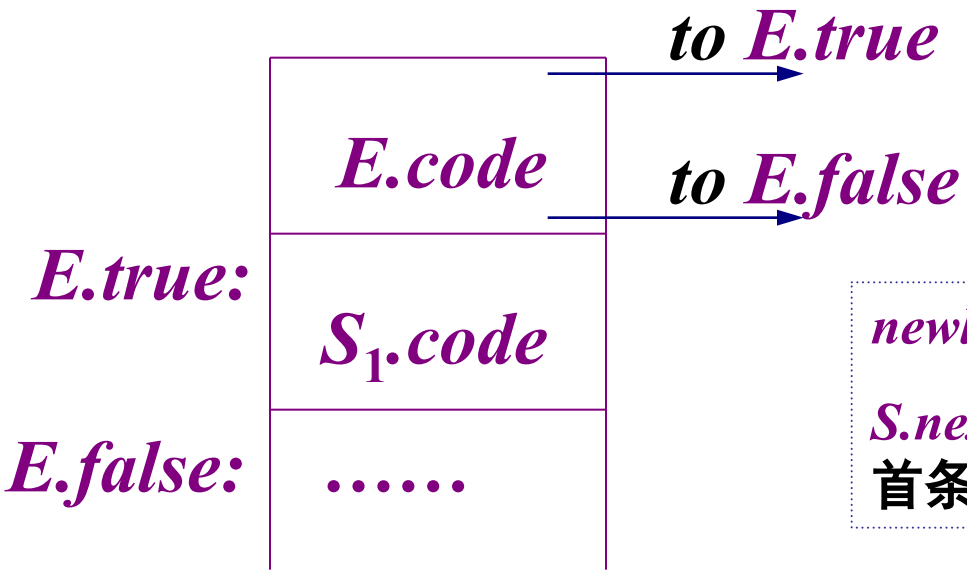
_a$ = -4
_b$ = -8
_c$ = -12
_d$ = -16
_e$ = -20
_f$ = -24
_g$ = -28



8.3.3.5 条件语句的语法制导翻译

1. if-then 语句（*L* 翻译模式）

$S \rightarrow$ if { *E.true* := *newlabel*； *E.false* := *S.next* } *E*
then { S_1.*next* := *S.next* } S_1
{ *S.code* := *E.code* || *gen*(*E.true* ':') || S_1.*code* }

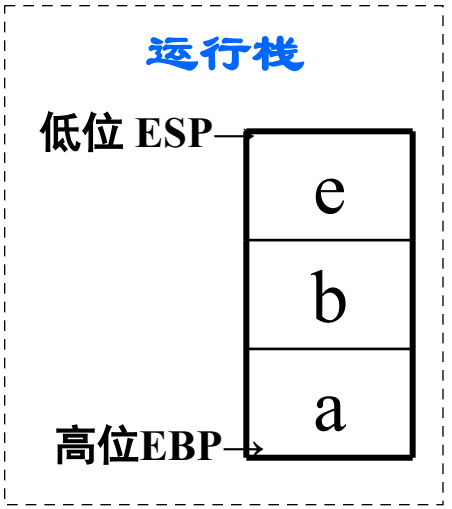


newlabel 返回一个新的语句标号

S.next 属性表示 *S* 之后要执行的首条 *TAC* 语句的标号

VS中对于if…then的处理

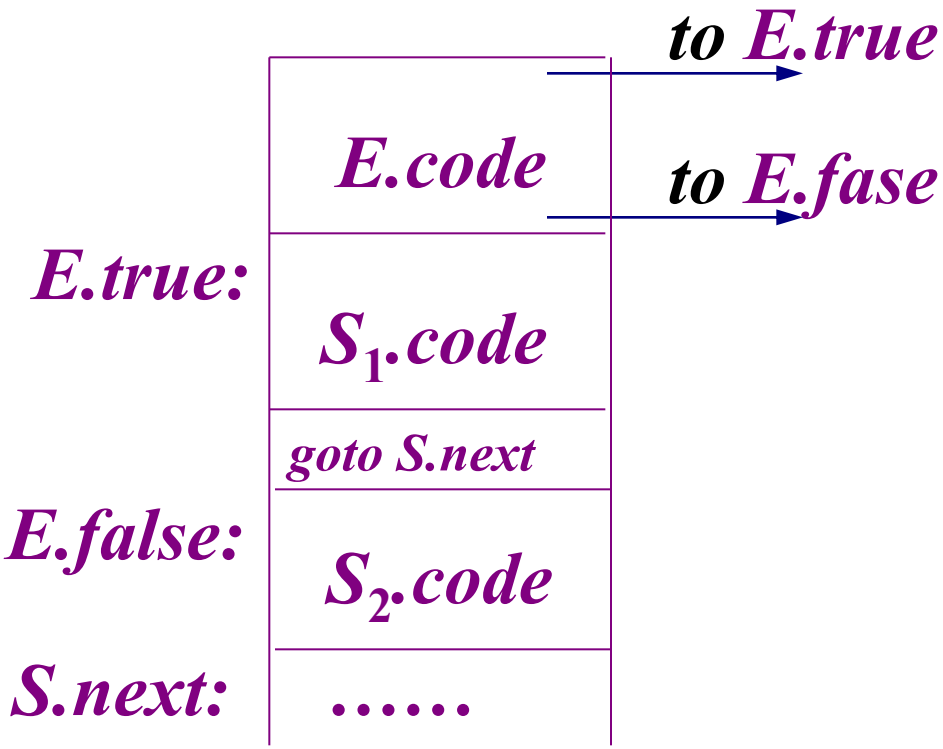
```
#include <stdio.h>
int main(){
 int a, b, e = 0;
 scanf("%d, %d", &a, &b);
 if (a<b)
   e = 1;
 printf("%d", e);
 return 0;
}
```




```
; 7    :   if (a<b)
    mov edx, DWORD PTR _a$[ebp]
    cmp edx, DWORD PTR _b$[ebp]
    jge   SHORT $L341
; 8    :     e = 1;
    mov DWORD PTR _e$[ebp], 1
$L341:
; 9    :   printf("%d", e);
    mov eax, DWORD PTR _e$[ebp]
    push    eax
    push    OFFSET FLAT:??_C@_02MECO@?$CFd?$AA@; `string'
    call  _printf
    add  esp, 8
```

_a$ = -4
_b$ = -8
_e$ = -12

2. if-then-else 语句（*L* 翻译模式）



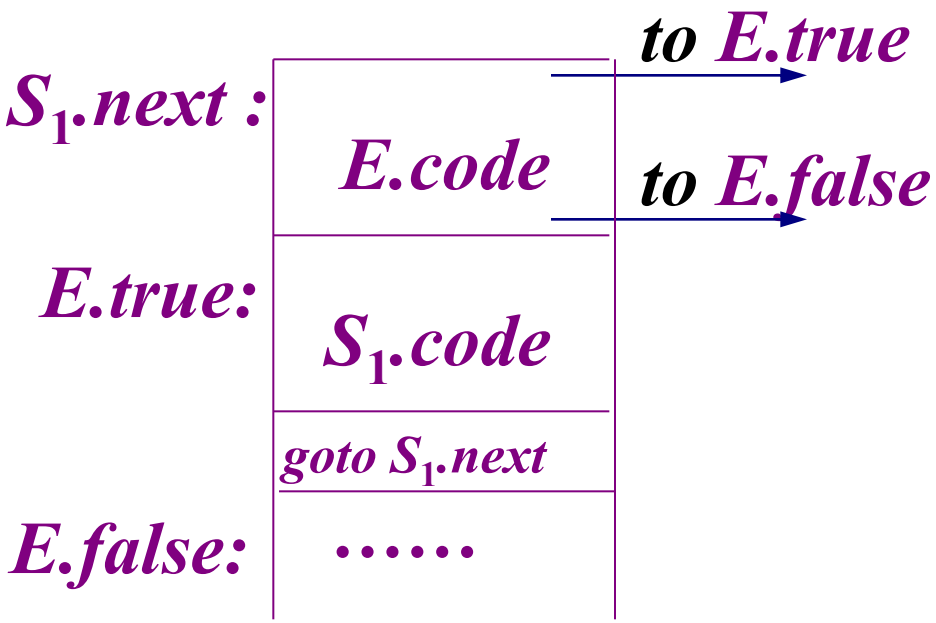
$S \rightarrow$ if { *E.true* := *newlabel*; *E.false* := *newlabel* }
E then { S_1.*next* := *S.next* }
S_1 else { S_2.*next* := *S.next* }
S_2
{ *S.code* := *E.code* ||
gen(E.true ':') ||
S_1.*code* ||
gen('goto' S.next) ||
gen(E.false ':') ||
S_2.*code*
}

VS中对于if..then..else的处理

```
int a, b, e;
scanf("%d, %d", &a, &b);
if (a<b)  e=1;  else  e=2;
printf("%d", a);
```

```
; 6    :   if (a<b)  e=1;  else  e=2;
    mov edx, DWORD PTR _a$[ebp]
    cmp edx, DWORD PTR _b$[ebp]
    jge   SHORT $L341
    mov DWORD PTR _e$[ebp], 1
    jmp  SHORT $L342
$L341:
    mov DWORD PTR _e$[ebp], 2
$L342:
; 7    :   printf("%d", a);
    mov eax, DWORD PTR _a$[ebp]
    push    eax
    push    OFFSET FLAT:??_C@_02MECO@?$CFd?$AA@; `string'
    call   _printf
    add  esp, 8
```

3. while 语句（*L* 翻译模式）



$S \rightarrow$ while
{ *E.true* := *newlabel*;
E.false := *S.next* }
E do
{ S_1.*next* := *newlabel* }
S_1
{ *S.code* := *gen*(S_1.*next* ':')
|| *E.code*
|| *gen*(*E.true* ':')
|| S_1.*code*
|| *gen*('goto' S_1.*next*)
}

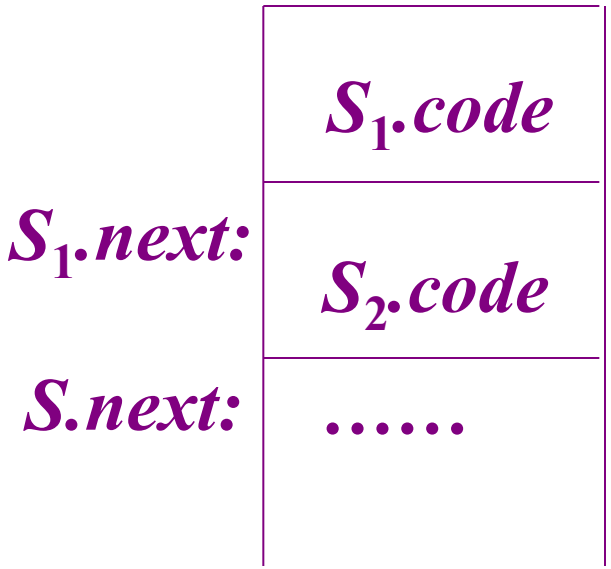
VS中对于while循环的处理

```
int a, b;
scanf("%d, %d", &a, &b);
while (a<b)
  a++;
printf("%d", a);
```

```
$L341:
; 6    :   while (a<b)
    mov edx, DWORD PTR _a$[ebp]
    cmp edx, DWORD PTR _b$[ebp]
    jge   SHORT $L342
; 7    :     a++;
    mov eax, DWORD PTR _a$[ebp]
    add  eax, 1
    mov DWORD PTR _a$[ebp], eax
    jmp  SHORT $L341
$L342:
; 8    :   printf("%d", a);
```

4. 顺序复合语句（*L* 翻译模式）

$S \rightarrow$ { $S_1.next := newlabel$ } S_1 ;
{ $S_2.next := S.next$ } S_2
{ $S.code := S_1.code$
$\| \; gen(S_1.next \text{ ':'})$
$\| \; S_2.code$
}



5. 含 *break* 语句的语法制导翻译

– 翻译模式

$P \rightarrow D\ ;\ \{ S.next := newlabel;\ S.break := newlabel \}\ S$
$\{ gen(S.next\ ':') \}$

$S \rightarrow if\ \{ E.true := newlabel;\ E.false := S.next \}\ E\ then$
$\{ S_1.next := S.next;\ S_1.break := S.break \}\ S_1$
$\{ S.code := E.code \| gen(E.\ true\ ':') \| S_1.code \}$

$S \rightarrow if\ \{ E.true := newlabel;\ E.false := newlabel \}\ E\ then$
$\{ S_1.next := S.next;\ S_1.break := S.break \}\ S_1\ else$
$\{ S_2.next := S.next;\ S_2.break := S.break \}\ S_2$
$\{ S.code := E.code \| gen(E.true\ ':') \| S_1.code \|$
$gen('goto'\ S.next) \| gen(E.false\ ':') \| S_2.code \}$

$S \rightarrow$ *while* { *E.true* := *newlabel*; *E.false* := *S.next* } *E do*
{ S_1.*next* := *newlabel*; S_1.*break* := *S.next* } S_1
{ *S.code* := *gen*(S_1.*next* ':') ||
E.code || *gen*(*E.true* ':') || S_1.*code* ||
gen('goto' S_1.*next*) }

$S \rightarrow$ { S_1.*next* := *newlabel*; S_1.*break* := *S.break* } S_1 ;
{ S_2.*next* := *S.next*; S_2.*break* := *S.break* } S_2
{ *S.code* := S_1.*code* || *gen*(S_1.*next* ':') || S_2.*code* }

$S \rightarrow$ *break* ; { *S.code* := *gen*('goto' *S.break*) }

6.3.3.6 拉链与代码回填（*backpatching*）

- 另一种控制流中间代码生成技术

比较：前面的方法采用 L-属性文法/翻译模式

下面的方法采用 S-属性文法/翻译模式

- **语义属性**

E.truelist：“真链”，链表中的元素表示 一系列跳转语句的地址，这些跳转语句的目标标号是体现布尔表达式 *E* 为“真”的标号

E. falselist：“假链”，链表中的元素表示 一系列跳转语句的地址，这些跳转语句的目标标号是体现布尔表达式 *E* 为假的标号

S. nextlist：“*next* 链”，链表中的元素表示 一系列跳转语句的地址，这些跳转语句的目标标号是在执行序列中紧跟在 *S* 之后的下条*TAC*语句的标号

- **语义函数/过程**

makelist(i)：创建只有一个结点 *i* 的表，对应存放目标 *TAC* 语句数组的一个下标

merge(p_1,p_2)：连接两个链表 p_1 和 p_2，返回结果链表

backpatch(p,i)：将链表 *p* 中每个元素所指向的跳转语句的标号置为 *i*

nextstm：下一条*TAC* 语句的地址

emit (…)：输出一条*TAC* 语句，并使 *nextstm* 加1

➢ 处理布尔表达式的翻译模式

$E \rightarrow E_1 \vee M\ E_2$ { *backpatch*(E_1.*falselist*,*M.gotostm*) ;
E.truelist := *merge*(E_1.*truelist*, E_2.*truelist*) ;
E.falselist := E_2.*falselist* }

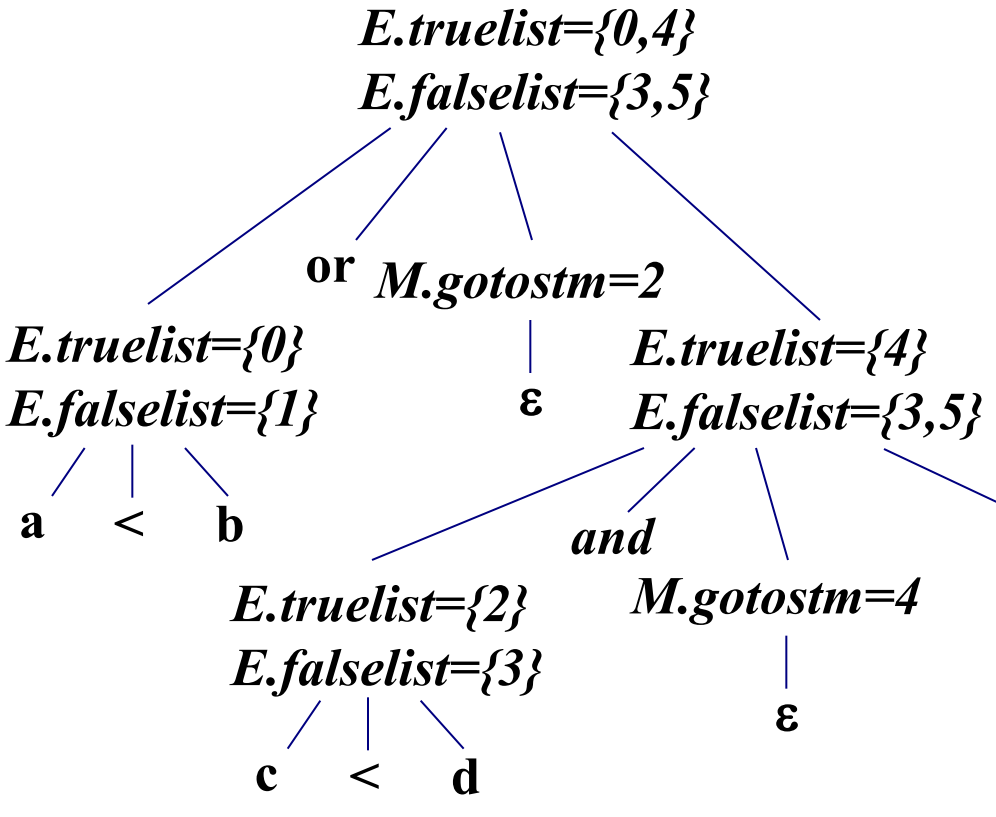
$E \rightarrow E_1 \wedge M\ E_2$ { *backpatch*(E_1.*truelist*,*M.gotostm*) ;
E.falselist := *merge*(E_1.*falselist*, E_2.*falselist*) ;
E.truelist := E_2.*truelist* }

$E \rightarrow \neg\ E_1$ { *E.truelist* := E_1.*falselist* ;
E.falselist := E_1.*truelist* }

注: 这里可以规定产生式的优先级依次递增来解决冲突问题（下同）

$E \rightarrow (E_1)$	{ *E.truelist := E_1.truelist ;* *E.falselist := E_1.falselist* }
$E \rightarrow \underline{id}_1$ rop $\underline{id}_2$	{ *E.truelist := makelist (nextstm);* *E.falselist := makelist (nextstm+1);* *emit (* 'if' $\underline{id}_1$.*place* rop.op $\underline{id}_2$.*place* 'goto _' *);* *emit (*'goto _'*)* }
$E \rightarrow$ true	{ *E.truelist := makelist (nextstm);* *emit (*'goto _'*)* }
$E \rightarrow$ false	{ *E.falselist := makelist (nextstm);* *emit (*'goto _'*)* }
$M \rightarrow \varepsilon$	{ *M.gotostm := nextstm* }

➢ 布尔表达式 $E = a<b \vee c<d \wedge e<f$ 的翻译示意



（0）if $a<b$ goto _

（1）goto _ (2)

（2）if $c<d$ goto _ (4)

（3）goto _

（4）if $e<f$ goto _

（5）goto _

➢ 处理条件语句的翻译模式

$S \rightarrow$ if E then $M\ S_1$

{ *backpatch(E.truelist,M.gotostm) ;*
S.nextlist := merge(E.falselist, S_1.nextlist) }

$S \rightarrow$ if E then $M_1\ S_1\ N$ else $M_2\ S_2$

{ *backpatch(E.truelist, M_1.gotostm) ;*
backpatch(E.falselist, M_2.gotostm) ;
S.nextlist := merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist)) }

$M \rightarrow \varepsilon$

{ *M.gotostm := nextstm* }

$N \rightarrow \varepsilon$

{ *N.nextlist := makelist(nextstm); emit('goto _')* }

➢ 处理循环、复合的翻译模式

$S \rightarrow$ while M_1 E do M_2 S_1

{ *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
backpatch(*E.truelist*, M_2.*gotostm*) ;
S.nextlist := *E.falselist*;
emit('goto', M_1.*gotostm*)}

$S \rightarrow S_1$; M S_2

{ *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
S.nextlist := S_2.*nextlist* }

$M \rightarrow \varepsilon$

{ *M.gotostm* := *nextstm* }

➢ 增加 *break* 语句后控制语句处理的翻译模式

$P \rightarrow D\ ;\ S\ M$ { *backpatch(S.nextlist, M.gotostm) ;*
backpatch(S.breaklist, M.gotostm) }

$S \rightarrow$ *if E then M* S_1 { *backpatch(E.truelist, M.gotostm) ;*
S.nextlist := merge(E.falselist, S_1*.nextlist) ;*
S.breaklist := S_1*.breaklist* }

$S \rightarrow$ *if E then* $M_1\ S_1\ N$ *else* $M_2\ S_2$ { *backpatch(E.truelist,* M_1*.gotostm) ;*
backpatch(E.falselist, M_2*.gotostm) ;*
S.nextlist := merge(S_1*.nextlist, merge(N.nextlist,* S_2*.nextlist) ;*
S.breaklist := merge(S_1*.breaklist,* S_2*.breaklist)* }

S. breaklist **：“*break* 链”，链表中的元素表示一系列跳转语句的地址，这些跳转语句的目标标号是直接所属 while 语句的结束位置**

$S \rightarrow$ *while* M_1 *E then* M_2 S_1

{ *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
backpatch(*E.truelist*, M_2.*gotostm*) ;
S.nextlist := *merge*(*E.falselist*, S_1.*breaklist*);
S.breaklist := "" ; *emit*('goto', M_1.*gotostm*) }

$S \rightarrow S_1$; M S_2 { *backpatch*(S_1.*nextlist*, *M.gotostm*) ;
S.nextlist := S_2.*nextlist* ;
S.breaklist := *merge*(S_1.*breaklist*, S_2.*breaklist*) }

$S \rightarrow$ *break* ; { *S.breaklist t* := *makelist* (*nextstm*) ; *S.nextlist* := "" ;
emit ('goto _') }

$M \rightarrow \varepsilon$ { *M.gotostm* := *nextstm* }

$N \rightarrow \varepsilon$ { *N.nextlist* := *makelist*(*nextstm*); *emit*('goto _') }

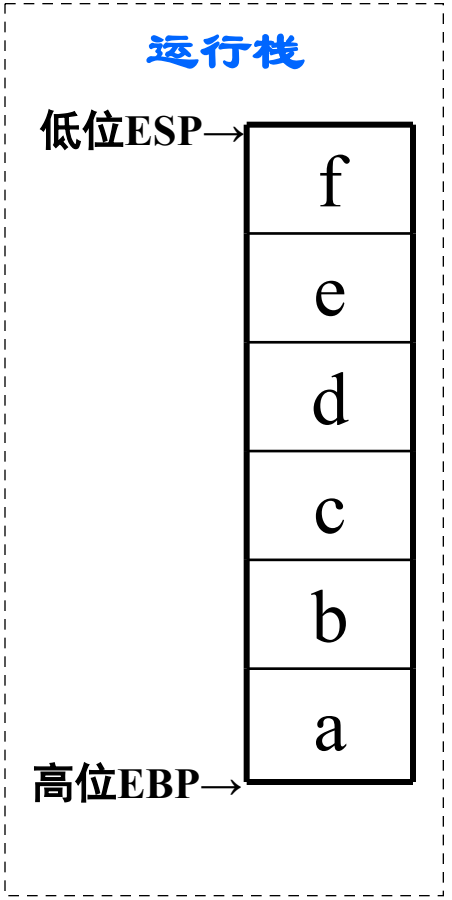
VS的翻译方法

```
void main(){
  int a, b, c, d, e, f, g;
  scanf("%d, %d, %d, %d, %d, %d", &a, &b, &c, &d, &e, &f);
  while (a<b) {
    if (c<d)
      g = e + f;
  }
  printf("%d", g);
}
```

VS的翻译结果

```
$L346:
; 6    :   while (a<b) {
    mov eax, DWORD PTR _a$[ebp]
    cmp eax, DWORD PTR _b$[ebp]
    jge   SHORT $L347
; 7    :     if (c<d)
    mov ecx, DWORD PTR _c$[ebp]
    cmp ecx, DWORD PTR _d$[ebp]
    jge   SHORT $L348
; 8    :       g = e + f;
    mov edx, DWORD PTR _e$[ebp]
    add  edx, DWORD PTR _f$[ebp]
    mov DWORD PTR _g$[ebp], edx
$L348:
; 9    :   }
    jmp  SHORT $L346
$L347:
; 10   :   printf("%d", g);
```

_a$ = -4
_b$ = -8
_c$ = -12
_d$ = -16
_e$ = -20
_f$ = -24
_g$ = -28



8.3.3.7 过程调用的语法制导翻译

- 简单过程调用的翻译
  - 示例：过程调用 call p（$a+b$，$a*b$）
    将被翻译为：

```
计算 a + b 置于 t 中的代码        // t := a + b
计算 a * b 置于 z 中的代码        // z := a * b
param   t                         // 第一个实参地址
param   z                         // 第二个实参地址
call  p , 2                       // 过程调用语句
```

➢ 简单过程调用的翻译模式

A.n : 参数个数
A.arglist :实参地址的列表
makelist :创建实参地址结点
append :在实参表中添加结点

$S \rightarrow$ call id (A)

{ *S*.code := *A*.code;

for *A.arglist* 中的每一项 *p* do

S.code := *S.code* || *gen*('param' *p*);

S.code := *S.code* || *gen* ('call' id.*place* , *A.n*) }

$A \rightarrow A_1$, E

{ $A.n := A_1.n + 1$;
$A.arglist := append(A_1.arglist, makelist(E.place))$;
$A.code := A_1.code \| E.code$ }

$A \rightarrow \varepsilon$

{ $A.n := 0$; $A.arglist := ""$; $A.code := ""$}

VS的翻译方法

举例：

```
#include <stdio.h>
void func(int, int);
void main(){
  int a, b;
  scanf("%d, %d", &a, &b);
  func(a, b);
  printf("%d, %d", a, b);
}
void func(int x, int y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

主程序调用func

```
; 7    :    func(a, b);

    mov edx, DWORD PTR _b$[ebp]
    push    edx
    mov eax, DWORD PTR _a$[ebp]
    push    eax
    call    _func
    add esp, 8
```

主要工作：实参入栈；调用函数；实参出栈

func函数开始

```
_func    PROC NEAR                          ; COMDAT

; 11    : {

    push    ebp
    mov ebp, esp
    sub esp, 68                            ; 00000044H
    push    ebx
    push    esi
    push    edi
```

主要工作：保存调用函数的栈底和栈顶；为本函数局部变量分配栈空间；保存寄存器值

func函数结束

```
; 16    : }

    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0
```

主要工作：恢复寄存器的值；释放局部变量的空间；恢复调用函数的栈底和栈顶