Lecture 7: 中间代码生成

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

中间表示形式

后缀表达式

$E \rightarrow E\,opE \mid uopE \mid (E) \mid \mathrm{id} \mid \mathrm{num}$

表达式E的后缀表示可以如下归纳定义：

表达式E	后缀式E'
id	id
num	num
$E_1\,opE_2$	$E_1'\,E_2'\,op$
$uopE$	$E'uop$
(E)	E'

后缀表达式

- **后缀表示不需要括号**
  - (8 – 5) + 2 的后缀表示是8 5 –2 +
- **后缀表示的最大优点是便于计算机处理表达式**

计算栈	输入串
	8 5 –2 +
8	5 –2 +
8 5	–2 +
3	2 +
3 2	+
5	

后缀表达式

- **后缀表示不需要括号**
  - (8 – 5) + 2 的后缀表示是8 5 –2 +
- **后缀表示的最大优点是便于计算机处理表达式**
- **后缀表示也可以拓广到表示赋值语句和控制语句，但很难用栈来描述控制语句的计算**

图形表示

- **语法树是一种图形化的中间表示**
  - 语法树中，公共子表达式每出现一次，就有一个对应的子树
- **有向无环图也是一种中间表示---有向无环图(Directed Acyclic Graph, DAG)**
  - 能够指出表达式中的公共子表达式，更简洁地表示表达式

DAG构造

- **可以用和构造抽象语法树一样的SDD来构造**

	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

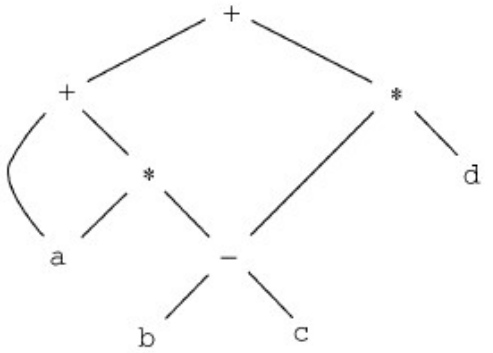


图 6-3　表达式 a + a * (b - c) + (b - c) * d 的 DAG

DAG构造

- **不同的处理**
  - 在函数Leaf和Node每次被调用时，构造新节点前先检查是否已存在同样的节点，如果已经存在，则返回这个已有的节点
- **构造过程示例**

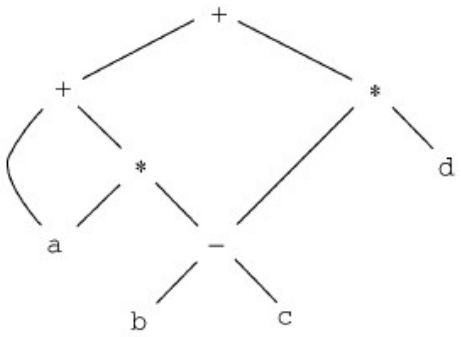


图 6-3　表达式 a + a * (b - c) + (b - c) * d 的 DAG

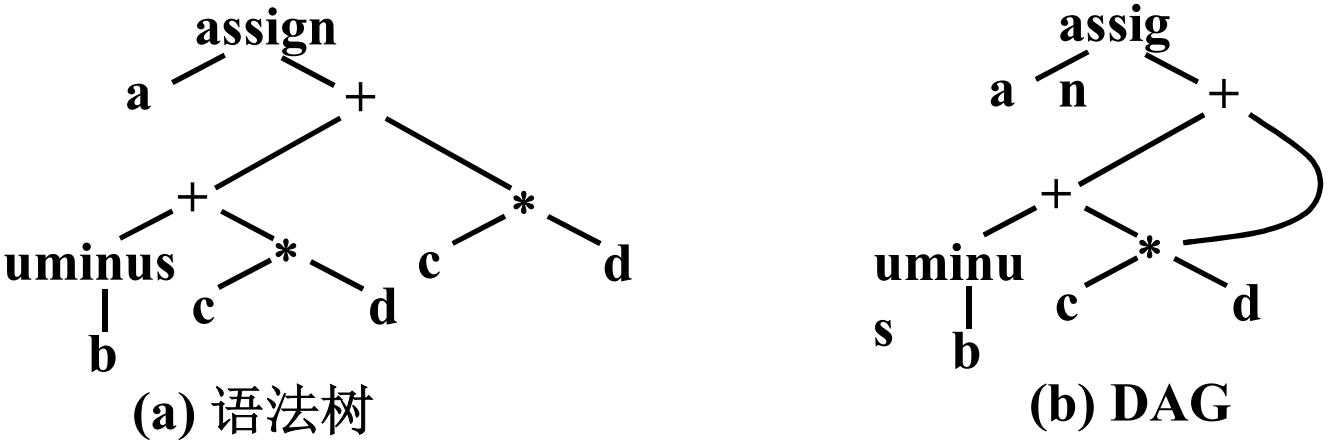
1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

图 6-5　图 6-3 所示的 DAG 的构造过程

DAG构造

构造赋值语句语法树的语法制导定义

修改构造结点的函数可生成有向无环图



a = (–b + c*d) + c*d的图形表示

DAG构造

构造赋值语句语法树的语法制导定义

修改构造结点的函数可生成有向无环图

产 生 式	语 义 规 则
$S \rightarrow \mathrm{id} = E$	*S.nptr* = *mkNode*('assign', *mkLeaf* (id, id.*entry*), *E.nptr*)
$E \rightarrow E_1 + E_2$	*E.nptr* = *mkNode*('+', E_1.*nptr*, E_2.*nptr*)
$E \rightarrow E_1 * E_2$	*E.nptr* = *mkNode*('*', E_1.*nptr*, E_2.*nptr*)
$E \rightarrow -E_1$	*E.nptr* = *mkUNode*('uminus', E_1.*nptr*)
$E \rightarrow (E_1)$	*E.nptr* = E_1.*nptr*
$F \rightarrow \mathrm{id}$	*E.nptr* = *mkLeaf* (id, id.*entry*)

三地址代码

- **每条指令右侧最多有一个运算符**
  - 一般情况可以写成x = y op z
- **允许的运算分量（地址）最多3个**
  - 名字：源程序中的名字作为三地址代码的地址
  - 常量：源程序中出现或生成的常量
  - 编译器生成的临时变量

三地址代码

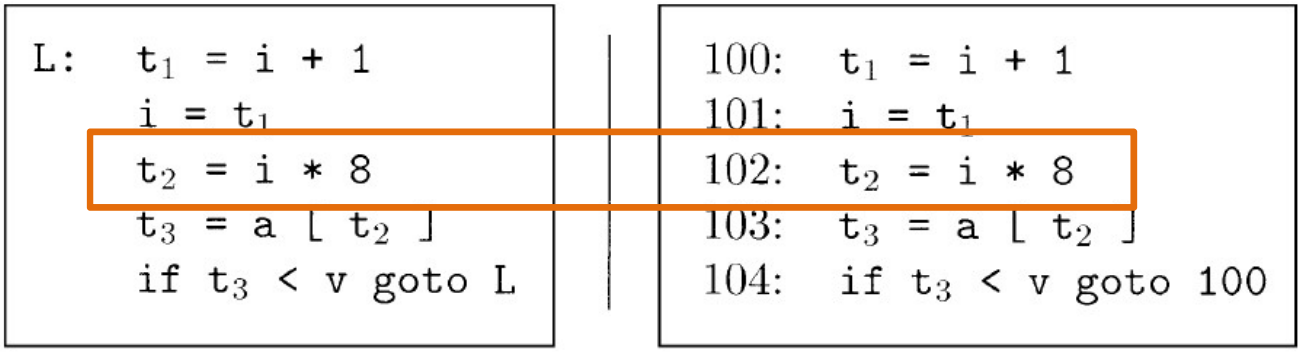
- **指令集合 （1）**
  - 运算/赋值指令：x=y op z　　x = op y
  - 复制指令：x=y
  - 无条件转移指令：goto L
  - 条件转移指令：if x goto L　　ifFalse x goto L
  - 条件转移指令：if x relop y goto L

三地址代码

- **指令集合（2）**
  - 过程调用/返回 p(x1, x2, …, xn)
    - param x1　　//设置参数
    - param x2
    - …
    - param xn
    - call p, n　　//调用子过程p，n为参数个数
  - 带下标的复制指令：x=y[i]　　x[i]=y
    - 注意：i表示离开数组位置第i个字节，而不是数组的第i个元素
  - 地址/指针赋值指令：
    - x=&y　　x=*y　　*x=y

三地址代码实例

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



a) 符号标号　　b) 位置号

三地址代码实例

- **三地址代码是语法树或DAG的一种线性表示**
  - 例：a = (–b + c∗d) + c∗d

语法树的代码

$t_1 = -b$

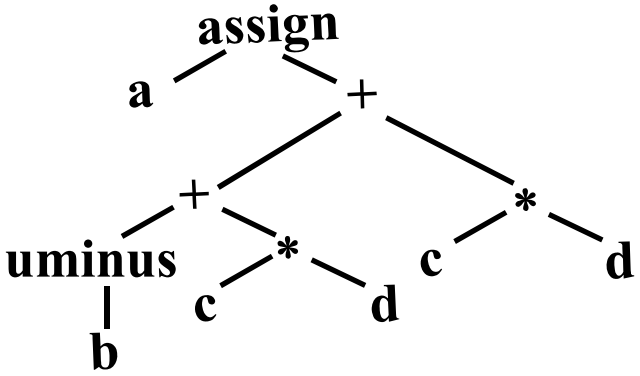
$t_2 = c * d$

$t_3 = t_1 + t_2$

$t_4 = c * d$

$t_5 = t_3 + t_4$

$a = t_5$



三地址代码实例

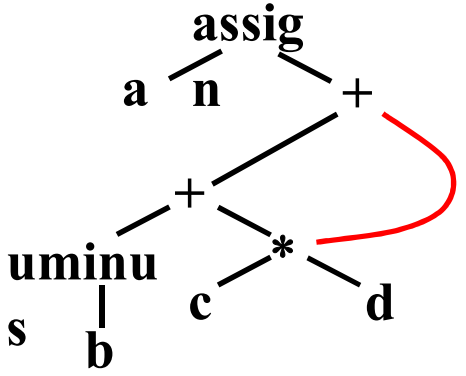
- **三地址代码是语法树或DAG的一种线性表示**
  - 例　　a = (–b + c∗d) + c∗d

语法树的代码

$t_1 = -b$
$t_2 = c * d$
$t_3 = t_1 + t_2$
$t_4 = c * d$
$t_5 = t_3 + t_4$
$a = t_5$

DAG的代码

$t_1 = -b$
$t_2 = c * d$
$t_3 = t_1 + t_2$
$t_4 = t_3 + t_2$
$a = t_4$



三地址指令的四元式表示方法

- **在实现时，可以使用四元式/三元式/间接三元式来表示三地址指令**
- **四元式：可以实现为纪录（或结构）**
- **格式（字段）：　　op　　arg1　　arg2　　result**
  - op: 运算符的内部编码
  - arg1,arg2,result是地址
  - x=y+z　　+　y　z　x
- **单目运算符不使用arg2**
- **param运算不使用arg2和result**
- **条件转移/非条件转移将目标标号放在result字段**

三地址指令的四元式表示方法

- **例：赋值语句：a=b* -c + b* -c**

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

a) 三地址代码

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

b) 四元式

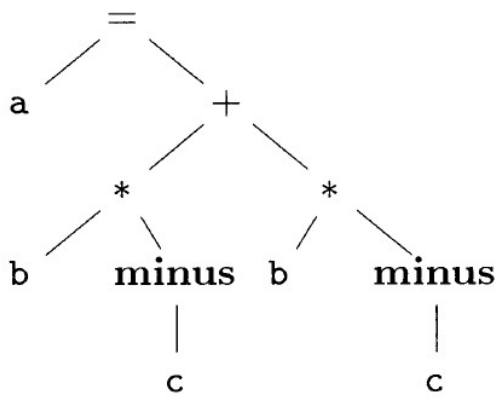
图 6-10　三地址代码及其四元式表示

三地址指令的三元式表示方法

- **三元式（triple） op arg1 arg2**
- **使用三元式的位置来引用三元式的运算结果**
  - x[i]=y需要拆分为两个三元式：求x[i]的地址，然后再赋值
  - x=y op z需要拆分为（这里？是编号）
    - (?) op y z
    - = x ?

三地址指令的三元式表示方法

- **例：a=b*-c + b * -c**



a) 语法树

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

b) 三元式

图 6-11　$a = b^* - c + b^* - c$ 的表示

问题：在优化时经常需要移动/删除/添加三元式，导致三元式的移动

间接三元式

- 包含了一个指向三元式的指针的列表
- 我们可以对这个列表进行操作，完成优化功能；操作时不需要修改三元式中的参数

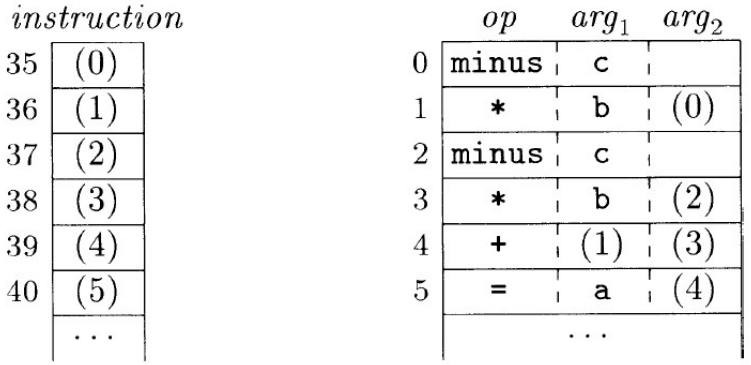


图 6-12　三地址代码的间接三元式表示

静态单赋值（SSA）

- **一种便于某些代码优化的中间表示**
- **和三地址代码的主要区别**
  - 所有赋值指令都是对不同名字的变量的赋值

三地址代码	静态单赋值形式
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

静态单赋值（SSA）

- **一种便于某些代码优化的中间表示**
- **和三地址代码的主要区别**
  - 所有赋值指令都是对不同名字的变量的赋值
  - 对于同一个变量在不同路径中定值的情况，可以使用φ函数来合并不同的定值

    if (flag) x = –1; else x = 1;

    y = x * a;

  改成

    if (flag) x_1 = –1; else x_2 = 1;

    $x_3 = \phi(x_1, x_2)$; //由flag的值决定用x_1还是x_2

声明语句的翻译

声明

- **考虑例子**
  - 含义
    - D生成一个声明列表
    - T生成不同的类型
    - B生成基本类型int/float
    - C表示分量，生成[num]序列
    - 注意record中也是声明列表，其中字段声明和变量声明的文法一致

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

除了确定类型和类型宽度，还有什么语义需要处理？符号表中的位置

声明

- **为什么需要确定类型**
  - 通过变量类型，我们可以知道该变量在运行时刻需要的内存数量；
  - 在编译时刻我们可以使用这些数量(类型的宽度)为每个名字分配一个相对地址(offset)：名字的类型和相对地址等信息保存在符号表
    - PS：变长数据用指针

声明

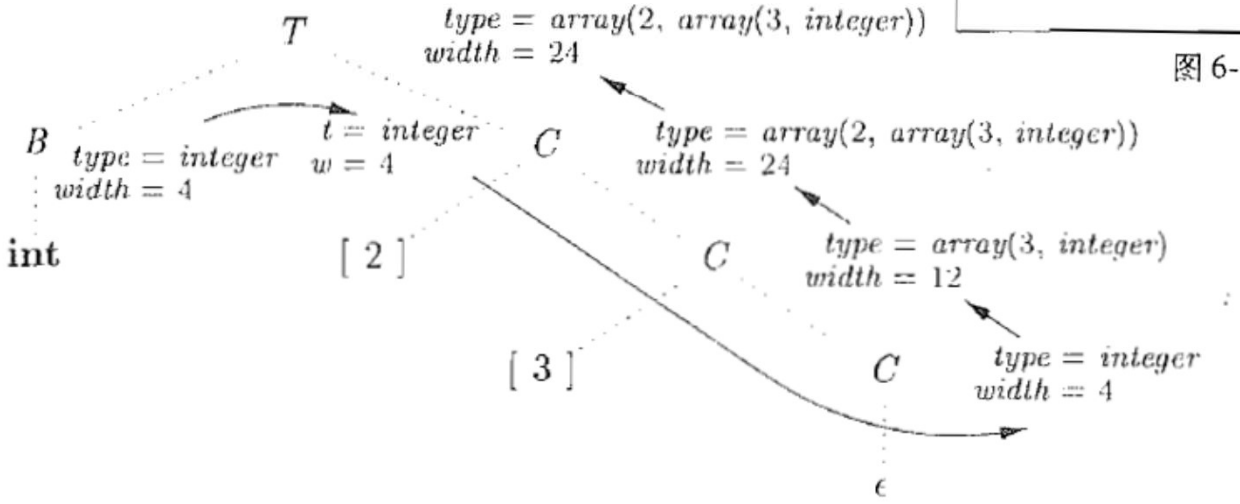
- 文法变量T(类型)的语义属性
  - type： 类型(表达式)
  - width：类型所占用的字节数
- 辅助子程序
  - enter：将变量的类型和地址填入符号表中
  - array：数组类型处理子程序
- 全局变量
  - offset：已分配空间字节数，用于计算相对地址

普通字段声明序列SDT (不考虑作用域问题)

- **计算被声明名字的类型和相对地址**
  - Int [2][3]

$T \rightarrow$	B	$\{ t = B.type;\ w = B.width; \}$
	C	$\{ T.type = C.type;\ T.width = C.width \}$
$B \rightarrow$	**int**	$\{ B.type = integer;\ B.width = 4; \}$
$B \rightarrow$	**float**	$\{ B.type = float;\ B.width = 8; \}$
$C \rightarrow$	ϵ	$\{ C.type = t;\ C.width = w; \}$
$C \rightarrow$	[**num**] C_1	$\{ C.type = array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width; \}$

图 6-15　计算类型及其宽度



普通字段声明序列SDT (不考虑作用域问题)

- **计算被声明名字的类型和相对地址**

enter是填表动作

$P \rightarrow$ {*offset* = 0; } D

$D \rightarrow T$ **id;**{*enter* (id.*lexeme*, *T.type*, *offset*);*offset* = *offset* + *T.width* } D_1

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

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

符号表与作用域

- **作用域：一个声明起作用的程序部分称为该声明的作用域。**
  - 过程中出现的名字，如果是在该过程的一个声明的作用域内，那么这个出现称为局部于该过程的；否则叫做非局部的。
  - 局部和非局部的区分也适用于其他任何可包含声明的语法结构。
  - 作用域是名字声明的一个性质
- **需要多张符号表来实现作用域信息的保存**

符号表与作用域

- **考虑嵌套变量的声明**

  $P \rightarrow D;\ S$

  $D \rightarrow D\ ;\ D\ \mid \text{id} : T \mid$

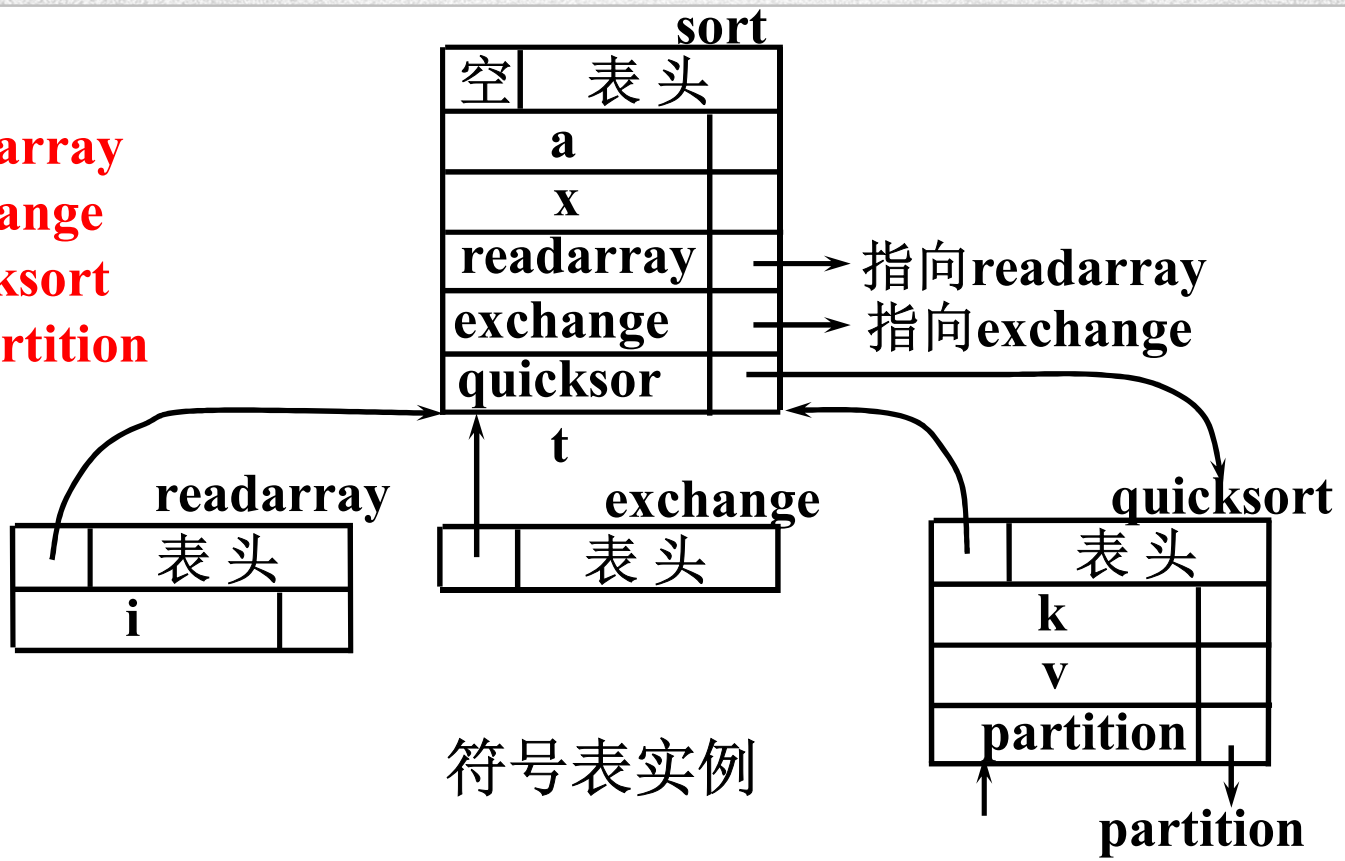
  proc id ; D ; S

```
sort
    var a:…; x:…;
    readarray
        var i:…;
    exchange
    quicksort
        var k, v:…;
        partition
                var i, j:…;
```

程序参数被略去

符号表与作用域

sort
  readarray
  exchange
  quicksort
    partition



符号表实例

符号表与作用域

- **符号表的特点**
  - 各过程有各自的符号表
  - 符号表之间有双向链
  - 构造符号表时需要符号表栈
  - 构造符号表需要活动记录栈

符号表与作用域

符号表栈（*tblptr)*

活动记录栈（*offset)*

- **语义动作用到的函数**

  mkTable(*previous*)

  enter(*table, name, type, offset*)

  addWidth(*table*, *width*)

  enterProc(*table*, *name*, *newtable*)

符号表与作用域

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*); *push*(*t*, *tblprt*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$
{*t*=*top*(*tblptr*);*addWidth*(*t*,*top*(*offset*));*pop*(*tblptr*);*pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

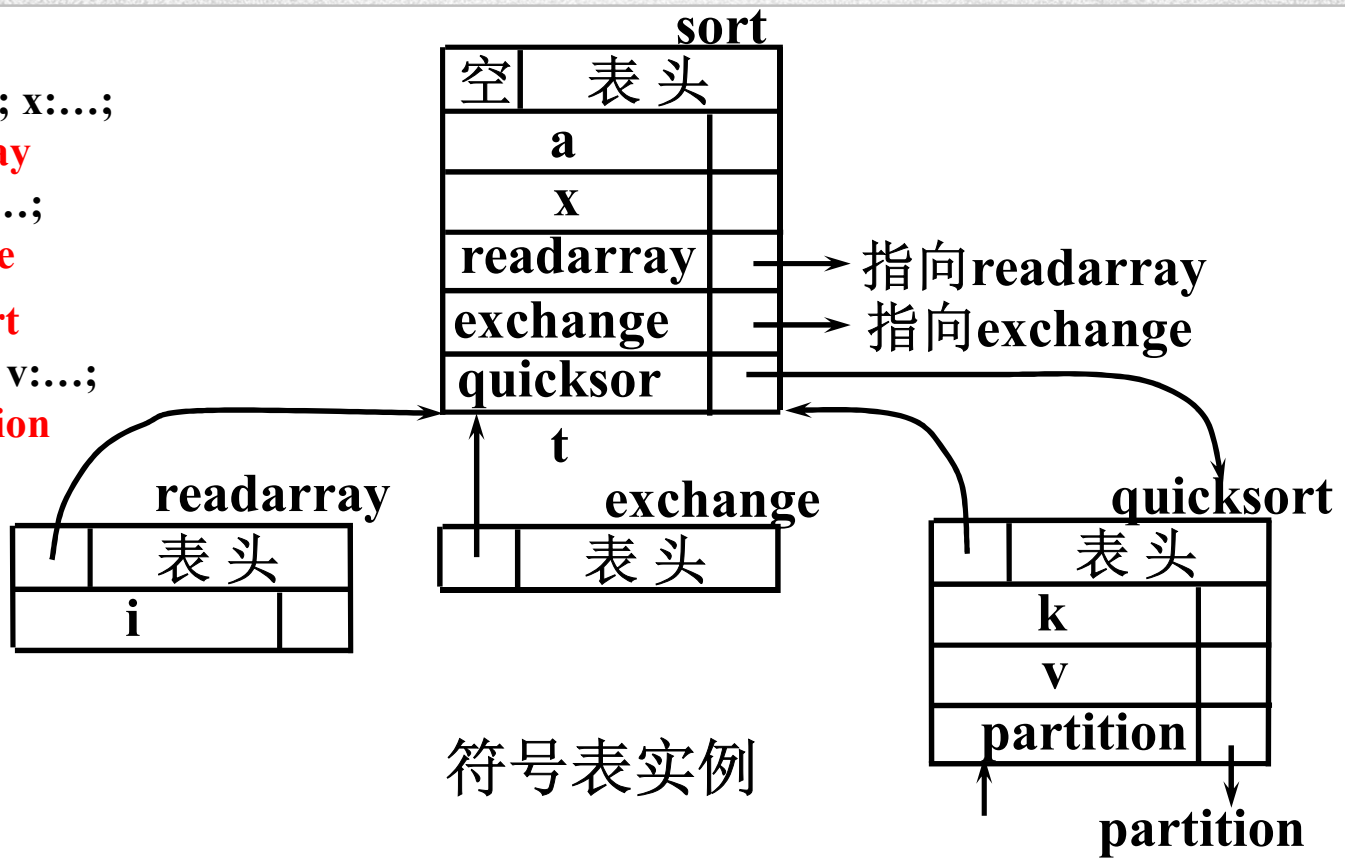
$D \rightarrow$ *id* :*T* {*enter*(*top*(*tblptr*), *id.lexeme*, *T.type*, *top*(*offset*));
top(*offset*) = *top*(*offset*) + *T.width* }

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*)); *push*(*t*, *tblptr*); *push*(0, *offset*) }

```
sort
    var a:…; x:…;
    readarray
        var i:…;
    exchange
    quicksort
        var k, v:…;
        partition
```

实现过程

```
sort
    var a:…; x:…;
    readarray
        var i:…;
    exchange
    quicksort
        var k, v:…;
        partition
```



符号表实例

记录的处理

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

不同作用域

Env是存储表的栈
Stack是存储offset值的栈
top是当前栈顶所指的表 (top.get, top.put)
offset是当前栈顶所指的表的整体偏移量

$T \rightarrow$	**record** '{'	{ *Env.push*(*top*); *top* = **new** *Env*(); *Stack.push*(*offset*); *offset* = 0; }
	D '}'	{ *T.type* = *record*(*top*); *T.width* = *offset*; *top* = *Env.pop*(); *offset* = *Stack.pop*(); }

- **为每个记录创建单独的符号表**
  - 首先创建一个新的符号表，压到栈顶
  - 然后处理对应于字段声明的D，字段都被加入到新符号表中
  - 最后根据栈顶的符号表构造出record类型表达式；符号表出栈

表达式和赋值语句的翻译

表达式代码的SDD

- **例：赋值语句：a=b* -c + b* -c**

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

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

a) 三地址代码

表达式代码的SDD

- **将表达式翻译成三地址指令序列**

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

top为当前符号表

code表示代码

addr表示存放表达式结果的地址（指针）- 临时变量

new Temp()可以生成一个临时变量

gen(...)生成一个指令

E为一个记录类型怎么办?

如何访问其某个字段

增量式翻译方案

- **主属性code满足增量式翻译的条件**

$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \mid\mid gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

gen依次执行直接生成指令序列

$S \rightarrow \mathbf{id} = E\ ;$	$\{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \}$
$E \rightarrow E_1 + E_2$	$\{\ E.addr = \mathbf{new}\ Temp\,();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \}$
$\mid\ - E_1$	$\{\ E.addr = \mathbf{new}\ Temp\,();$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr);\ \}$
$\mid\ (\ E_1\ )$	$\{\ E.addr = E_1.addr;\ \}$
$\mid\ \mathbf{id}$	$\{\ E.addr = top.get(\mathbf{id}.lexeme);\ \}$

数组元素的寻址

- **数组元素存储在一块连续的存储空间中，以方便快速的访问它们**

重要

  - n个数组元素是0,1,…,n-1进行顺序编号的
  - 假设每个数组元素宽度是w，那么数组A的第i个元素的开始地址为base + i*w
    - base是A[0]的相对地址

数组元素的寻址

- **二维数组**
  - $A[i_1][i_2]$表示第i_1行第i_2个元素。假设一行的宽度是w_1，同一行中每个元素的宽度是w_2，$A[i_1][i_2]$的相对地址是$base+i_1*w_1+i_2*w_2$
  - 根据第j维上的数组元素的个数n_j和该数组每个元素的宽度w进行计算的，如二维数组$A[i_1][i_2]$的地址$base+(i_1*n_2+i_2)*w$
- **对于k维数组$A[i_1][i_2]\ldots[i_k]$，推广**
  - $base+i_1*w_1+i_2*w2+\ldots+i_k*w_k$
  - 于k维数组$A[i_1][i_2]\ldots[i_k]$ 的地址$base+((\ldots(i_1*n_2+i_2)\ *n_3+i_3)\ldots)*n_k+i_k)*w$

数组元素的寻址

- **有时下标不一定从0开始，**
  - 比如一维数组编号low,low+1,…,high，此时base是A[low]的相对地址。计算A[i]的地址变成base+(i-low)*w
- **预先计算技术**
  - base+i*w和base+(i-low)*w都可以改写成i*w+c的形式，其中c=base-low*w可以在编译时刻预先计算出来，计算A[i]的相对地址只要计算i*w再加上c就可以了

数组元素的寻址

- **上述地址的计算是按行存放的**

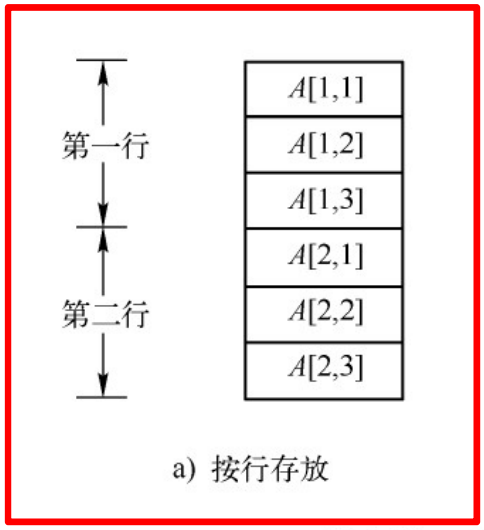


图 6-21　二维数组的存储布局

按行存放策略和按列存放策略可以推广到多维数组中

数组引用的翻译

- **为数组引用生成代码要解决的主要问题**
  - 数组引用的文法和地址计算相关联
- **假定数组编号从0开始，基于宽度来计算相对地址**
- **数组引用相关文法**
  - 非终结符号L生成一个数组名字加上一个下标表达式序列

$$L \rightarrow L[E] \mid \mathbf{id}[E]$$

数组引用生成代码的翻译方案

- **非终结符号L的三个综合属性**
  - L.addr指示一个临时变量，计算数组引用的偏移量$i_j * w_j$
  - L.array是一个指向数组名字对应的符号表条目的指针，L.array.base为该数组的基地址
  - L.type是L生成的子数组的类型，对于任何数组类型t，其宽度由t.width给出，t.elem给出其数组元素的类型

数组引用生成代码的翻译方案

核心是确定数组引用的地址

$S \rightarrow$	**id** = E ;	{ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$; }
|	L = E ;	{ $gen(L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr)$; }
$E \rightarrow$	E_1 + E_2	{ $E.addr = \textbf{new}\ Temp()$; $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }
|	**id**	{ $E.addr = top.get(\textbf{id}.lexeme)$; }
|	L	{ $E.addr = \textbf{new}\ Temp()$; $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; }

$L \rightarrow$	**id** [E]	{ $L.array = top.get(\textbf{id}.lexeme)$; $L.type = L.array.type.elem$; $L.addr = \textbf{new}\ Temp()$; $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width)$; }
|	L_1 [E]	{ $L.array = L_1.array$; $L.type = L_1.type.elem$; $t = \textbf{new}\ Temp()$; $L.addr = \textbf{new}\ Temp()$; $gen(t\ '='\ E.addr\ '*'\ L.type.width)$; $gen(L.addr\ '='\ L_1.addr\ '+'\ t)$; }

图 6-22　处理数组引用的语义动作

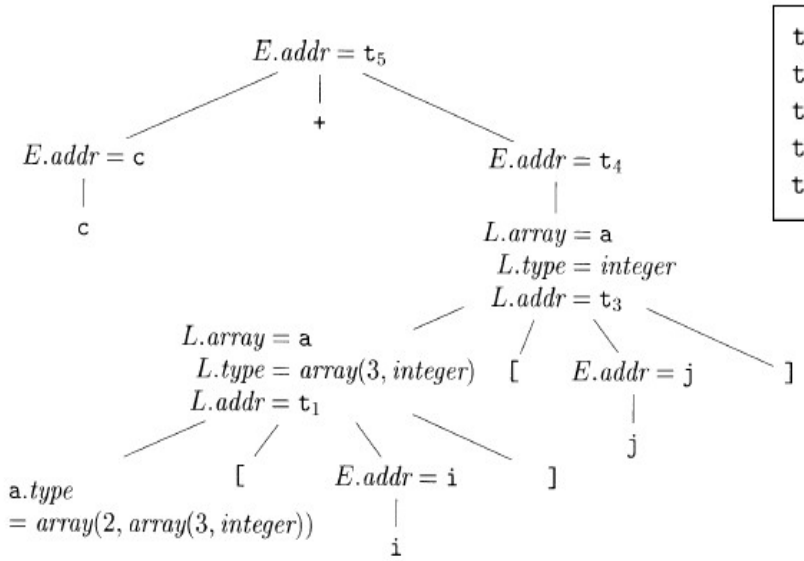
数组引用翻译示例

- **基于数组引用的翻译方案，表达式c+a[i][j]的注释语法树及三地址代码序列**
- **假设a是一个2*3的整数数组，c、i、j都是整数**
  - 设整数宽度为x(x=4)
  - 那么a的类型是array(2, array(3,integer)), a的宽度是6x=24
  - a[i]的类型是array(3,integer)，宽度是3x=12
  - a[i][j]的类型是整型

什么时候得到的这些信息?

数组引用翻译示例

$S \rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' E.*addr*); }
|	L = E ;	{ *gen*(L.*array* .*base* '[' L.*addr* ']' '=' E.*addr*); }
$E \rightarrow$	E_1 + E_2	{ E.*addr* = **new** *Temp* (); *gen*(E.*addr* '=' E_1.*addr* '+' E_2.*addr*); }
|	**id**	{ E.*addr* = *top.get*(**id**.*lexeme*); }
|	L	{ E.*addr* = **new** *Temp* (); *gen*(E.*addr* '=' L.*array*.*base* '[' L.*addr* ']'); }
$L \rightarrow$	**id** [E]	{ L.*array* = *top.get*(**id**.*lexeme*); L.*type* = L.*array*.*type*.*elem*; L.*addr* = **new** *Temp* (); *gen*(L.*addr* '=' E.*addr* '*' L.*type*.*width*); }
|	L_1 [E]	{ L.*array* = L_1.*array*; L.*type* = L_1.*type*.*elem*; t = **new** *Temp* (); L.*addr* = **new** *Temp* (); *gen*(t '=' E.*addr* '*' L.*type*.*width*); *gen*(L.*addr* '=' L_1.*addr* '+' t); }




```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

图 6-23　c + a[i][j]的注释语法分析树

L.addr指示一个临时变量，计算数组引用的偏移量
L.array是一个指向数组名字对应的符号表条目的指针，L.array.base为该数组的基地址
L.type是L生成的子数组的类型，对于任何数组类型t，其宽度由t.width给出，t.elem给出其数组元素的类型

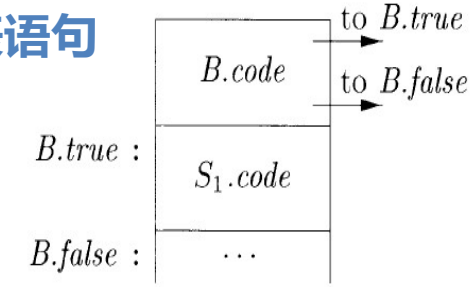
作业

- **教材p237：6.2.2 (2)**
- **教材p247：6.4.3 (2)**

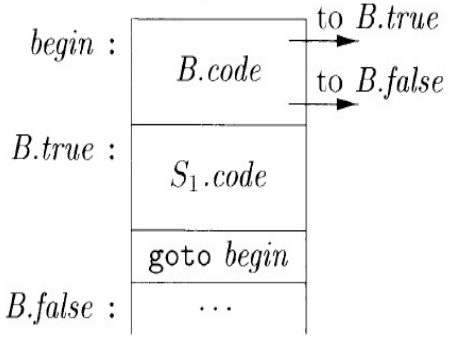
控制流翻译

控制流语句的翻译

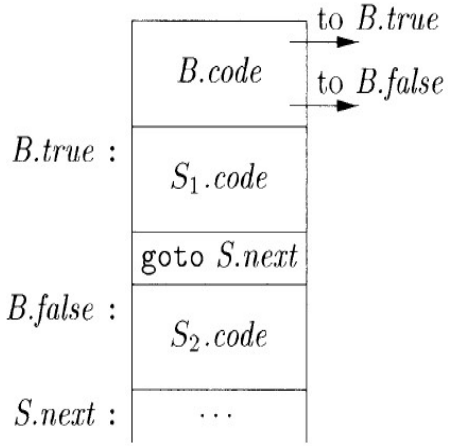
- **文法：B表示布尔表达式，S代表语句**
  - S → **if** (B) S_1
  - S → **if** (B) S_1 **else** S_2
  - S → **while** (B) S_1
- **代码的布局见右图**
- **继承属性**
  - B.true：B为真的跳转目标
  - B.false：B为假的跳转目标
  - S.next：S执行完毕时的跳转目标



a) if



c) while



b) if-else

控制流语句的翻译

控制流语句的翻译

$S \rightarrow$ if B then S_1

| if B then S_1 else S_2

| while B do S_1

| S_1; S_2

产生式	语义规则
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

图 6-36　控制流语句的语法制导定义

控制流语句的翻译

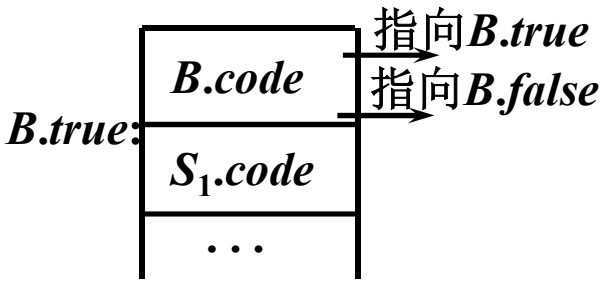
$S \rightarrow$ if B then S_1

{*B.true = newLabel*();

B.false = S.next;

S_1*.next = S.next*;

S.code = B.code || *gen*(*B.true*, ':') || S_1*.code* }



(a) if-then

label(B.true)

控制流语句的翻译

$S \rightarrow$ if B then S_1 else S_2

{*B.true* = *newLabel*();

B.false = *newLabel*();

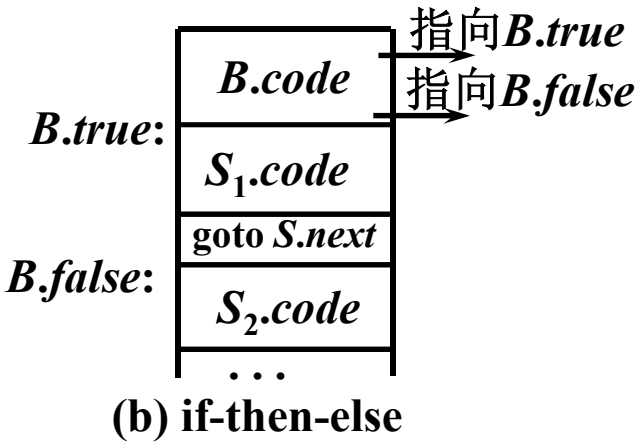
S_1.*next* = *S.next*;

S_2.*next* = *S.next*;

S.code = *B.code* || *gen*(*B.true*, ':') || S_1.*code* ||

gen('goto', *S.next*) || *gen*(*B.false*, ':') ||

S_2.*code*}



(b) if-then-else

控制流语句的翻译

$S \rightarrow$ while B do S_1

{*S.begin = newLabel*();

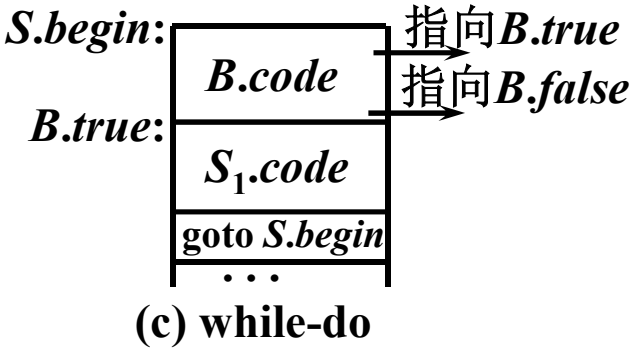
B.true = newLabel();

B.false = S.next;

S_1*.next = S.begin*;

S.code = gen(*S.begin*, ':') || *B.code* ||

gen(*B.true*, ':') || S_1*.code* || *gen*('goto', *S.begin*) }



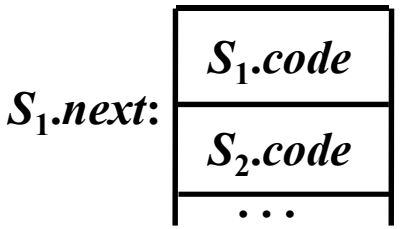
B.code怎么获得？

控制流语句的翻译

$S \rightarrow S_1; S_2$

{$S_1.next = newLabel()$; $S_2.next = S.next$;

$S.code = S_1.code \,||\, gen(S_1.next, ':') \,||\, S_2.code$ }



(d) $S_1; S_2$

布尔表达式翻译

布尔表达式

- **布尔表达式有两个基本目的**
  - 在控制流语句中用作条件表达式
  - 计算逻辑值
- **本节所用的布尔表达式文法**

$B \rightarrow B$ or B | B and B | not B | (B) | E relop E | true | false

布尔表达式

布尔表达式的完全计算

- 布尔值计算和算数表达式计算非常类似，可仿照算数表达式翻译方法，为每个产生式写出语义子程序：值的表示数值化

产生式	语义子程序
(1)$E \rightarrow E_a^{(1)}$ rop $E_a^{(2)}$	{T=NEWTEMP; GEN(rop, $E_a^{(1)}$•PLACE , $E_a^{(2)}$•PLACE ,T); E•PLACE =T }
(2)$E \rightarrow E^{(1)}$ bop $E^{(2)}$	{T=NEWTEMP; GEN(bop, $E_a^{(1)}$•PLACE , $E_a^{(2)}$•PLACE ,T); E•PLACE =T }

效率低下！！

(3)$E \rightarrow \neg E^{(1)}$	{T=NEWTEMP; GEN(¬ , $E^{(1)}$•PLACE , _ ,T); E•PLACE =T }
(4)$E \rightarrow (E^{(1)})$	{E•PLACE =$E^{(1)}$•PLACE }
(5)$E \rightarrow i$	{E•PLACE = ENTRY(i)}

布尔表达式

- **如何优化？布尔表达式的“短路”计算**
  - 用控制流来实现计算，即用程序中的位置来表示值，因为布尔表达式通常用来决定控制流走向
  - B_1 or B_2　定义成　if B_1 then true else B_2
  - B_1 and B_2 定义成　if B_1 then B_2 else false

两种不同计算方式会导致程序的结果不一样
有时B中有副作用

布尔表达式的控制流语句的翻译

- **布尔表达式可以用于改变控制流/计算逻辑值**
  - B → B||B | B && B | !B | (B) | E **rel** E | **true** | false
- **短路代码**
  - 通过跳转指令实现控制流的处理
  - 逻辑运算符本身不在代码中出现
- **语义**
  - $B_1||B_2$中B_1为真时，不计算B_2，整个表达式为真，因此，当B_1为真时可以跳过B_2的代码：定义成 if B_1 then true else B_2
  - $B_1\&\&B_2$中B_1为假时，可以不计算B_2，整个表达式为假：定义成 if B_1 then B_2 else false

布尔表达式的控制流翻译

产生式	语义规则
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ // 短路 $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ // 短路 $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$

布尔表达式的控制流翻译

$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\texttt{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \texttt{'goto'}\ B.true)$ $\mid\mid gen(\texttt{'goto'}\ B.false)$
$B \rightarrow$ **true**	$B.code = gen(\texttt{'goto'}\ B.true)$
$B \rightarrow$ **false**	$B.code = gen(\texttt{'goto'}\ B.false)$

如果*B*是*a* < *b*的形式，
那么代码是：

if *a* < *b* goto *B.true*
goto *B.false*

布尔表达式的控制流语句的翻译

- **例 表达式*a* < *b* or *c* < *d* and *e* < *f* 的代码是：**

if *a* < *b* goto L_{true}

goto L_1

L_1: if *c* < *d* goto L_2

goto L_{false}

L_2: if *e* < *f* goto L_{true}

goto L_{false}

B.true和B.false要根据具体应用场景来定

场景一：布尔表达式用于决定控制流走向

- **以 if (B) then S1 else S2 为例进行说明**
- **生成的代码执行时跳转到两个标号之一**
  - 表达式的值为真时，跳转到B.true
  - 表达式的值为假时，跳转到B.false

Where is B.true and B.false

场景一：布尔表达式用于决定控制流走向

- **B.true和B.false是两个继承属性，根据B所在的上下文指向不同的位置**
  - 如果B是if语句的条件表达式，分别指向then分支和else分支；如果没有else分支，则指向if语句的下一条指令
  - 如果B是while语句的条件表达式，分别指向循环体的开头和循环出口处

场景一：布尔表达式用于决定控制流走向

- **以 if (B) then S1 else S2 为例进行说明**
  - if (T) then S1 else S2
  - if (E1 rop E2) then S1 else S2
  - if (B1 || B2) then S1 else S2
  - if (B1 && B2) then S1 else S2
  - if (!B) then S1 else S2

场景一：布尔表达式用于决定控制流走向

- **短路代码例子：if (x<100 || x>200 && x!= y) x = 0;**

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

生成的中间代码

```
        if x < 100 goto L2
        if False x > 200 goto L1
        if False x != y goto L1
L2:     x = 0
L1:     接下来的代码
```

优化过的中间代码

场景二：布尔表达式用于赋值

- **程序中出现布尔表达式的目的可能就是求出它的值**

x=a<b && c<d

```
        ifFalse a < b goto L1
        ifFalse c < d goto L1
        t = true
        goto L2
L1:     t = false
L2:     x = t
```

图 6-42　通过计算一个临时变量的值来翻译一个布尔类型的赋值语句

场景三：General cases

- **处理方法**
  - 首先建立表达式的语法树，然后根据表达式的不同角色来处理
- **文法**
  - S → id = E; | if (E) S | while (E) S | S S
  - E → E||E | E && E | E rel E | ...
- **根据E的语法树结点所在的位置**
  - S → while (E) S_1中的E，生成跳转代码
  - 对于S → id = E，生成计算右值的代码

Switch语句的翻译

Switch语句翻译

- **分支数较少时**

switch *E*
begin
case V_1: S_1
case V_2: S_2
. . .
case V_{n-1}: S_{n-1}
default: S_n
end

t := E的代码
if t≠ V_1 goto L_1
S_1 的代码
goto next
L_1: if t≠ V_2 goto L_2
S_2 的代码
goto next
L_2: ...
...
L_{n-2}: if t≠ V_{n-1} goto L_{n-1}
S_{n-1}的代码
goto next
L_{n-1}: S_n 的代码
next:

Switch语句翻译

- **分支较多时，将分支测试代码集中在一起，便于生成较好的分支测试代码**

```
        t := E的代码
        goto test
L1:     S1 的代码
        goto next
L2:     S2 的代码
        goto next
        ...
Ln-1:   Sn-1的代码
        goto next
Ln:     Sn 的代码
        goto next
test:   if t= V1 goto L1
        if t= V2 goto L2
        ...
        if t=Vn-1 goto Ln-1
        goto Ln
next:
```

Switch语句翻译

- **中间代码增加一种case语句，便于代码生成器对它进行特别处理**

```
test:   case V1      L1
        case V2      L2
        . . .
        case Vn-1    Ln-1
        case  t  Ln
next:
```

简洁的办法，构造一个关系对照表，表中每个关系包含一个敞亮和对应的**label**，运行时进行查找。

N较小的时候，效率可接受
N较大的时候：二分查找，**hash table**，为每个可能值建立一个单元空间

过程调用的翻译

过程调用的翻译

例 6.25 假定 a 是一个整数数组，并且 f 是一个从整数到整数的函数。那么赋值语句

```
n = f(a[i]);
```

可以被翻译成如下的三地址代码。

```
1)  t1 = i * 4
2)  t2 = a [ t1 ]
3)  param t2
4)  t3 = call f, 1
5)  n = t3
```

语言中函数定义

D	$\rightarrow$	**define** T **id** (F) { S }
F	$\rightarrow$	ϵ | T **id** , F
S	$\rightarrow$	**return** E ;
E	$\rightarrow$	**id** (A)
A	$\rightarrow$	ϵ | E , A

D定义了函数的signature;
F定义了形参;
S定义了(返回)语句;
E定义了表达式，包含函数调用;
A定义了实参

过程调用翻译需要考虑的点

- **函数类型：包含返回类型和形参类型 --- 在其列表上应用fun构造算子构造；**
- **符号表：用栈来管理**
- **类型检查：注意类型转换的规则**
- **函数调用：为id(E, E, …, E)生成三地址代码时，只需对各个参数E生成三地址执行，将所有E.addr放到一个队列中，然后为每个参数生成一条param指令，清除队列**

作业

- **教材P263: 6.6.1**

回填

回填（1）

很重要

- **为布尔表达式和控制流语句生成目标代码的关键问题：某些跳转指令应该跳转到哪里**
- **例如：if (B) S**
  - 按照短路代码的翻译方法，B的代码中有一些跳转指令在B为假时执行，
  - 这些跳转指令的目标应该跳过S对应的代码,生成这些指令时，S的代码尚未生成，因此目标不确定
  - 通过语句的继承属性next来传递。需要第二趟处理
- **如何一趟处理完毕呢?**

回填（2）

- **基本思想**
  - 记录B的代码中跳转指令goto S.next，if … goto S.next的位置，但是不生成跳转目标
  - 这些位置被记录到B的综合属性B.falseList中
  - 当S.next的值已知时（即S的代码生成完毕时），把B.falseList中的所有指令的目标都填上这个值
- **回填技术**
  - 生成跳转指令时暂时不指定跳转目标标号，而是使用列表记录这些不完整的指令
  - 等知道正确的目标时再填写目标标号
  - 每个列表中的指令都指向同一个目标

布尔表达式的回填翻译（1）

- **布尔表达式用于语句的控制流时，它总是在取值true时和取值false时分别跳转到某个位置**
- **引入两个综合属性**
  - truelist: 包含跳转指令（位置）的列表，这些指令在取值true时执行
  - falselist：包含跳转指令（位置）的列表，这些指令在取值false时执行
- **辅助函数**
  - Makelist(i)：创建一个只包含i的列表
  - Merge(p1,p2)：将p1和p2指向的列表合并
  - Backpatch(p,i)：将i作为目标标号插入到p所指列表中的各指令中

布尔表达式的回填翻译（2）

1)	$B \rightarrow B_1 \ \| \| \ M \ B_2$	{ $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }
2)	$B \rightarrow B_1 \ \&\& \ M \ B_2$	{ $backpatch(B_1.truelist, M.instr)$; $B.truelist = B_2.truelist$; $B.falselist = merge(B_1.falselist, B_2.falselist)$; }
3)	$B \rightarrow \ ! \ B_1$	{ $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
4)	$B \rightarrow (\ B_1 \ )$	{ $B.truelist = B_1.truelist$; $B.falselist \ = \ B_1.falselist$; }
5)	$B \rightarrow E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}' \ E_1.addr$ **rel**$.op \ E_2.addr \ '\texttt{goto} \ \_')$; $gen('\texttt{goto} \ \_')$; }
6)	$B \rightarrow$ **true**	{ $B.truelist = makelist(nextinstr)$; $gen('\texttt{goto} \ \_')$; }
7)	$B \rightarrow$ **false**	{ $B.falselist = makelist(nextinstr)$; $gen('\texttt{goto} \ \_')$; }
8)	$M \rightarrow \epsilon$	{ $M.instr = nextinstr$; }

回填和非回填方法的比较

$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\text{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \text{'goto'}\ B.true)$ $\mid\mid gen(\text{'goto'}\ B.false)$
$B \rightarrow$ **true**	$B.code = gen(\text{'goto'}\ B.true)$
$B \rightarrow$ **false**	$B.code = gen(\text{'goto'}\ B.false)$

$B \rightarrow E_1$ **rel** E_2	$\{\ B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $gen(\text{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \text{'goto'}\ \_');$ $gen(\text{'goto'}\ \_');\ \}$
$B \rightarrow$ **true**	$\{\ B.truelist = makelist(nextinstr);$ $gen(\text{'goto'}\ \_');\ \}$
$B \rightarrow$ **false**	$\{\ B.falselist = makelist(nextinstr);$ $gen(\text{'goto'}\ \_');\ \}$

- 回填时生成指令坯，然后加入相应的list
- 原来跳转到B.true的指令，现在被加入到B.truelist中

回填和非回填方法的比较

$$
\begin{array}{l|l}
B \rightarrow B_1 \;||\; B_2 & B_1.true = B.true \\
& B_1.false = newlabel() \\
& B_2.true = B.true \\
& B_2.false = B.false \\
& B.code = B_1.code \;||\; label(B_1.false) \;||\; B_2.code
\end{array}
$$

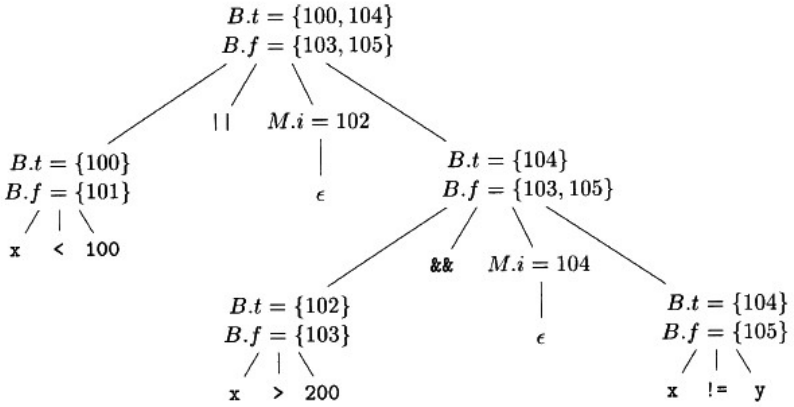
回填；并将真假出口相对应

$$
\begin{array}{ll}
B \rightarrow B_1 \;||\; M \; B_2 & \{\; backpatch(B_1.falselist, M.instr); \\
& \quad B.truelist = merge(B_1.truelist, B_2.truelist); \\
& \quad B.falselist = B_2.falselist; \;\} \\
M \rightarrow \epsilon & \{\; M.instr = nextinstr; \;\}
\end{array}
$$

- true/false属性的赋值，在回填方案中对应为相应的list的赋值或者merge
- 原来生成label的地方，在回填方案中使用M来记录相应的代码位置，M.inst需要对应label的标号
- 原方案生成的指令goto B_1.false，现在生成了goto M.inst

布尔表达式的回填例子

- **x<100 || x>200 && x!=y**




```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto _
103:  goto _
104:  if x != y goto _
105:  goto _
```

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

a) 将 104 回填到指令 102 中之后

```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

b) 将 102 回填到指令 101 中之后

1)	$B \rightarrow B_1$ **||** $M\ B_2$	{ *backpatch*(B_1.*falselist*, M.*instr*); B.*truelist* = *merge*(B_1.*truelist*, B_2.*truelist*); B.*falselist* = B_2.*falselist*; }
2)	$B \rightarrow B_1$ **&&** $M\ B_2$	{ *backpatch*(B_1.*truelist*, M.*instr*); B.*truelist* = B_2.*truelist*; B.*falselist* = *merge*(B_1.*falselist*, B_2.*falselist*); }
3)	$B \rightarrow$ **!** B_1	{ B.*truelist* = B_1.*falselist*; B.*falselist* = B_1.*truelist*; }
4)	$B \rightarrow$ **(** B_1 **)**	{ B.*truelist* = B_1.*truelist*; B.*falselist* = B_1.*falselist*; }
5)	$B \rightarrow E_1$ **rel** E_2	{ B.*truelist* = *makelist*(*nextinstr*); B.*falselist* = *makelist*(*nextinstr* + 1); *gen*('**if**' E_1.*addr* **rel**.*op* E_2.*addr* '**goto** _'); *gen*('**goto** _'); }
6)	$B \rightarrow$ **true**	{ B.*truelist* = *makelist*(*nextinstr*); *gen*('**goto** _'); }
7)	$B \rightarrow$ **false**	{ B.*falselist* = *makelist*(*nextinstr*); *gen*('**goto** _'); }
8)	$M \rightarrow \epsilon$	{ M.*instr* = *nextinstr*; }

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

控制转移语句的回填

$$S \rightarrow \textbf{if}\,(\,B\,)\,S \mid \textbf{if}\,(\,B\,)\,S\,\textbf{else}\,S \mid \textbf{while}\,(\,B\,)\,S$$
$$\mid \{\,L\,\} \mid A$$
$$L \rightarrow L\,S \mid S$$

- **语句的综合属性：nextlist**
  - nextlist中的跳转指令的目标应该是S执行完毕之后紧接着执行的下一条指令的位置
  - 考虑S是while语句、if语句的子语句时，分别应该跳转到哪里

控制转移语句的回填

- M的作用就是用M.instr记录下一个指令的位置
- N的作用是生成goto指令坯，N.nextlist只包含这个指令的位置

1) $S \rightarrow \textbf{if}\ (\ B\ )\ M\ S_1$ { $backpatch(B.truelist,\ M.instr)$;
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist)$; }

2) $S \rightarrow \textbf{if}\ (\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$
{ $backpatch(B.truelist,\ M_1.instr)$;
$backpatch(B.falselist,\ M_2.instr)$;
$temp\ =\ merge(S_1.nextlist,\ N.nextlist)$;
$S.nextlist\ =\ merge(temp,\ S_2.nextlist)$; }

6) $M \rightarrow \epsilon$ { $M.instr\ =\ nextinstr$; }

7) $N \rightarrow \epsilon$ { $N.nextlist\ =\ makelist(nextinstr)$;
$gen('\texttt{goto}\ \_')$; }

控制转移语句的回填

3) $S \rightarrow$ **while** M_1 (B) M_2 S_1
{ *backpatch*(S_1.*nextlist*, M_1.*instr*);
backpatch(B.*truelist*, M_2.*instr*);
S.*nextlist* = B.*falselist*;
gen('goto' M_1.*instr*); }

4) $S \rightarrow$ { L } { S.*nextlist* = L.*nextlist*; }

5) $S \rightarrow A$; { S.*nextlist* = **null**; }

8) $L \rightarrow L_1$ M S { *backpatch*(L_1.*nextlist*, M.*instr*);
L.*nextlist* = S.*nextlist*; }

9) $L \rightarrow S$ { L.*nextlist* = S.*nextlist*; }

Break、Continue的处理

- **虽然break、continue在语法上是一个独立的句子，但是它的代码和外围语句相关**
- **方法：(break语句)**
  - 跟踪外围语句S，
  - 生成一个跳转指令坯
  - 将这个指令坯的位置加入到S的nextlist中
- **跟踪的方法**
  - 在符号表中设置break条目，令其指向外围语句
  - 在符号表中设置指向S的nextlist的指针，然后把这个指令坯的位置直接加入到nextlist中

作业

- **教材P268: 6.7.1**
- **给定下列代码**

```
do {
  x = y + z;
  if (a > b || !(c > d)) continue;
  else x = x + 1;
} while (e > f && !(g > h || i > j));
```

其对应的三地址码如下所示

```
L0: t0 := y + z                |      x := t1
    x := t0                    | L1: [   ] (e > f) goto L__
    [   ] (a > b) goto L__     |     [   ] (g > h) goto L__
    [   ] (c > d) goto L__     |     [   ] (i > j) goto L__
    t1 := x + 1                | L2:
```

试为其中空白“__”填上正确的标号编号，并为空白“[ ]”填上 if 或 ifnot.

类型检查

什么是类型

- **一个程序变量在程序执行期间的值可以设想为有一个范围，这个范围的一个界叫做该变量的类型。**
  - 变量都被给定类型的语言叫做类型化语言（typed language）。
  - 语言若不限制变量值的范围，则被称作未类型化的语言（untyped language ）

类型系统

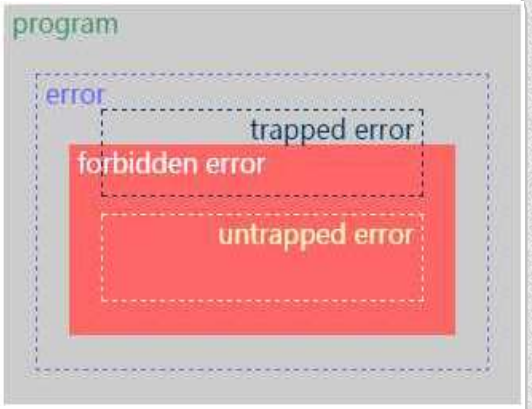
- 类型化语言的类型系统（type system）是该语言的一个组成部分，它始终监视着程序中变量的类型，通常还包括所有表达式的类型。
- 一个类型系统主要由一组定型规则（typing rules）构成，这组规则用来给各种语言构造（程序、语句、表达式等）指派类型。

类型系统

- **程序运行时的执行错误分成两类**
  - 会被捕获的错误（*trapped error*）
    - 例：非法指令错误、非法内存访问、除数为零
    - 引起计算立即停止
  - 不会被捕获的错误（*untrapped error*）
    - 例：下标变量的访问越过了数组的末端
    - 例：跳到一个错误的地址，该地址开始的内存正好代表一个指令序列
    - 错误可能会有一段时间未引起注意

类型系统

- **禁止错误（*forbidden error*）**
  - *untrapped error*集合 + *trapped error*的一个子集
  - 为语言设计类型系统的目标是在排除禁止错误



- **良行为的程序(well-behaved)**
  - A program fragment that will not produce forbidden errors at run time (不同场合对良行为的定义略有区别)
- **安全语言**
  - 任何合法程序都没有`forbidden error`

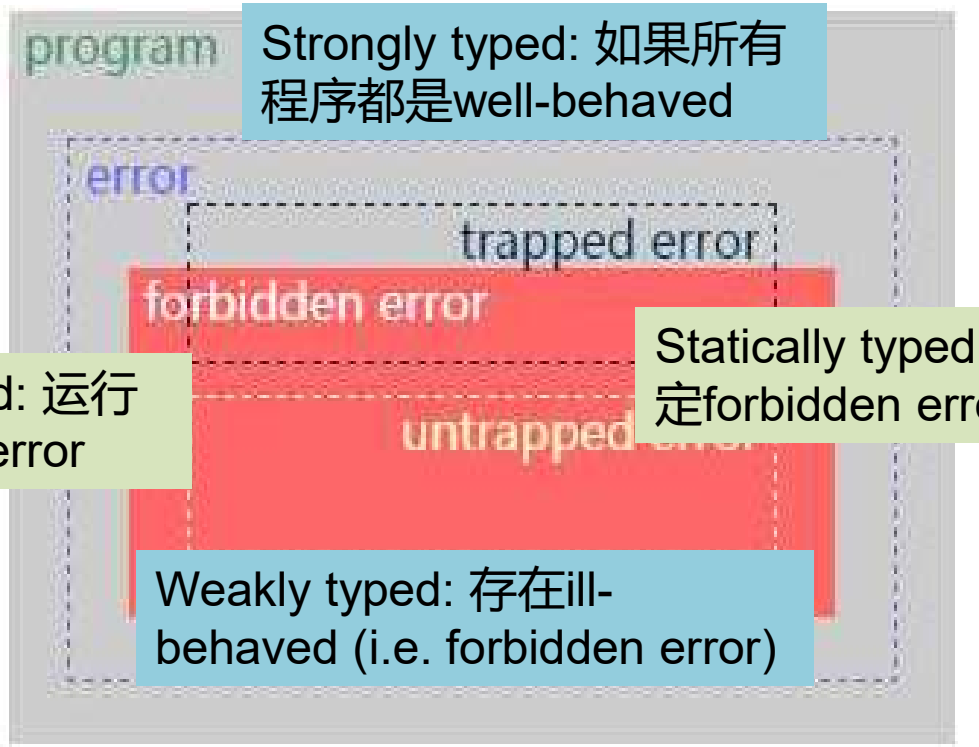
类型系统

- **类型化的语言**
  - 变量都被给定类型的语言：表达式、语句等程序构造的类型都可以静态确定，例如，类型*boolean*的变量*x*在程序每次运行时的值只能是布尔值，*not* (*x*)总有意义
- **未类型化的语言**
  - 不限制变量值范围的语言：一个运算可以作用到任意的运算对象，其结果可能是一个有意义的值、一个错误、一个异常或一个语言未加定义的结果, 例如：LISP语言

类型系统

- **显式类型化语言**
  - 类型是语法的一部分
- **隐式类型化的语言**
  - 不存在隐式类型化的主流语言，但可能存在忽略类型信息的程序片段，例如不需要程序员声明函数的参数类型

类型系统


program
error
trapped error
forbidden error
untrapped
Strongly typed: 如果所有程序都是well-behaved
Statically typed: 编译时确定forbidden error
Dynamically typed: 运行时确定forbidden error
Weakly typed: 存在ill-behaved (i.e. forbidden error)

类型表达式

- **类型本身也有结构，我们使用类型表达式 (type expression)来表示这种结构**
  - 基本类型：Boolean, integer, float, char, void；或
  - 类名；或

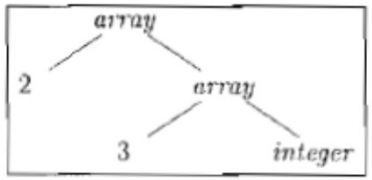


图 6-14　int[2][3]的类型表达式

类型表达式

- **类型本身也有结构，我们使用类型表达式 (type expression)来表示这种结构**
  - 通过将“类型构造算子”作用于类型表达式而得，例如：
    - array[数字，类型表达式]
    - record[字段/类型对的列表]（可以用符号表表示）
    - 函数类型构造算子→：参数类型→结果类型
    - 笛卡尔积：s X t: struct { int a[10]; float f;} st 对应于：record((a × array(0..9, int)) × (f × real))
    - 可以包含取值为类型表达式的变量

类型表达式的例子

- **类型例子**
  - 元素个数为3X4的二维数组
  - 数组的元素的记录类型
  - 该记录类型中包含两个字段: x和y,其类型分别是float和integer
- **类型表达式**
  - array[3,
    array[4, record[(x,float),(y,integer)]
    ]
    ]

类型等价

- **不同的语言有不同的类型等价的定义**
- **结构等价**
  - 或者它们是相同的基本类型
  - 或者是相同的构造算子作用于结构等价的类型而得到的。
  - 或者一个类型是另一个类型表达式的名字
- **名等价**
  - 类型名仅仅代表其自身

静态类型信息在编译中的作用

- **应用：静态类型检查**
  - 编译时确定forbidden error
- **类型系统**
  - 给每一个组成部分赋予一个类型表达式
  - 通过一组逻辑规则来表示这些类型表达式必须满足的条件
- **可发现错误、提高代码效率、确定临时变量的大小…**

类型系统的分类

- **类型综合**
  - 根据子表达式的类型构造出表达式的类型

    if f 的类型为s → t且x的类型为s

    then f(x)的类型为t
- **类型推导**
  - 根据语言结构的使用方式来确定该结构的类型：

    if f(x)是一个表达式

    then 对于某些类型α,β；f的类型为α → β且x的类型为α

类型转换

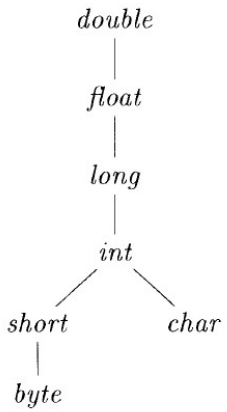
- **假设在表达式x*i中，x为浮点数、i为整数，则结果应该是浮点数**
  - x和i使用不同的二进制表示方式
  - 浮点*和整数*使用不同的指令
  - t1 = (float) i
  - t2 = x fmul t1
- **类型转换比较简单时的SDD**
  - E → E1 + E2 {
    if(E1.type = integer and E2.type = integer) E.type = integer;
    else if (E1.type = float and E2.type= integer) E.type = float;
    }
  - 这个规则没有考虑生成类型转换代码

类型的widening和narrowing

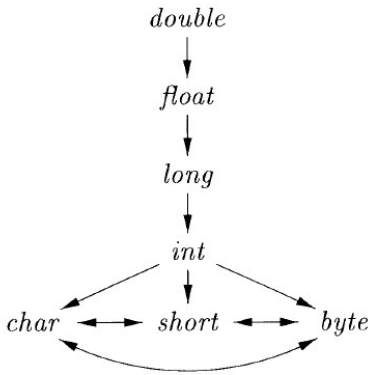
- **不同语言有不同的类型转换规则，例如，Java转换规则区分了拓宽和窄化转换**
  - 拓宽：保持原有信息

    较低层的类型可以被拓宽为较高层的类型
  - 窄化：可能丢失信息

    如果存在一条链s → t, 则s可以被转化为t



a) 拓宽类型转换



b) 窄化类型转换

类型的widening和narrowing

- **编译器自动完成的转换为隐式转换（也成为自动类型转化）**
  - 很多语言中，仅限于拓宽
- **程序员用代码指定的转换为显式转换（也成为强制类型转化）**

类型的widening和narrowing

- **检查E → E1 + E2的语义动作用到了两个函数**
  - 函数Max求的是两个参数在拓宽层次结构中的最大者，或最小公共祖先
    - 对于不存在于层次结构中的类型(e.g. 数组)，报错
  - Widen函数已经生成了必要的类型转换代码
    - Widen (a, t, w)将类型为t的地址a中的内容转换为类型为w的值
    - t和w相同类型，返回a本身

$$
\begin{array}{l}
E \rightarrow E_1 + E_2 \quad \{\ E.type = max(E_1.type, E_2.type); \\
\qquad\qquad\qquad\quad\ \ a_1 = widen(E_1.addr, E_1.type, E.type); \\
\qquad\qquad\qquad\quad\ \ a_2 = widen(E_2.addr, E_2.type, E.type); \\
\qquad\qquad\qquad\quad\ \ E.addr = \textbf{new}\ Temp\,(); \\
\qquad\qquad\qquad\quad\ \ gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{array}
$$

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

函数/运算符的重载

- **通过查看参数来解决函数重载问题**
- **$E \rightarrow f(E_1)$**

  { **if** f.typeset = $\{s_i \rightarrow t_i | 1 <= i <= k\}$ and E_1.type=s_k

  then E.type = t_k

  }

Thank you!