第六章　中间代码生成

概要

- 中间代码表示
- 类型和声明翻译
- 表达式翻译
- 类型检查
- 控制流翻译
- 总结

编译器的前端

- 前端是对源语言进行分析并产生中间表示
- 处理与源语言相关的细节，与目标机器无关
- 前端后端分开的好处：不同的源语言、不同的机器可以得到不同的编译器组合

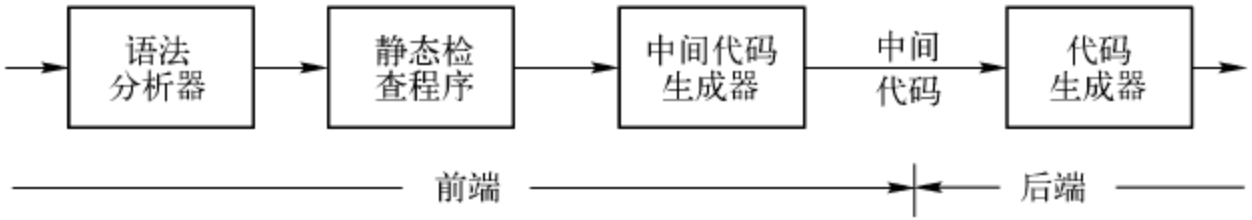
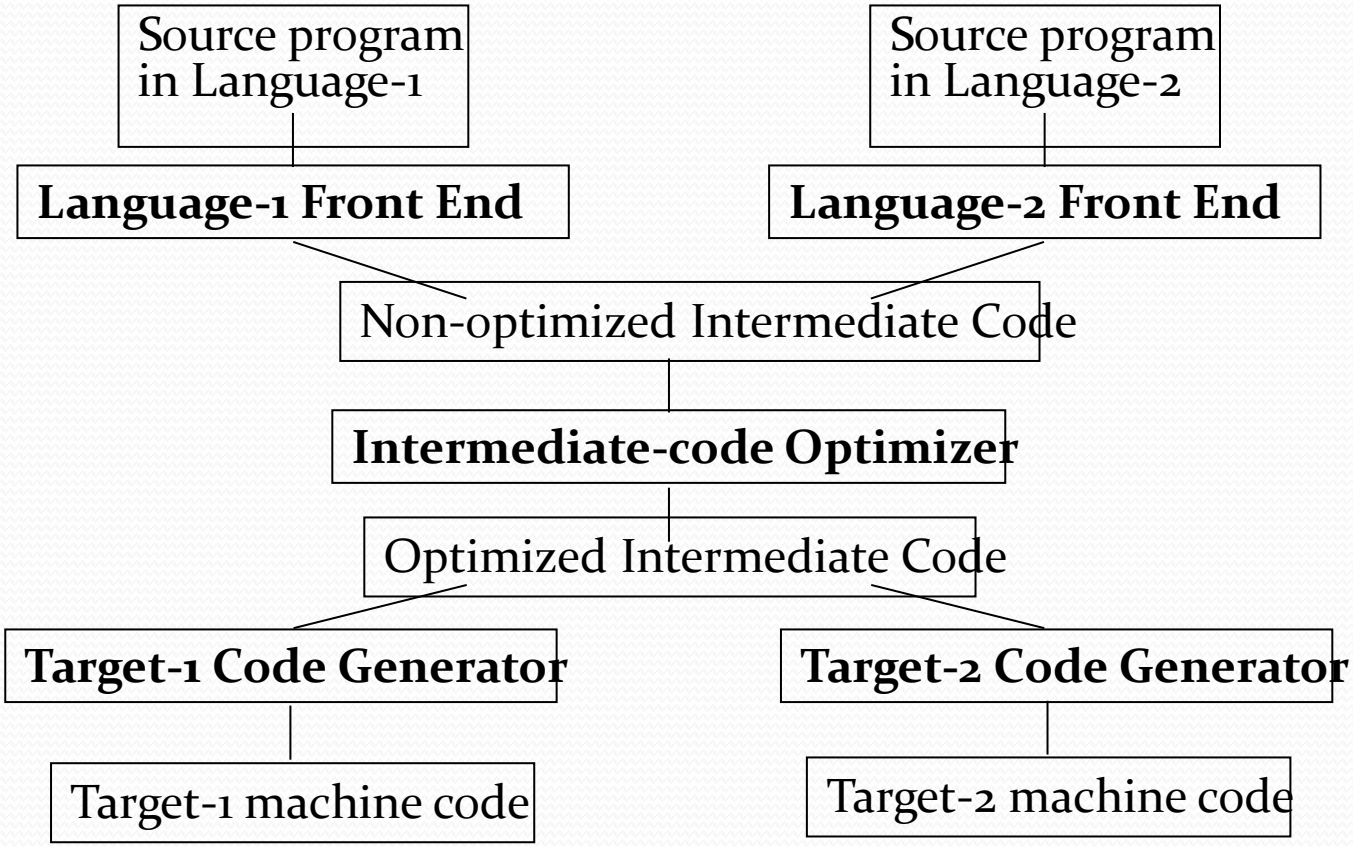


图 6-1　一个编译器前端的逻辑结构

中间代码表示及其好处

- 形式
  - 多种中间表示，不同层次
  - 抽象语法树
  - 三地址码
- 重定位
  - 为新的机器建编译器，只需要做从中间代码到新的目标代码的翻译器（前端独立）
- 高层次的优化
  - 优化与源语言和目标机器都无关

建立组合编译的做法


Source program in Language-1
Source program in Language-2
Language-1 Front End
Language-2 Front End
Non-optimized Intermediate Code
Intermediate-code Optimizer
Optimized Intermediate Code
Target-1 Code Generator
Target-2 Code Generator
Target-1 machine code
Target-2 machine code

中间代码的实现

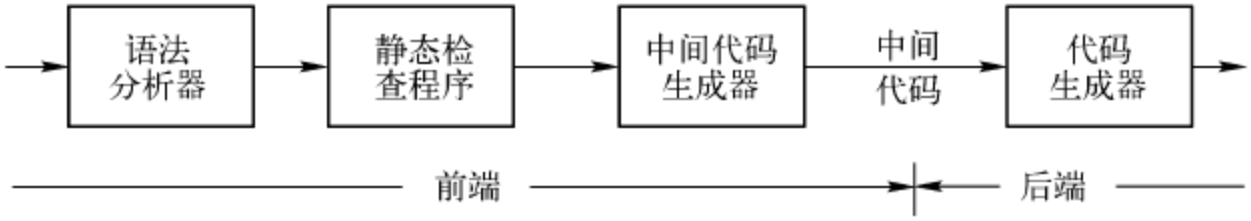


图 6-1　一个编译器前端的逻辑结构

- 静态类型检查和中间代码生成的过程都可以用语法制导的翻译来描述和实现
- 对于抽象语法树这种中间表示的生成，第五章已经介绍过

生成语法分析树的语法制导定义

	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)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

- a+a*(b-c)+(b-c)*d 的抽象语法树

表达式的有向无环图

- 语法树中，公共子表达式每出现一次，就有一个对应的子树
- 表达式的有向无环图(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。

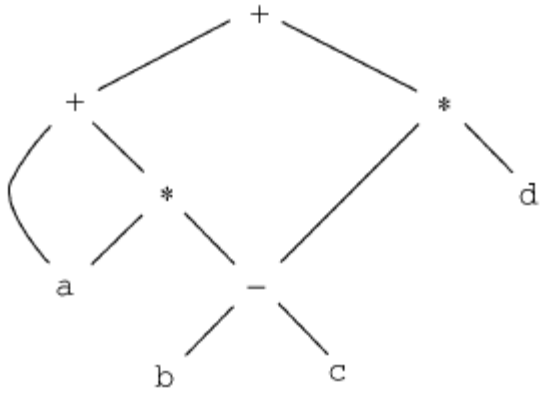


图 6-3　表达式 a + a * (b - c) + (b - c) * d 的 DAG

DAG构造

- 可以用和构造抽象语法树一样的SDD来构造
- 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

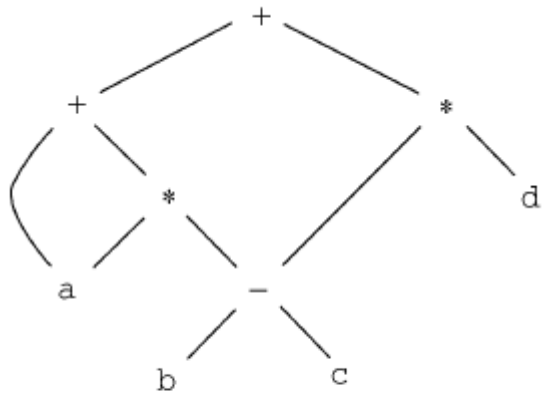


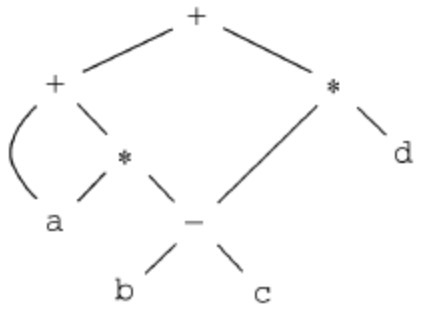
图 6-3　表达式 a + a * (b - c) + (b - c) * d 的 DAG

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

三地址代码

- 三地址代码是抽象语法树或DAG的线性表示形式
- 三地址代码中，一条指令右侧最多有一个运算符。三地址码将多运算符算数表达式和控制流语句进行拆分。



a) DAG

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

b) 三地址代码

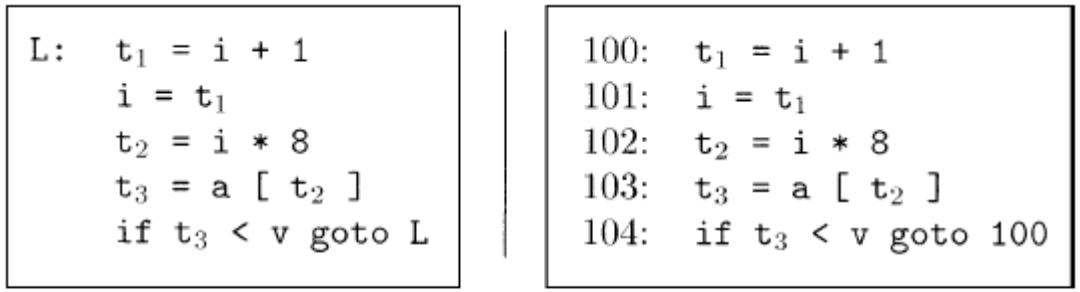
图 6-8　一个 DAG 及其对应的三地址代码

三地址代码中的地址和指令

- 地址：
  - 名字（变量）
  - 常量
  - 编译器生成的临时变量
- 指令形式：
  - 赋值指令 x=y op z, x=op y, x=y
  - 无条件转移指令 goto L
  - 条件转移指令 if x goto L, if False x goto L
  - 带有关系运算符的转移指令 if x relop y goto L
  - 过程调用和返回指令 param x, call p,n , return y
  - 带下标的复制指令x=y[i], x[i]=y
  - 地址和指针赋值指令 x=&y x=*y *x=y
- 三地址码的指令足以描述语言的各种构造，并且可以用于高效地生成目标机器代码

三地址指令示例

- 语句 do i=i+1; while (a[i]<v);



a) 符号标号　　b) 位置号

图 6-9　给三地址指令指定标号的两种方法

类型和声明

- 类型检查
  - 利用一组逻辑规则来推理一个程序在运行时刻的行为。明确地讲，类型检查保证运算分量的类型和运算符的预期类型一致。
- 翻译时
  - 根据一个名字的类型，编译器确定这个名字在运行时刻需要多大的存储空间。
- 在过程或类中声明的变量，要考虑其存储空间的布局问题
  - 编译时刻，用相对地址（相对于数据区域开始位置的偏移量）进行布局。

类型表达式

- 类型表达式：用来表示类型的结构
  - 可能是基本类型
  - 也可能通过“类型构造算子”运算符作用于类型表达式得到
- 如int[2][3]，表示两个数组组成的数组。array(2,array(3,integer))
- array是运算符，有两个参数：数字和类型

类型表达式的定义

- 基本类型是一个类型表达式。如boolean char integer float void
- 类型名是一个类型表达式
- 类型构造算子array作用于一个数字和一个类型表达式得到一个类型表达式
- 类型构造算子record作用于字段名和相应的类型可以得到一个类型表达式
- 应用类型构造算子→可以构造得到函数类型的类型表达式
- 如果s,t是类型表达式，其笛卡尔积s→ t也是类型表达式
- 类型表达式中可以包含取值为类型表达式的变量
- 示例：struct {int a[10]; float f} st;

  record((a →array(0..9, int)) × (f→real))

类型的声明

- 处理基本类型、数组类型或记录类型的文法

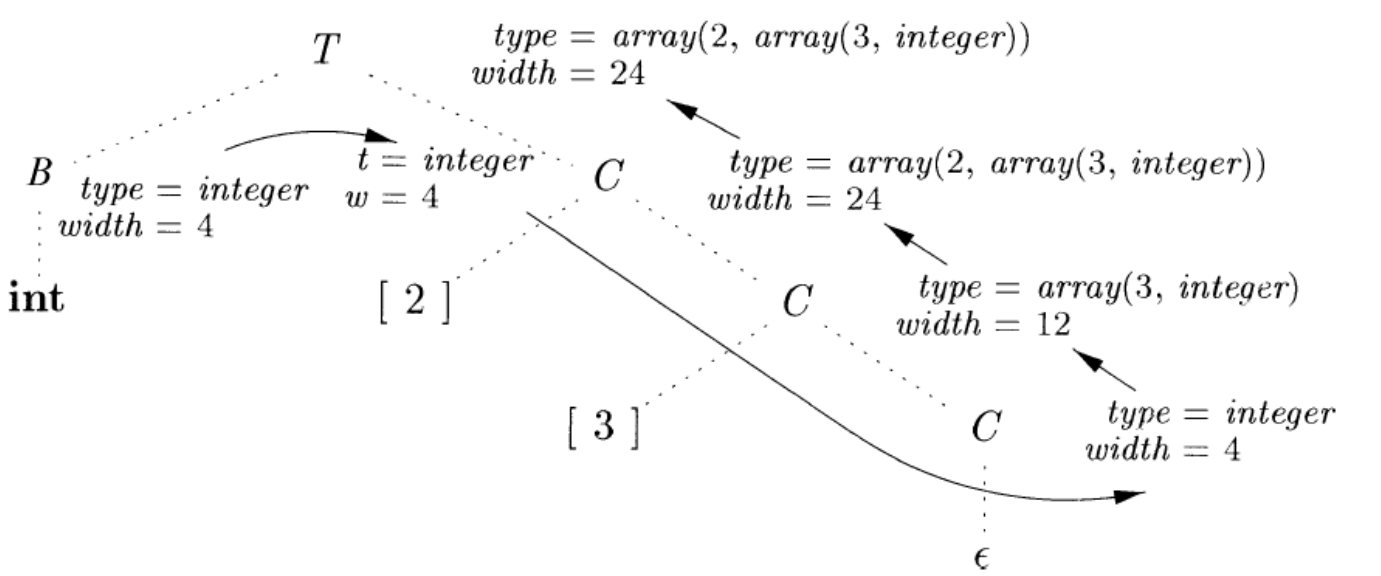
$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

- 应用该文法及其对应的语法制导定义，除了进行得到类型表达式之外，还得进行各种类型的存储布局

局部变量名的存储布局

- 对于在编译时刻即可知道其在运行时刻需要的内存数量的名字，我们可以根据其类型，为每个名字分配一个相对地址。
- 类型和相对地址信息保存在符号表中
- 约定：
  - 存储区域是连续的字节块
  - 字节是可寻址的最小内存单位
  - 一个字节通常是一个8个二进制位，若干字节组成一个机器字
- 类型的宽度：该类型一个对象所需的存储单元的数量。比如，一个整型数的宽度是4。一个浮点数的宽度是8。

计算



- 分析其中的属性、变量

T → *B* *{t=B.type; w=B.width;}*
C *{T.type=C.type;T.width=C.width;}*

B → int *{B.type=interger; B.width=4;}*

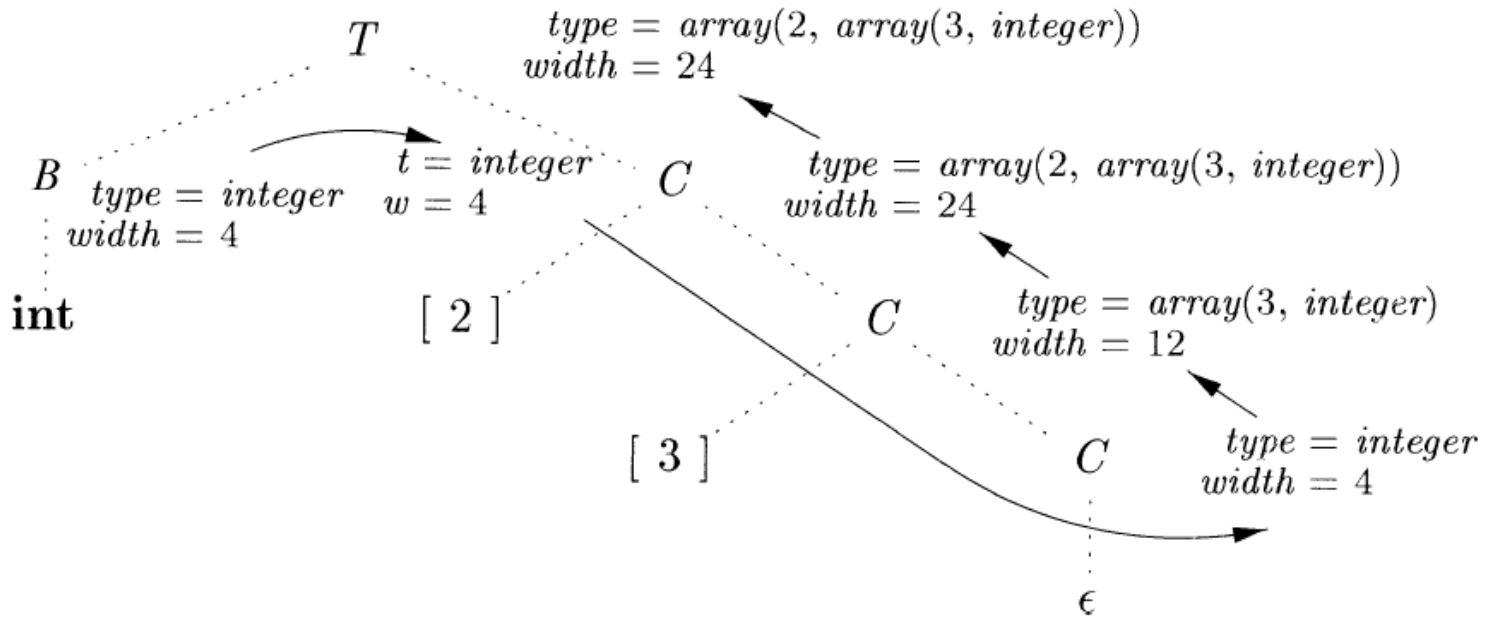
B → float *{B.type=float; B.width=8;}*

C →ε *{C.type=t; C.width=w;}*

C →[num] *C1* *{C.type=array(*num.*value,C1.type);*
*C.width=*num.*value*C1.width}*

数组类型的语法制导翻译

- 分析求值过程 int[2][3]



声明序列

- 插入符号表条目，跟踪下一个可用的相对地址

$P \rightarrow$	D	$\{\ offset = 0;\ \}$
$D \rightarrow$	T **id** ; D_1	$\{\ top.put(\textbf{id}.lexeme,\ T.type,\ offset);$ $offset = offset + T.width;\ \}$
$D \rightarrow$	ϵ	

图 6-17　计算被声明变量的相对地址

记录和类中的字段

- 记录变量声明的翻译方案
- 约定：
  - 一个记录中各个字段的名字必须互不相同
  - 字段名的偏移量（相对地址），是相对于该记录的数据区字段而言的。
- 记录类型使用一个专用的符号表，对它们的各个字段类型和相对地址进行单独编码。
- 如记录类型record(t),record是类型构造算子，t是一个符号表对象，保存该记录类型的各个字段信息。

记录和类中的字段(续)

$T \rightarrow \textbf{record}\ '\{'\ D\ '\}'$

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*(); }

注：记录类型存储方式可以推广到类

表达式的翻译

- 从表达式到三地址代码的翻译
- 表达式的运算
- *S.code*和*E.code*属性分别表示*S*和*E*对应的三地址代码，*E.addr*属性表示存放*E*的值的地址

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

表达式翻译成三地址代码示例

- a=b+-c

```
t1 = minus c
t2 = b + t1
a = t2
```

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

图 6-19　表达式的三地址代码

表达式的增量翻译

- 类似于上一章中所述边扫描边生成
- gen不仅构造新的三地址指令，还要将它添加到至今为止已生成的指令序列之后。
- 不需要code指令保存已有的代码，而是对gen的连续调用生成一个指令序列

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

图 6-20　增量生成表达式的三地址代码

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
- 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。
- 推广到二维或多维数组。$A[i_1][i_2]$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$base+i_1*w_1+i_2*w_2$
- 对于k维数组$A[i_1][i_2]...[i_k]$，推广$base+i_1*w_1+i_2*w_2+...+i_k*w_k$

数组元素的寻址(续)

- 另一种计算数组引用相对地址的方法，是根据第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$A[i_1][i_2]...[i_k]$ 的地址$base+((...(i_1*n_2+i_2)*n_3+i_3)...)*n_k+i_k)*w$
- 有时下标不一定从o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：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(续)

- 上述地址的计算是按行存放的

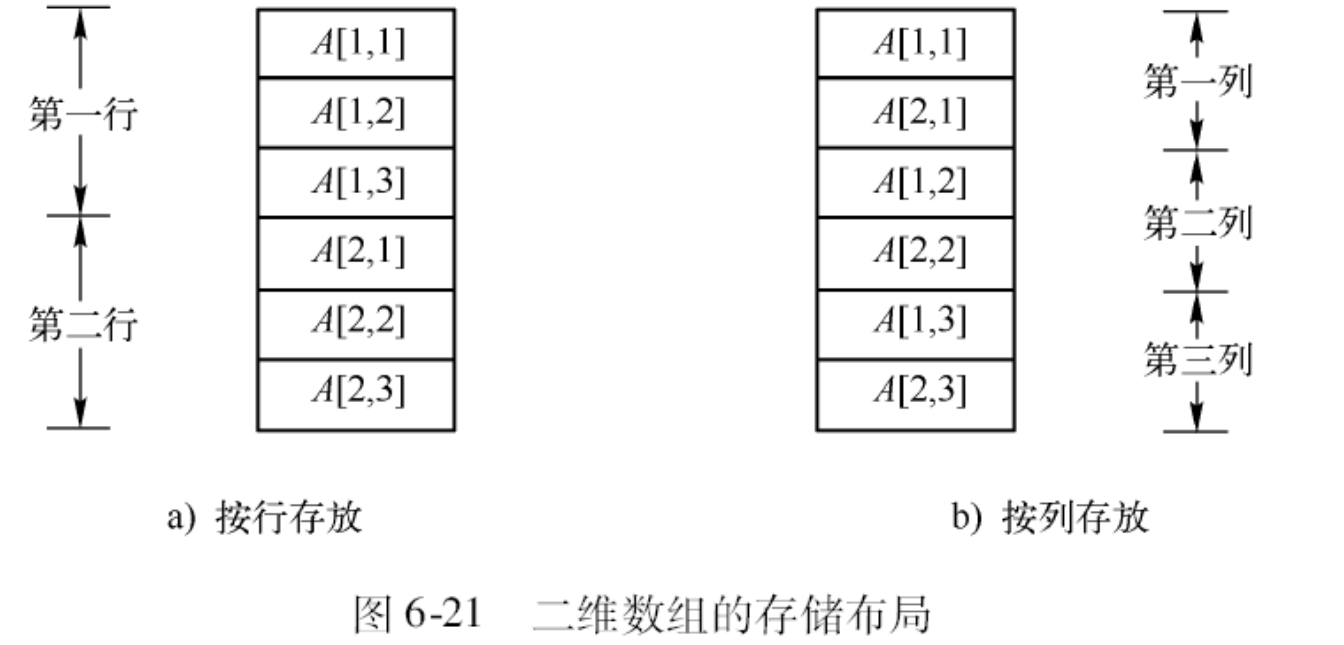


图 6-21　二维数组的存储布局

- 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o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
  - 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

$S \rightarrow$	**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** $Temp()$; $gen(t$ $'='$ $E.addr$ $'*'$ $L.type.width)$; $gen(L.addr$ $'='$ $L_1.addr$ $'+'$ $t)$; }

图 6-22　处理数组引用的语义动作

核心是确定数组引用的地址

数组引用翻译示例

- 假设a是一个2*3的整数数组，c、i、j都是整数。
- 那么a的类型是array(2, array(3,integer)), a的宽度是24。a[i]的类型是array(3,integer)，宽度是12。a[i][j]的类型是整型。
- 基于数组引用的翻译方案，表达式c+a[i][j]的注释树及三地址代码序列如下：

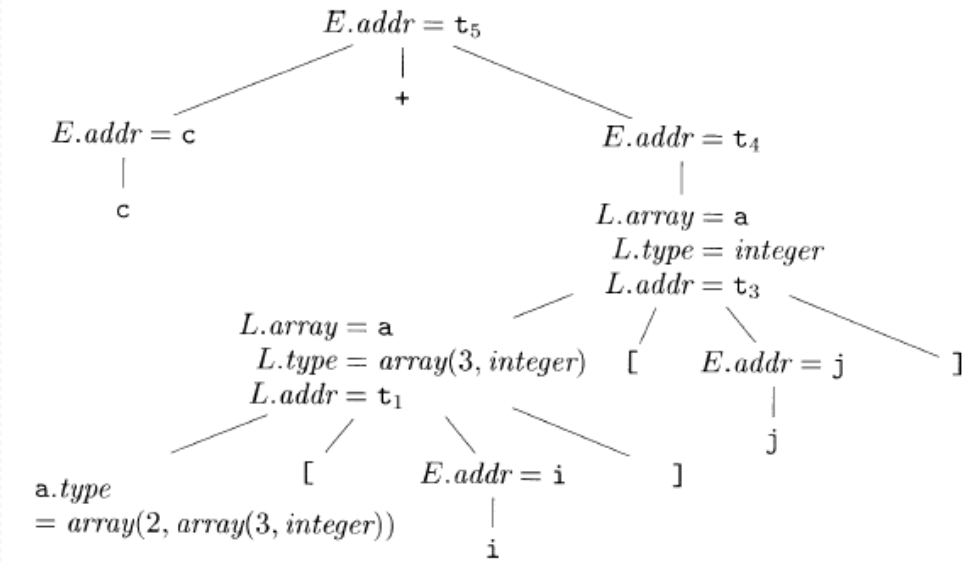


图 6-23　c + a [i] [j] 的注释语法分析树

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

图 6-24　表达式 c + a [i] [j] 的三地址代码

控制流语句翻译

- if-else语句，while语句
- 翻译目标：指令形式：
  - 赋值指令 x=y op z, x=op y, x=y
  - 无条件转移指令 goto L
  - 条件转移指令 if x goto L, if False x goto L
  - 带有关系运算符的转移指令 if x relop y goto L
  - 过程调用和返回指令 param x, call p,n , return y
  - 带下标的复制指令x=y[i], x[i]=y
  - 地址和指针赋值指令 x=&y x=*y *x=y

控制流语句翻译

- if-else语句，while语句
- 需要将语句的翻译和布尔表达式的翻译结合在一起
- 布尔表达式是被用作语句中改变控制流的条件表达式，通常用来
  - 改变控制流。布尔表达式的值由程序到达的某个位置隐含地指出。
  - 计算逻辑值。可以使用带有逻辑运算符的三地址指令进行求值。
- 布尔表达式的使用意图要根据其语法上下文确定
  - 跟在关键字if后面的表达式用来改变控制流
  - 一个赋值语句右部的表达式用来计算一个逻辑值
  - 可以使用两个不同的非终结符号或其它方法来区分这两种使用

布尔表达式

- 将布尔运算符作用在布尔变量或关系表达式上，构成布尔表达式
- 引入新的非终结符号B表示布尔表达式
- 布尔运算符: && 、 || 、 !
- 关系表达式 E1 rel E2
- 关系运算符：<、<=、=、!=、>、>=
- 其中布尔运算符&&和||是左结合的，优先级||最低，其次是&&，！最高
- 表示布尔表达式的文法

$$B \rightarrow B \mid\mid B \mid B \;\&\&\; B \mid ! \; B \mid (B) \mid E \;\textbf{rel}\; E \mid \textbf{true} \mid \textbf{false}$$

布尔表达式的高效求值

- B1 || B2, B1为真，则不用求B2也能断定整个表达式为真
- B1&&B2, B1为假，则整个表达式肯定为假
- 如果某些程序设计语言允许这种高效的求值方式，则编译器可以优化布尔表达式的求值过程，只要已经求值部分足以确定整个表达式的值就可以了。

短路（跳转）代码

- 布尔运算符&&、||、!被翻译成跳转指令。由跳转位置隐含的指出布尔表达式的值。
- if(x<100 || x>200 && x!=y) x=0;

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

图 6-34　跳转代码

控制流语句翻译

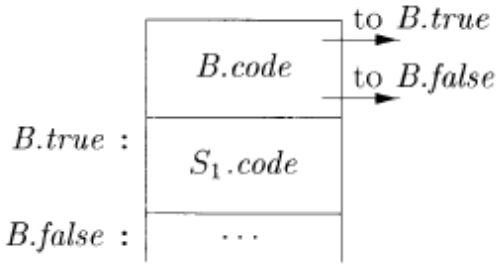
- 语句及文法

$S \rightarrow$ **if** (B) S_1

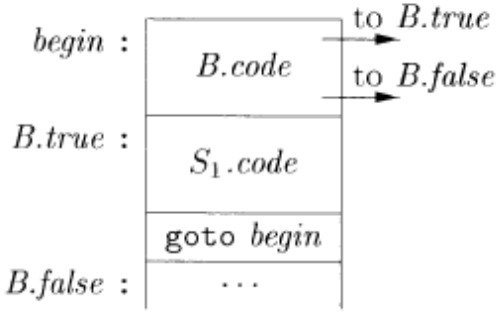
$S \rightarrow$ **if** (B) S_1 **else** S_2

$S \rightarrow$ **while** (B) S_1

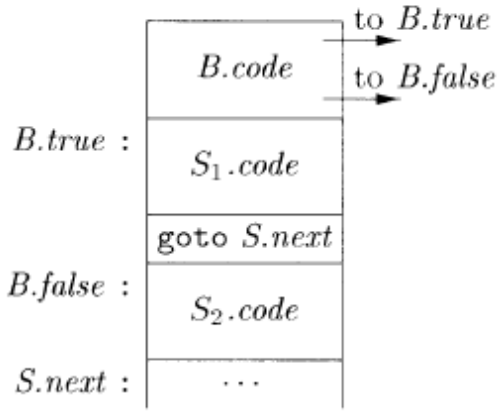
- *B*和*S*有综合属性*code*，表示翻译得到的三地址代码。
- *B*的继承属性*true*和*false*，*S*的继承属性*next*，表示跳转的位置。



a) if



c) while



b) if-else

控制流语句翻译分析

- 翻译S→if (*B*) S1, 创建B.true标号，并指向S1的第一条指令。
- 翻译S →if (*B*) S1 else S2, B为真时，跳转到S1代码的第一条指令；当B为假时跳转到S2代码的第一条指令。然后，控制流从S1或S2转到紧跟在S的代码后面的三地址指令，该指令由继承属性S.next指定。
- while语句中有个begin局部变量
- ……

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	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

布尔表达式的控制流翻译及分析

- 布尔表达式*B*被翻译成三地址指令，生成的条件或无条件转移指令反映*B*的值。
- *B*→*E1 rel E2*，直接翻译成三地址比较指令，跳转到正确位置。
- *B*→*B1* || *B2*, 如果*B1*为真，*B*一定为真，所以*B1.true*和*B.true*相同。如果*B1*为假，那就要对*B2*求值。因此*B1.false*指向*B2*的代码开始的位置。*B2*的真假出口分别等于*B*的真假出口。
- ……

产生式	语义规则
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **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**	$B.code = gen(\texttt{'goto'}\ B.false)$

控制流语句及布尔表达式翻译

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	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

产生式	语义规则
$B \rightarrow B_1$ `||` B_2	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1$ && B_2	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow$! B_1	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr$ **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**	$B.code = gen('\texttt{goto}'\ B.false)$

布尔表达式及控制流语句翻译示例

- 布尔表达式翻译，a<b

  if a<b goto *B.true*

  goto *B.false*

- 控制流语句翻译 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:
```

避免冗余的goto指令

- 在上面的例子中goto L3是冗余的
- X>200翻译成

```
        if x > 200 goto L4
        goto L1
L4:     ...
```

- 可以替换成

```
        ifFalse x > 200 goto L1
L4:     ...
```

- 减少了一条goto指令
- 引入一个特殊标号“fall”(穿越，fall through)，表示不要生成任何跳转指令。

- S→if (*B*) S1的新语义规则

$$B.true = fall$$
$$B.false = S_1.next = S.next$$
$$S.code = B.code \,||\, S_1.code$$

- 对于if-else和while语句的规则也将B.true设为fall

利用“穿越”修改布尔表达式的语义规则

$$
\begin{aligned}
test &= E_1.addr\ \mathbf{rel}.op\ E_2.addr \\
s &= \mathbf{if}\ B.true \neq fall\ \mathbf{and}\ B.false \neq fall\ \mathbf{then} \\
&\quad\quad gen(\text{'if'}\ test\ \text{'goto'}\ B.true) \parallel gen(\text{'goto'}\ B.false) \\
&\quad \mathbf{else\ if}\ B.true \neq fall\ \mathbf{then}\ gen(\text{'if'}\ test\ \text{'goto'}\ B.true) \\
&\quad \mathbf{else\ if}\ B.false \neq fall\ \mathbf{then}\ gen(\text{'ifFalse'}\ test\ \text{'goto'}\ B.false) \\
&\quad \mathbf{else}\ '' \\
B.code &= E_1.code \parallel E_2.code \parallel s
\end{aligned}
$$

图 6-39　$B \rightarrow E_1$ **rel** E_2 的语义规则

$$
\begin{aligned}
B_1.true &= \mathbf{if}\ B.true \neq fall\ \mathbf{then}\ B.true\ \mathbf{else}\ newlabel() \\
B_1.false &= fall \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= \mathbf{if}\ B.true \neq fall\ \mathbf{then}\ B_1.code \parallel B_2.code \\
&\quad\quad \mathbf{else}\ B_1.code \parallel B_2.code \parallel label(B_1.true)
\end{aligned}
$$

图 6-40　$B \rightarrow B_1 \,||\, B_2$ 的语义规则

注意B.true=fall时，还得为B1.true new一个label

B →B1&&B2带“穿越”的语义规则

{B_1.*false*=if (*B*.false=*fall*) *newlabel*() *else B.false*
B_1.*true* = *fall*
B_2.*true* = *B.true*
B_2.*false* = *B.false*
B.code = if (B.false=fall) then B_1.*code*||B_2.*code*|| *label* (B_1.false)
else B_1.*code* || B_2.*code* }

使用标号fall的控制流语句翻译示例

$$S.next = newlabel()$$
$$P.code = S.code \;||\; label(S.next)$$

- if (x<100 ||x>200 && x!=y) x=0;

$$B.true = fall$$
$$B.false = S_1.next = S.next$$
$$S.code = B.code \;||\; S_1.code$$

$$B_1.true = \textbf{if } B.true \neq fall \textbf{ then } B.true \textbf{ else } newlabel()$$
$$B_1.false = fall$$
$$B_2.true = B.true$$
$$B_2.false = B.false$$
$$B.code = \textbf{if } B.true \neq fall \textbf{ then } B_1.code \;||\; B_2.code \textbf{ else } B_1.code \;||\; B_2.code \;||\; label(B_1.true)$$

图 6-40　$B \rightarrow B_1 \,||\, B_2$ 的语义规则

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

图 6-41　使用控制流穿越技术翻译的 if 语句

$$test = E_1.addr \;\textbf{rel}.op\; E_2.addr$$

$$s = \textbf{if } B.true \neq fall \textbf{ and } B.false \neq fall \textbf{ then } gen('\texttt{if}'\ test\ '\texttt{goto}'\ B.true) \;||\; gen('\texttt{goto}'\ B.false)$$
$$\textbf{else if } B.true \neq fall \textbf{ then } gen('\texttt{if}'\ test\ '\texttt{goto}'\ B.true)$$
$$\textbf{else if } B.false \neq fall \textbf{ then } gen('\texttt{ifFalse}'\ test\ '\texttt{goto}'\ B.false)$$
$$\textbf{else } ''$$

$$B.code = E_1.code \;||\; E_2.code \;||\; s$$

图 6-39　$B \rightarrow E_1$ **rel** E_2 的语义规则

回填

- 布尔表达式和控制流语句生成目标代码时，一个重要问题是跳转指令需要准确给出跳转到的目标
- If (B) S, 在对B翻译生成的代码中，当B为假时应该跳转到紧跟在S的代码之后的指令处。在前面的做法中，是将S.next作为继承属性传递给B.false，指明要跳转到的位置。这样做，需要两趟处理。
- 一趟处理的技术：回填
  - 生成跳转指令时暂时不指定该跳转指令的目标。等到能够确定正确的目标标号时再去填充这些指令的目标位置。
  - 需要回填的指令将被放在一个列表中。

回填（续）

- 为非终结符号*B*引入两个综合属性*truelist*和*falselist*
  - 生成*B*的代码时，跳转指令是不完整的，跳转到的真假入口位置尚未填写，这些不完整的跳转指令将被存放在*truelist*或*falselist*的列表中
  - 回填时，在*truelist*存放的指令中插入*B*为真时控制流应该转向的标号
  - 同样，在*falselist*存放的指令中插入*B*为假时控制流应该转向的标号
- 类似地，语句*S*引入一个综合属性*nextlist*，用来存放跳转指令列表，这些指令应该跳转到紧跟在*S*代码之后。

回填（续）

- 用于处理跳转指令列表的三个函数
  - *Makelist*(*i*)，创建一个包含i的列表，i是三地址码的标号，返回一个指向新创建的列表的指针
  - *Merge*(p_1,p_2),将p_1和p_2指向的列表进行合并，返回新的列表
  - *Backpatch*(*p*,*i*), 将i作为目标标号插入到列表p中的所有指令中。

$B \rightarrow B_1 \ || \ M \ B_2 \mid B_1 \ \&\& \ M \ B_2 \mid ! \ B_1 \mid (\ B_1 \ ) \mid E_1 \ \textbf{rel} \ E_2 \mid \textbf{true} \mid \textbf{false}$

$M \rightarrow \epsilon$

布尔表达式的回填

- 文法中引入标记非终结符号*M*。它的作用是在适当的时候获取将要生成的下一条指令的标号。
- 1)中
  - *B1.truelist*和*B2.truelist*合并成*B.truelist*
  - B_1为假时，跳转目标是B_2的代码的起始位置。回填*backpatch*(B_1.*falselist*,*M.instr*)
- 8)中
  - 记录下一条指令的位置，即B2.code开始的位置。

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

布尔表达式回填示例

- x<100 || x>200 && x!=y
- 注：所有的动作都在产生式右部的最后，因此可以在自底向上分析的规约是执行上述动作。
- 语法树.....
- 注释语法树.....

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	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}\ \_')$; $gen('\texttt{goto}\ \_')$; }
6)	$B \rightarrow$ **true**	{ $B.truelist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
7)	$B \rightarrow$ **false**	{ $B.falselist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
8)	$M \rightarrow \epsilon$	{ $M.instr = nextinstr$; }

控制转移语句的回填

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

$L \rightarrow L\,S \mid S$

- 除了3）和7），其它地方都没有产生新的代码，语句所有的代码都由赋值语句和表达式相关的语义代码产生。根据控制流进行回填，将赋值语句和表达式的求值过程关联起来。

记录需要回填的指令位置

记录转移的目标位置

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

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

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

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

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

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

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

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

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