第六章 中间代码生成

许畅

南京大学计算机系

2014年春季

本章内容

- 中间代码表示
  - 表达式的有向无环图
  - 三地址代码：x = y op z
- 类型检查
  - 类型、表达式的翻译、类型检查
- 中间代码生成
  - 控制流、回填

编译器前端的逻辑结构

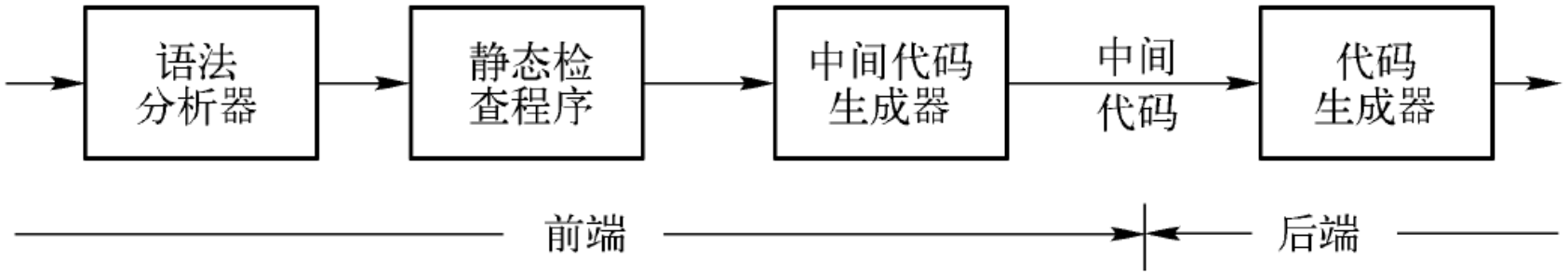


图 6-1　一个编译器前端的逻辑结构

- 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

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**
  - **三地址代码**
- 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

中间代码的实现

- 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

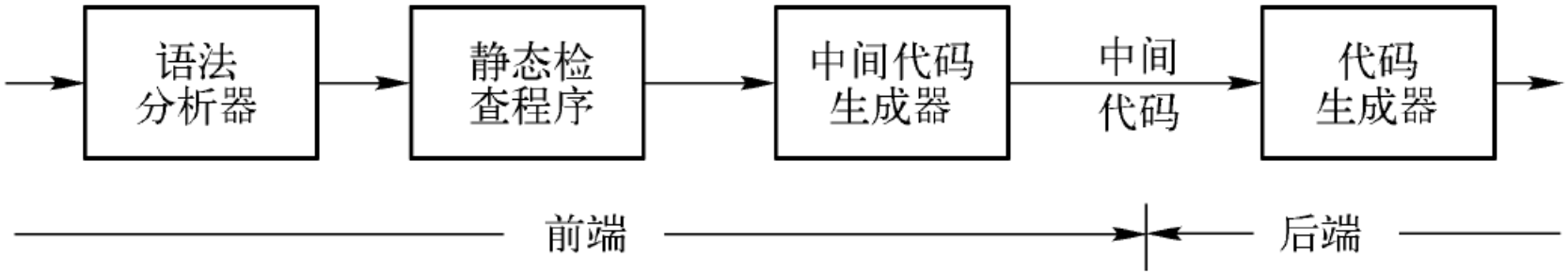


图 6-1　一个编译器前端的逻辑结构

生成抽象语法树的语法制导定义

- a + a * (b – c) + (b – c) * d的抽象语法树

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

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

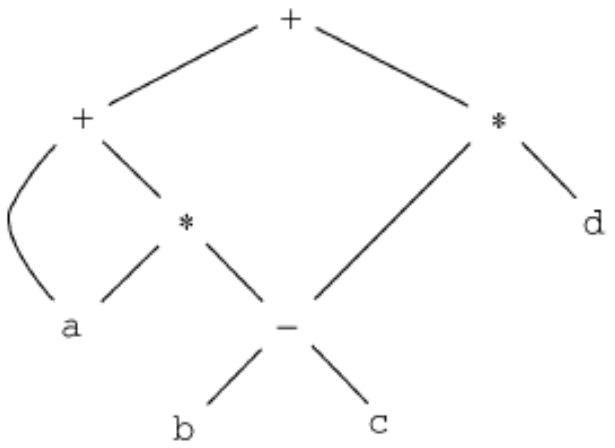


图 6-3　表达式 a + a * (b - c) + (b - c) * d 的 DAG

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

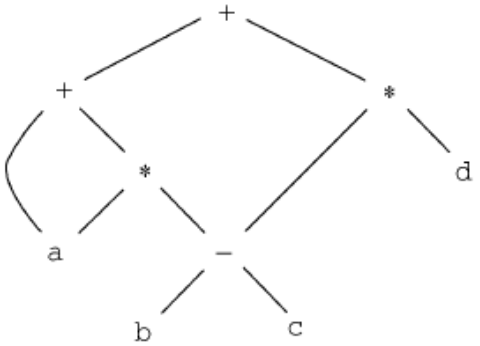


图 6-3　表达式 a + a * (b - c) + (b - c) * d 的 DAG

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 (1)

- 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
  - 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 (2)

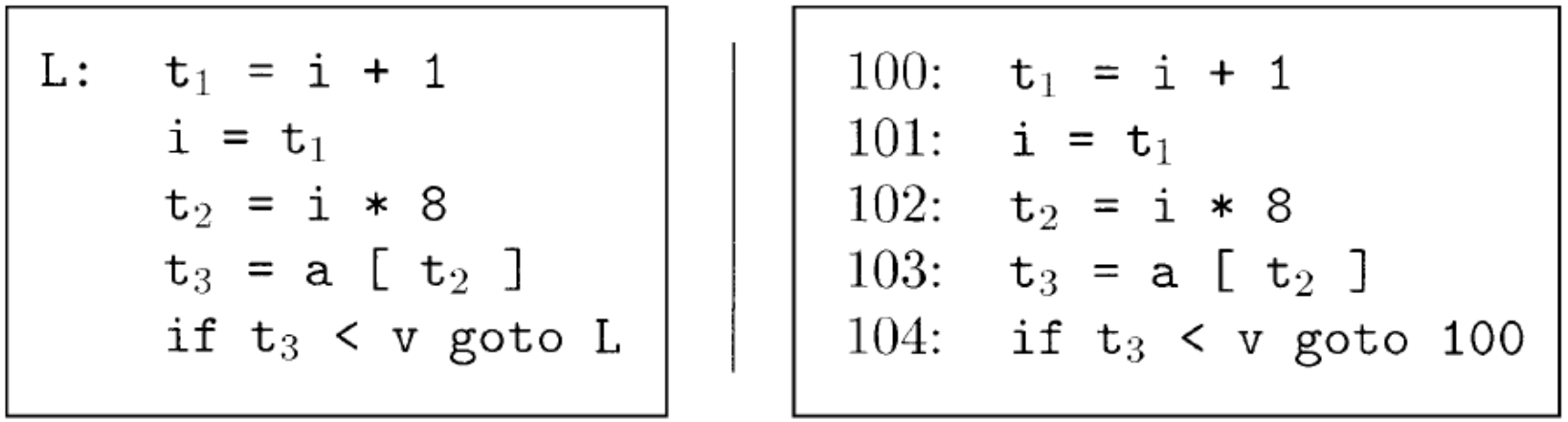
- 指令集合 (1)
  - 运算/赋值指令：x = y op z　　x = op y
  - 复制指令：x = y
  - 无条件转移指令：goto L
  - 条件转移指令：if x goto L　　if False x goto L
  - 条件转移指令：if x relop y goto L

三地址代码 (3)

- 指令集合 (2)
  - 过程调用/返回
    - 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

例子

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



a) 符号标号　　b) 位置号

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/静态单赋值**来表示三地址指令
- 四元式：可以实现为记录 (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

四元式的例子

```
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

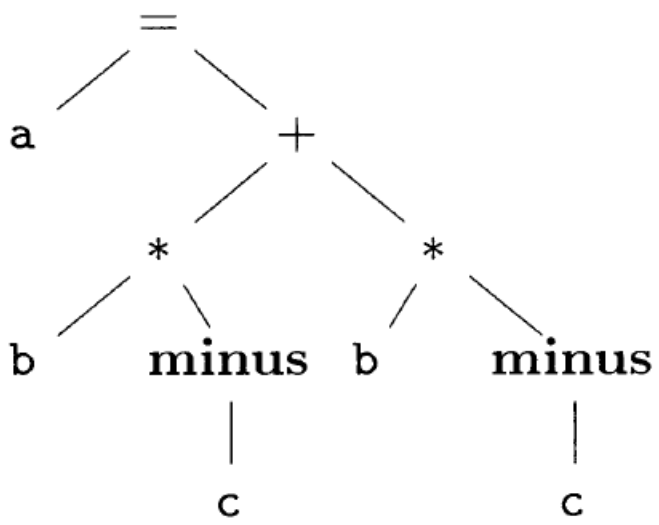
图 6-10　三地址代码及其四元式表示

- 赋值语句：a = b * – c + b * – c

三元式表示

- 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 = y op z需要拆分为 (这里?是编号)
  - ?　　op　　y　　z
  - 　　=　　x　　(?)
- x[i] = y需要拆分为两个三元式
  - 求x[i]的地址，然后再赋值
- 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

三元式的例子



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

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

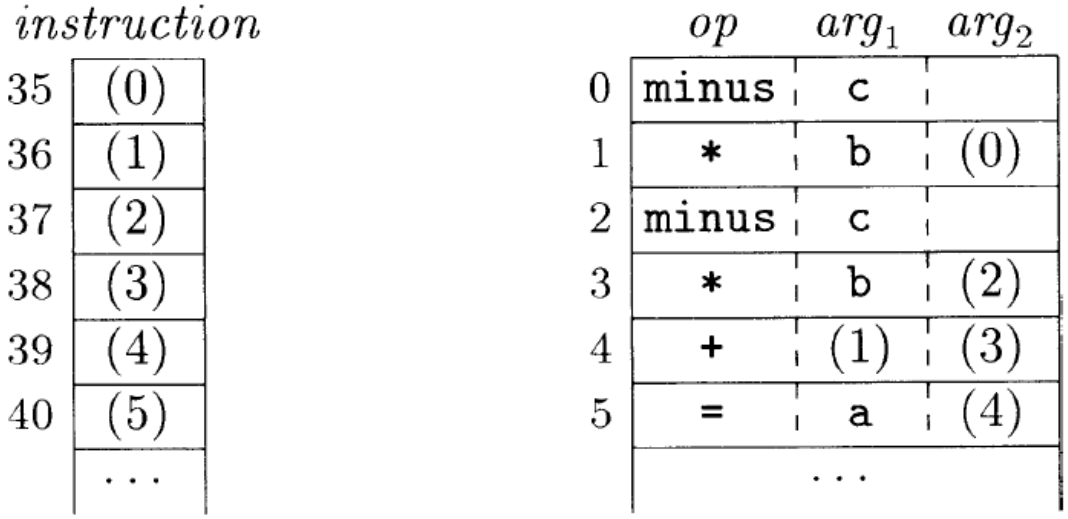


图 6-12　三地址代码的间接三元式表示

静态单赋值 (SSA)

- Static Single Assignment (SSA) 中的所有赋值都是针对不同名的变量
- 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;
  - if (flag) x_1 = –1; else x_2 = 1;　$x_3 = \varphi(x_1, x_2)$;　y = x_3 * a;

类型和声明

- 类型检查 (Type checking)
  - 利用一组规则来检查**运算分量**的类型和**运算符**的预期类型是否匹配
- 类型信息的用途
  - 查错、确定名字需要的内存空间、计算数组元素的地址、类型转换、选择正确的运算符
- 本节的内容
  - 确定名字的类型
  - 变量的存储空间布局 (相对地址)

类型表达式

- 类型表达式 (Type expression)：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”作用于类型表达式得到
- 如int [2][3]，表示由两层数组组成的数组
  - array(2, array(3, integer))
  - array是**类型构造算子**，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
  - 如：struct { int *a*[10]; float *f*; } *st*;
  - 对应于：record((*a* × array(0..9, int)) × (*f* × real))
- 类型表达式中可以包含取值为类型表达式的变量

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
  - 或者由相同的构造算子作用于结构等价的类型而得到
  - 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 (仅有前两个条件)

类型的声明

- 处理基本类型、数组类型或记录类型的文法
  - $D \rightarrow T$ id; $D \mid \varepsilon$
  - $T \rightarrow B\ C \mid$ record '{' D '}'
  - $B \rightarrow$ int | float
  - $C \rightarrow \varepsilon \mid$ [num] C
- 应用该文法及其对应的语法制导定义，除了得到**类型表达式**之外，还得进行各种类型的**存储布局**

局部变量的存储布局

- 变量的类型可以确定变量需要的内存
  - 即类型的**宽度** (该类型一个对象所需的存储单元的数量)
  - 可变大小的数据结构只需要考虑指针
- 函数的局部变量总是分配在连续的区间
  - 因此给每个变量分配一个相对于这个区间开始处的**相对地址**
- 变量的类型信息保存在**符号表**中

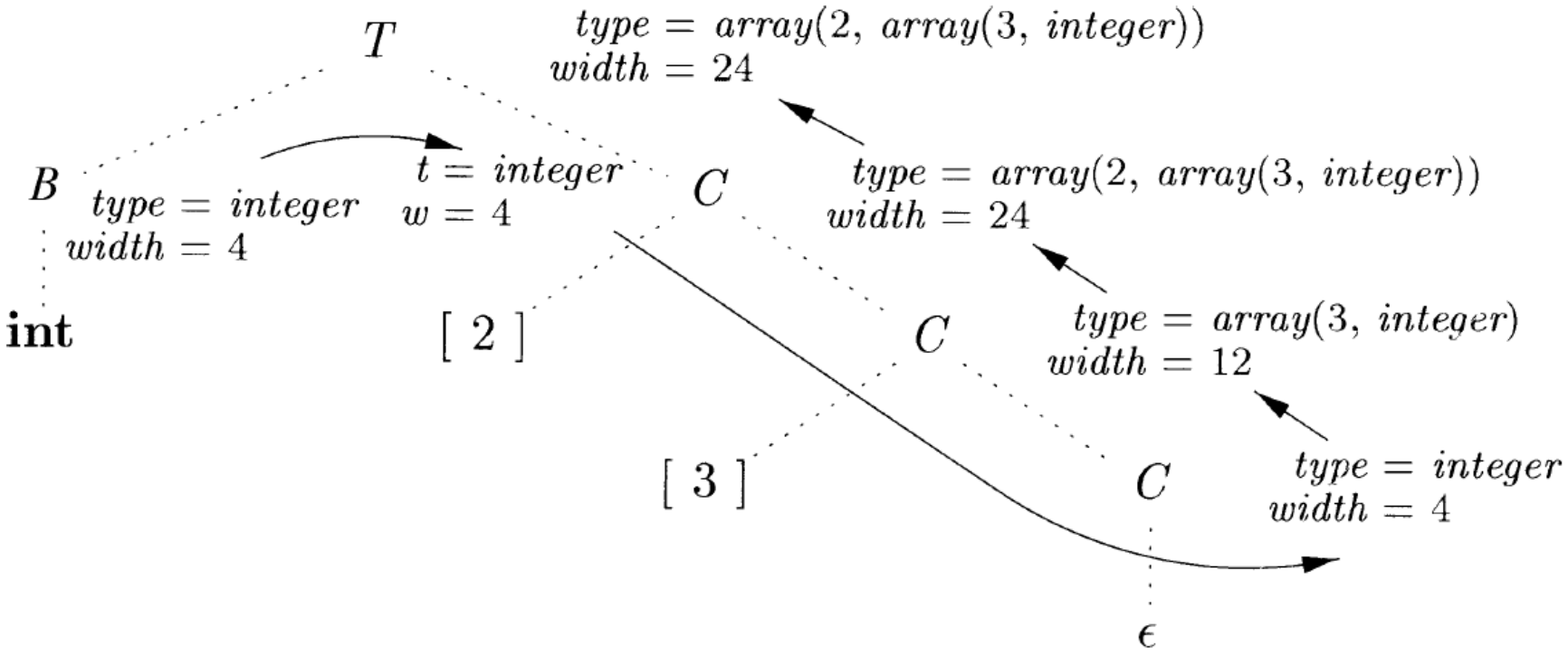
计算*T*的类型和宽度的SDT

- 综合属性：*type*, *width*
- 全局变量*t*和*w*用于将**类型**和**宽度**信息从*B*传递到*C* → ε
  - 相当于*C*的继承属性 (也可以把*t*和*w*替换为*C.t*和*C.w*)

$T \rightarrow B$		$\{ t = B.type;\ w = B.width; \}$
	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	$\{ C.type = array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width; \}$

SDT运行的例子

- 输入：int [2][3]



声明序列的SDT (1)

- 在处理一个过程/函数时，局部变量应该放到**单独的符号表**中去
- 这些变量的内存布局独立
  - 相对地址从0开始
  - 假设变量的放置和声明的**顺序相同**
- SDT的处理方法
  - 变量*offset*记录当前**可用的相对地址**
  - 每“分配”一个变量，*offset*的值增加相应的值 (加宽度)
- *top.put*(*id.lexeme*, *T.type*, *offset*)
  - 在当前符号表 (位于栈顶) 中创建符号表条目，记录标识符的**类型**和**偏移量**

声明序列的SDT (2)

- 我们可以把*offset*看作*D*的继承属性
  - *D.offset*表示*D*中第一个变量的相对地址
  - $P \rightarrow \{\ D.offset = 0;\ \}\ D$
  - $D \rightarrow T\ \text{id};\ \{\ D_1.offset = D.offset + T.width;\ \}\ D_1$

$$
\begin{array}{lll}
P & \rightarrow & \{\ offset = 0;\ \} \\
 & D & \\
D & \rightarrow & T\ \mathbf{id}\ ;\quad \{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset); \\
 & & \qquad offset\ =\ offset + T.width;\ \} \\
 & D_1 & \\
D & \rightarrow & \epsilon
\end{array}
$$

记录和类中的字段 (1)

- 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**
- 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 (2)

$$
\begin{array}{lll}
T \rightarrow \textbf{record}\ '\{' & \{\ Env.push(top);\ top = \textbf{new}\ Env();\\
 & \quad Stack.push(offset);\ offset = 0;\ \} \\
\quad D\ '\}' & \{\ T.type = record(top);\ T.width = offset;\\
 & \quad top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$

注：记录类型存储方式可以推广到类

表达式代码的SDD

- 将表达式翻译成三地址指令序列
- 表达式的SDD
  - 属性*code*表示代码
  - *addr*表示存放表达式**结果的地址** (临时变量)
  - new *Temp*()可以生成一个临时变量
  - *gen*(…)生成一个指令

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

$$a = b + -c$$

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

增量式翻译方案

- 类似于上一章中所述的**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*); }

数组元素的寻址

- 假设数组元素被存放在**连续**的存储空间中
- 元素从0到$n-1$编号，第i个元素的地址为

  $$base + i * w$$

- K维数组的寻址：假设数组**按行存放**，即首先存放$A[0][i_2]\ldots[i_k]$，然后存放$A[1][i_2]\ldots[i_k]$，…
- $A[i_1][i_2]\ldots[i_k]$的地址

  $$base + i_1 * w_1 + i_2 * w_2 + \ldots + i_k * w_k$$

- 或者

  $$base + ((\ldots((i_1 * n_2 + i_2) * n_3 + i_3)\ldots) * n_k + i_k) * w$$

- 其中：$base, w, i, n$的值可以从符号表中找到

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 (1)

- 非终结符号*L*的三个综合属性
  - *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.addr*指示一个临时变量，计算数组引用的**偏移量**
  - *L.type*是*L*生成的**子数组的类型**
    - 对于任何 (子) 数组类型*L.type*，其宽度由*L.type.width*给出，*L.type.elem*给出其数组**元素的类型**

数组引用生成代码的翻译方案 (2)

- 核心是确定数组引用的**地址**

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

数组引用生成代码的翻译方案 (3)

- *L*的代码只计算了**偏移量**
- 数组元素的存放地址应该根据偏移量进一步计算，即*L*的**数组基址**加上**偏移量**
- 使用三地址指令x = a[i]

$$
\begin{array}{lll}
E \rightarrow E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp\,(); & \\
 & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} & \\
\mid\ \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \} & \\
\mid\ L & \{\ E.addr = \textbf{new}\ Temp\,(); & \\
 & \quad gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \} &
\end{array}
$$

数组引用生成代码的翻译方案 (4)

- 使用三地址指令a[i] = x

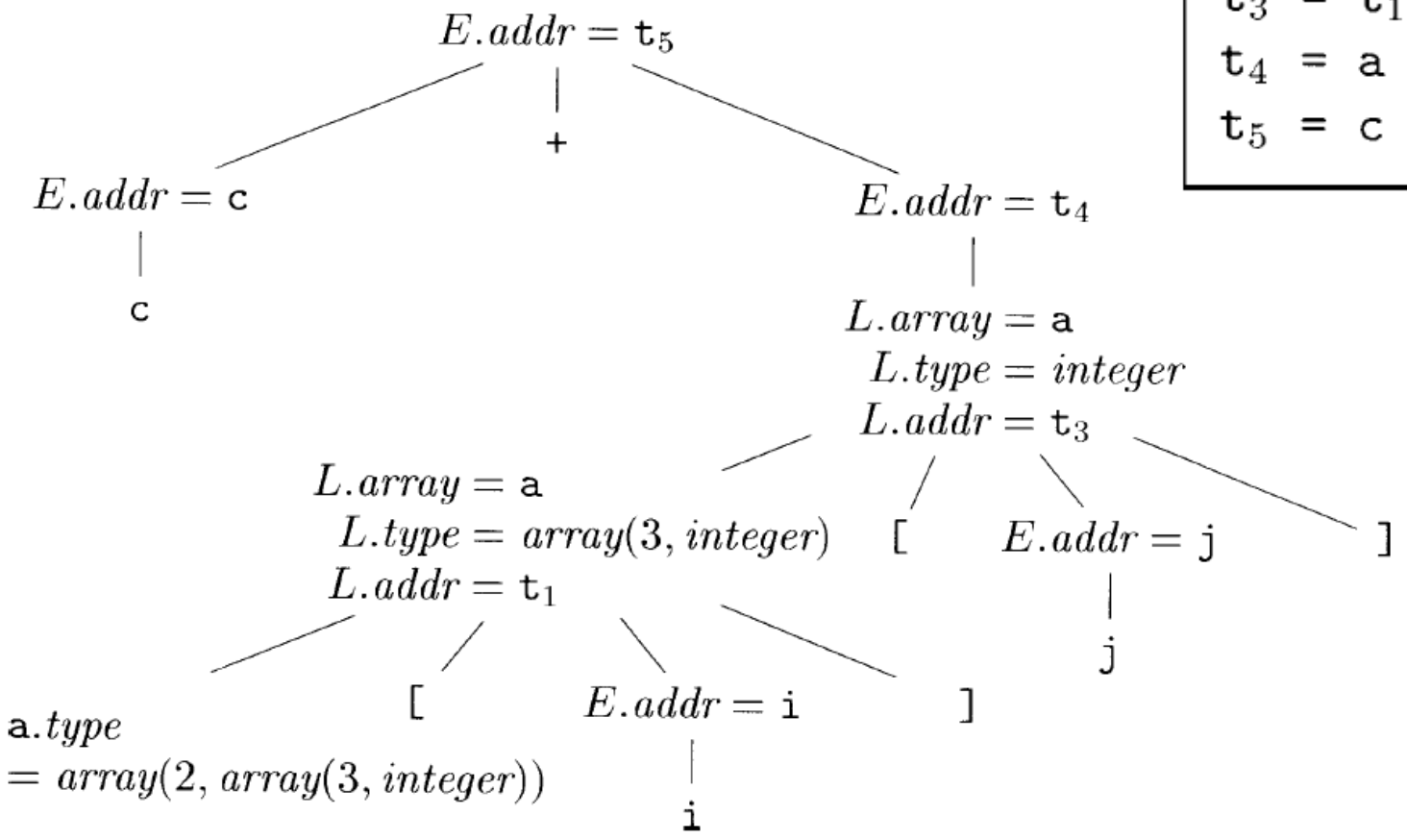
$$S \rightarrow \mathbf{id} = E\ ; \quad \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \}$$

$$\mid\ L = E\ ; \quad \{\ gen(L.array\ .base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \}$$

例子

- 表达式：c + a[i][j]

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



类型检查和转换

- 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、…
- 类型检查可以分为**动态**和**静态**两种
- 强类型的：如果编译器中的类型系统能够保证它接受的程序在运行时刻不会发生类型错误，则该语言的这种实现称为**强类型的**

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$s \to t$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和β，f的类型为$\alpha \to \beta$且x的类型为α

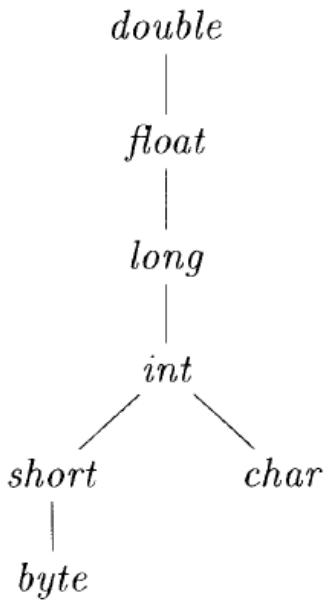
类型转换

- 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
  - $t_1 = (\text{float})\ i$
  - $t_2 = x\ \text{fmul}\ t_1$
- 类型转换比较简单时的SDD
  - $E \rightarrow E_1 + E_2$

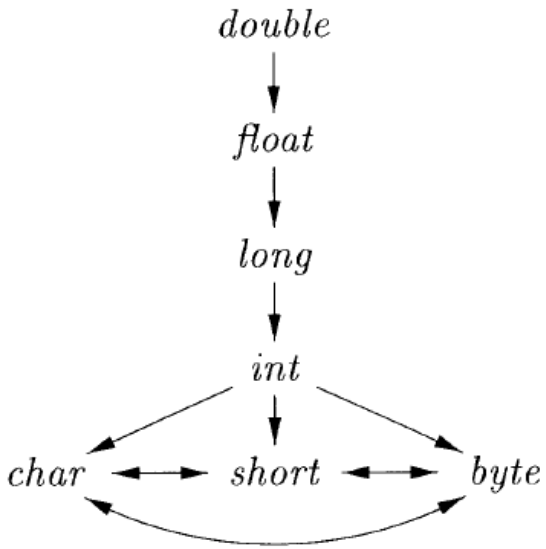
    { if (E_1.*type* = integer and E_2.*type* = integer) E.*type* = integer;
    else if (E_1.*type* = float and E_2.*type* = integer) E.*type* = float;
    … }

类型的拓宽widening和窄化narrowing

- Java的类型转换规则
- 编译器自动完成的转换为**隐式转换**，程序员用代码指定的转换为**显式转换**



a) 拓宽类型转换



b) 窄化类型转换

处理类型转换的SDT

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

```
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*max*求两个参数在拓宽层次结构中的**最小公共祖先**
- 函数*widen*生成必要的类型转换代码

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 $f.typeset = \{ s_i \rightarrow t_i \mid 1 <= i <= n \}$ and $E_1.type = s_k$
  then $E.type = t_k \}$

控制流的翻译

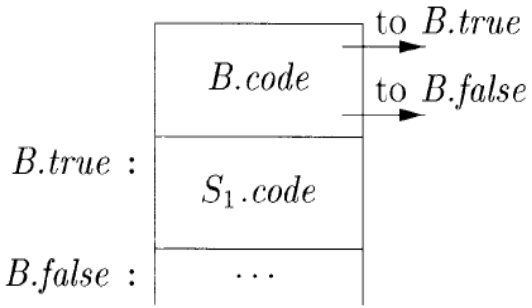
- 布尔表达式可以用于改变控制流/计算逻辑值
- 文法
  - $B \rightarrow B \,\|\, B \mid B \,\&\&\, B \mid !\, B \mid (\, B \,) \mid E$ **rel** $E \mid$ **true** $\mid$ **false**
- 语义
  - $B_1 \,\|\, B_2$中B_1为真时，不计算B_2，整个表达式为真；因此，当B_1为真时应该跳过B_2的代码
  - $B_1 \,\&\&\, B_2$中B_1为假时，不计算B_2，整个表达式为假
- 短路代码
  - 通过**跳转指令**实现控制流的处理
  - 逻辑运算符本身不在代码中出现

短路代码的例子

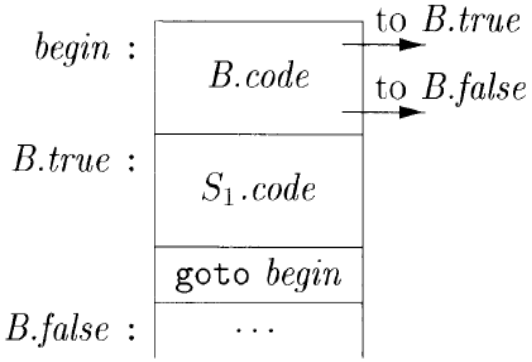
- 语句
  - if (x < 100 || x > 200 && x != y) x = 0;
- 代码
  - if x < 100 goto L_2
  - if False x > 200 goto L_1
  - if False x != y goto L_1
  - L_2: x = 0
  - L_1: 接下来的代码

控制流语句的翻译

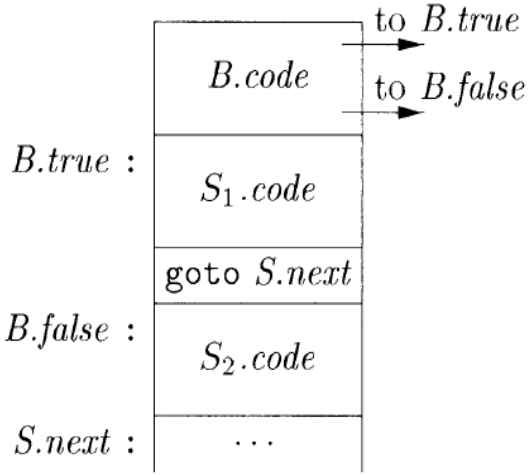
- *B*表示布尔表达式，*S*代表语句
  - $S \rightarrow$ **if** $(B)\ S_1$
  - $S \rightarrow$ **if** $(B)\ S_1$ **else** S_2
  - $S \rightarrow$ **while** $(B)\ S_1$
- 代码布局见右图
- 继承属性
  - *B.true*：*B*为真时的跳转目标
  - *B.false*：*B*为假时的跳转目标
  - *S.next*：*S*执行完毕时的跳转目标



a) if



c) while



b) if-else

语法制导的定义 (1)

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

语法制导的定义 (2)

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

- 增量式生成代码

S → **while** (

{ *begin* = *newlabel*(); *B.true* = *newlabel*(); *B.false* = *S.next*; *gen*(*begin* ':'); }

B)

{ S_1.*next* = *begin*; *gen*(*B.true* ':'); } S_1 { *gen*('goto' *begin*); }

布尔表达式的控制流翻译

- 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*
- *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*B.false*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

布尔表达式的代码的SDD (1)

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 $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$

布尔表达式的代码的SDD (2)

$B \rightarrow E_1$ **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**	$B.code = gen('\texttt{goto}'\ B.false)$

布尔表达式代码的例子

- if ($x < 100 \,||\, x > 200 \,\&\&\, x \,!= y$) $x = 0$; 的代码

```
        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:
```

原始代码

```
        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
```

优化过的代码

布尔值和跳转代码

- 程序中出现布尔表达式的目的可能就是求出它的值，比如$x = a < b$
- 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
  - $S \rightarrow \text{id} = E; \mid \text{if } (E)\ S \mid \text{while } (E)\ S \mid S\ S$
  - $E \rightarrow E \parallel E \mid E \,\&\&\, E \mid !\, E \mid E \text{ rel } E \mid \ldots$
- 根据E的语法树结点所在的位置
  - $S \rightarrow \text{while } (\ E\ )\ S_1$中的$E$，生成**跳转代码**
  - 对于$S \rightarrow \text{id} = E$，生成**计算右值的代码**

回填 (1)

- 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*的标号，但是不生成跳转目标
  - 这些标号被记录到*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，而是使用列表记录这些不完整的指令
  - 等知道正确的跳转目标时再填写目标
  - 列表中的每个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控制流时，它在取值true和false时分别跳转到某个目标
- 综合属性
  - *truelist*：包含跳转指令标号的列表，这些指令在取值true时执行
  - *falselist*：包含跳转指令标号的列表，这些指令在取值false时执行
- 辅助函数
  - *makelist*(*i*)：创建一个包含跳转指令标号*i*的列表
  - *merge*(p_1, p_2)：将p_1和p_2指向的标号列表合并然后返回
  - *backpatch*(*p*, *i*)：将*i*作为跳转目标插入到*p*的所有指令中

布尔表达式的回填翻译 (2)

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)	$B \rightarrow B_1$ `||` 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	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}'\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ '\texttt{goto}\ \_')$; $gen('\texttt{goto}\ \_')$; }
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 (1)

$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$

1) $B \rightarrow B_1 \mid\mid M\ B_2$ { $backpatch(B_1.falselist, M.instr)$;
$B.truelist = merge(B_1.truelist, B_2.truelist)$;
$B.falselist = B_2.falselist$; }

- *true/false*属性的赋值，在回填方案中对应为相应的*truelist/falselist*的赋值或者*merge*

回填和非回填方法的比较 (2)

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

5) $B \rightarrow E_1 \ \mathbf{rel} \ E_2$

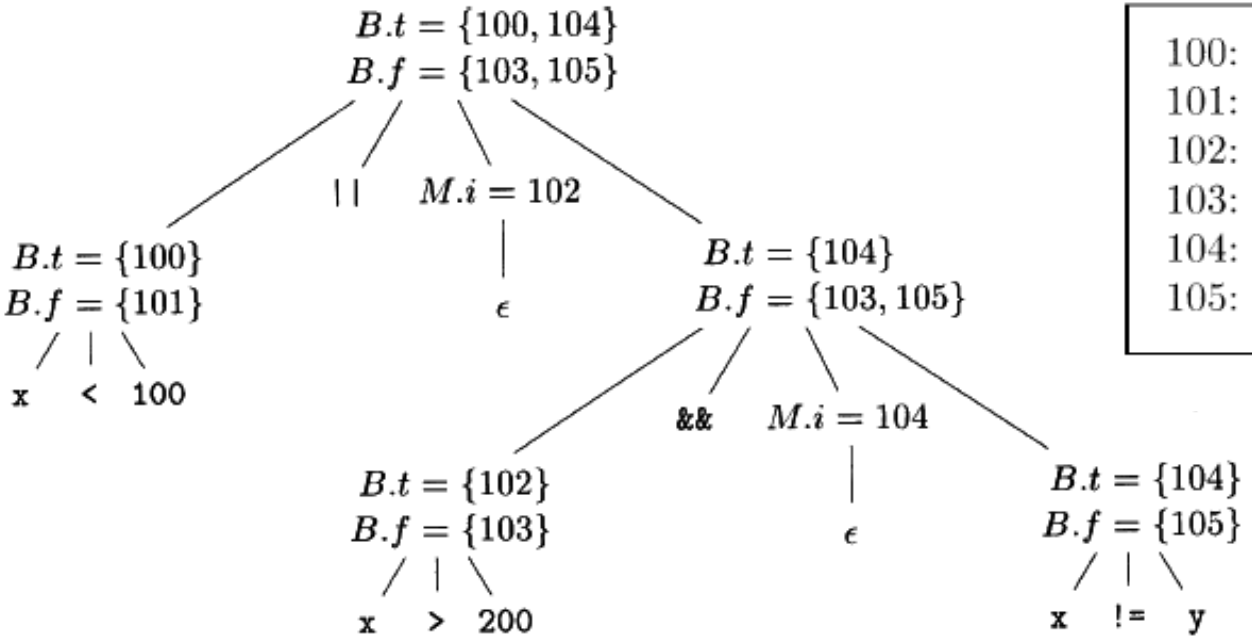
{ $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$gen('\texttt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\texttt{goto}\ \_');$
$gen('\texttt{goto}\ \_');$ }

- 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
- 原来跳转到*B.false*的指令，现在被加入到*B.falselist*中

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

控制转移语句的回填 (1)

- $S \rightarrow$ if (B) S | if (B) S else S |
  while (B) S | { L } | 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标号
  - 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 (2)

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

- *M*的作用就是用*M.instr*记录下一个指令的标号
  - 规则1中记录了then分支的代码起始标号
  - 规则2中，分别记录了then分支和else分支的起始标号
- *N*的作用是生成goto指令坯，*N.nextlist*只包含该指令的标号

控制转移语句的回填 (3)

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

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)
  - 跟踪外围循环语句*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