第六章 中间代码生成

陈 林

本章内容

- 中间代码表示
  - 抽象语法树
  - 三地址代码
- 中间代码生成
  - 表达式
  - 类型检查
  - 控制流

编译器前端的逻辑结构

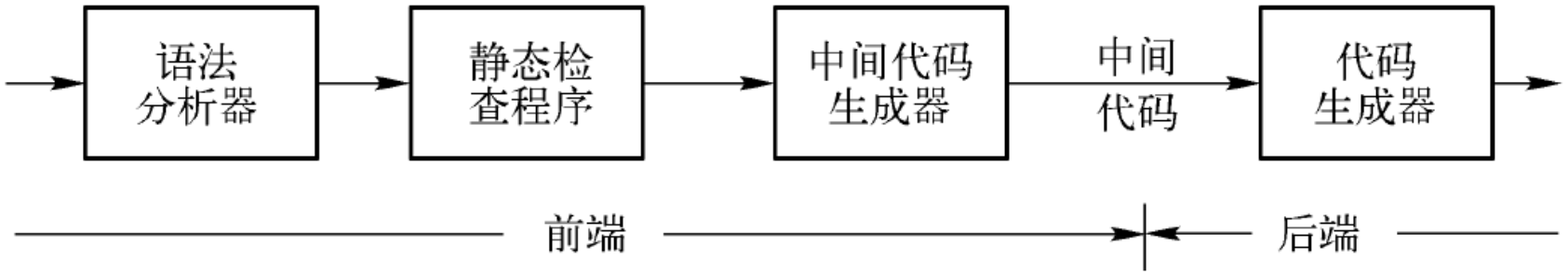


图 6-1　一个编译器前端的逻辑结构

- 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

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

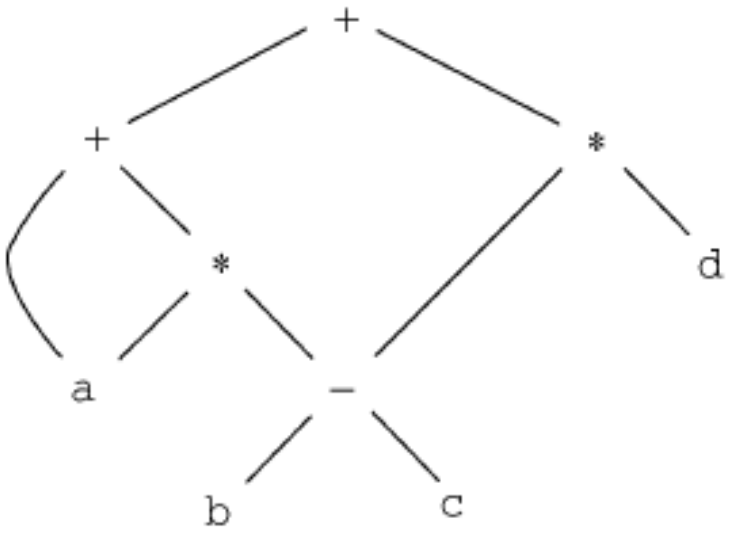


图 6-3　表达式 a + a * (b - c) + (b - c) * d 的 DAG

DAG构造

- 可以用和构造抽象语法树一样的SDD来构造

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

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

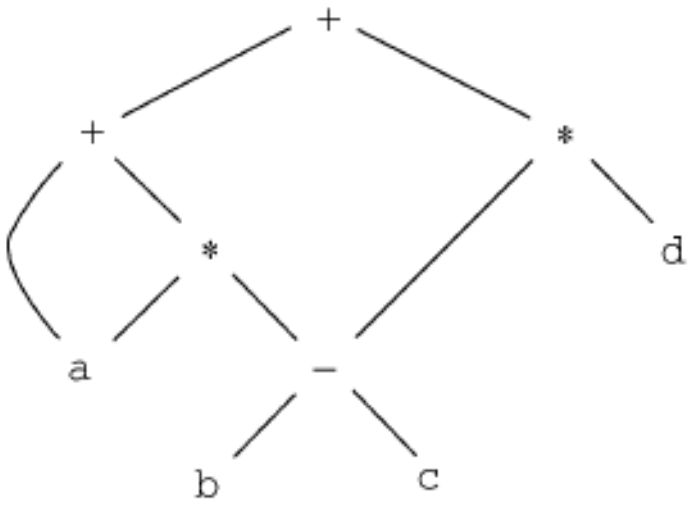


图 6-3　表达式 a + a * (b - c) + (b - c) * d 的 DAG

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

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

三地址代码实例

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

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

a) 符号标号

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

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

四元式的例子

- 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

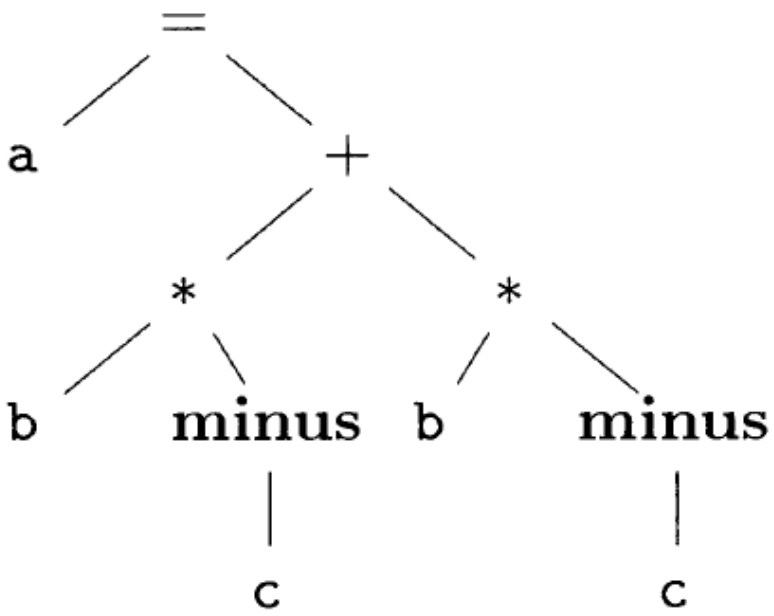
图 6-10　三地址代码及其四元式表示

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[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 ?

- 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

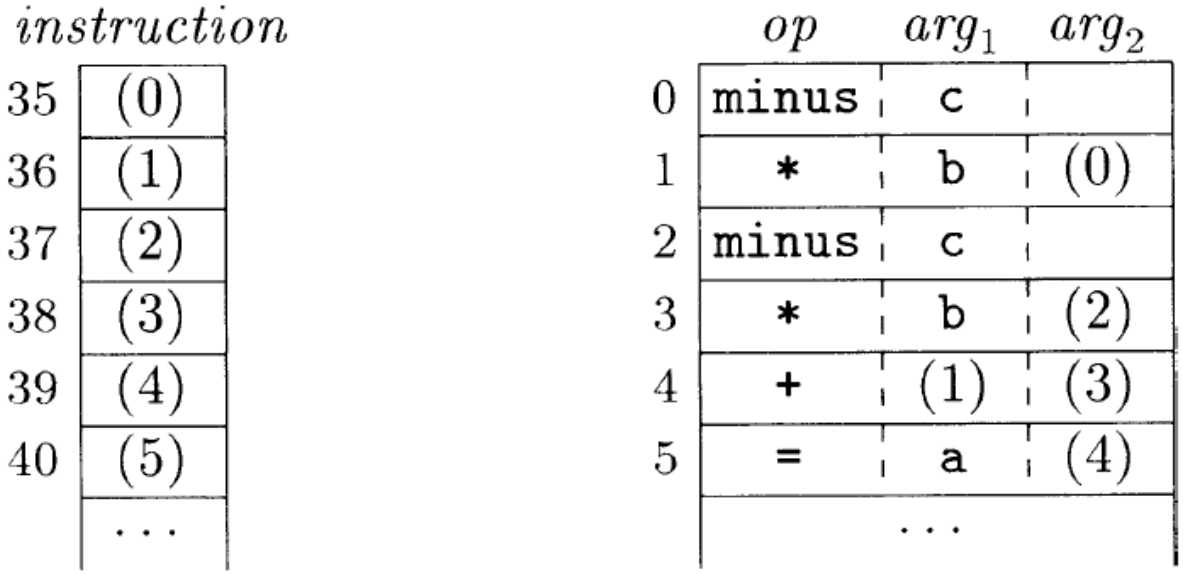


图 6-12　三地址代码的间接三元式表示

静态单赋值（SSA）

- 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;
    $x_3=\Phi(x_1, x_2)$;
    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
- 主要内容
  - 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
  - 基本类型
  - 类名
  - 类型构造算子作用于类型
    - 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
  - 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

声明

- 文法

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

- 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中包含了各个字段的声明，字段声明和变量声明的文法一致

局部变量名的存储布局

- 变量的类型可以确定变量需要的内存
  - 即类型的宽度
  - 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

计算类型和宽度的SDT

$$
\begin{aligned}
T &\rightarrow B \\
&\quad\ C \\
B &\rightarrow \mathbf{int} \\
B &\rightarrow \mathbf{float} \\
C &\rightarrow \epsilon \\
C &\rightarrow [\ \mathbf{num}\ ]\ C_1
\end{aligned}
$$

计算类型和宽度的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，因为总是通过拷贝来传递，所以在SDT中只赋值一次，也可以把t和w替换为C.t和C.w

$$
\begin{aligned}
T &\rightarrow B \\
&\quad\ C \\
B &\rightarrow \textbf{int} \\
B &\rightarrow \textbf{float} \\
C &\rightarrow \epsilon \\
C &\rightarrow [\ \textbf{num}\ ]\ C_1
\end{aligned}
$$

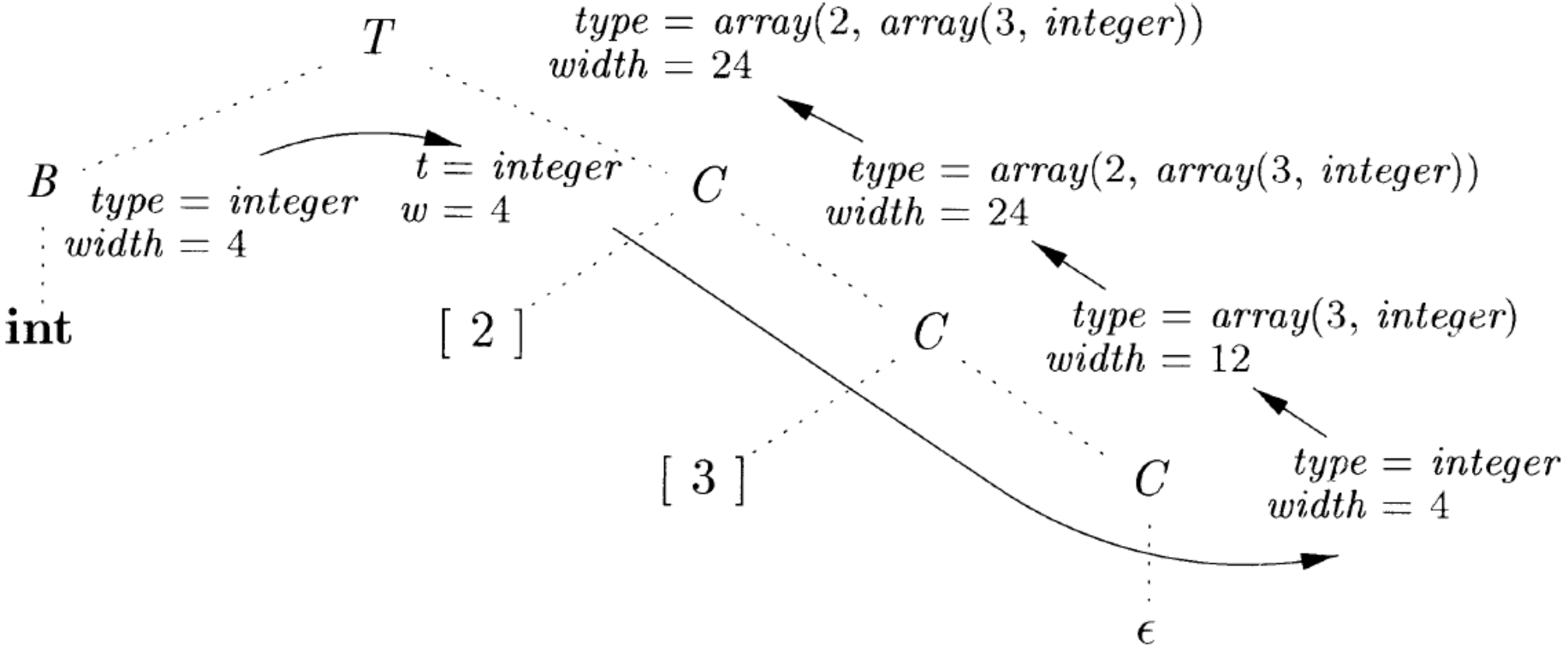
计算类型和宽度的SDT

- 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，因为总是通过拷贝来传递，所以在SDT中只赋值一次，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

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

- 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
- 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，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 → {D.offset =0} D
  - D → T id; {D_1.offset = D.offset + T.width;} D_1

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

记录字段的处理

- T → record ‘{‘ D ‘}’
- 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

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

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
$S \rightarrow \mathbf{id} = E\ ;$
$E \rightarrow E_1 + E_2$
$\mid\ - E_1$
$\mid\ (E_1)$
$\mid\ \mathbf{id}$

表达式代码的SDD

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

增量式翻译方案

- 主属性code满足增量式翻译的条件
- 注意
  - top.get(…)从栈顶符号表开始，逐个向下寻找id的信息
  - 这里的gen发出相应的代码

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

数组元素的寻址

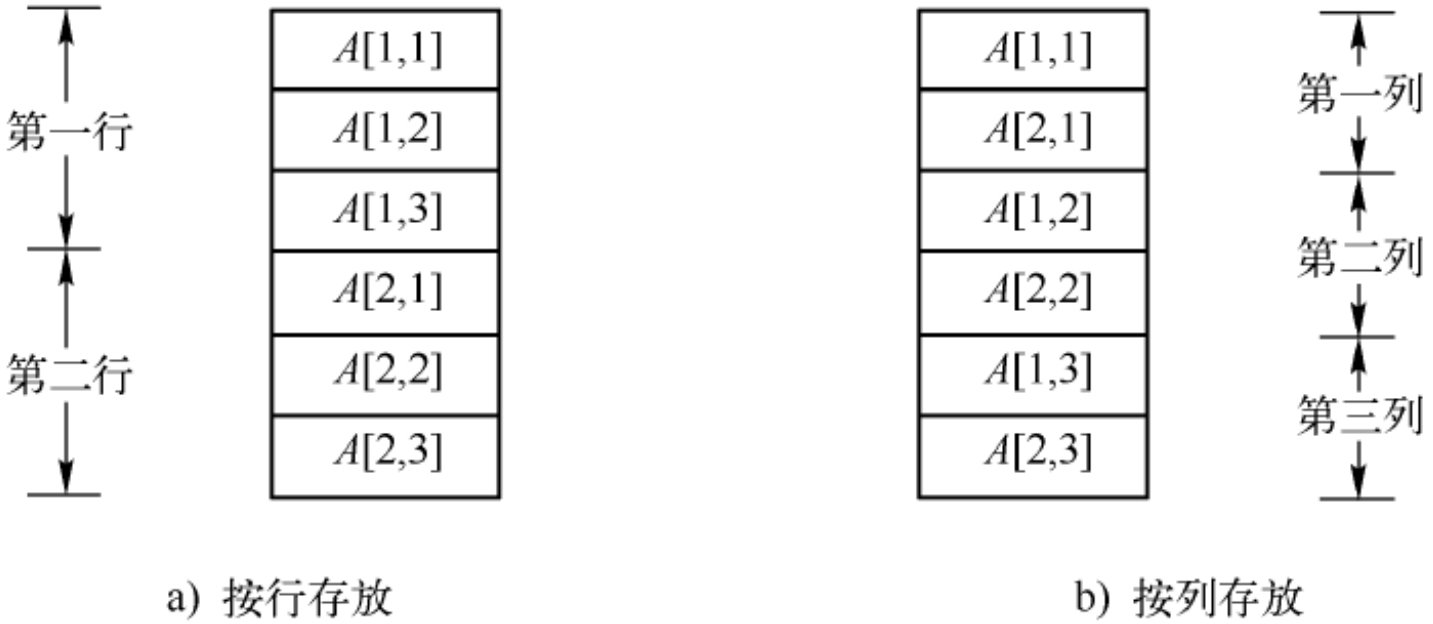
- 数组元素存储在一块连续的存储空间中，以方便快速的访问它们
- n个数组元素是0,1,…,n-1进行顺序编号的
- 假设每个数组元素宽度是w，那么数组A的第i个元素的开始地址为base+i*w, base是A[0]的相对地址。
- 推广到二维或多维数组。$A[i_1][i_2]$表示第i_1行第i_2个元素。假设一行的宽度是w_1，同一行中每个元素的宽度是w_2，$A[i_1][i_2]$的相对地址是 $base+i_1*w_1+i_2*w_2$
- 对于k维数组$A[i_1][i_2]\ldots[i_k]$，推广
  - $base+i_1*w_1+i_2*w2+\ldots+i_k*w_k$

数组元素的寻址(续)

- 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 $base+(i_1*n_2+i_2)*w$
- 于k维数组$A[i_1][i_2]\ldots[i_k]$ 的地址$base+((\ldots(i_1*n_2+i_2)*n_3+i_3)\ldots)*n_k+i_k)*w$
- 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
  - 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



a) 按行存放　　b) 按列存放

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
  - 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 ;	{ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr);$ }
	|	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

核心是确定数组引用的地址

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
  - 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

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

$E \rightarrow E_1 + E_2$ { $E.addr = \textbf{new}\ Temp()$;
$gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }

| **id** { $E.addr = top.get(\textbf{id}.lexeme)$; }

| L { $E.addr = \textbf{new}\ Temp()$;
$gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; }

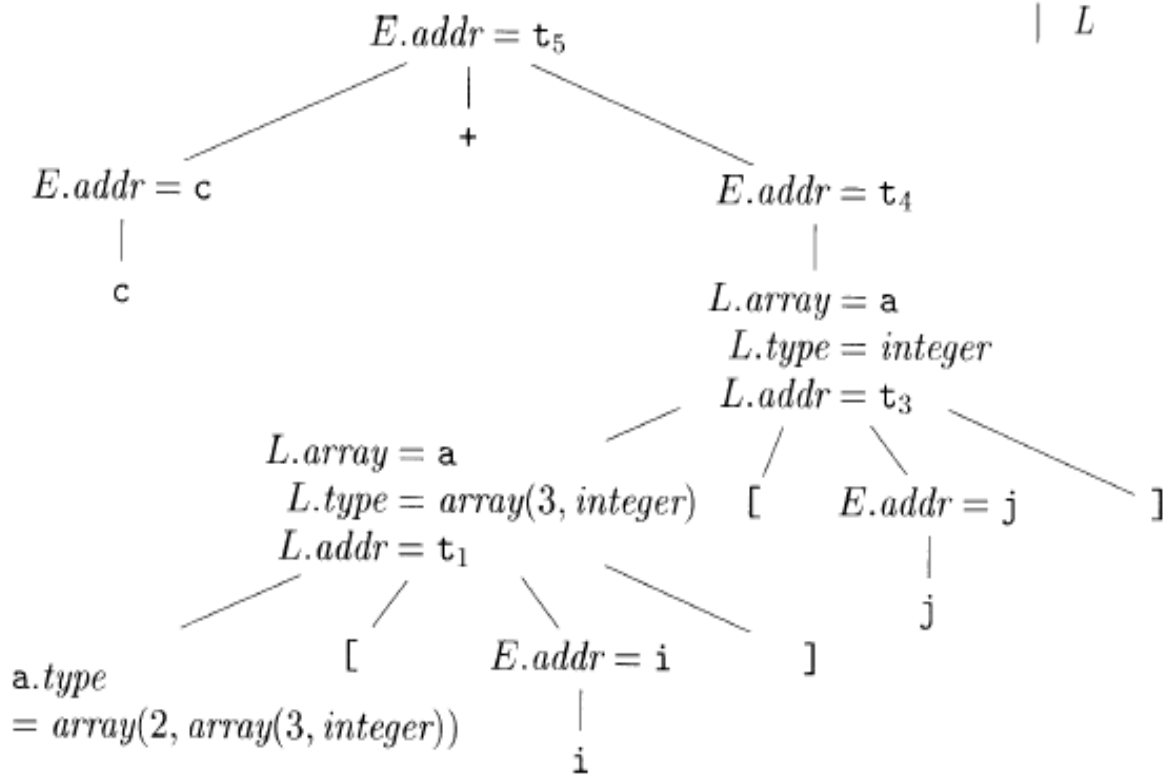


图 6-23 c + a[i][j]的注释语法分析树

```
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

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 {

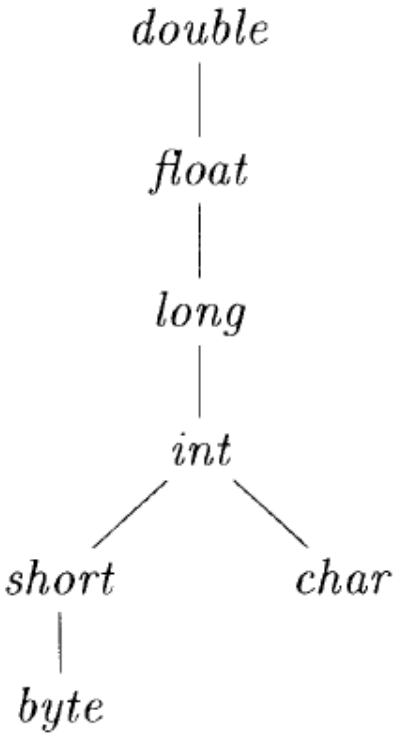
    if(E1.type = integer and E2.type = integer) E.type = integer;

    else if (E1.type = float and E2.type= integer) E.type = float;

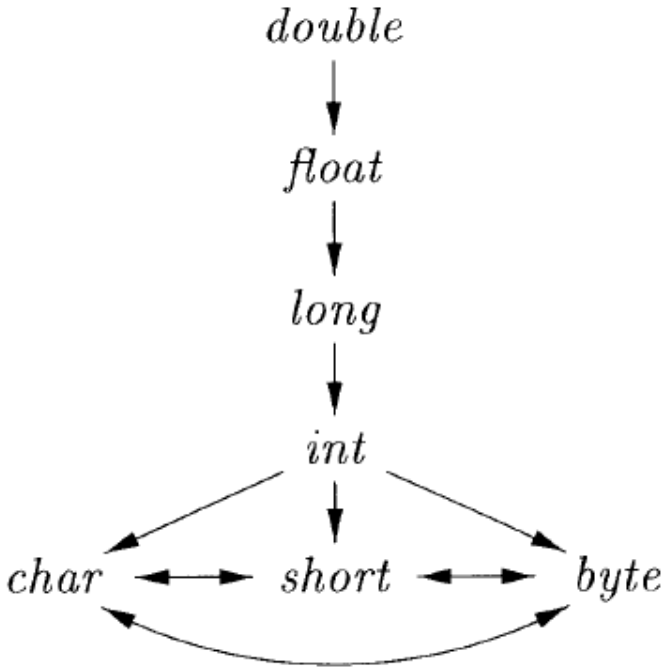
    }
  - 这个规则没有考虑生成类型转换代码

类型的widening和narrowing

- 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

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

```
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求的是两个参数在拓宽层次结构中的最小公共祖先
- Widen函数已经生成了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 = $\{s_i \rightarrow t_i \mid 1 <= i <= k\}$ and
E_1.type=s_k
then E.type = t_k
}

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

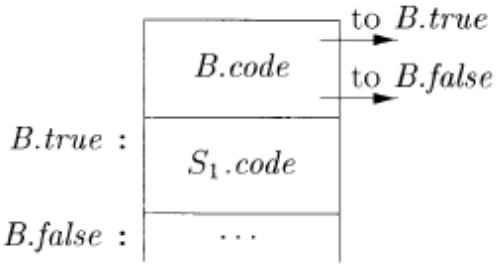
- 语句及文法

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

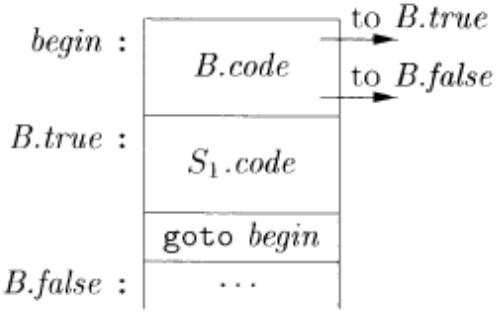
$S \rightarrow$ **if** (B) S_1 **else** S_2

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

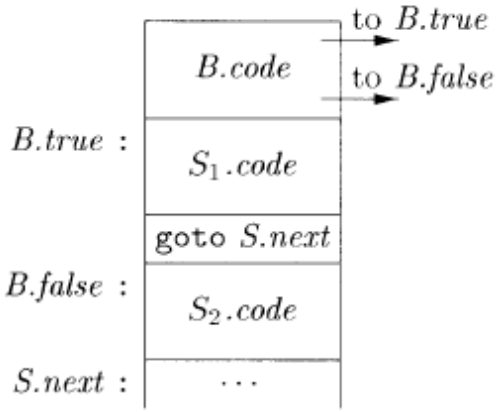
- *B*和*S*有综合属性*code*，表示翻译得到的三地址代码。
- *B*的继承属性*true*和*false*，*S*的继承属性*next*，表示跳转的位置。



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
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \;\mathbf{rel}\; E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\texttt{'if'}\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ \texttt{'goto'}\ B.true)$ $\mid\mid gen(\texttt{'goto'}\ B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen(\texttt{'goto'}\ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen(\texttt{'goto'}\ B.false)$

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*) S_1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

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

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

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

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

- 注意B.true=fall时，还得为B_1.true new一个label

$$
\begin{aligned}
B_1.true &= B.true \\
B_1.false &= newlabel() \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= B_1.code \parallel label(B_1.false) \parallel B_2.code
\end{aligned}
$$

B →B_1&&B_2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

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

使用标号fall的控制流语句翻译示例

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

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

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

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

图 6-40　$B \rightarrow B_1 \mid\mid B_2$ 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

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

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

$B.code = E_1.code \mid\mid E_2.code \mid\mid s$

图 6-39　$B \rightarrow E_1 \textbf{ rel } E_2$ 的语义规则

布尔表达式的两个功能

- 改变控制流，跳转
  - 刚刚前面重点讨论，用非终结符号B表示此种功能的布尔表达式以示区别
- 求值
  - 如*x*=true，*x*=*a*<*b*
- 统一处理

$$S \rightarrow \mathbf{id} = E; \mid \mathbf{if}\ (E)\ S \mid \mathbf{while}\ (E)\ S \mid S\ S$$

$$E \rightarrow E \mid\mid E \mid E \&\& E \mid E\ \mathbf{rel}\ E \mid E + E \mid (E) \mid \mathbf{id} \mid \mathbf{true} \mid \mathbf{false}$$

  - 使用不同的代码生成函数处理表达式的两种角色。E.n对应于抽象语法树上的表达式节点。两个函数：
    - jump, 对于出现在S → while(E)S1中的E，调用E.n.jump(t,f)
    - rvalue, 对于出现在S → id=E中的E，在节点E.n上调用rvalue。如果E是算术表达式，按照算术表达式的翻译生成代码。如果E是布尔表达式，如E1&E2，首先为E生成跳转代码，然后在跳转代码的真假出口分别将true或false赋给一个新的临时变量t。

示例

- 赋值语句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

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

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

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

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

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

$$B \rightarrow\ !\, B_1 \quad \Big| \quad \begin{array}{l} B_1.true = B.false \\ B_1.false = B.true \\ B.code = B_1.code \end{array}$$

$$B \rightarrow\ !\, B_1 \quad \{\ \begin{array}{l} B.truelist = B_1.falselist; \\ B.falselist = B_1.truelist;\ \} \end{array}$$

$$B \rightarrow (\ B_1\ ) \quad \{\ \begin{array}{l} B.truelist = B_1.truelist; \\ B.falselist\ =\ B_1.falselist;\ \} \end{array}$$

$$B \rightarrow E_1 \ \mathbf{rel} \ E_2 \quad \Big| \quad \begin{array}{l} B.code = E_1.code \ || \ E_2.code \\ \qquad || \ gen(\text{'}\texttt{if}\text{'} \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ \text{'}\texttt{goto}\text{'} \ B.true) \\ \qquad || \ gen(\text{'}\texttt{goto}\text{'} \ B.false) \end{array}$$

$$B \rightarrow E_1 \ \mathbf{rel} \ E_2 \qquad \begin{array}{l} \{ \ B.truelist = makelist(nextinstr); \\ \quad B.falselist = makelist(nextinstr + 1); \\ \quad emit(\text{'}\texttt{if}\text{'} \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ \text{'}\texttt{goto}\ \_\text{'}); \\ \quad emit(\text{'}\texttt{goto}\ \_\text{'}); \ \} \end{array}$$

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

$B \rightarrow \textbf{true}$	$\{\ B.truelist = makelist(nextinstr);$ $emit('\texttt{goto}\ \_');\ \}$
$B \rightarrow \textbf{false}$	$\{\ B.falselist = makelist(nextinstr);$ $emit('\texttt{goto}\ \_');\ \}$
$M \rightarrow \epsilon$	$\{\ M.instr = nextinstr;\ \}$

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

回填和非回填方法的比较（2）

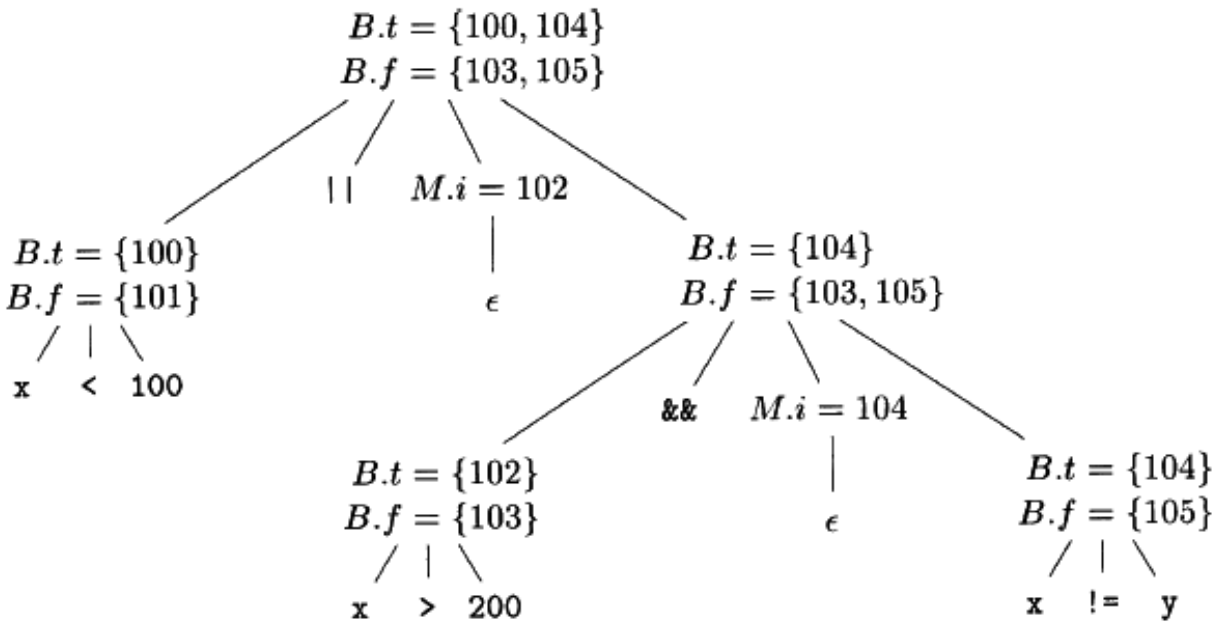
$$B \rightarrow E_1 \ \mathbf{rel} \ E_2 \quad \Big| \quad \begin{aligned} B.code = {} & E_1.code \ || \ E_2.code \\ & || \ gen(\texttt{'if'} \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ \texttt{'goto'} \ B.true) \\ & || \ gen(\texttt{'goto'} \ B.false) \end{aligned}$$

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

- 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

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 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

```
      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 → **if** (B) S | **if** (B) S **else** S | **while** (B) S | { L } | A

  L → L S | 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

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记录下一个指令的位置
  - 规则1中记录了then分支的代码起始位置；
  - 规则2中，分别记录了then分支和else分支的起始位置；
- N的作用是生成goto指令坯，N.nextlist只包含这个指令的位置

控制转移语句的回填

3) $S \rightarrow$ **while** M_1 (B) M_2 S_1

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

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

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

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

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

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

$$
\begin{aligned}
&begin = newlabel() \\
&B.true = newlabel() \\
&B.false = S.next \\
&S_1.next = begin \\
&S.code = label(begin) \,||\, B.code \\
&\qquad\qquad ||\, label(B.true) \,||\, S_1.code \\
&\qquad\qquad ||\, gen(\texttt{'goto'}\ begin)
\end{aligned}
$$

$S \rightarrow$ **while** M_1 (B) M_2 S_1

$$
\begin{aligned}
\{\ & backpatch(S_1.nextlist,\ M_1.instr); \\
& backpatch(B.truelist,\ M_2.instr); \\
& S.nextlist = B.falselist; \\
& emit(\texttt{'goto'}\ M_1.instr);\ \}
\end{aligned}
$$

$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

$S \rightarrow \{\ L\ \}$	$\{\ S.nextlist = L.nextlist;\ \}$
$S \rightarrow A\ ;$	$\{\ S.nextlist = \textbf{null};\ \}$
$M \rightarrow \epsilon$	$\{\ M.instr = nextinstr;\ \}$
$N \rightarrow \epsilon$	$\{\ N.nextlist = makelist(nextinstr);$ $emit('\texttt{goto}\ \_');\ \}$
$L \rightarrow L_1\ M\ S$	$\{\ backpatch(L_1.nextlist,\ M.instr);$ $L.nextlist = S.nextlist;\ \}$
$L \rightarrow S$	$\{\ L.nextlist = S.nextlist;\ \}$

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