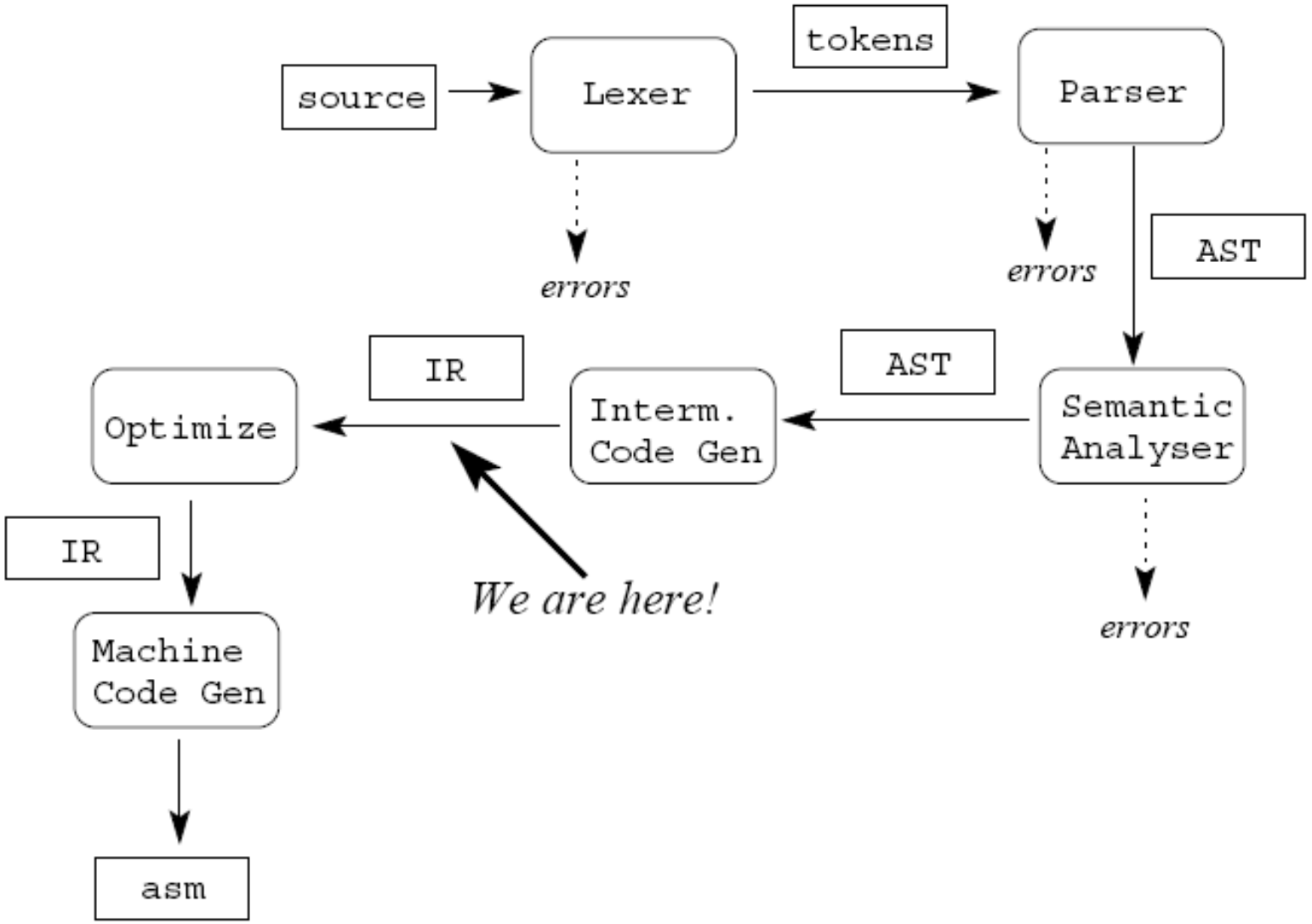
第6章 中间代码生成

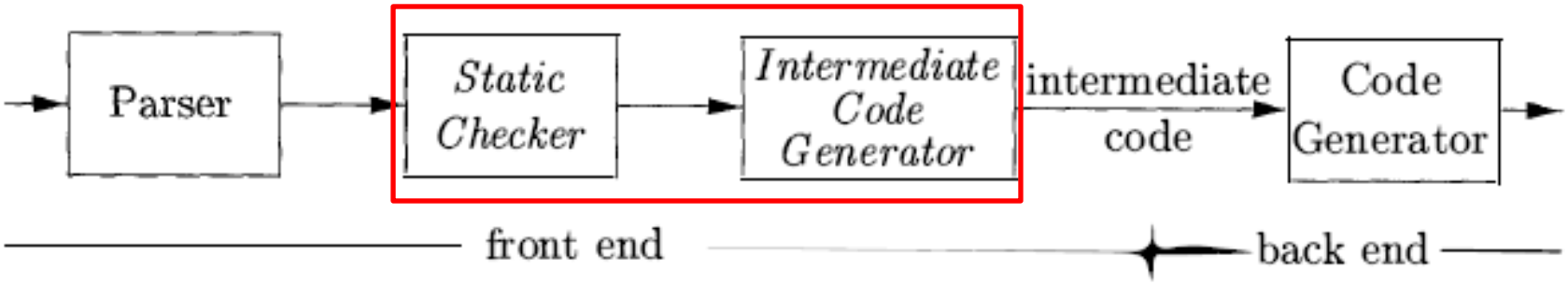
西北大学信息学院计算机科学系 付丽娜

中间代码生成

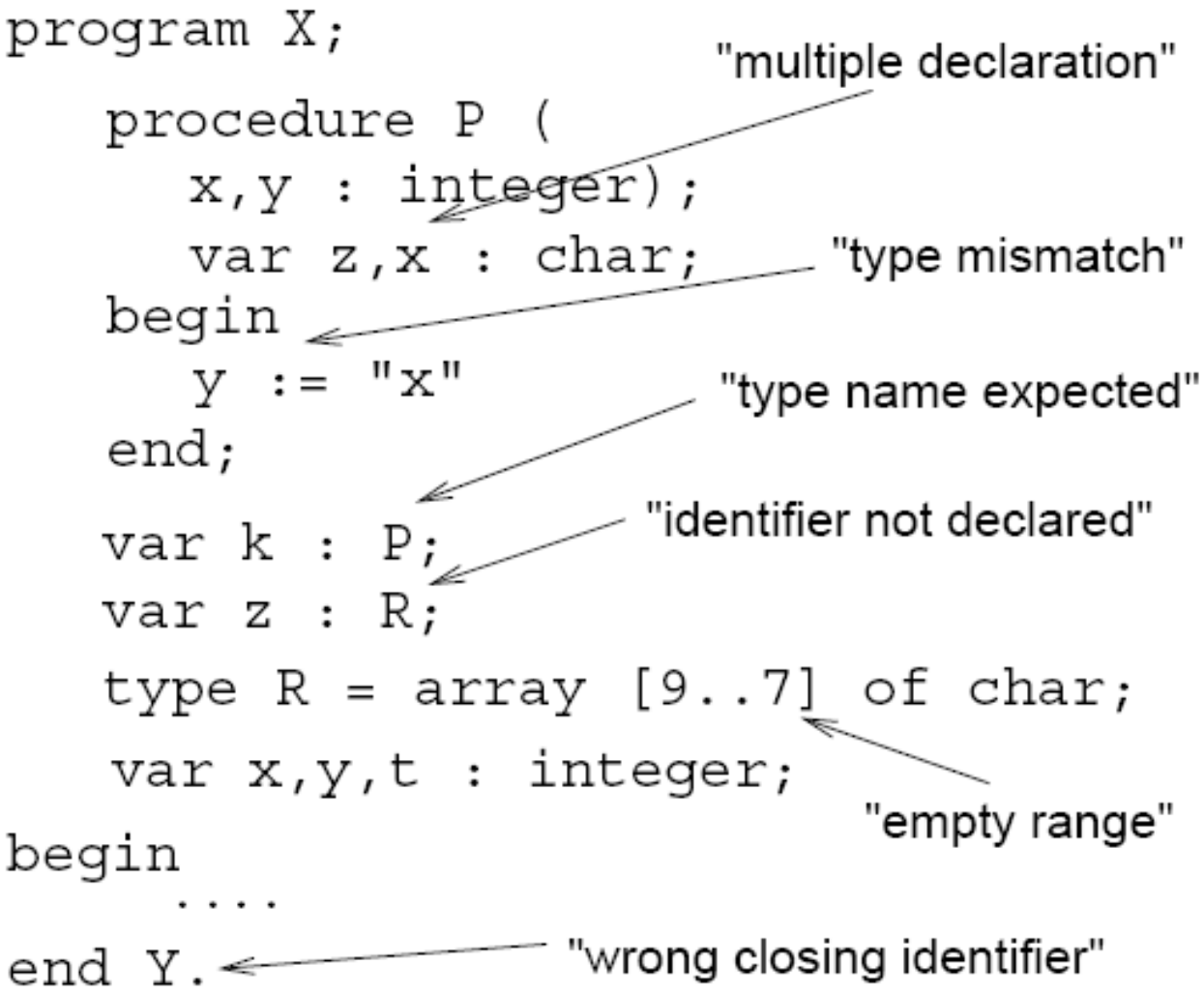

source
Lexer
tokens
Parser
errors
errors
AST
IR
AST
Optimize
Interm.
Code Gen
Semantic
Analyser
We are here!
errors
IR
Machine
Code Gen
asm

Overview

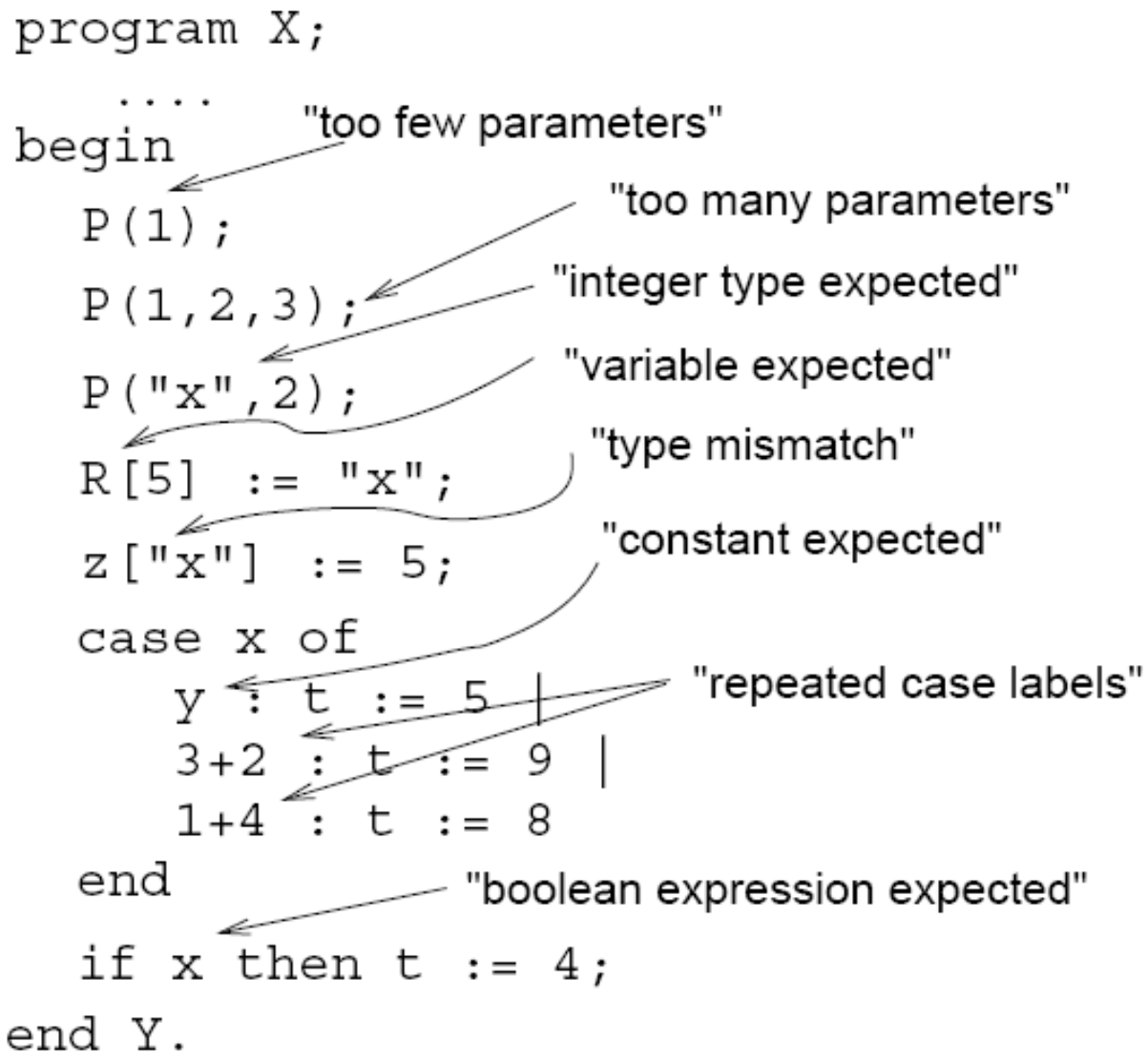
- 本章讨论
  - 中间代码表示
    - 抽象语法树（DAG）和三地址码
    - 编译器可能使用一系列的中间表示，高层的表示接近源语言，低层的表示接近目标语言，更适用于机器相关的处理任务
  - 静态类型检查
    - 包括类型检查和在语法分析之后进行的所有语法检查
  - 中间代码生成



静态语义错误……1


program X;
procedure P (
x,y : integer);
var z,x : char;
begin
y := "x"
end;
var k : P;
var z : R;
type R = array [9..7] of char;
var x,y,t : integer;
begin
....
end Y.
"multiple declaration"
"type mismatch"
"type name expected"
"identifier not declared"
"empty range"
"wrong closing identifier"

静态语义错误……2


program X;
....
begin
P(1);
P(1,2,3);
P("x",2);
R[5] := "x";
z["x"] := 5;
case x of
y : t := 5 |
3+2 : t := 9 |
1+4 : t := 8
end
if x then t := 4;
end Y.
"too few parameters"
"too many parameters"
"integer type expected"
"variable expected"
"type mismatch"
"constant expected"
"repeated case labels"
"boolean expression expected"

Contents

中间代码形式

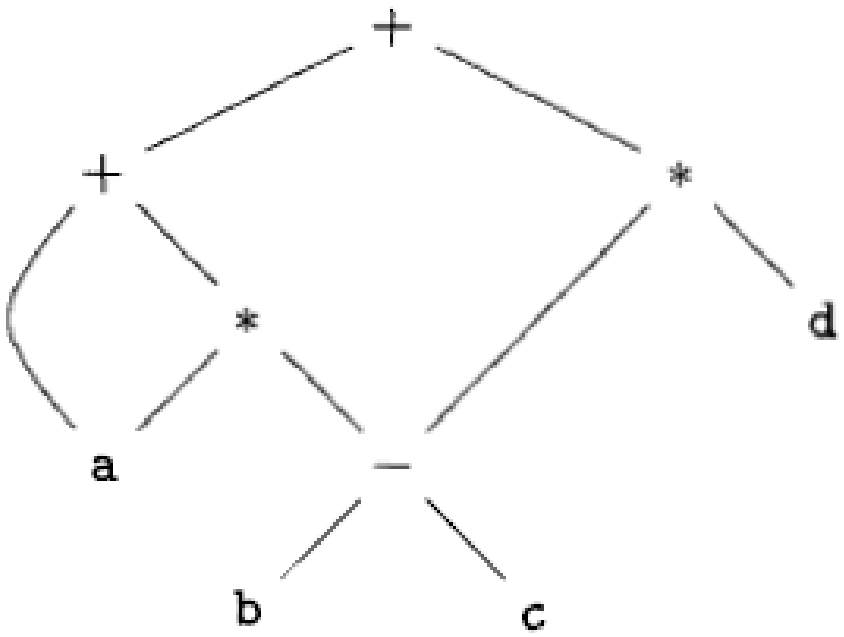
抽象语法树、DAG和三地址码

语法树的变体

- 抽象语法树
  - 结点代表源程序中的构造
  - 结点的子结点反映了该结点对应构造的有意义的组成部分
- 有向无环图（*Directed Acyclic Graph, DAG*）
  - 抽象语法树的变体，其中指出了表达式中的公共子表达式（多次出现的子表达式）；能更简洁地表示表达式，便于优化
  - 叶子结点对应于原子运算分量，内部结点对应于运算符
  - 如果结点N表示一个公共子表达式，则N可能有多个父结点

表达式的有向无环图

- 例6.1
  - 表达式 a + a * (b-c) + (b-c) * d 的DAG



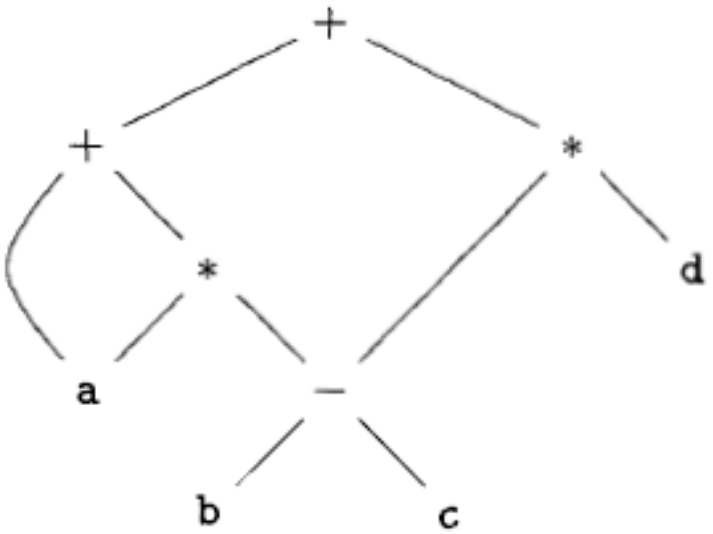
构造语法树和DAG的SDD

- 说明
  - 函数Node和Leaf调用一次都构造一个新结点
  - 构造DAG时，函数在构造新结点之前先检查是否已经存在这样的结点
    - 如果存在一个已创建的结点，就返回这个已有结点

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

例……构造DAG

- 例6.2
  - 构造表达式 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) $p_{13} = Node('+', p_7, p_{12})$

练习6.1(课堂练习)

练习 6.1.1：为下面的表达式构造 DAG

$$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$$

练习 6.1.2：为下列表达式构造 DAG，假定 + 是左结合的。

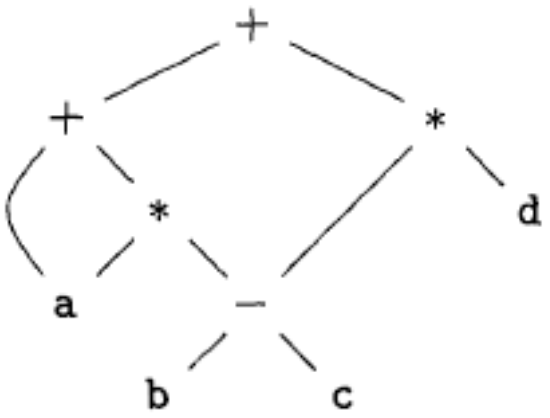
1) $a+b+(a+b)$

2) $a+b+a+b$

3) $a+a+(a+a+a+(a+a+a+a))$

三地址代码

- 三地址代码的一般形式 $x = y\ op\ z$
  - y和z是两个运算分量，x是结果变量
  - op是运算符，一条三地址代码指令的右侧最多有一个运算符
- 三地址代码是语法树或DAG的线性表示
  - 三地址代码中的名字对应图中的内部结点



a) DAG

$$
\begin{aligned}
t_1 &= b - c \\
t_2 &= a * t_1 \\
t_3 &= a + t_2 \\
t_4 &= t_1 * d \\
t_5 &= t_3 + t_4
\end{aligned}
$$

b) 三地址代码

地址和指令

- 三地址代码基于两个基本概念
  - 地址
  - 指令
- 地址可以具有如下形式之一
  - 名字：允许源程序中的名字作为三地址代码中的地址
    - 实现时名字被替换为指向符号表条目的指针
  - 常量：各种类型的常量
  - 编译器生成的临时变量：每次需要临时变量时产生一个新名字

常见的三地址指令形式

指令形式	描述
$x = y$ ***op*** z	op是双目运算符或逻辑运算符，x,y,z是地址
$x =$ ***op*** y	op是单目运算符
$x = y$	复制指令，将y的值赋给x
goto L	无条件转移，L是下一步要执行的指令标号
if x ***goto*** L ***if*** *False* x ***goto*** L	当x为真或为假时，转向L，否则执行下一条指令
if x ***relop*** y ***goto*** L	对x和y应用关系运算，满足relop关系转向L
param x ***call*** p, n $y =$ ***call*** p, n ***return*** y	过程调用和返回指令 传递参数，调用，返回
$x = y[i]$ $x[i] = y$	带下标的复制指令，x和y是地址，i是偏移量
$x = \&y$ $x = *y$ $*x = y$	地址和指针指令

例……三地址指令

- 例6.5 下面的do-while语句的两种翻译

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

  - i*8 计算每个元素占8个存储单元的数组元素的偏移量

```
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

四元式表示

- 三地址指令在数据结构中的表示方法
  - 四元式、三元式、间接三元式
- 四元式（*quadruple*）
  - 一个四元式有四个字段：*op arg1 arg2 result*
    - op：运算符的内部编码
  - 三地址码 *x = y op z* 对应的四元式是 *op y z x*
    a) 单目运算符指令 x = op y和赋值指令 x = y不使用arg2
    b) param运算符不使用arg2和result
    c) 条件或无条件转移指令的目标标号L放入result字段

例……三地址代码及其四元式表示

❖ 例6.6

赋值语句 a = b* -c + b * -c的三地址代码和四元式序列

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

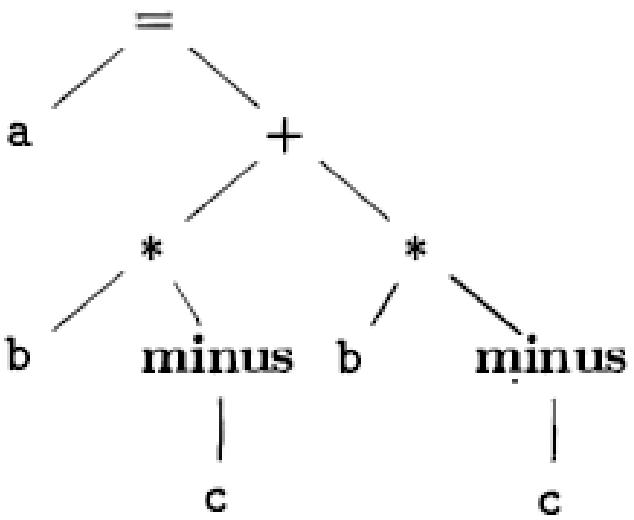
$t_5 = t_2 + t_4$

$a = t_5$

	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
	...			

三元式表示

- 三元式（*triple*）
  - 一个三元式有三个字段 *op arg1 arg2*
  - 使用三元式时，用运算的位置来表示其结果，而不是用显式的临时变量名字
  - 例6.7： **a = b* -c * b * -c**的语法树和三元式序列



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

四元式VS.三元式

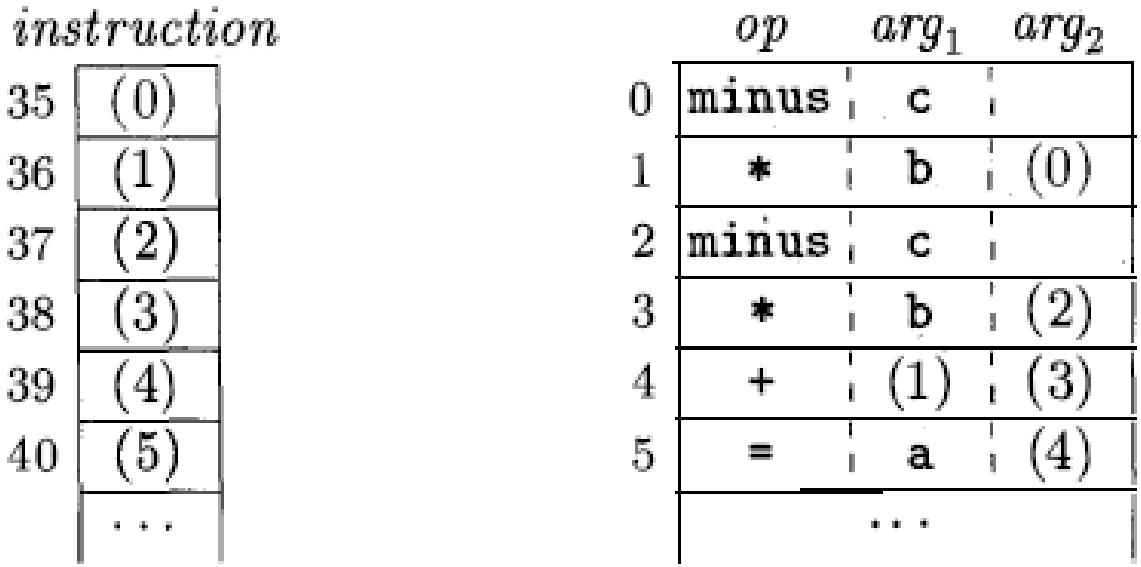
- 四元式和三元式的优缺点
  - 四元式表示中使用大量临时变量，但便于优化
  - 三元式表示不便于优化
    - 优化时指令位置常常发生变化，用位置表示结果会引起相应指令的修改

	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
	...			

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

间接三元式

- 间接三元式（*indirect triple*）
  - 包含一个执行三元式的指针列表*instruction*
  - 优化时对*instruction*列表重新排序来移动指令的位置，不影响三元式本身



练习6.2（作业）

练习 6.2.1：将算术表达式 a + -(b + c)翻译成

1）抽象语法树

2）四元式序列

3）三元式序列

4）间接三元式序列

练习 6.2.2：对下列赋值语句重复练习 6.2.1。

1) `a = b[i] + c[j]`

2) `a[i] = b*c - b*d`

3) `x = f(y+1) + 2`

4) `x = *p + &y`

类型和声明

类型和声明

- 类型的应用：

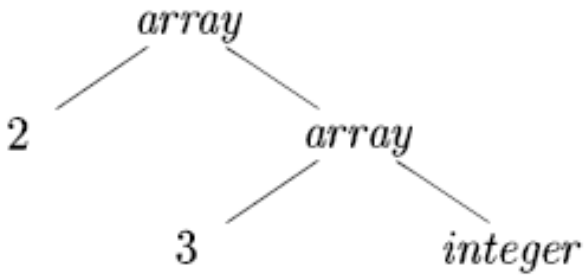
1. 类型检查（*type checking*）
   - 类型检查利用一组逻辑规则来推理一个程序在运行时刻的行为
   - 类型检查保证运算分量的类型和运算符的预期类型相匹配
2. 翻译时的应用（*translation application*）
   - 根据一个名字的类型，编译器可以确定这个名字在运行时刻需要多大的存储空间
   - 其他应用包括计算数组引用指向的地址、插入显式类型转换、选择正确版本的运算符等

类型和声明

- 名字的类型和存储空间布局
  - 名字的类型在过程中或类中声明
  - 一个过程调用的实际存储空间是在运行时刻调用该过程时进行分配的
  - 对象的实际存储空间是在运行时刻创建该类对象时分配
  - 在编译时刻检查局部声明，可以进行相对地址的布局
- 相对地址（*relative address*）
  - 一个名字或某个数据结构分量的相对地址是指它相对于数据区域开始位置的偏移量

类型表达式

- 类型表达式（*type expression*）
  - 表示类型自身的结构
  - 可以是基本类型，也可以是把类型构造算子运算符应用于类型表达式得到
  - 基本类型的集合和类型构造算子根据被检查的具体语言而定
  - 可以用图表示类型表达式：内部结点表示类型构造算子，叶子结点是基本类型、类型名或类型变量
- 例6.8 数组类型 `int[2][3]` 的类型表达式和图表示
  - *array(2, array(3, integer))*
  - *array*运算符有两个参数
    - 数字，类型



类型表达式

- 类型表达式的定义
  - 基本类型是一个类型表达式
  - 类名是一个类型表达式
  - 将类型构造算子*array*作用于一个数字和一个类型表达式可以得到一个类型表达式
  - 一个记录是包含有名字段的数据结构；将*record*类型构造算子应用于字段名和相应的类型可以构造得到一个类型表达式
  - 使用类型构造算子→可以构造得到函数类型的类型表达式，$s \rightarrow t$表示从类型s到类型t的函数
  - 如果s和t是类型表达式，则其笛卡尔积s×t也是类型表达式
  - 类型表达式可以包含取值为类型表达式的变量

类型等价

- 类型表达式的等价
  - 用图表示类型表达式的时候，两种类型之间结构等价（*structurally equivalent*），当且仅当下面的某个条件为真：
    - 它们是相同的基本类型
    - 它们是将相同的类型构造算子应用于结构等价的类型而构造得到
    - 一个类型是另一个类型表达式的名字
  - 类型表达式的名等价（*name equivalence*）关系
    - 如果类型名仅仅代表它自身，那么上述定义中的前两个条件定义了类型表达式的名等价关系
    - 注意：类型名除了代表自身还可以被看作是另一个类型表达式的一种缩写形式

声明

- 简化的声明文法
  - 一次只声明一个名字

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

存储布局

- 局部变量名的存储布局
  - 变量的类型决定了该变量在运行时刻需要的内存数量；编译时刻可以使用这些数量为每个名字分配一个相对地址
  - 名字的类型和相对地址信息保存在相应的符号表条目中
  - 变长数据和动态数组只有运行时刻才能确定其大小，编译时刻为指向这些数据的指针保留一个已知的固定大小的存储区域
- 数据对象的存储布局受目标机器的寻址约束的影响
  - 地址对齐（*aligned*）和补白（*padding*）
  - 压缩（*pack*）

存储布局

- 存储区域
  - 存储区域是连续字节块，字节是可寻址的最小内存单位；
  - 一个字节通常有8个二进制位，若干字节组成一个机器字；
  - 多字节数据对象往往被存储在一段连续的字节中，并以初始字节的地址作为该数据对象的地址
- 类型的宽度（*width*）
  - 指该类型的一个对象所需的存储单元的数量

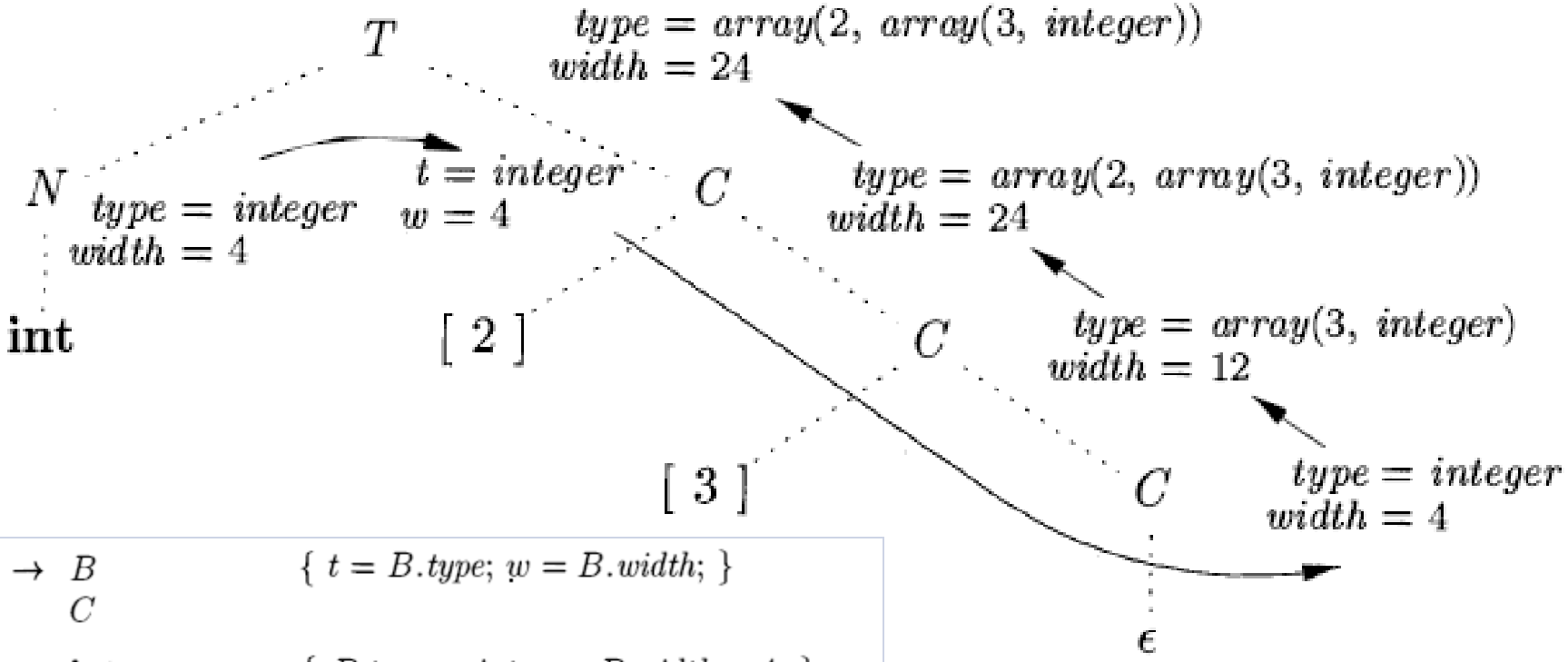
计算类型及其宽度的SDT

- 属性和变量
  - 每个非终结符有综合属性type和width
  - 变量t和w的用途是将类型和宽度信息从语法分析树的B结点传递到对应于产生式C→ε的结点；
    - 在SDD中t和w是C的继承属性

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

例……数组类型的语法制导翻译

❖ 例6.9 int[2][3]



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

声明的序列

- 处理声明序列的SDT
  - 变量*offset*跟踪下一个可用的相对地址

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

记录和类中的字段

- 记录类型
  - 在图6-15中加入产生式 $T \rightarrow \textbf{record}\ '\{'\ D\ '\}'$
  - 用图6-17的方法确定字段的类型和相对地址
    - 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（或相对地址）是相对于该记录的数据区字段而言的
- 处理记录中字段名的语义动作

$$T \rightarrow \textbf{record}\ '\{'\quad \{\ Env.push(top);\ top = \textbf{new}\ Env();\ Stack.push(offset);\ offset = 0;\ \}$$

$$D\ '\}'\quad \{\ T.type = record(top);\ T.width = offset;\ top = Env.pop();\ offset = Stack.pop();\ \}$$

练习6.3

练习6.3.1：确定下列声明序列中各个标识符的类型和相对地址。

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

表达式的翻译

将表达式翻译为三地址代码

表达式的三地址代码

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

表达式中的运算

- SDD中的属性和函数
  - *code*：赋值语句和表达式对应的三地址代码
  - *addr*：存放表达式值的地址
  - *gen(x '=' y 'op' z)*：生成一条三地址指令 *x= y op z*
  - *new Temp()*：产生一个临时变量
- 例6.11：翻译赋值语句 `a = b + -c;`
  - 翻译得到的三地址代码序列

$$t_1 = \text{minus } c$$
$$t_2 = b + t_1$$
$$a = t_2$$

增量翻译

- 增量生成表达式的三地址代码的SDT
  - 不需要*code*属性，对*gen*的连续调用生成一个指令序列

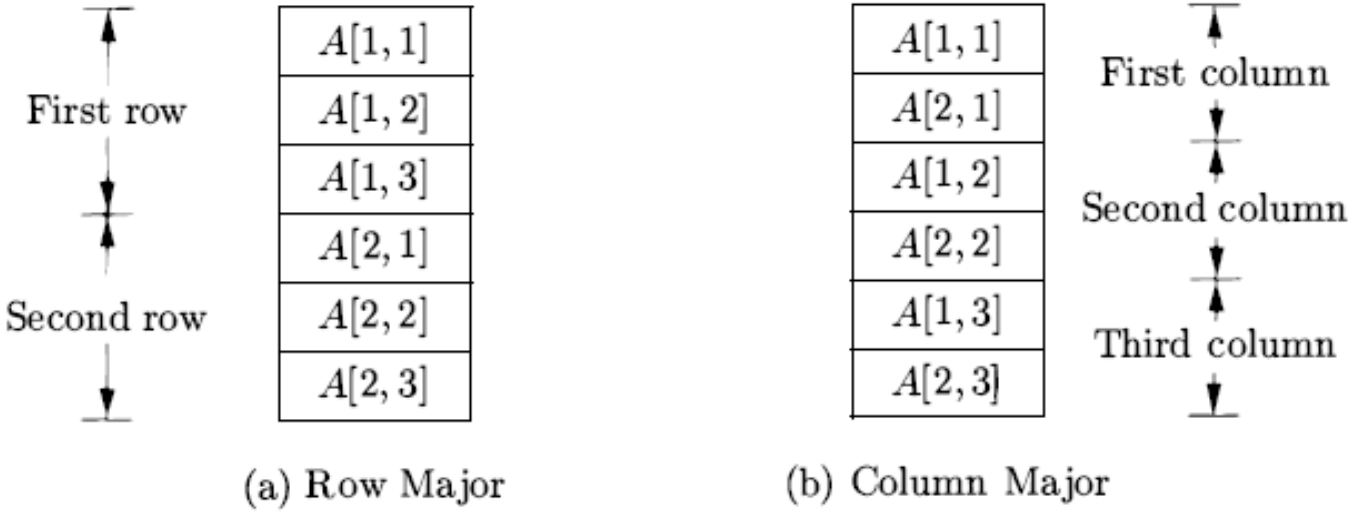
$$
\begin{array}{lll}
S \rightarrow \mathbf{id} = E\ ; & \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \} \\
E \rightarrow E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\ ();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
\mid\ -\ E_1 & \{\ E.addr = \mathbf{new}\ Temp();\ gen(E.addr\ '='\ '\mathbf{minus}'\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

数组元素的寻址

- 一维数组
  - 元素$A[i]$的开始地址=$base + i * w$
    - 假设：有n个元素的数组中元素的下标从0到$n-1$；数组下标不从0开始，从low开始时将i 换为 $i-low$
    - $base$是分配给数组A的内存块的相对地址，即$A[0]$的相对地址；w是每个数组元素的宽度
- 二维数组
  1. 元素$A[i_1][i_2]$的相对地址= $base + i_1 * w_1 + i_2 * w_2$
     - w_1是一行的宽度，w_2是同一行中每个元素的宽度
  2. $A[i_1][i_2]$的相对地址= $base + (i_1 * n_2 + i_2) * w$
     - w是数组每个元素的宽度，的n_2是第2维（列）的元素个数

数组元素的寻址

- k维数组A的元素$A[i_1]...[i_k]$的相对地址计算公式
  1. $base + i_1 * w_1 + i_2 * w_2 + ... + i_k * w_k$
  2. $base + ((...(i_1 * n_2 + i_2) * n_3 + i_3)...)) * n_k + i_k) * w$
- 按行存放和按列存放
  - 上面的公式是按行存放的计算方法



(a) Row Major　　(b) Column Major

数组引用的翻译

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

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

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

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

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

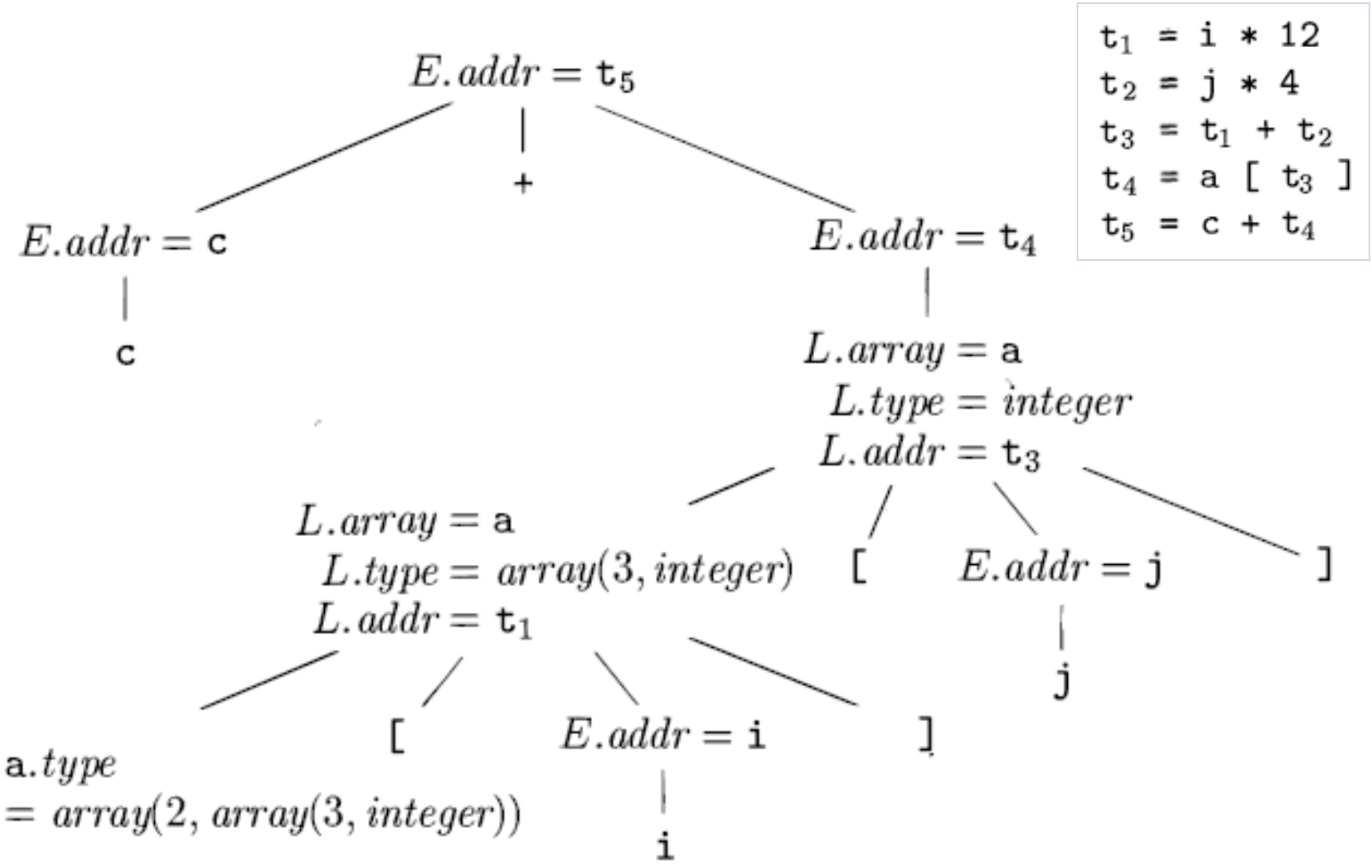
$L \rightarrow \mathbf{id}\ [\ E\ ]$ { $L.array = top.get(\mathbf{id}.lexeme);$
$L.type = L.array.type.elem;$
$L.addr = \mathbf{new}\ Temp();$
$gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);$ }

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

数组引用的翻译

- 非终结符号L的综合属性
  - *L.addr*：用于累加 $i_j * w_j$ 的临时变量，计算数组引用的偏移量
  - *L.array*：指向数组名字对应的符号表条目的指针，*L.array.base*是数组的基地址
  - *L.type*：L生成的子数组的类型
- 例6.12：表达式 `c + a[i][j]` 的注释语法分析树和翻译
  - *a*表示2×3的整型数组，*c*, *i*, *j*都是整数
  - *a*的类型是 *array(2, array(3, integer))*
  - 假设一个整数的宽度是4，那么*a*类型的宽度是24
  - *a[i]*的类型是 *array(3, integer)*

例……数组引用的翻译


$E.addr = \mathtt{t}_5$
+
$E.addr = \mathtt{c}$
c
$E.addr = \mathtt{t}_4$
$L.array = \mathtt{a}$
$L.type = integer$
$L.addr = \mathtt{t}_3$
$L.array = \mathtt{a}$
$L.type = array(3, integer)$
$L.addr = \mathtt{t}_1$
[
$E.addr = \mathtt{j}$
]
j
[
$E.addr = \mathtt{i}$
]
i
$\mathtt{a}.type$
$= array(2, array(3, integer))$
$\mathtt{t}_1$ = i * 12
$\mathtt{t}_2$ = j * 4
$\mathtt{t}_3$ = $\mathtt{t}_1$ + $\mathtt{t}_2$
$\mathtt{t}_4$ = a [$\mathtt{t}_3$]
$\mathtt{t}_5$ = c + $\mathtt{t}_4$

练习6.4

练习 6.4.1：向图 6-19 的翻译方案中加入对应于下列产生式的规则：

1）$E \rightarrow E_1 * E_2$

2）$E \rightarrow +E_1$（单目加）

练习 6.4.2：使用图 6-20 中的增量式翻译方案重复练习 6.4.1。

类型检查

- 编译器的类型检查
  - 编译器给源程序的每个组成部分赋予一个类型表达式
  - 然后，编译器确定这些类型表达式是否满足一组逻辑规则
- 这些规则称为源语言的**类型系统**（*type system*）
  - 原则上，如果目标代码在保存元素值的同时保存了元素类型的信息，那么任何检查都可以动态地进行
  - 一个健全（*sound*）的类型系统可以消除对动态类型错误检查的需要
  - 如果一个编译器可以保证它接受的程序在运行时刻不会发生类型错误，那么该语言的这个实现就被称为**强类型的**

类型检查规则

- 类型检查有综合和推导两种形式
- 类型综合（*type synthesis*）
  - 根据子表达式的类型构造出表达式的类型；即表达式E_1+E_2的类型是根据E_1和E_2的类型定义的
  - 要求名字先声明再使用
- 一个典型的类型综合规则具有如下形式

$$\begin{array}{l}\textbf{if } f \text{ has type } s \to t \textbf{ and } x \text{ has type } s, \\ \textbf{then} \text{ expression } f(x) \text{ has type } t\end{array} \qquad (6.8)$$

类型检查规则

- 类型推导（*type inference*）
  - 根据一个语言结构的使用方式来确定该结构的类型
  - 代表类型表达式的变量使得我们可以考虑未知类型
    - 可以用希腊字母α、β等作为类型表达式中的类型变量
- 一个典型的类型推导规则具有如下形式

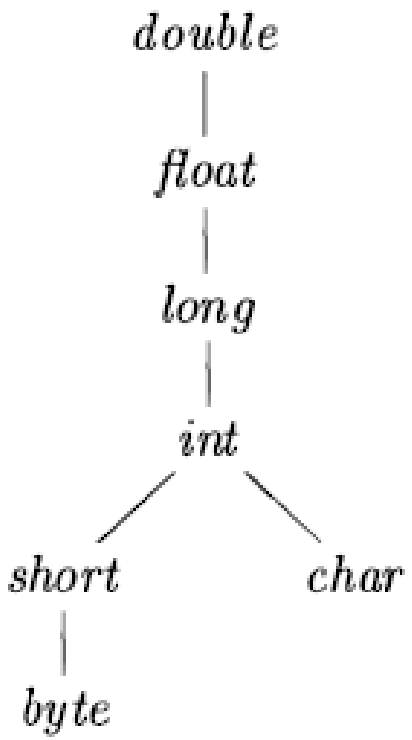
if $f(x)$ is an expression,
then for some α and β, f has type $\alpha \to \beta$ **and** x has type α (6.9)

类型转换

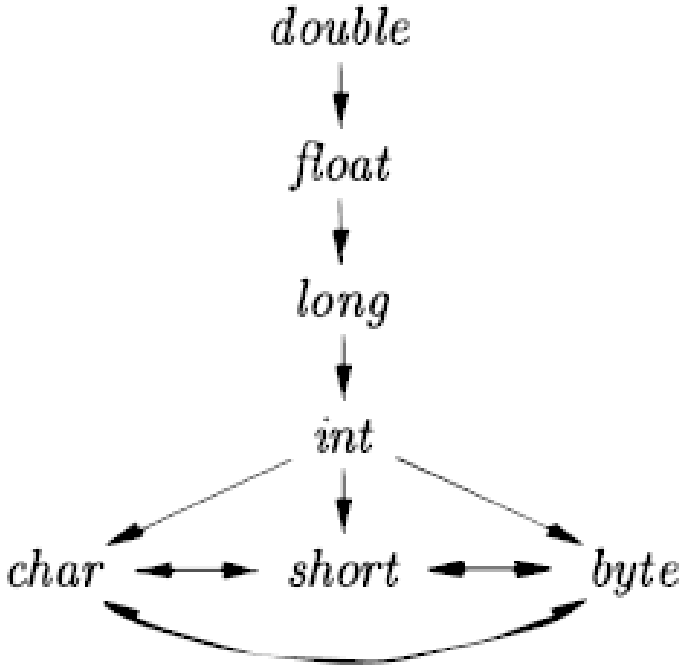
- 类型转换
  - 编译器有时候需要把运算的某个分量进行转换，以保证在进行运算时运算分量具有相同的类型
- 不同语言具有不同的类型转换规则
  - 如Java的拓宽（*widening*）转换和窄化（*narrowing*）转换
- 隐式转换（自动类型转换）
  - 如果类型转换由编译器自动完成，那么这样的转换称为隐式转换（*coercing*）
- 显式转换（强制类型转换）
  - 如果程序员必须写出某些代码来引发类型转换运算，那么这个转换称为显式转换

例……类型转换

- Java中简单类型的转换



(a) Widening conversions



(b) Narrowing conversions

例……类型转换

- 表达式求值中引入的类型转换

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

```
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;
}
```

图 6-26　*widen* 函数的伪代码

控制流的翻译

布尔表达式和控制语句的翻译

布尔表达式

- 布尔表达式在程序设计语言中的作用
  1. 改变控制流
     - 用作语句中改变控制流的条件表达式
     - 这些布尔表达式的值由程序到达的某个位置隐含地指出
  2. 计算逻辑值
     - 布尔表达式的值可以表示true或false
     - 这样的布尔表达式可以像算术表达式一样，使用带有逻辑运算符的三地址指令求值
  - 布尔表达式的使用意图根据上下文确定

布尔表达式

- 生成布尔表达式的文法
  - 布尔表达式由作用于布尔变量或关系表达式的布尔运算符构成

$$B \rightarrow B \,||\, B \mid B \,\&\&\, B \mid !\, B \mid (\, B\,) \mid E \,\mathbf{rel}\, E \mid \mathbf{true} \mid \mathbf{false}$$

短路代码

- 布尔表达式的短路语义
  - 给定表达式$B_1||B_2$，如果B_1为真，则整个表达式为真
  - 给定$B_1\&\&B_2$，如果B_1为假，则整个表达式为假
- 短路（跳转）代码
  - 在短路代码中，布尔运算符&&、||、!被翻译为跳转指令
  - 运算符本身不出现在代码中，布尔表达式的值是通过代码序列中的位置来表示的
  - 例6.14

```
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:
```

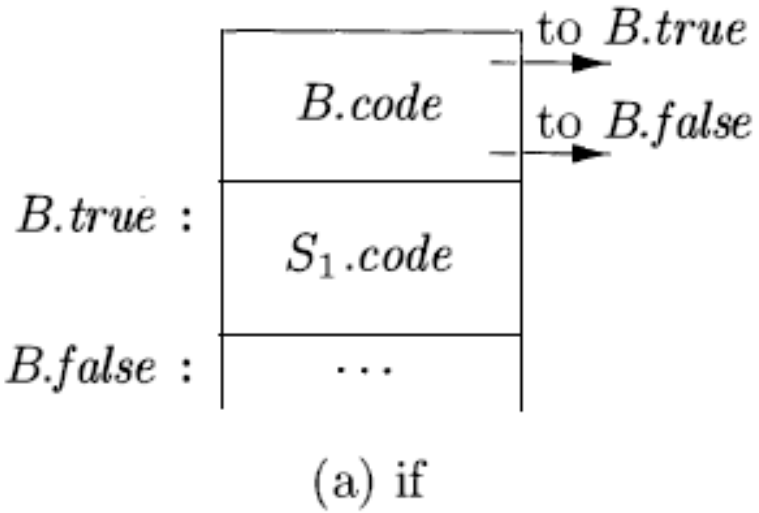
控制流语句

- 布尔表达式作控制语句的条件肘的翻译
  - 考虑在以下文法生成的语句的上下文中，如何把布尔表达式翻译成三地址代码

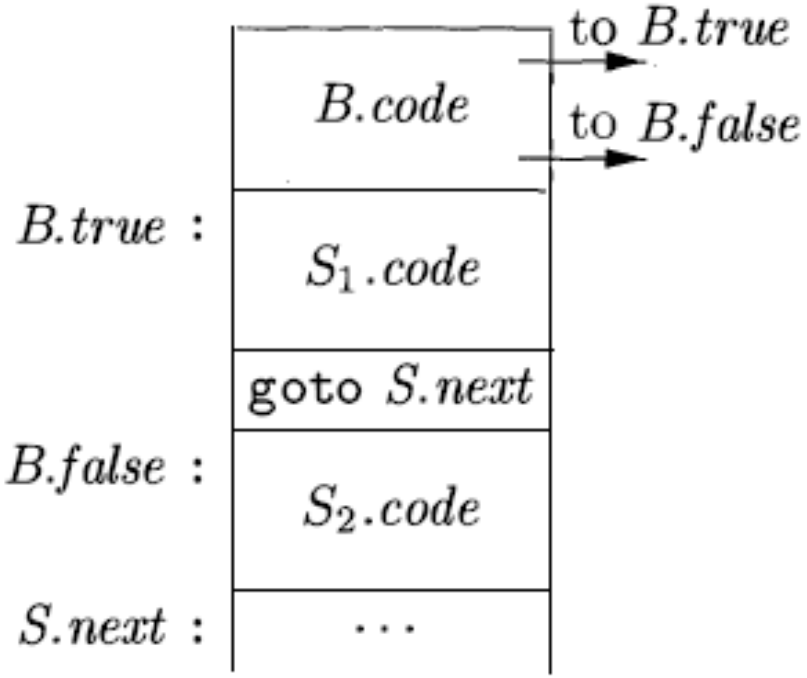
$$
\begin{aligned}
S &\rightarrow \textbf{if}\ (\ B\ )\ S_1 \\
S &\rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2 \\
S &\rightarrow \textbf{while}\ (\ B\ )\ S_1
\end{aligned}
$$

  - 综合属性*code*：给出翻译得到的三地址指令
  - 继承属性*B.true*和*B.false*：表示跳转标号
  - 继承属性*S.next*：紧跟在S代码之后的指令的标号

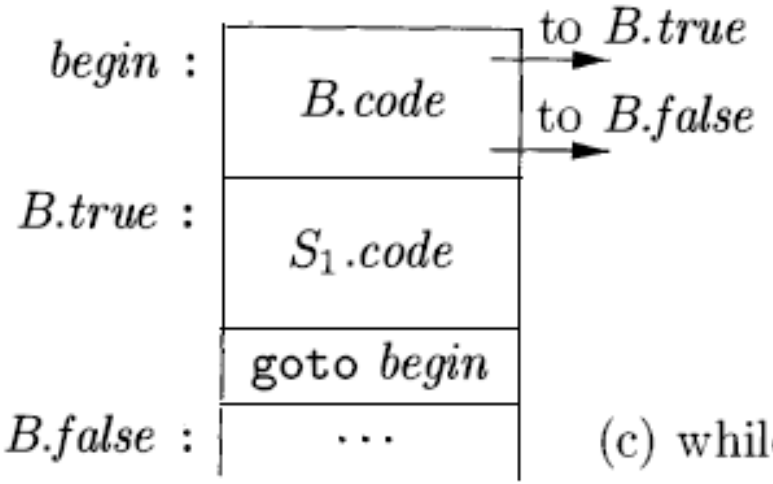
if, if-else, while语句的代码



(a) if



(b) if-else



(c) while

控制流语句的语法制导定义（1）

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

控制流语句的语法制导定义（2）

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

为布尔表达式生成三地址代码（1）

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$

为布尔表达式生成三地址代码（2）

PRODUCTION	SEMANTIC RULES
$B \rightarrow\ !\ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1\ \mathbf{rel}\ E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\text{'if'}\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ \text{'goto'}\ B.true)$ $\mid\mid gen(\text{'goto'}\ B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen(\text{'goto'}\ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen(\text{'goto'}\ B.false)$

表达式的三地址代码

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

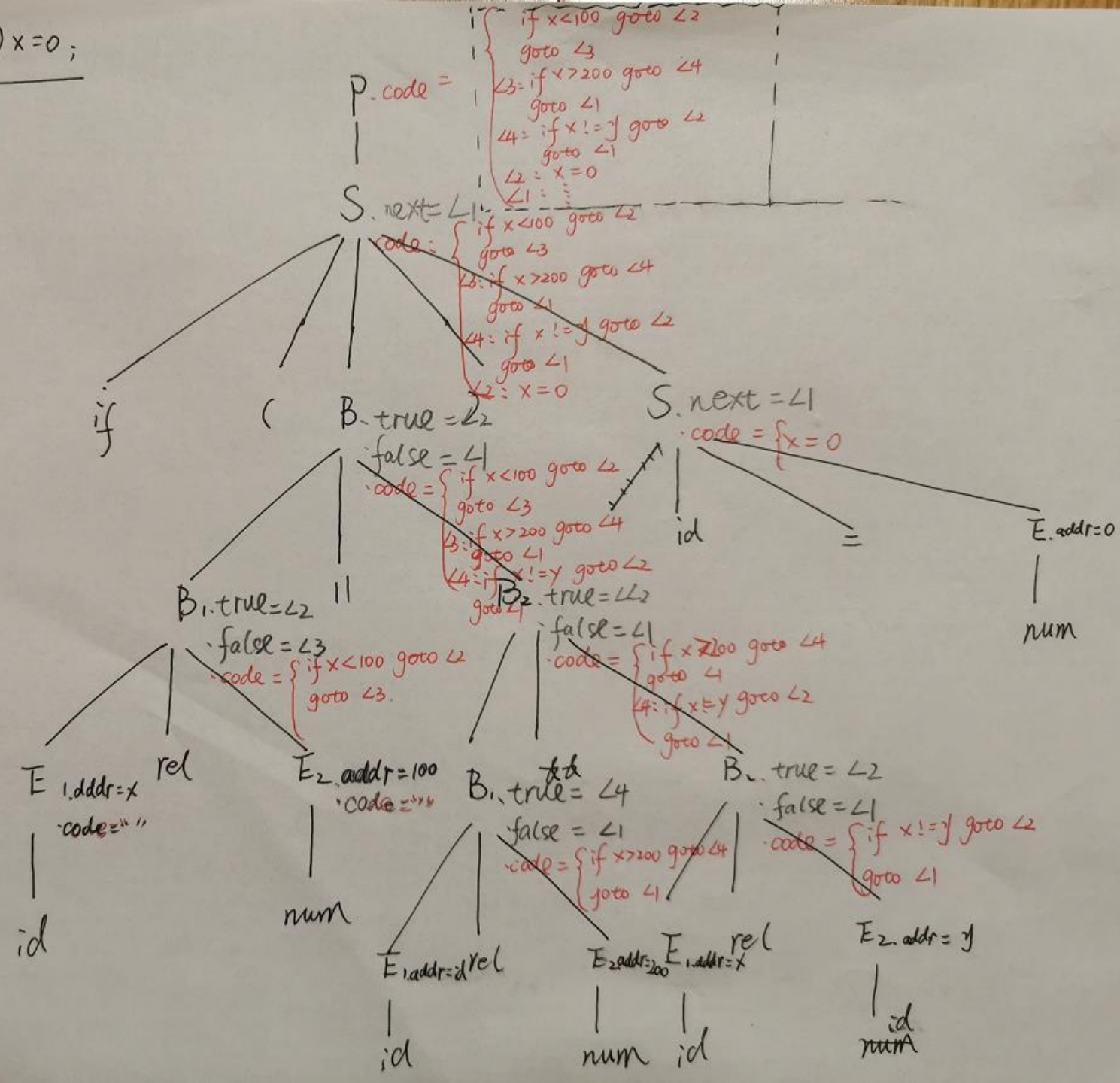
例……if语句的控制流翻译

- 例6.15

```
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:
```

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



避免生成冗余的goto指令

- 冗余goto指令的问题
  - 对比两段指令序列

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

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

  - 第二段中的*ifFalse*指令利用了控制流在指令序列中会从一个指令自然流动到下一个指令的性质，当x>200时，控制流直接穿越到标号L_4，减少了一个跳转指令
- 可以使用特殊标号*fall*，修改控制流语句和布尔表达式翻译的语义规则，支持控制流的直接穿越
  - *fall*表示不要生成任何跳转指令

避免冗余的语义规则

$S \rightarrow \textbf{if}\ (B)\ S_1$

$$
\begin{aligned}
B.true &= fall \\
B.false &= S_1.next\ =\ S.next \\
S.code &= B.code\ ||\ S_1.code
\end{aligned}
$$

$B \rightarrow E_1\ \textbf{rel}\ E_2$

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

$B \rightarrow B_1\ ||\ B_2$

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

例……使用控制流穿越技术翻译的if语句

- 例6.16

```
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:
```

布尔值和跳转代码

- 布尔表达式的两种用途
  1. 改变语句中的控制流
  2. 求出布尔表达式的值
- 处理布尔表达式的方法
  1. 使用两趟处理
     - 为输入构造完整的抽象语法树，然后以深度优先顺序遍历这棵抽象语法树，依据语义规则的描述计算得到翻译结果
  2. 对语句进行一趟处理，但对表达式进行两趟处理
     - 首先翻译语句中的布尔表达式E，再处理语句S_1
     - 翻译E时要建立它的抽象语法树，然后再遍历它，求值

布尔值和跳转代码

- 布尔表达式的文法
  - 非终结符E代表表达式

$$S \rightarrow \mathbf{id} = E\ ; \mid \mathbf{if}\ (\ E\ )\ S \mid \mathbf{while}\ (\ E\ )\ S \mid S\ S$$
$$E \rightarrow E \mid\mid E \mid E\ \&\&\ E \mid E\ \mathbf{rel}\ E \mid E + E \mid (E) \mid \mathbf{id} \mid \mathbf{true} \mid \mathbf{false}$$

  - 例：通过计算临时变量的值翻译一个布尔类型的赋值语句

```
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
```

练习6.6

练习 6.6.1：在图 6-30 的语法制导定义中添加处理下列控制流构造的规则：

1) A repeat-statment **repeat** S **while** B

2) A for-loop **for** $(S_1;\ B;\ S_2)\ S_3$

回填

- 为布尔表达式和控制流语句生成代码时的一个问题
  - 如何将一个跳转指令和该指令的目标匹配起来？
    - 例如，对`if(B)S`中的B的翻译结果中包含一条跳转指令，B为假时，跳转到紧跟在S之后的指令处
    - 问题：在一趟式翻译中，布尔表达式B必须在处理S之前翻译完毕，那么跳过S的*goto*指令的目标是什么？
- 解决方法
  1. 将标号作为继承属性传递到生成相关跳转指令的地方
     - 这种方法要求再进行一趟处理，将标号和具体地址绑定
  2. 回填（*backpatching*）技术
     - 以综合属性的形式传递跳转指令组成的列表，一趟式翻译

回填技术

- 回填技术的思想
  - 生成一个跳转指令时暂时不指定该跳转指令的目标
  - 将这些没有确定目标的指令放入一个跳转指令组成的列表中
  - 等到能够确定正确的目标标号时，才去填充这些指令的目标标号
  - 回填技术可以用来在一趟扫描中完成对布尔表达式或控制流语句的代码生成
    - 生成的代码形式和6.6节中相同，但处理标号的方法不同

回填技术

回填在SDT中的实现

- 非终结符B的综合属性*truelist*和*falselist*管理布尔表达式的跳转代码中的标号
  - *B.truelist*是一个包含跳转或条件跳转指令的列表，必须向这些指令中插入适当的标号，即B为真时控制流应该转向的标号
  - *B.falselist*也是一个包含跳转或条件跳转指令的列表，这些指令中最终插入的标号是B为假时控制流应该转向的标号
  - 生成B的代码时，跳转到真或假出口的跳转指令是不完整的，标号字段尚未填写；这些不完整的跳转指令被保存在*B.truelist*或*B.falselist*指向的列表中
- 语句S的综合属性*S.nexlist*也是一个跳转指令序列，这些指令应该跳转到紧跟在S的代码之后的指令

回填技术的实现

- 指令标号
  - 我们将生成的指令放在一个指令数组中，标号就是这个数组的下标
- 处理跳转指令列表的三个函数
  - *makelist(i)*：创建一个只包含*i*的列表，这里*i*是指令数组的下标；*makelist*返回一个指向新创建的列表的指针
  - *merge(p1, p2)*：合并*p1*和*p2*指向的列表，返回指向合并后的列表的指针
  - *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

布尔表达式的翻译（1）

- 自底向上语法分析过程中为布尔表达式生成目标代码的翻译方案

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

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

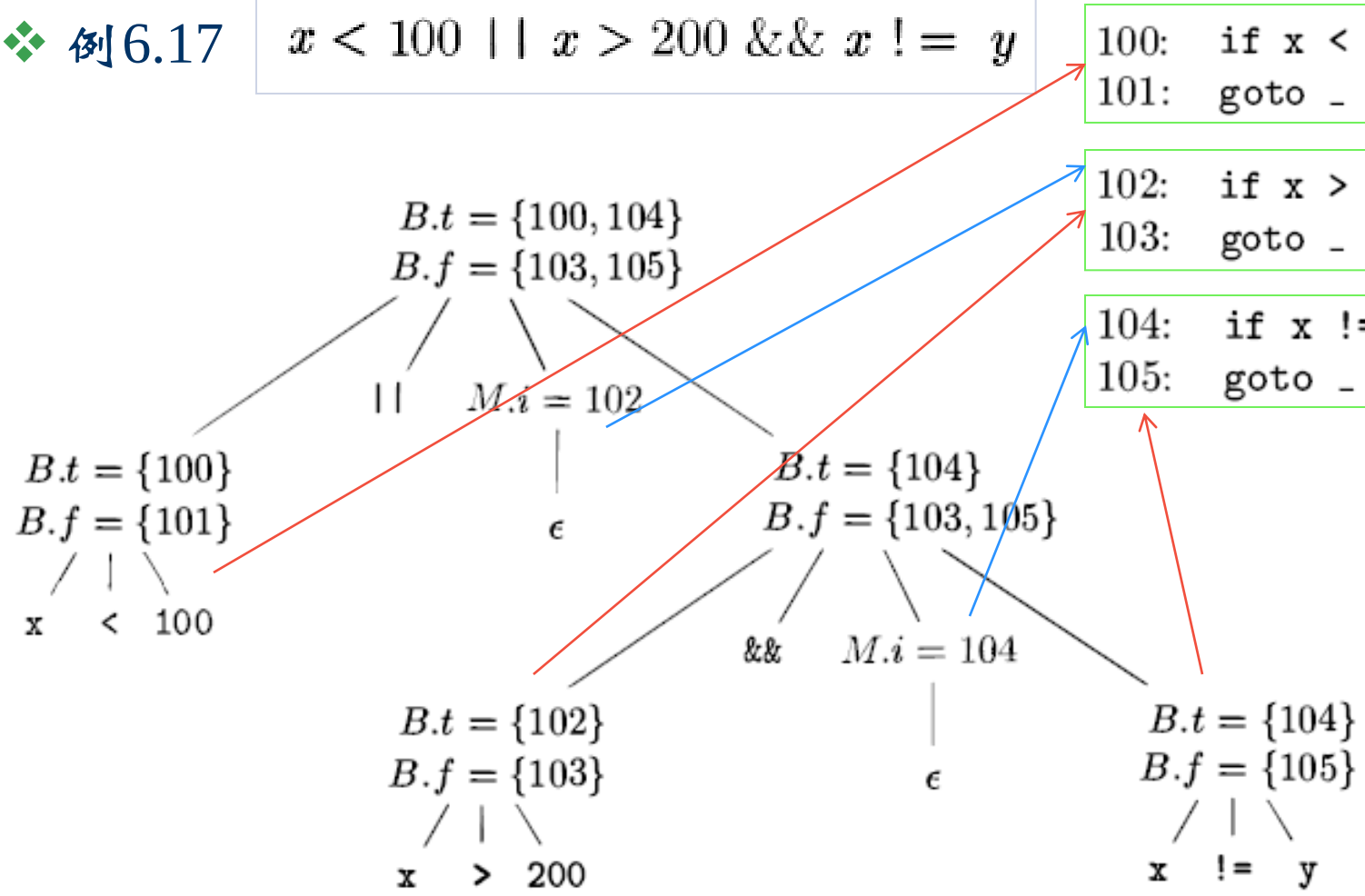
3) $B \rightarrow \ \texttt{!}\ B_1$ { $B.truelist = B_1.falselist$;
$B.falselist = B_1.truelist$; }

布尔表达式的翻译（2）

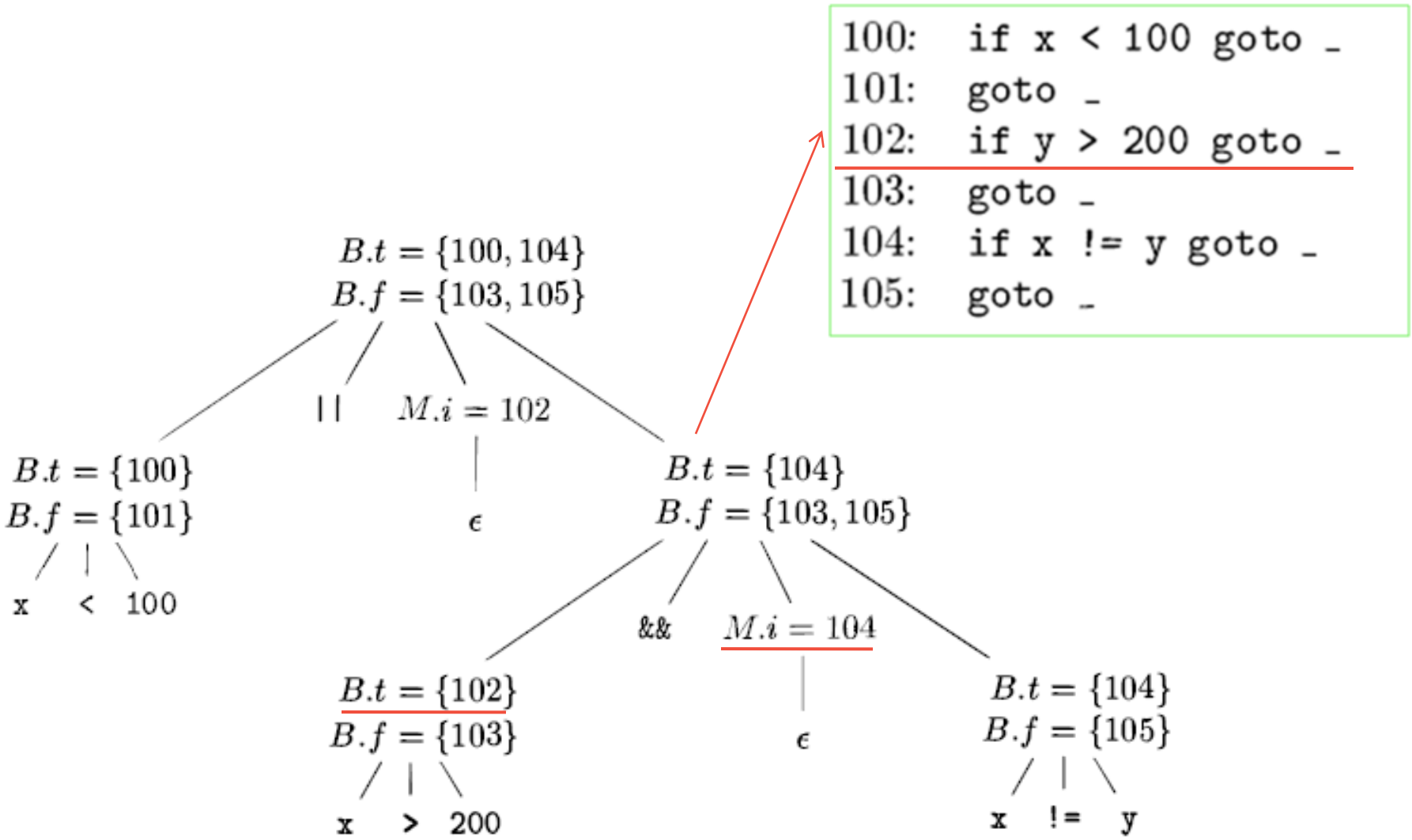
4)	$B \rightarrow (\ B_1\ )$	$\{\ B.truelist = B_1.truelist;$ $B.falselist\ =\ B_1.falselist;\ \}$
5)	$B \rightarrow E_1\ \mathbf{rel}\ E_2$	$\{\ B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr+1);$ $emit(\texttt{'if'}\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ \texttt{'goto\ \_'});$ $emit(\texttt{'goto\ \_'});\ \}$
6)	$B \rightarrow \mathbf{true}$	$\{\ B.truelist = makelist(nextinstr);$ $emit(\texttt{'goto\ \_'});\ \}$
7)	$B \rightarrow \mathbf{false}$	$\{\ B.falselist = makelist(nextinstr);$ $emit(\texttt{'goto\ \_'});\ \}$
8)	$M \rightarrow \epsilon$	$\{\ M.instr = nextinstr;\ \}$

例……回填（1）

❖ 例6.17 $x < 100 \;||\; x > 200 \;\&\&\; x != y$

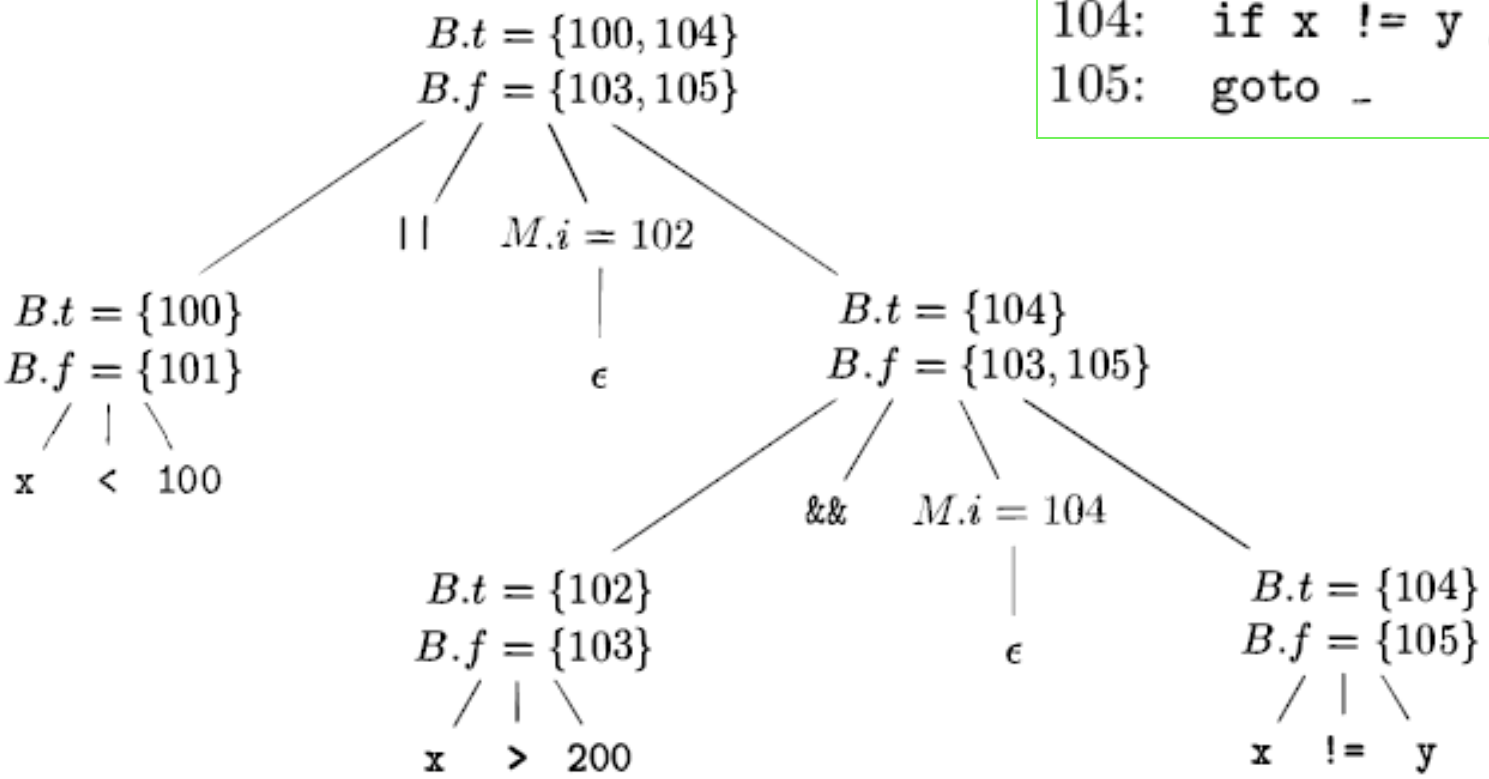


例……回填（2）

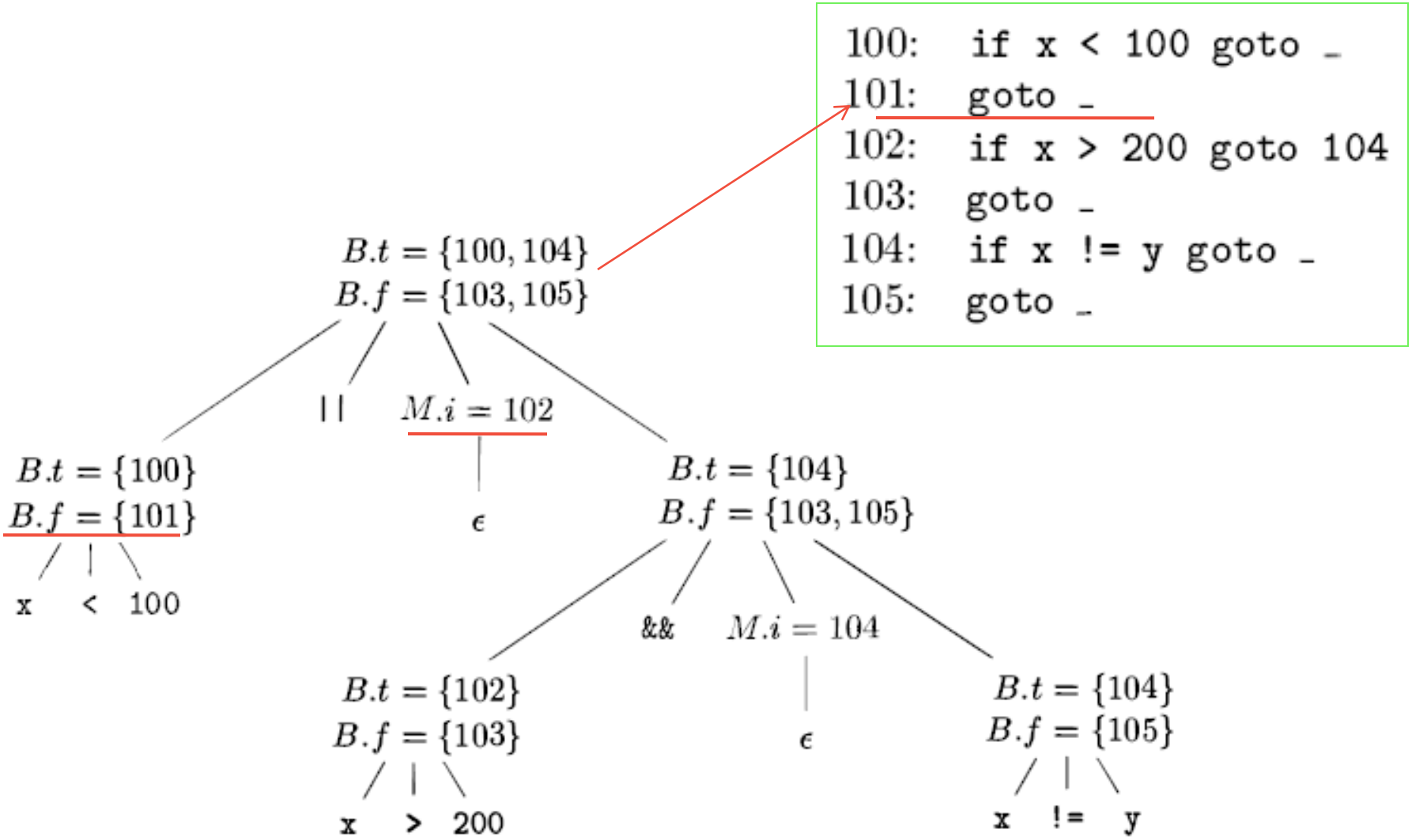

100: if x < 100 goto _
101: goto _
102: if y > 200 goto _
103: goto _
104: if x != y goto _
105: goto _
B.t = {100, 104}
B.f = {103, 105}
||
M.i = 102
B.t = {100}
B.f = {101}
x < 100
ϵ
B.t = {104}
B.f = {103, 105}
&&
M.i = 104
B.t = {102}
B.f = {103}
x > 200
ϵ
B.t = {104}
B.f = {105}
x != y

例……回填（3）

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

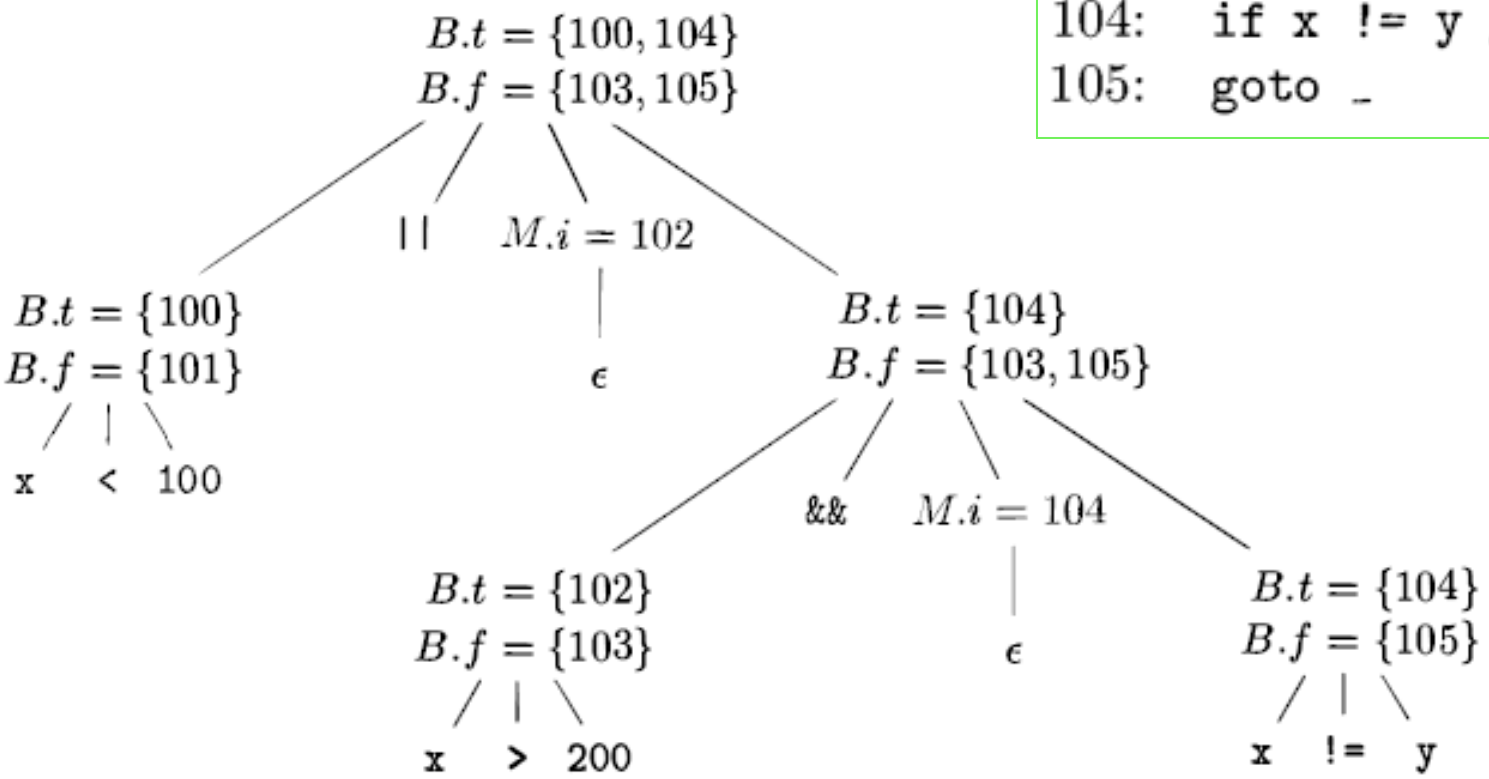


例……回填（4）


100: if x < 100 goto _
101: goto _
102: if x > 200 goto 104
103: goto _
104: if x != y goto _
105: goto _
B.t = {100, 104}
B.f = {103, 105}
||
M.i = 102
B.t = {100}
B.f = {101}
x < 100
ϵ
B.t = {104}
B.f = {103, 105}
&&
M.i = 104
B.t = {102}
B.f = {103}
x > 200
ϵ
B.t = {104}
B.f = {105}
x != y

例……回填（5）

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



控制转移语句的翻译（1）

- 使用回填技术在一趟扫描中完成控制流语句的翻译
  - 文法

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

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

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

控制转移语句的翻译（2）

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

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

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

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

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

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

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

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

例……控制转移语句的翻译

- 将下面的C语言程序段翻译为三地址码：

```
while(c > min && c < max)
{
        if(x < y)
            x = y - a;
        else
            y = x + b;
        c = x + y;
}
```

例……控制转移语句的翻译(1)

(1) if c> min goto ____

(2) goto ____

(3)

```
while(c > min && c < max)
{
        if(x < y)
            x = y - a;
        else
            y = x + b;
        c = x + y;
}
```

例……控制转移语句的翻译(2)

```
while(c > min && c < max)
{
        if(x < y)
            x = y - a;
        else
            y = x + b;
        c = x + y;
}
```

(1) if c> min goto 3

(2) goto ____

(3) if c<max goto ____

(4) goto ____

(5)

例……控制转移语句的翻译(3)

```
while(c > min && c < max)
{
        if(x < y)
            x = y - a;
        else
            y = x + b;
        c = x + y;
}
```

(1) if c> min goto 3
(2) goto ____
(3) if c<max goto 5
(4) goto ____
(5) if x<y goto ____
(6) goto ____
(7)

例……控制转移语句的翻译(4)

```
while(c > min && c < max)
{
        if(x < y)
            x = y - a;
        else
            y = x + b;
        c = x + y;
}
```

(1) if c> min goto 3
(2) goto ____
(3) if c<max goto 5
(4) goto ____
(5) if x<y goto 7
(6) goto ____
(7) t1 = y – a
(8) x = t1
(9) goto ____
(10)

例……控制转移语句的翻译(5)

```
while(c > min && c < max)
{
        if(x < y)
            x = y – a;
        else
            y = x + b;
        c = x + y;
}
```

(1) if c> min goto 3
(2) goto ____
(3) if c<max goto 5
(4) goto ____
(5) if x<y goto 7
(6) goto 10
(7) t1 = y – a
(8) x = t1
(9) goto ____
(10) t2 = x + b
(11) y = t2
(12)

例……控制转移语句的翻译(6)

```
while(c > min && c < max)
{
        if(x < y)
            x = y – a;
        else
            y = x + b;
        c = x + y;
}
```

(1) if c> min goto 3
(2) goto ____
(3) if c<max goto 5
(4) goto ____
(5) if x<y goto 7
(6) goto 10
(7) t1 = y – a
(8) x = t1
(9) goto 12
(10) t2 = x + b
(11) y = t2
(12) t3 = x + y
(13) c = t3
(14)

例……控制转移语句的翻译(7)

```
while(c > min && c < max)
{
        if(x < y)
            x = y – a;
        else
            y = x + b;
        c = x + y;
}
```

(1) if c> min goto 3
(2) goto ____
(3) if c<max goto 5
(4) goto ____
(5) if x<y goto 7
(6) goto 10
(7) t1 = y – a
(8) x = t1
(9) goto 12
(10) t2 = x + b
(11) y = t2
(12) t3 = x + y
(13) c = t3
(14) goto 1
(15)

例……控制转移语句的翻译(8)

```
while(c > min && c < max)
{
        if(x < y)
            x = y – a;
        else
            y = x + b;
        c = x + y;
}
```

(1) if c> min goto 3
(2) goto 15
(3) if c<max goto 5
(4) goto 15
(5) if x<y goto 7
(6) goto 10
(7) t1 = y – a
(8) x = t1
(9) goto 12
(10) t2 = x + b
(11) y = t2
(12) t3 = x + y
(13) c = t3
(14) goto 1
(15)

break、continue和goto

- goto语句
  - 是改变程序控制流的基本语言结构
- break语句
  - 使控制流跳出外围的语言结构
  - 为break生成代码的步骤
    1. 跟踪外围循环语句S
    2. 为该break语句生成未完成的跳转指令
    3. 将这些指令放到S.nextlist中
- continue语句
  - 和break的处理类似，区别在于生成的跳转指令的目标不同

练习6.7(作业)

- 练习6.7.1：使用图6-37中的翻译方案翻译下列表达式

  1) `a==b && (c==d || e==f)`

  2) `(a==b || c==d) || e==f`

  3) `(a==b && c==d) && e==f`

- 练习6.7.2：使用图6-40中的翻译方案翻译下列代码

  1)
```
while (a>b && a>c)
    if (a>0) a = a-2;
    else a = a-1;
```

  2)
```
if (a>b) while (a>0) a = a-2;  (课堂练习)
    else if (a<d) while(a!>0) a = a+1;
```

switch语句

- *switch*语句的语法

```
switch ( E ) {
        case V1: S1
        case V2: S2
                 ...
        case Vn−1: Sn−1
        default: Sn
}
```

- *switch*语句的翻译结果是完成如下工作的代码
  1. 计算表达式E的值
  2. 在*case*列表中寻找与表达式值相同的值V_j；当在*case*列表中明确列出的值都不和表达式匹配时，用*default*和表达式匹配
  3. 执行和匹配值关联的语句S_j

switch语句的语法制导翻译（1）

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

switch语句的语法制导翻译（2）

```
        code to evaluate E into t
        if t != V1 goto L1
        code for S1
        goto next
L1:     if t != V2 goto L2
        code for S2
        goto next
L2:
        ...
Ln-2:   if t != Vn-1 goto Ln-1
        code for Sn-1
        goto next
Ln-1:   code for Sn
next:
```

过程的中间代码

- 在源语言中加入函数

$$
\begin{array}{lcl}
D & \rightarrow & \textbf{define}\ T\ \textbf{id}\ (\ F\ )\ \{\ S\ \} \\
F & \rightarrow & \epsilon \mid T\ \textbf{id}\ ,\ F \\
S & \rightarrow & \textbf{return}\ E\ ; \\
E & \rightarrow & \textbf{id}\ (\ A\ ) \\
A & \rightarrow & \epsilon \mid E\ ,\ A
\end{array}
$$

- 例6.18 包含函数调用的赋值语句的翻译

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

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

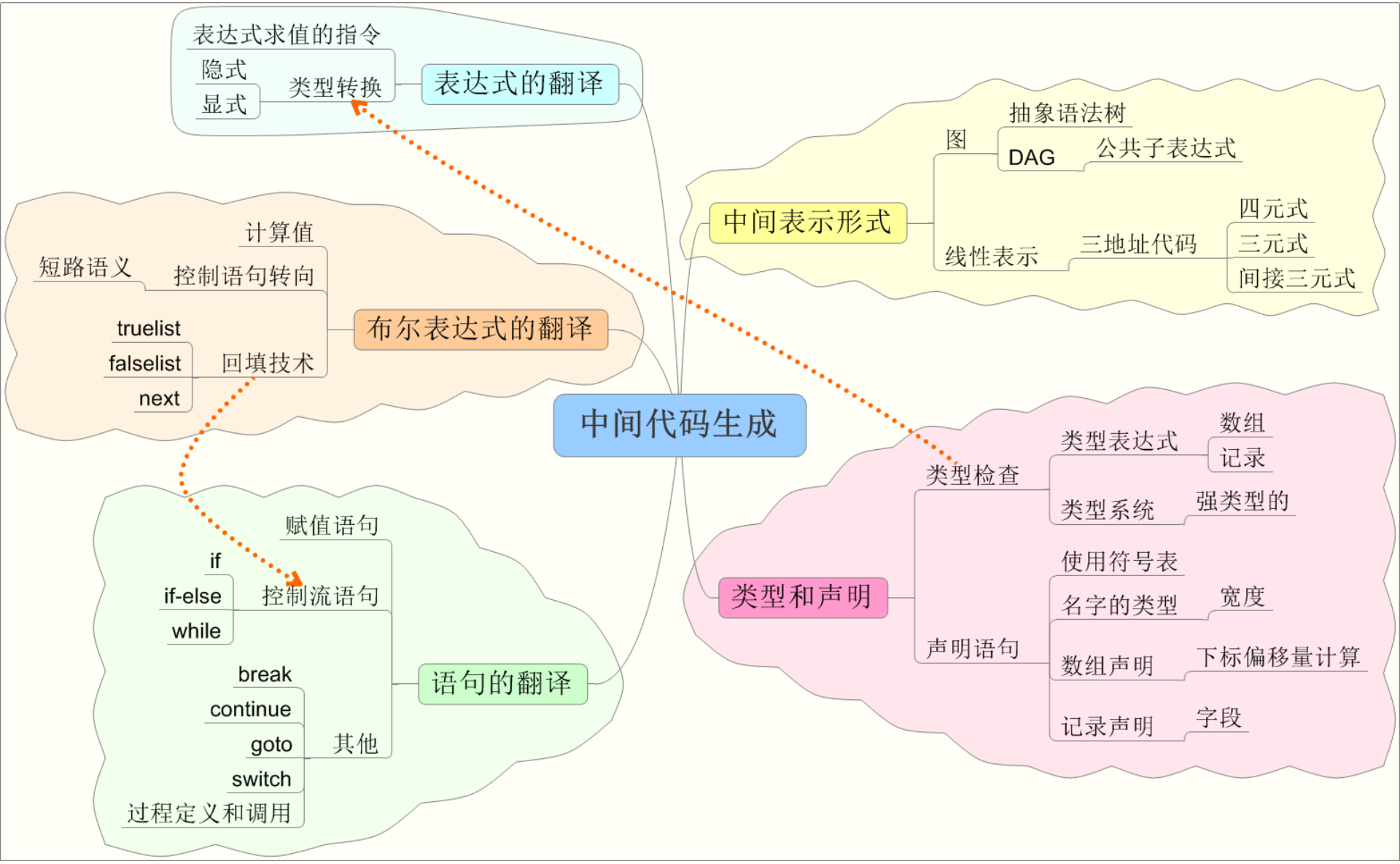
函数定义和函数调用的翻译

- 函数类型
  - 函数类型包含它的返回值类型和形式参数类型
- 符号表
  - 处理函数定义时，函数名被放入最上层的符号表s；
  - 看到*define*关键字和函数名之后，将s压栈并建立新的符号表t；t可以被链接到s；
  - t用于这个函数的函数体的翻译；
  - 翻译完成后，恢复之前的符号表s；
  - 函数的形参用类似于记录字段名的方式处理；

函数定义和函数调用的翻译

- 类型检查
  - 在表达式中，函数和运算符的处理方法相同
  - 6.5节讨论的类型检查规则（包括自动类型转换）仍然可用
- 函数调用
  - 为函数调用*id(E, E,..., E)*生成三地址指令时，只要生成对各个参数E求值的三地址指令，或者生成将各个参数E归约为地址的三地址指令；然后再为每个参数生成一条*param*指令即可
  - 也可以将每个表达式的属性E.addr放到一个数据结构（如队列）中，一旦所有表达式翻译完成，可以在情况队列的同时生成*param*指令

本章小结


中间代码生成
表达式的翻译
表达式求值的指令
类型转换
隐式
显式
布尔表达式的翻译
计算值
控制语句转向
短路语义
回填技术
truelist
falselist
next
语句的翻译
赋值语句
控制流语句
if
if-else
while
其他
break
continue
goto
switch
过程定义和调用
中间表示形式
图
抽象语法树
DAG
公共子表达式
线性表示
三地址代码
四元式
三元式
间接三元式
类型和声明
类型检查
类型表达式
数组
记录
类型系统
强类型的
声明语句
使用符号表
名字的类型
宽度
数组声明
下标偏移量计算
记录声明
字段