



中国科学技术大学
University of Science and Technology of China

中间语言与中间代码生成 II

《编译原理和技术(H)》、《编译原理(H)》

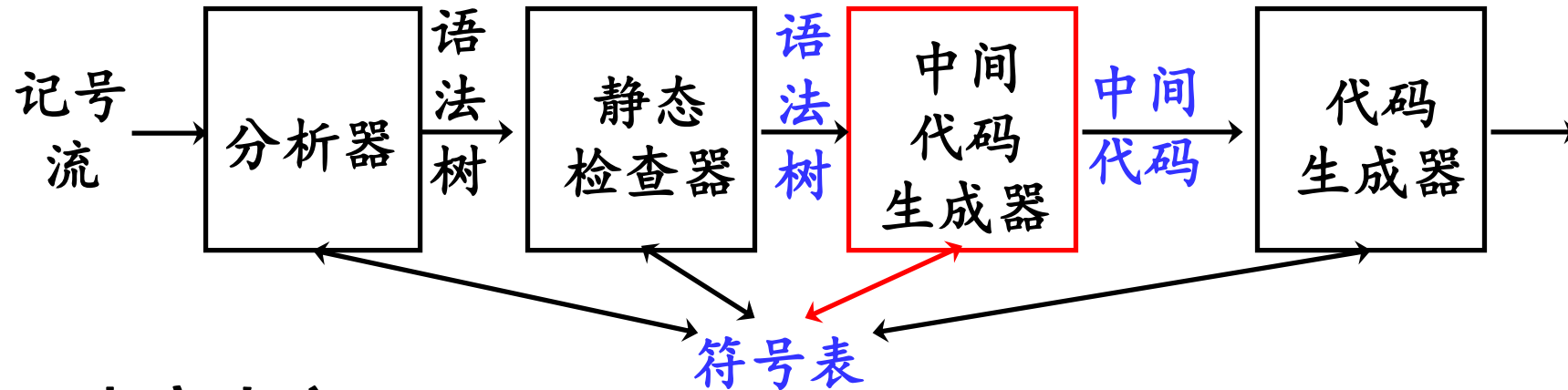
张昱

0551-63603804, yuzhang@ustc.edu.cn

中国科学技术大学
计算机科学与技术学院



本章内容



本章内容

- 中间语言：常用的中间表示 (Intermediate Representation)
 - 后缀表示、图表示、三地址代码、[LLVM IR](#)
- 基本块和控制流图
- 中间代码的生成
 - 声明语句 (=>更新符号表)
 - 表达式、赋值语句 (=>产生临时变量、查符号表)
 - 布尔表达式、控制流语句 (=>标号/回填技术、短路计算)



7.4 中间代码生成概述

- 方法和关键问题
- 名字与作用域
- 符号表结构的变化



中间代码生成的方法

□ 边解析边生成中间代码

- 语法制导的翻译方案
- 难点：理解分析器的运转机制、继承属性的处理

□ 基于树访问的中间代码生成

■ 重点

树结构的设计、访问者模式、节点类的 **visit/enter/exit** 接口及实现

本节将以基于树访问的中间代码生成方法为主来讲解，
这是现代编译器使用的主流方法。



中间代码生成的关键问题

假设采取的中间语言类似三地址代码

□ 类型与符号表的变化

- 多样化类型 \Rightarrow 整型(字节、字)、浮点型、类型符号表
- 1个某类型的数据 \Rightarrow m 个字节(m 为类型对应的字宽)

□ 语句的翻译

- 声明语句：不生成指令，但会更新符号表（作用域，字宽及存放的相对地址）
- 赋值语句：引入临时变量、数组/记录元素的地址计算、类型转换
- 控制流语句：跳转目标的确定(引入标号或者使用回填技术)、短路计算



符号表的设计

□ 类型检查后的符号表

- 符号表条目：（标识符、存储类别、类型信息）
- 存储类别：extern, static, register, ...
- 类型信息：（类别标识, 该类别关联的其他信息）
 - 如数组(Array, (len, elemtype))

□ 本章符号表的变化

- 作用域 => 多个符号表
- 变量：字宽、存储的相对地址（以字节为单位）
- 记录类型：用符号表管理各个成员的字宽、相对地址



5. 声明语句

- ☐ 分配存储单元，更新符号表
- ☐ 作用域的管理
- ☐ 记录类型的管理



声明语句的翻译

□ 主要任务

- 为局部名字分配存储单元

符号表条目：名字、类型、字宽、偏移

- 作用域信息的保存

- 记录类型的管理

不产生中间代码指令，但是要更新符号表



块中无变量声明时的翻译

计算被声明名字的类型和**相对地址**

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

$D \rightarrow D ; D$

相对地址初始化为0

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

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

更新符号表信息

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

类型=>字宽

$T \rightarrow array [num] of T_1$

$\{T.type = array (num.val, T_1.type); T.width = num.val \times T_1.width\}$

$T \rightarrow \uparrow T_1 \quad \{T.type = pointer (T_1.type); T.width = 4 \}$



仅有主过程时的翻译

基于树访问的代码生成

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

$D \rightarrow D; D$

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

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

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

$T \rightarrow array [num] \text{ of } T_1$

$\{T.type = array (num.val, T_1.type);$

$T.width = num.val \times T_1.width \}$

$T \rightarrow \uparrow T_1 \quad \{T.type = pointer (T_1.type); T.width = 4 \}$

enterP时处理

visitD时处理
(只有访问D时才
知道D是哪种结构)

~~exitD~~时处理

~~exitT~~时处理

visitT时处理
(只有访问T时才知
道T是哪种结构)



允许自定义过程时的翻译

□ 所讨论的语言的文法

$$P \rightarrow D; S$$
$$D \rightarrow D ; D / \text{id} : T /$$
$$\text{proc id} ; D ; S$$

□ 管理作用域

- 每个过程内声明的符号要置于该过程的符号表中
- 方便地找到子过程和父过程对应的符号表

sort

var a:....; x:....;

readarray

var i:....;

exchange

quicksort

var k, v:....;

partition

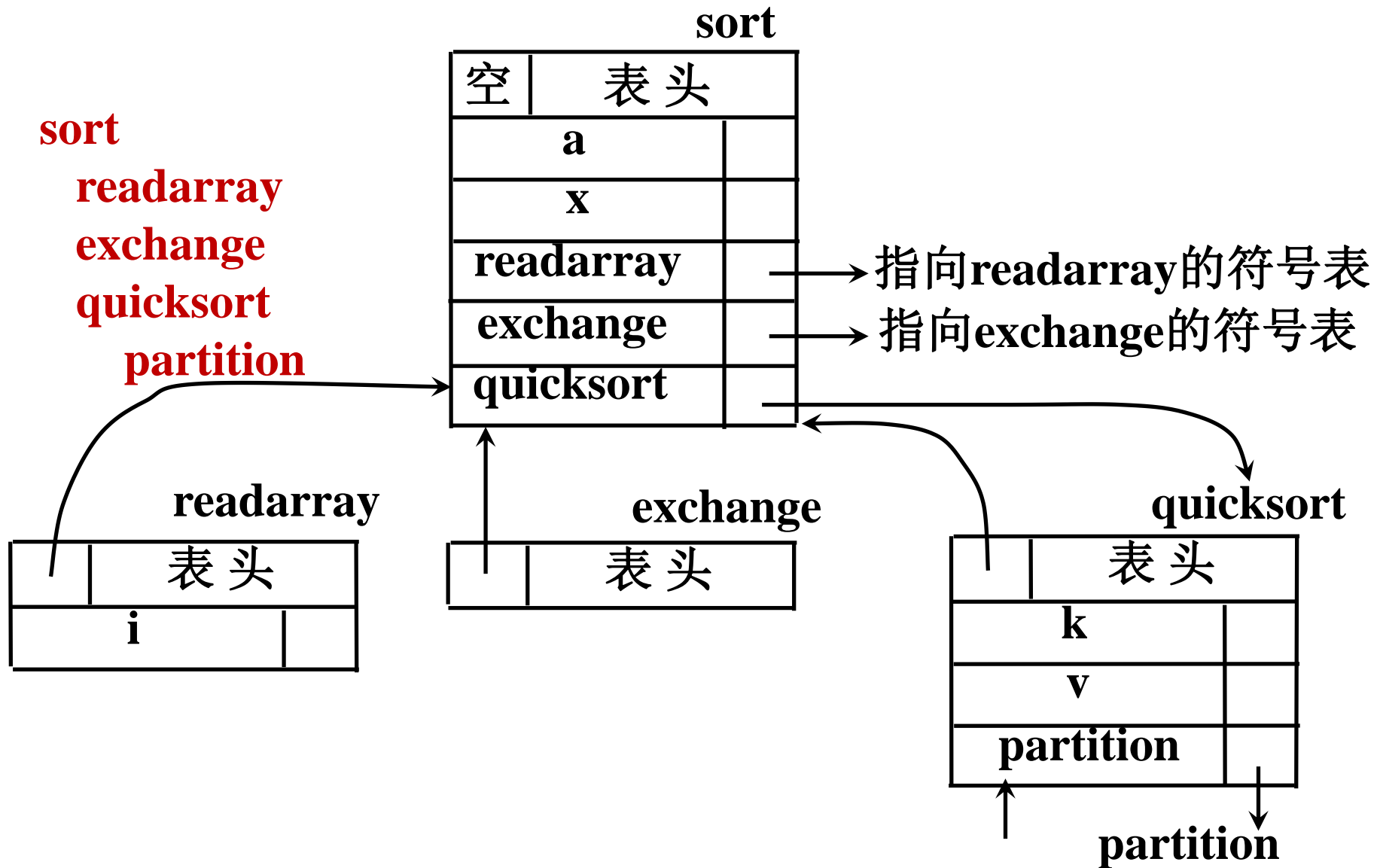
var i, j:....;

P186,图6.14

(过程参数被略去)



各过程的符号表





符号表的组织与管理

□ 相关的数据结构设计

- 符号表：哈希表
- 符号表之间的连接(双向链)
 - 父→子：过程中包含哪些子过程定义：
 - 子→父：分析完子过程后继续分析父过程
- 一遍分析时，需要维护符号表栈

□ 本章使用的符号表相关的函数

mkTable(previous)
enter(table, name, type, offset)
addWidth(table, width)
enterProc(table, name, newtable)

sort

var a:....; x:....;

readarray

var i:....;

exchange

quicksort

var k, v:....;

partition

var i, j:....;

P186,图6.14

(过程参数被略去)



声明语句的处理

$P \rightarrow D; S$

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

proc id ; $D ; S$

tblptr: 符号表栈

offset: 偏移量栈

enterP: $t = \text{mkTable}(\text{nil}); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset})$

visitD:

1) **id : T**: 更新符号表中id对应的条目

$\text{enter}(\text{top}(\text{tblptr}), \text{id.lexeme}, T.\text{type}, \text{top}(\text{offset}));$

$\text{top}(\text{offset}) = \text{top}(\text{offset}) + T.\text{width}$



声明语句的处理

$P \rightarrow D; S$

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

proc $\text{id} ; D ; S$

tblptr: 符号表栈

offset: 偏移量栈

enterP: $t = \text{mkTable}(\text{nil}); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset})$

visitD:

1) **id : T**: 更新符号表中**id**对应的条目

2) **proc id ; D ; S**:

访问**D**前: 新建该过程的符号表, 进入该过程的作用域

$t = \text{mkTable}(\text{top}(\text{tblptr})); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset})$



声明语句的处理

$P \rightarrow D; S$

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

$\text{proc id} ; D ; S$

tblptr: 符号表栈

offset: 偏移量栈

enterP: $t = \text{mkTable}(\text{nil}); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset})$

visitD:

1) **id : T**: 更新符号表中id对应的条目

2) **proc id ; D ; S**:

访问D前: 新建该过程的符号表, 进入该过程的作用域

访问S后: 将该过程符号信息插入到父符号表, 退出作用域

$t = \text{top}(\text{tblptr}); \text{addWidth}(t, \text{top}(\text{offset}));$

$\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \text{enterProc}(\text{top}(\text{tblptr}), \text{id.lexeme}, t)$

如果S中存在对该过程的
递归调用,则在分析S前
将该过程名插入符号表



声明语句的处理

$P \rightarrow D; S$

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

proc **id** ; D ; S

tblptr: 符号表栈

offset: 偏移量栈

enterP: $t = mkTable (nil); push(t, tblptr); push (0, offset)$

visitD:

1) **id : T**: 更新符号表中**id**对应的条目

2) **proc id ; D ; S**:

exitP:

$addWidth (top (tblptr), top (offset)); pop(tblptr); pop (offset)$



记录的域名管理

□ 关联的文法

$T \rightarrow \text{record } D \text{ end}$

记录类型单独建符号表(类型表达式),域相对地址从0开始

visit T : **record** D **end**

访问 D **之前**: 建立符号表, 进入作用域

$t = \text{mkTable}(\text{nil}); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset})$

结尾: 设置记录的类型表达式和宽度, 退出作用域

$T.\text{type} = \text{record } (\text{top}(\text{tblptr}));$

$T.\text{width} = \text{top}(\text{offset}); \text{pop}(\text{tblptr}); \text{pop}(\text{offset})$



6. 赋值语句

- 分配临时变量，存储表达式计算的中间结果
- 数组元素的地址计算
- 类型转换



赋值语句的翻译

□ 主要任务

- 复杂的表达式 \Rightarrow 多条计算指令组成的序列
- 分配临时变量保存中间结果
- id: 查符号表获得其存储的场所
- 数组元素: 元素地址计算
 - 符号表中保存数组的基址和用于地址计算的常量表达式的值
 - 数组元素在中间代码指令中表示为“基址[偏移]”
- 可以进行一些语义检查
 - 类型检查、变量未定义/重复定义/未初始化
- 类型转换: 因为目标机器的运算指令是区分类型的



赋值语句的中间代码生成

□ 关联的文法

$S \rightarrow \text{id} := E$ $E \rightarrow E_1 + E_2 / -E_1 / (E_1) \mid \text{id}$

visitS: **id** := **E**

结尾: 获取id的地址和存放E结果的场所, 发射赋值指令

$p = \text{lookup}(\text{id.lexeme});$

if $p \neq \text{nil}$ then **emit** (p , '=', $E.place$) else **error**

visitE:

E → **E₁** + **E₂** **结尾**: 发射加法指令

$E.place = \text{newTemp}();$

emit ($E.place$, '=', $E_1.place$, '+', $E_2.place$)



赋值语句的中间代码生成

□ 关联的文法

$S \rightarrow \text{id} := E$ $E \rightarrow E_1 + E_2 / -E_1 / (E_1) \mid \text{id}$

visitE:

$E \rightarrow E_1 + E_2$ 结尾: 发射加法指令

$E \rightarrow -E_1$ 结尾: 发射负号运算指令

$E.place = newTemp();$

$emit(E.place, '=', uminus, E_1.place)$

$E \rightarrow (E_1)$ 结尾: $E.place = E_1.place;$

$E \rightarrow \text{id}$ 结尾: 获取id的地址并作为E的场所

$p = lookup(id.lexeme);$

if $p \neq nil$ then $E.place = p$ else error



数组元素的地址计算

□ 一维数组元素的地址计算

A的第*i*个元素的地址: $base + (i - low) \times w$

变换成: $i \times w + (base - low \times w)$

$low \times w$ 是常量, 编译时计算, 减少运行时计算

□ 二维数组元素的地址计算

■ 列为主序(列优先)? 行为主序?

行为主序时: $base + ((i_1 - low_1) \times n_2 + (i_2 - low_2)) \times w$

($A[i_1, i_2]$ 的地址, 其中 $n_2 = high_2 - low_2 + 1$)

变换成: $((i_1 \times n_2) + i_2) \times w +$
 $(base - (low_1 \times n_2 + low_2) \times w)$



数组元素的地址计算

□ 多维数组元素的地址计算

■ 以行为主序

下标变量 $A[i_1, i_2, \dots, i_k]$ 的地址表达式

$$\begin{aligned} & ((\dots ((i_1 \times n_2 + i_2) \times n_3 + i_3) \dots) \times n_k + i_k) \times w \\ & + \text{base} - ((\dots ((low_1 \times n_2 + low_2) \times n_3 + low_3) \dots) \times n_k + low_k) \times w \end{aligned}$$

□ 翻译的主要任务

■ 发射地址计算的指令

■ “基址[偏移]” 相关的中间指令： $t = b[o]$, $b[o] = t$



数组元素的访问处理

□ 关联的文法

$$S \rightarrow L := E \qquad L \rightarrow \text{id} [Elist] \mid \text{id}$$
$$Elist \rightarrow Elist, E \mid E \qquad E \rightarrow L \mid \dots$$

□ 采用语法制导的翻译方案时存在的问题

$Elist \rightarrow Elist, E \mid E$ 由 $Elist$ 的结构只能得到各维的下标值，但无法获得数组的信息（如各维的长度）

需要改写文法为： $L \rightarrow Elist] \mid \text{id}$ $Elist \rightarrow \text{id} [E / Elist, E$

$Elist \rightarrow \text{id} [E$ 由这个定义可以获得数组的信息，并从左到右传播下去，达到边分析边计算的目的



数组元素的访问处理

□ 关联的文法

基于树来生成会简单多了，
不用改写文法

$S \rightarrow L := E$ $L \rightarrow \text{id} [Elist] \mid \text{id}$ $Elist \rightarrow Elist, E \mid E$

visitL: $L \rightarrow \text{id} [E_1, E_2, \dots, E_n]$

访问 E_1 之后: $ndim = 1$; $place = E_1.place$; // 局部变量

每次访问 E_i 之后计算: $t = newTemp()$; $ndim ++$;

$emit(t, '=', place, '*', limit(id.place, ndim))$;

$emit(t, '=', t, '+', E_i.place)$; $place = t$;

结尾: $L.place = newTemp()$;

$emit(L.place, '=', base(id.place), '-', invariant(id.place))$;

$L.offset = newTemp()$;

$emit(L.offset, '=', place, '*', width(id.place))$;



数组元素的访问处理

□ 关联的文法

$$S \rightarrow L := E \quad E \rightarrow L$$

visitE: $E \rightarrow L$

结尾: if ($L.offset == \text{null}$) /* 简单变量 */ $E.place = L.place$
else { $E.place = \text{newTemp}()$;
 $\text{emit}(E.place, '=', L.place, '[', L.offset, ']');$ }

visitS: $S \rightarrow L := E$

结尾: if ($L.offset == \text{null}$) $\text{emit}(L.place, '=', E.place);$
else $\text{emit}(L.place, '[', L.offset, ']', '=', E.place);$



类型转换

例 $x = y + i * j$
(x 和 y 的类型是real, i 和 j 的类型是integer)

中间代码

$t_1 = i \text{ int} \times j$

$t_2 = \text{intto real } t_1$

$t_3 = y \text{ real} + t_2$

$x = t_3$

目标机器的运算指令是区分整型和浮点型的

高级语言中的重载算符 \Rightarrow 中间语言中的多种具体算符



类型转换的处理

□ 以 $E \rightarrow E_1 + E_2$ 为例说明

visitE: $E \rightarrow E_1 + E_2$

结尾：判断 E_1 和 E_2 的类型，看是否要进行类型转换；若需要，则分配存放转换结果的临时变量并发射类型转换指令

$E.place = newTemp();$

if ($E_1.type == integer \ \&\& \ E_2.type == integer$) {
 $emit(E.place, '=', E_1.place, 'int+', E_2.place);$

$E.type = integer;$

} else if ($E_1.type == integer \ \&\& \ E_2.type == real$) {
 $u = newTemp();$ $emit(u, '=', 'inttoreal', E_1.place);$
 $emit(E.place, '=', u, 'real+', E_2.place);$ $E.type = real;$
}



7. 布尔表达式和控制流语句

- ☐ 布尔表达式：短路计算
- ☐ 控制流语句的翻译：标号、回填技术
- ☐ switch的翻译优化
- ☐ 过程调用的中间代码格式与翻译



中间代码生成的主要任务

□ 主要任务

- 布尔表达式的计算：完全计算、短路计算
- 控制流语句
 - 分支结构(if、switch)、循环结构、过程/函数的调用
- 各子结构的布局+无条件或有条件转移指令
- 跳转目标的两种处理方法
 - 标号技术：新建标号，跳转到标号
 - 回填技术：先构造待回填的指令链表，待跳转目标确定时再回填链表中各指令缺失的目标信息



布尔表达式

□ 布尔表达式的作用

- 计算逻辑值
- 作为控制流语句中的条件

□ 本节关联的布尔表达式文法

$B \rightarrow B \text{ or } B \mid B \text{ and } B \mid \text{not } B \mid (B) \mid E \text{ relop } E \mid \text{true} \mid \text{false}$

□ 布尔表达式的计算

- 完全计算：各子表达式都要被计算
- 短路计算： $B_1 \text{ or } B_2$ 定义成 $\text{if } B_1 \text{ then true else } B_2$
 $B_1 \text{ and } B_2$ 定义成 $\text{if } B_1 \text{ then } B_2 \text{ else false}$



控制流语句的翻译

□ 关联的控制流语句

$S \rightarrow \text{if } B \text{ then } S_1$

$\quad / \text{if } B \text{ then } S_1 \text{ else } S_2$

$\quad / \text{while } B \text{ do } S_1$

$\quad / \text{switch } E \text{ begin case } V_1: S_1 \dots$

$\quad \quad \text{case } V_{n-1}: S_{n-1}$

$\quad \quad \text{default: } S_n$

$\quad \text{end}$

$\quad | \text{ call id } (Elist)$

$\quad / S_1; S_2$



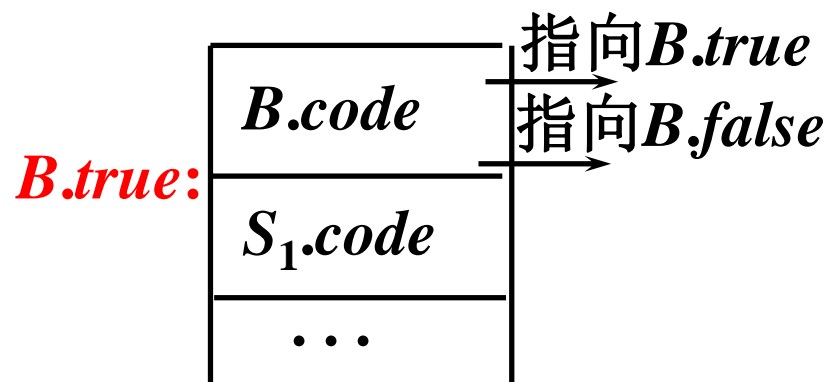
if 语句的中间代码布局

□ 问题与对策

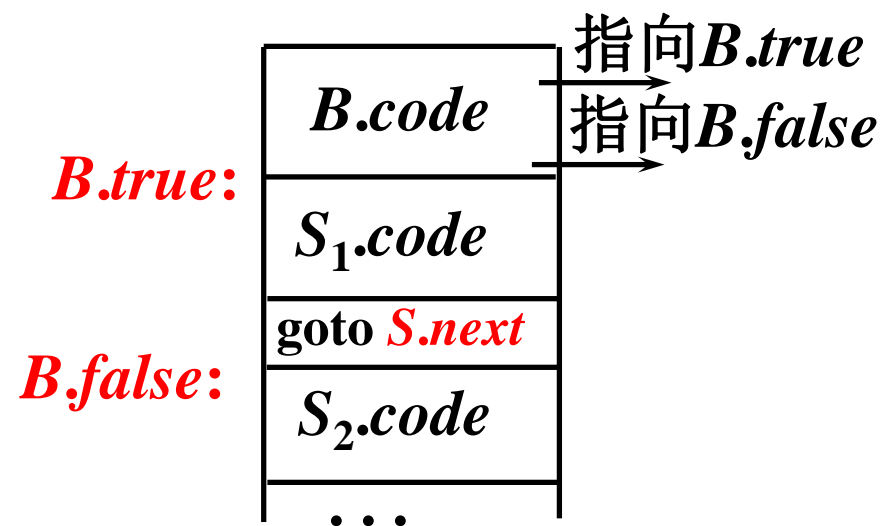
- B 的短路计算中，需要知道其为真或假时的跳转目标
- B 、 S_1 、 S_2 分别会发射多少条指令是不确定的

引入**标号**：先确定标号，在目标确定时发射**标号指令**

可调用 `newLabel()` 产生新标号，**每条语句有 *next* 标号**



(a) if-then



(b) if-then-else



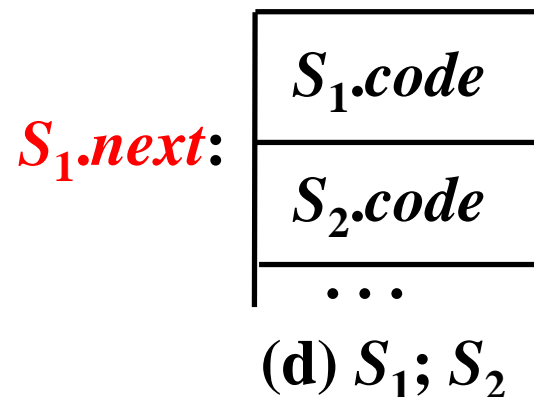
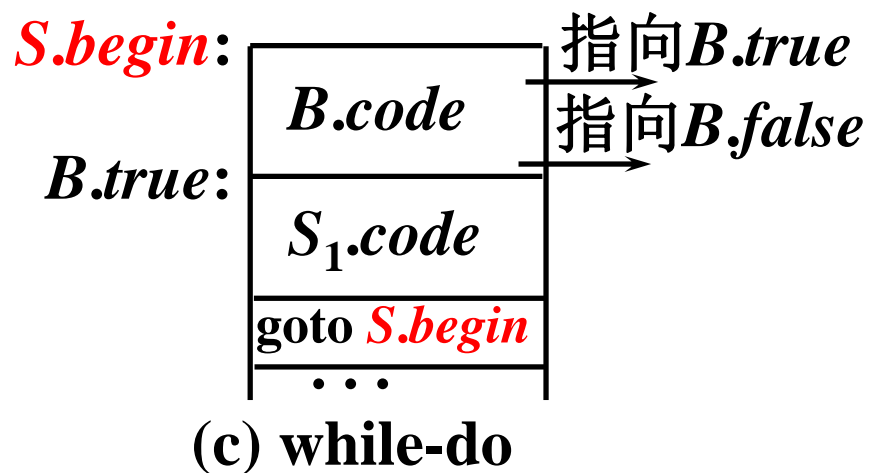
while语句和顺序结构

□ while循环语句的中间代码

- 引入开始标号 *S.begin*, 作为循环的跳转目标

□ 顺序结构

- 为每一语句 *S₁* 引入其后的下一条语句的标号 *S₁.next*





if 语句的中间代码生成

□ 问题与对策

■ B 的短路计算中，需要知道其为真或假时的跳转目标

■ B 、 S_1 、 S_2 分别会发射多少条指令是不确定的

引入**标号**：先确定标号，在目标确定时发射**标号指令**

可调用 `newLabel()` 产生新标号

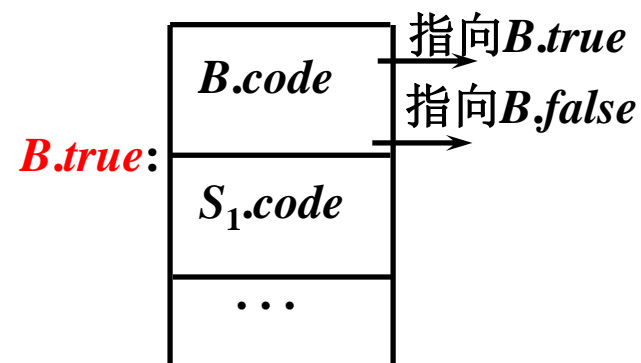
visitIf-then:

访问 B 前： $B.true = newLabel();$

$B.false = S.next;$ // 继承属性

进入 S_1 前： $S_1.next = S.next;$

访问 S_1 后： $S.code = B.code \parallel gen(B.true, ':') \parallel S_1.code$



(a) if-then



if 语句的中间代码生成

回填：仅使用综合属性

- 把跳转到同一个标号的指令放到同一张指令表中

如，为 B 引入综合属性 $truelist$ 和 $falselist$ 分别收集要回填的跳转指令，为 S 引入 $nextlist$ 收集要回填的跳转指令

- 等目的标号确定时，再把它填到表中的各条指令中

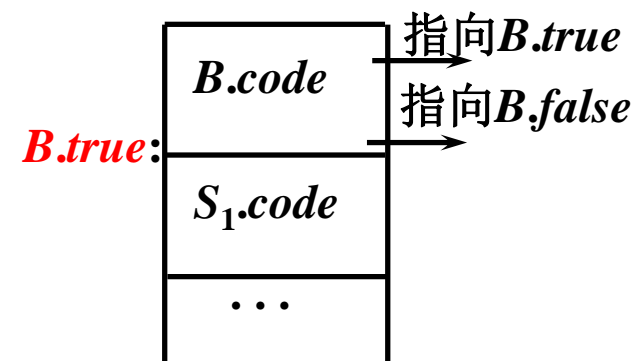
visitIf-then: $S \rightarrow \text{if } B \text{ then } S_1$

准备访问 S_1 前: $instr = nextinstr$;

访问 S_1 后:

$backPatch(B.truelist, instr)$; //回填

$S.nextlist = merge(B.falselist, S_1.nextlist)$;



(a) if-then



if 语句的中间代码生成

$S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$ (标号技术)

访问 B 前: $B.true = newLabel(); B.false = newLabel();$

进入 S_1 前: $S_1.next = S.next;$

进入 S_2 前: $S_2.next = S.next;$

访问 S_2 后: $S.code = B.code \parallel gen(B.true, ':') \parallel S_1.code \parallel$
 $gen('goto', S.next) \parallel gen(B.false, ':') \parallel S_2.code$



if 语句的中间代码生成

$S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$ (标号技术)

访问 B 前: $B.true = newLabel(); B.false = newLabel();$

进入 S_1 前: $S_1.next = S.next;$

进入 S_2 前: $S_2.next = S.next;$

访问 S_2 后: $S.code = B.code \parallel gen(B.true, ':') \parallel S_1.code \parallel$
 $gen('goto', S.next) \parallel gen(B.false, ':') \parallel S_2.code$

回填

进入 S_1 前: $instr1 = nextinstr;$

访问 S_1 后: $nextlist = makeList(nextinstr); emit('goto _');$

进入 S_2 前: $instr2 = nextinstr;$

访问 S_2 后: $backPatch(B.truelist, instr1); \quad backPatch(B.falselist, instr2);$

$S.nextlist = merge(merge(S_1.nextlist, nextlist), S_2.nextlist);$



while语句的中间代码生成

$S \rightarrow \text{while } B \text{ do } S_1$

访问while前: $S.begin = \text{newLabel}();$

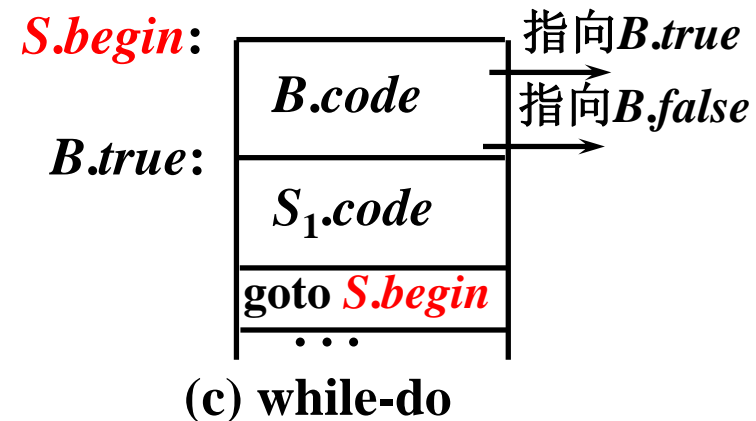
访问B前: $B.true = \text{newLabel}();$

$B.false = S.next;$

进入 S_1 前: $S_1.next = S.begin;$

访问 S_1 后: $S.code = \text{gen}(S.begin, ':') \parallel B.code \parallel$

$\text{gen}(B.true, ':') \parallel S_1.code \parallel \text{gen}(\text{'goto'}, S.begin)$





while语句的中间代码生成

$S \rightarrow \text{while } B \text{ do } S_1$

访问while前: $S.begin = \text{newLabel}();$

访问B前: $B.true = \text{newLabel}();$

$B.false = S.next;$

进入 S_1 前: $S_1.next = S.begin;$

访问 S_1 后: $S.code = \text{gen}(S.begin, ':') \parallel B.code \parallel$

$\text{gen}(B.true, ':') \parallel S_1.code \parallel \text{gen}(\text{'goto'}, S.begin)$

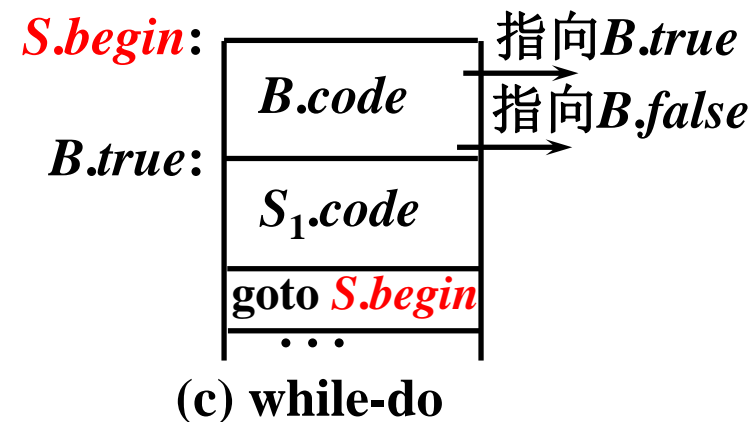
回填 进入B前: $\text{instr1} = \text{nextinstr};$

进入 S_1 前: $\text{instr2} = \text{nextinstr};$

访问 S_1 后: $\text{backPatch}(S_1.nextlist, \text{instr1});$

$\text{backPatch}(B.truelist, \text{instr2});$

$S.nextlist = B.falselist; \text{emit}(\text{'goto'}, \text{instr1});$





布尔表达式的控制流翻译

如果 B 是 $a < b$ 的形式,

那么代码是:

if $a < b$ goto $B.true$

goto $B.false$

基于回填链

1. if $a < b$ goto ____

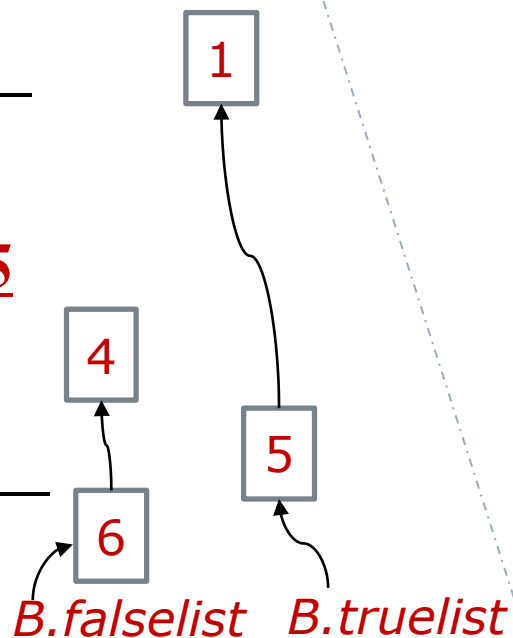
2. goto 3

3. if $c < d$ goto 5

4. goto ____

5. if $e < f$ goto ____

6. goto ____



例 表达式 B

$a < b$ or $c < d$ and $e < f$

的代码是:

基于标号

if $a < b$ goto L_{true}

goto L_1

L_1 : if $c < d$ goto L_2

goto L_{false}

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

goto L_{false}

多了标号语句
的存储开销



布尔表达式的翻译

$B \rightarrow B_1 \text{ or } B_2$ (标号技术)

访问 B_1 前: $B_1.true = B.true; B_1.false = newLabel();$

访问 B_2 前: $B_2.true = B.true; B_2.false = B.false;$

访问 B_2 后: $B.code = B_1.code \parallel gen(B_1.false, ':') \parallel B_2.code$

$B \rightarrow \text{not } B_1$ (标号技术)

访问not前: $B_1.true = B.false; B_1.false = B.true;$

访问 B_1 后: $B.code = B_1.code$



布尔表达式的翻译

$B \rightarrow B_1 \text{ and } B_2$ (标号技术)

访问 B_1 前: $B_1.true = newLabel(); B_1.false = B.false;$

访问 B_2 前: $B_2.true = B.true; B_2.false = B.false;$

访问 B_2 后: $B.code = B_1.code \parallel gen(B_1.true, ':') \parallel B_2.code$

$B \rightarrow (B_1)$ (标号技术)

访问(前: $B_1.true = B.true; B_1.false = B.false;$

访问)后: $B.code = B_1.code$



布尔表达式的翻译

$B \rightarrow E_1 \text{ relop } E_2$ (标号技术)

访问 E_2 后: $B.code = E_1.code \parallel E_2.code$

$\parallel \text{ gen('if', } E_1.place, \text{ relop.op, } E_2.place, \text{'goto', } B.true)$

$\parallel \text{ gen('goto', } B.false)$

$B \rightarrow \text{true}$ (标号技术)

访问true后: $B.code = \text{gen('goto', } B.true)$

$B \rightarrow \text{false}$ (标号技术)

访问false后: $B.code = \text{gen('goto', } B.false)$



布尔表达式的翻译(回填)

$B \rightarrow B_1 \text{ or } M B_2$ { *backPatch*(B_1 .*false*list, M .*instr*);
 B .*false*list = B_2 .*false*list;
 B .*true*list = *merge*(B_1 .*true*list, B_2 .*true*list); }

$M \rightarrow \varepsilon$ { M .*instr* = *nextinstr*; }

$B \rightarrow B_1 \text{ and } M B_2$ { *backPatch*(B_1 .*true*list, M .*instr*);
 B .*true*list = B_2 .*true*list;
 B .*false*list = *merge*(B_1 .*false*list, B_2 .*false*list); }

$B \rightarrow \text{not } B_1$ { B .*true*list = B_1 .*false*list;
 B .*false*list = B_1 .*true*list; }



布尔表达式的翻译(回填)

$B \rightarrow (B_1)$ $\{ B.truelist = B_1.truelist;$
 $B.falselist = B_1.falselist; \}$

$B \rightarrow E_1 \text{ relop } E_2$ $\{ B.truelist = makeList(nextinstr);$
 $B.falselist = makeList(nextinstr+1);$
 $emit('if', E_1.place, relop.op, E_2.place, 'goto _');$
 $emit('goto _'); \}$

$B \rightarrow \text{true}$ $\{$ $B.truelist = makeList(nextinstr);$
 $B.falselist = null; emit('goto _'); \}$

$B \rightarrow \text{false}$ $\{$ $B.falselist = makeList(nextinstr);$
 $B.truelist = null; emit('goto _'); \}$



switch语句的翻译

switch E

begin

case $V_1: S_1$

case $V_2: S_2$

...

case $V_{n-1}: S_{n-1}$

default: S_n

end

注意：这里 S_i 执行后就退出switch，相当于C语言中每个case处理后有break

分支数较少时

$t = E$ 的代码

if $t \neq V_1$ goto L_1

S_1 的代码

goto next

L_1 : if $t \neq V_2$ goto L_2

S_2 的代码

goto next

L_2 : ...

L_{n-2} : if $t \neq V_{n-1}$ goto L_{n-1}

S_{n-1} 的代码

goto next

L_{n-1} : S_n 的代码

next:

当要多次判断 t 的值时，由于跳转目标不是邻近的语句，代码的局部性不好，会引起比较多的 cache miss
→ 代码性能不高



switch语句的翻译

分支较多时，将分支测试代码集中在一起，便于生成较好的分支测试代码

| | | |
|---------------------------|----------------------------------|---------------|
| $t = E$ 的代码 | L_n : S_n 的代码 | |
| goto test | goto next | |
| L_1 : S_1 的代码 | test: if $t == V_1$ goto L_1 | |
| goto next | if $t == V_2$ goto L_2 | |
| L_2 : S_2 的代码 | ... | |
| goto next | if $t == V_{n-1}$ goto L_{n-1} | |
| ... | goto L_n | |
| L_{n-1} : S_{n-1} 的代码 | next: | 多次判断 t 的值的代 |
| goto next | | 码是邻近的 |
| | | →改善代码的局部性, |
| | | 降低cache miss |
| | | →代码性能好 |



switch语句的翻译

中间代码增加一种case语句，便于代码生成器对它进行特别处理

```
test:  case  $V_1$        $L_1$   
       case  $V_2$        $L_2$   
       ...  
       case  $V_{n-1}$      $L_{n-1}$   
       case t         $L_n$   
  
next:
```

代码生成器可做两种优化：

- 用二分查找确定该执行的分支
- 建立入口地址表，直接找到该执行的分支

(例子见第244页习题8.8)



switch语句的翻译：习题8.8

```
int i;  
i=50;  
switch(i*i){  
    case 10: i=10; break;  
    case 80: i=80; break;  
    case 50: i=50; break;  
    case 70: i=70; break;  
    case 20: i=20; break;  
    default: i=40;  
}  
switch(i*i){  
    case 7: i=7; break;  
    case 1: i=1; break;  
    case 6: i=6; break;  
    case 9: i=9; break;  
    case 5: i=5; break;  
    case 10: i=10; break;  
    case 2: i=2; break;  
    default: i=40;  
}
```

二分查找



```
movl $50,-4(%ebp)  
movl -4(%ebp),%eax  
imull -4(%ebp),%eax  
cmpl $50,%eax  
je .L5  
cmpl $50,%eax  
jg .L10  
cmpl $10,%eax  
je .L3  
cmpl $20,%eax  
je .L7  
jmp .L8
```

```
.L10:  
    cmpl $70,%eax  
    je .L6  
    cmpl $80,%eax  
    je .L4  
    jmp .L8  
    .p2align 4,,7  
.L3:  
    movl $10,-4(%ebp)  
    jmp .L2  
    .p2align 4,,7  
.L4:  
    movl $80,-4(%ebp)  
    jmp .L2  
    .p2align 4,,7  
.L5:  
    movl $50,-4(%ebp)  
    jmp .L2  
    .p2align 4,,7  
.L6:  
    movl $70,-4(%ebp)  
    jmp .L2  
    .p2align 4,,7  
.L7:  
    movl $20,-4(%ebp)  
    jmp .L2  
    .p2align 4,,7  
.L8:  
    movl $40,-4(%ebp)
```



switch语句的翻译：习题8.8

```
int i;  
i=50;  
switch(i*i){  
    case 10: i=10; break;  
    case 80: i=80; break;  
    case 50: i=50; break;  
    case 70: i=70; break;  
    case 20: i=20; break;  
    default: i=40;  
}  
switch(i*i){  
    case 7: i=7; break;  
    case 1: i=1; break;  
    case 6: i=6; break;  
    case 9: i=9; break;  
    case 5: i=5; break;  
    case 10: i=10; break;  
    case 2: i=2; break;  
    default: i=40;  
}
```

建立入口地址表



L2:

```
movl -4(%ebp),%edx  
imull -4(%ebp),%edx  
leal -1(%edx),%eax  
cmpl $9,%eax  
ja .L19  
movl .L20(,%eax,4),%eax  
jmp *%eax  
...
```

计算 $i*i-1$

直接找到该
执行的分支

L20:

```
.long .L13  
.long .L18  
.long .L19  
.long .L19  
.long .L16  
.long .L14  
.long .L12  
.long .L19  
.long .L15  
.long .L17
```



过程调用的翻译

$S \rightarrow \text{call id } (Elist)$

$Elist \rightarrow Elist, E$

$Elist \rightarrow E$

过程调用 $\text{id}(E_1, E_2, \dots, E_n)$ 的
中间代码结构

$E_1.place = E_1$ 的代码

$E_2.place = E_2$ 的代码

...

$E_n.place = E_n$ 的代码

param $E_1.place$

param $E_2.place$

...

param $E_n.place$

call $\text{id}.place, n$



过程调用的翻译

$S \rightarrow \text{call id } (Elist)$

结尾:

为长度为 n 的队列中的每个 $E.place$, $\text{emit}(\text{'param'}, E.place);$
 $\text{emit}(\text{'call'}, id.place, n);$

$Elist \rightarrow Elist, E$

结尾: 把 $E.place$ 放入队列末尾

$Elist \rightarrow E$

结尾: 将队列初始化, 并让它仅含 $E.place$



例题1

Pascal语言的标准将for语句

for v := initial to final do stmt

能否定义成和下面的代码序列有同样的含义？

begin

$t_1 := \text{initial}; t_2 := \text{final};$

$v := t_1;$

while $v \leq t_2$ do begin

stmt; $v := \text{succ}(v)$

end

end



例题1

Pascal语言的标准将for语句

for v := initial to final do stmt

能否定义成和下面的代码序列有同样的含义？

begin

$t_1 := \text{initial}; t_2 := \text{final};$

$v := t_1;$

while $v \leq t_2$ do begin

stmt; $v := \text{succ}(v)$

end

end

**final为最大整数时，
succ(final)会导致越界错误**



例题1

Pascal语言的标准将for语句

for $v := \text{initial}$ to final do stmt

的中间代码结构如下:

$t_1 = \text{initial}$

$t_2 = \text{final}$

if $t_1 > t_2$ goto L1

$v = t_1$

L2: stmt

if $v == t_2$ goto L1

$v = v + 1$

goto L2

L1:



例题2

C语言的for语句有下列形式

for (e_1 ; e_2 ; e_3) stmt

它和

e_1 ;

while (e_2)do begin

stmt;

e_3 ;

end

有同样的语义

e_1 的代码

L1: e_2 的代码

L2: e_3 的代码

goto L1

stmt的代码

goto L2

假转，由外层语句决定

真转



例题3

□ Pascal语言

`var a,b : array[1..100] of integer;`

`a:=b` // 允许数组之间赋值

也允许在相同类型的记录之间赋值

□ C语言

数组类型不行，结构体类型可以

为这种赋值选用或设计一种三地址语句，它要便于生成目标代码

答：仍然用中间代码复写语句 $x = y$ ，在生成目标代码时，必须根据它们类型的 size，产生一连串的值传送指令



中间代码生成的关键问题

假设采取的中间语言类似三地址代码

□ 类型与符号表的变化

- 多样化类型 \Rightarrow 整型(字节、字)、浮点型、类型符号表
- 1个某类型的数据 \Rightarrow m 个字节(m 为类型对应的字宽)

□ 语句的翻译

- 声明语句：不生成指令，但会更新符号表（作用域，字宽及存放的相对地址）
- 赋值语句：引入临时变量、数组/记录元素的地址计算、类型转换
- 控制流语句：跳转目标的确定(引入标号或者使用回填技术)、短路计算



中国科学技术大学
University of Science and Technology of China

下期预告：代码生成