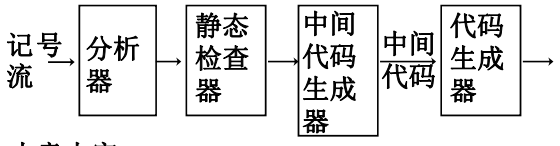
第6章　中间代码生成



本章内容

- 介绍几种常用的中间表示：后缀表示、图形表示和三地址代码
- 用语法制导定义和翻译方案来说明源语言的各种构造怎样被翻译成中间形式

6.1 中　间　语　言

6.1.1 后缀表示

表达式E的后缀表示可以如下归纳定义

- 如果E是变量或常数，那么E的后缀表示就是E本身
- 如果E是形式为$E_1\ op E_2$的表达式，那么E的后缀表示是$E_1'\ E_2'\ op$，其中E_1'和E_2'分别是E_1和E_2的后缀表示
- 如果E是形式为(E_1)的表达式，那么E_1的后缀表示也是E的后缀表示

6.1 中　间　语　言

- 后缀表示不需要括号

  (8 – 5) + 2 的后缀表示是**8 5 –2 +**
- 后缀表示的最大优点是便于计算机处理表达式

计算栈	输入串
	8 5 –2 +
8	5 –2 +
8 5	–2 +
3	2 +
3 2	+
5	

6.1 中　间　语　言

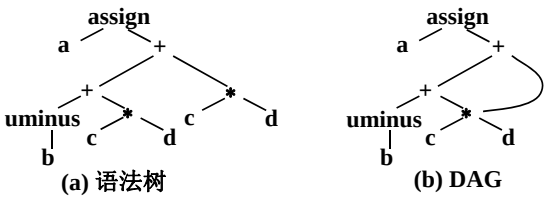
- 后缀表示不需要括号

  (8 – 5) + 2 的后缀表示是**8 5 –2 +**
- 后缀表示的最大优点是便于计算机处理表达式
- 后缀表示很容易拓广到含一元算符的表达式
- 后缀表示也可以拓广到表示赋值语句和控制语句，但很难用栈来描述它的计算

6.1 中　间　语　言

6.1.2 图形表示

- 语法树是一种图形化的中间表示
- 有向无环图（DAG）也是一种中间表示



(a) 语法树　　(b) DAG

a = (–b + c*d) + c*d的图形表示

6.1 中　间　语　言

构造赋值语句语法树的翻译方案

修改构造结点的函数可生成有向无环图

产 生 式	语 义 动 作
$S \rightarrow$ id $=E$	{*S.nptr* = *mkNode*(‘assign’, *mkLeaf* (id, id.*entry*), *E.nptr*);}
$E \rightarrow E_1 + E_2$	{*E.nptr* = *mkNode*(‘+’, E_1.*nptr*, E_2.*nptr*);}
$E \rightarrow E_1 * E_2$	{*E.nptr* = *mkNode*(‘*’, E_1.*nptr*, E_2.*nptr*);}
$E \rightarrow -E_1$	{*E.nptr* = *mkUNode*(‘uminus’, E_1.*nptr*);}
$E \rightarrow (E_1)$	{*E.nptr* = E_1.*nptr*;}
$E \rightarrow$ id	{*E.nptr* = *mkLeaf* (id, id.*entry*);}

6.1 中 间 语 言

6.1.3 三地址代码

一般形式：$x = y\ op\ z$

- 例 表达式$x + y * z$翻译成的三地址语句序列是

  $t_1 = y * z$

  $t_2 = x + t_1$

6.1 中 间 语 言

- 三地址代码是语法树或DAG的一种线性表示
- 例 **a = (–b + c*d) + c*d**

  语法树的代码

  $t_1 = -b$

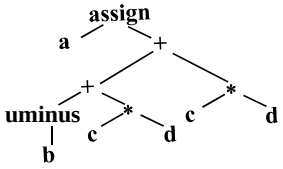
  $t_2 = c * d$

  $t_3 = t_1 + t_2$

  $t_4 = c * d$

  $t_5 = t_3 + t_4$

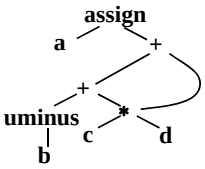
  $a = t_5$



6.1 中 间 语 言

- 三地址代码是语法树或DAG的一种线性表示
- 例 **a = (–b + c*d) + c*d**

语法树的代码	DAG的代码
$t_1 = -b$	$t_1 = -b$
$t_2 = c * d$	$t_2 = c * d$
$t_3 = t_1 + t_2$	$t_3 = t_1 + t_2$
$t_4 = c * d$	$t_4 = t_3 + t_2$
$t_5 = t_3 + t_4$	$a = t_4$
$a = t_5$	



6.1 中 间 语 言

本书常用的三地址语句

- 赋值语句$x = y\ op\ z$， $x = op\ y$， $x = y$
- 无条件转移**goto** L
- 条件转移**if** $x\ relop\ y$ **goto** L
- 过程调用**param** x 和**call p,** n
- 过程返回 **return** y
- 索引赋值$x = y[i]$和 $x[i] = y$
- 地址和指针赋值$x = \&y$，$x = *y$和$*x = y$

6.1 中 间 语 言

6.1.4 静态单赋值形式（SSA）

- 一种便于某些代码优化的中间表示
- 和三地址代码的主要区别
  - 所有赋值指令都是对不同名字的变量进行赋值

三地址代码	静态单赋值形式
p = a +b	$p_1 = a + b$
q = p – c	$q_1 = p_1 - c$
p = q * d	$p_2 = q_1 * d$
p = e – p	$p_3 = e - p_2$
q = p + q	$q_2 = p_3 + q_1$

6.1 中 间 语 言

6.1.4 静态单赋值形式（SSA）

- 一种便于某些代码优化的中间表示
- 和三地址代码的主要区别
  - 所有赋值指令都是对不同名字的变量进行赋值
  - 一个变量在不同路径上都被定值的解决办法

    if (flag) x = –1; else x = 1;

    y = x * a;

  的条件语句改成

    if (flag) $x_1 = -1$**; else** $x_2 = 1$**;**

    $x_3 = \phi(x_1, x_2)$**;** //由**flag**的值决定用x_1还是x_2

6.2 声 明 语 句

本节介绍

- 为局部名字建立符号表条目
- 为局部名字分配存储单元
- 符号表中包含名字的类型和分配给它的存储单元的相对地址等信息

6.2 声 明 语 句

6.2.1 过程中的声明

6.2 声 明 语 句

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
$D \rightarrow$ id : T {*enter* (id.*lexeme*, *T.type*, *offset*);
offset = *offset* + *T.width* }
$T \rightarrow$ integer {*T.type* = integer; *T.width* = 4; }
$T \rightarrow$ real {*T.type* = real; *T.width* = 8; }
$T \rightarrow$ array [num] of T_1
{*T.type* = *array* (num.*val*, T_1.*type*);
T.width = num.*val* × T_1.width;}
$T \rightarrow \uparrow T_1$ {*T.type* = *pointer* (T_1.*type*); *T.width* = 4;}

6.2 声 明 语 句

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
$D \rightarrow$ id : T {*enter* (id.*lexeme*, *T.type*, *offset*);
offset = *offset* + *T.width* }
$T \rightarrow$ integer {*T.type* = integer; *T.width* = 4; }
$T \rightarrow$ real {*T.type* = real; *T.width* = 8; }
$T \rightarrow$ array [num] of T_1
{*T.type* = *array* (num.*val*, T_1.*type*);
T.width = num.*val* × T_1.width;}
$T \rightarrow \uparrow T_1$ {*T.type* = *pointer* (T_1.*type*); *T.width* = 4; }

6.2 声 明 语 句

6.2.2 作用域信息的保存

- 所讨论语言的文法

  $P \rightarrow D$; S
  $D \rightarrow D$; D | id : T | proc id ; D ; S

- 语义动作用到的函数

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

6.2 声 明 语 句

$P \rightarrow M\ D$; S {*addWidth* (*top* (*tblStack*), *top* (*offsetStack*));
pop(*tblStack*); *pop* (*offsetStack*); }
$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblStack*); *push* (0, *offsetStack*); }
$D \rightarrow D_1$; D_2
$D \rightarrow$ proc id ; $N\ D_1$; S {*t* = *top*(*tblStack*);
addWidth(*t*, *top*(*offsetStack*));
pop(*tblStack*); *pop*(*offsetStack*);
enterProc(*top*(*tblStack*), id.*lexeme*, *t*); }
$D \rightarrow$ id : T {*enter*(*top*(*tblStack*), id.*lexeme*, *T.type*, *top*(*offsetStack*));
top(*offsetStack*) = *top*(*offsetStack*) + *T.width*; }
$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblStack*));
push(*t*, *tblStack*); *push*(0, *offsetStack*); }

6.2 声 明 语 句

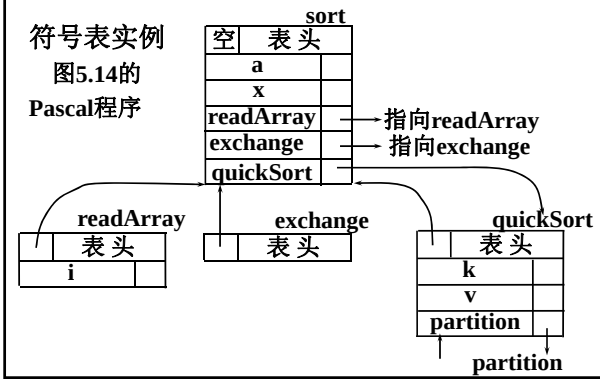
$P \rightarrow M\ D$; S {*addWidth* (*top* (*tblStack*), *top* (*offsetStack*));
pop(*tblStack*); *pop* (*offsetStack*); }
$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblStack*); *push* (0, *offsetStack*); }
$D \rightarrow D_1$; D_2
$D \rightarrow$ proc id ; $N\ D_1$; S {*t* = *top*(*tblStack*);
addWidth(*t*, *top*(*offsetStack*));
pop(*tblStack*); *pop*(*offsetStack*);
enterProc(*top*(*tblStack*), id.*lexeme*, *t*); }
$D \rightarrow$ id : T {*enter*(*top*(*tblStack*), id.*lexeme*, *T.type*, *top*(*offsetStack*));
top(*offsetStack*) = *top*(*offsetStack*) + *T.width*; }
$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblStack*));
push(*t*, *tblStack*); *push*(0, *offsetStack*); }

6.2 声 明 语 句

$P \rightarrow M\ D$; S {*addWidth* (*top* (*tblStack*), *top* (*offsetStack*));
pop(*tblStack*); *pop* (*offsetStack*); }
$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblStack*); *push* (0, *offsetStack*); }
$D \rightarrow D_1$; D_2
$D \rightarrow$ proc id ; $N\ D_1$; S {*t* = *top*(*tblStack*);
addWidth(*t*, *top*(*offsetStack*));
pop(*tblStack*); *pop*(*offsetStack*);
enterProc(*top*(*tblStack*), id.*lexeme*, *t*); }
$D \rightarrow$ id : T {*enter*(*top*(*tblStack*), id.*lexeme*, *T.type*, *top*(*offsetStack*));
top(*offsetStack*) = *top*(*offsetStack*) + *T.width*; }
$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblStack*));
push(*t*, *tblStack*); *push*(0, *offsetStack*); }

6.2 声 明 语 句

符号表实例
图5.14的
Pascal程序



6.2 声 明 语 句

6.2.3 记录的域名

$T \rightarrow$ record D end

记录类型单独建符号表，作为类型表达式的主要部分，域的相对地址从0开始

$T \rightarrow$ record $L\ D$ end
{*T.type* = *record* (*top*(*tblStack*));
T.width = *top*(*offsetStack*);
pop(*tblStack*); *pop*(*offsetStack*); }
$L \rightarrow \varepsilon$ {*t* = *mkTable*(*nil*);
push(*t*, *tblStack*); *push*(0, *offsetStack*); }

6.3 赋 值 语 句

6.3.1 符号表的中名字

$S \rightarrow$ id := E { *p* = *lookup*(id.*lexeme*);
if (*p* != *nil*)
emit (*p*, '=', *E.place*);
else *error*; }
$E \rightarrow E_1 + E_2$
{ *E.place* = *newTemp*( );
emit (*E.place*, '=', E_1.*place*, '+', E_2.*place*); }

6.3 赋 值 语 句

6.3.1 符号表的中名字

$E \rightarrow -E_1$ { *E.place* = *newTemp*( );
emit (*E.place*, '=', 'uminus', E_1.*place*); }
$E \rightarrow (E_1)$ { *E.place* = E_1.*place*; }
$E \rightarrow$ id { *p* = *lookup*(id.*lexeme*);
if (*p* != *nil*) *E.place* = *p*; else *error*; }

6.3 赋 值 语 句

6.3.2 数组元素的地址计算

一维数组A的第*i*个元素的地址计算

$base + (i - low) \times w$

变换成

$i \times w + (base - low \times w)$

减少了运行时的计算

6.3 赋 值 语 句

二维数组A: array[1..2, 1..3] of T

- 列为主

  A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]
- 行为主

  A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]

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

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

  变换成 $((i_1 \times n_2) + i_2) \times w +$

  $(base - ((low_1 \times n_2) + low_2) \times w)$

6.3 赋 值 语 句

多维数组下标变量A[i_1, i_2, ..., i_k]的地址表达式

$((\ldots ((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w$
$+ base - ((\ldots ((low_1 \times n_2 + low_2) \times n_3 + low_3) \ldots$
$) \times n_k + low_k) \times w$

$((\ldots ((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w$
$+ invariant(A)$

6.3 赋 值 语 句

6.3.3 数组元素地址计算的翻译方案

- 下标变量访问的产生式

  $L \rightarrow$ id [*Elist*] | id

  $Elist \rightarrow Elist, E \mid E$

  不便于在处理下标表达式时到符号表取信息
- 为便于写语义动作，改成等价的

  $L \rightarrow Elist$] | id

  $Elist \rightarrow Elist, E$ | id [*E*

6.3 赋 值 语 句

- 所有产生式

$S \rightarrow L := E$

$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow L$

$L \rightarrow Elist$]

$L \rightarrow$ id

$Elist \rightarrow Elist, E$

$Elist \rightarrow$ id [*E*

6.3 赋 值 语 句

$L \rightarrow$ id {*L.place* = id.*place*; *L.offset* = null; }

$Elist \rightarrow$ id [*E* {*Elist.place* = *E.place*; *Elist.ndim* = 1;
Elist.array = id.*place*; }

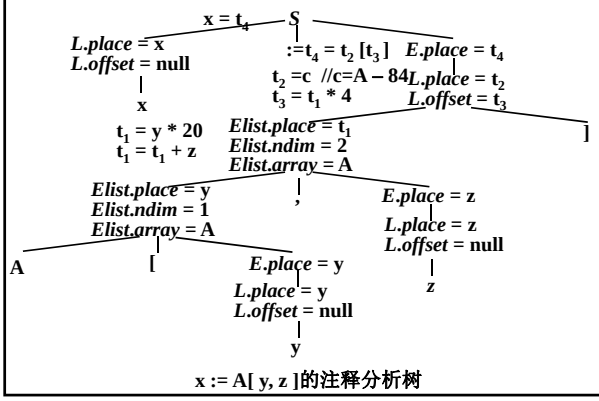
$Elist \rightarrow Elist_1, E$ { $t = newTemp( )$; $m = Elist_1.ndim + 1$;
emit (*t*, '=', $Elist_1$.*place*, '*', *limit*($Elist_1$.*array*, *m*));
emit (*t*, '=', *t*, '+', *E.place*);
$Elist.array = Elist_1.array$;
Elist.place = *t*; *Elist.ndim* = *m*;}

$L \rightarrow Elist$] { *L.place* = *newTemp*( );
emit (*L.place*, '=', *invariant* (*Elist.array*));
L.offset = *newTemp*( );
emit (*L.offset*, '=', *Elist.place*,'*',*width*(*Elist.array*));}

6.3 赋 值 语 句

$E \rightarrow L$ {if ($L.offset$ == null) /* L是简单变量 */
$E.place = L.place$;
else { $E.place = newTemp( )$;
$emit$ ($E.place$, '=', $L.place$, '[', $L.offset$, ']');
} }
$E \rightarrow E_1 + E_2$ {$E.place = newTemp( )$;
$emit$ ($E.place$, '=', $E_1.place$, '+', $E_2.place$); }
$E \rightarrow (E_1)$ { $E.place = E_1.place$;}
$S \rightarrow L := E$ { if ($L.offset$ == null) /* L是简单变量 */
$emit$ ($L.place$, '= ', $E.place$)
else $emit$ ($L.place$, '[', $L.offset$, ']', '=',
$E.place$); }

6.3 赋 值 语 句



x := A[y, z]的注释分析树

6.3 赋 值 语 句

6.3.4 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 int* j
t_2 = inttoreal t_1
t_3 = y real+ t_2
x = t_3

6.3 赋 值 语 句

$E \rightarrow E_1 + E_2$
$E.place = newTemp( )$;
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;
}
. . .

6.4 布尔表达式和控制流语句

6.4.1 布尔表达式
- 布尔表达式有两个基本目的
  - 计算逻辑值
  - 在控制流语句中用作条件表达式
- 本节所用的布尔表达式文法

$B \rightarrow B$ or B | B and B | not B | (B)
| E relop E | true | false

6.4 布尔表达式和控制流语句

6.4.1 布尔表达式
- 布尔表达式的完全计算
  - 值的表示数值化
  - 其计算类似于算术表达式的计算
- 布尔表达式的“短路”计算
  - B_1 or B_2 定义成 if B_1 then true else B_2
  - B_1 and B_2 定义成 if B_1 then B_2 else false
  - 用控制流来实现计算，即用程序中的位置来表示值，因为布尔表达式通常用来决定控制流走向
- 两种不同计算方式可能导致程序结果不一样

6.4 布尔表达式和控制流语句

- 例

语句if x < 100 or x > 200 and x != y then x := 0 的短路计算代码如下

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

6.4 布尔表达式和控制流语句

6.4.2 控制流语句的中间代码结构

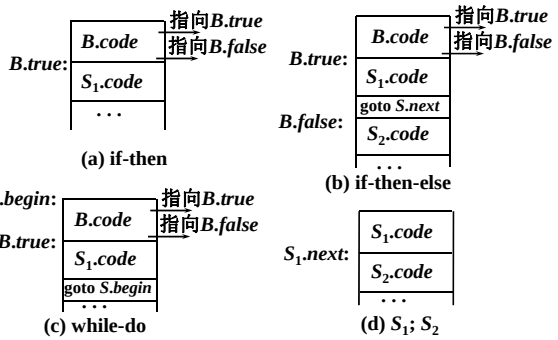
$S \rightarrow$ **if** B **then** S_1

| **if** B **then** S_1 **else** S_2

| **while** B **do** S_1

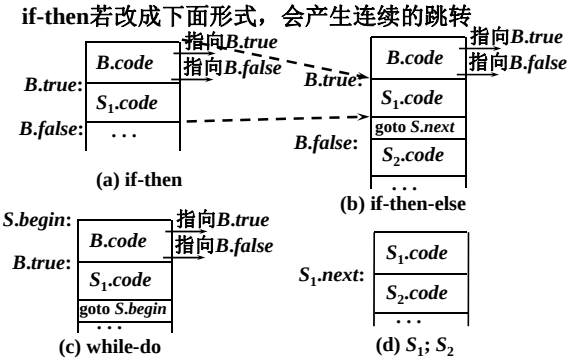
| S_1; S_2

6.4 布尔表达式和控制流语句



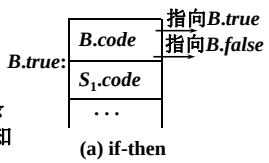
6.4 布尔表达式和控制流语句

if-then若改成下面形式，会产生连续的跳转



6.4 布尔表达式和控制流语句

- 设计翻译方案的困难
  - *B.true, B.false, S.begin* 和*S.next*都是继承属性
  - 由于*B.code*究竟有多少条指令需等*B* 翻译结束才能知道，故在翻译*B*的过程中难以知道*B.true*在三地址指令序列中的准确位置
  - 对于*B.false*来说，情况更严重，图中没有给出该标号的定义
  - 采用增加标记非终结符、使用全局变量和改写文法等方式都无法解决这种继承属性带来的困难

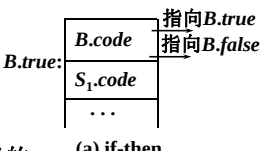


6.4 布尔表达式和控制流语句

- 解决办法——回填技术

1、从概念上说

  - 把跳转到同一个标号的指令放到同一张指令表中
  - 等目的标号确定时再把目的标号填写到该表中的各条指令中



2、从写翻译方案的角度

  - 非终结符*B*的综合属性*truelist* 和*falselist* 用来管理布尔表达式的跳转代码
  - 等跳转的目的标号确定时通过过程调用来回填

6.4 布尔表达式和控制流语句

6.4.3 布尔表达式的回填

$B \to B_1$ or $M\ B_2$ { *backPatch*(B_1.*falselist*, *M*.*instr*);
B.*falselist* = B_2.*falselist*;
B.*truelist* = *merge*(*B*1.*truelist*, *B*2.*truelist*);}

$B \to B_1$ and $M\ B_2$ { *backPatch*(B_1.*truelist*, *M*.*instr*);
B.*truelist* = B_2.*truelist*;
B.*falselist*=*merge*(B_1.*falselist*, B_2.*falselist*); }

$B \to$ not B_1 { *B*.*truelist* = B_1.*falselist*;
B.*falselist* = B_1.*truelist*; }

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

6.4 布尔表达式和控制流语句

6.4.3 布尔表达式的回填

$B \to E_1$ 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 _'); }

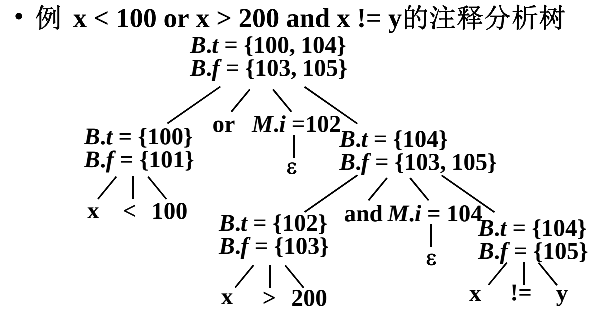
$B \to$ true { *B*.*truelist* = *makeList*(*nextinstr*);
B.*falselist* = null; *emit*('goto _'); }

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

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

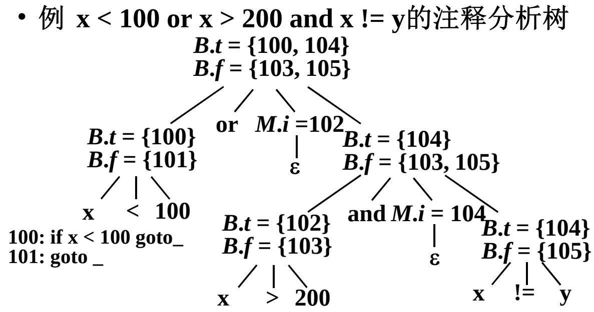
6.4 布尔表达式和控制流语句

- 例 x < 100 or x > 200 and x != y的注释分析树



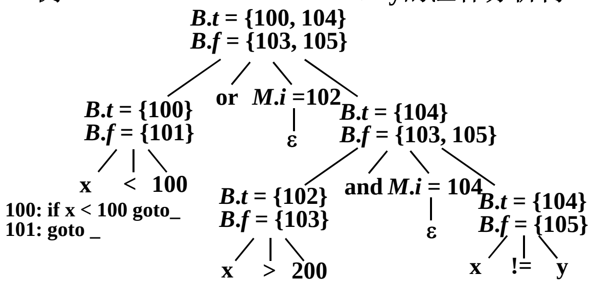
6.4 布尔表达式和控制流语句

- 例 x < 100 or x > 200 and x != y的注释分析树



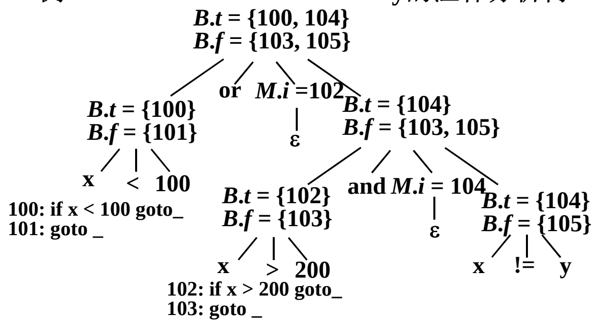
6.4 布尔表达式和控制流语句

- 例 x < 100 or x > 200 and x != y的注释分析树



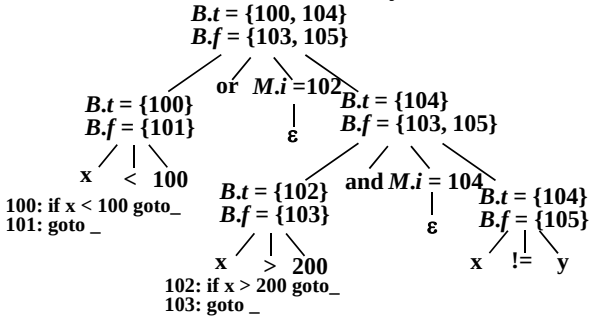
6.4 布尔表达式和控制流语句

- 例 x < 100 or x > 200 and x != y的注释分析树



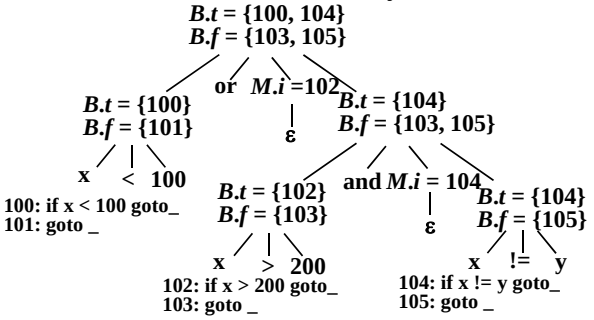
6.4 布尔表达式和控制流语句

• 例 x < 100 or x > 200 and x != y的注释分析树



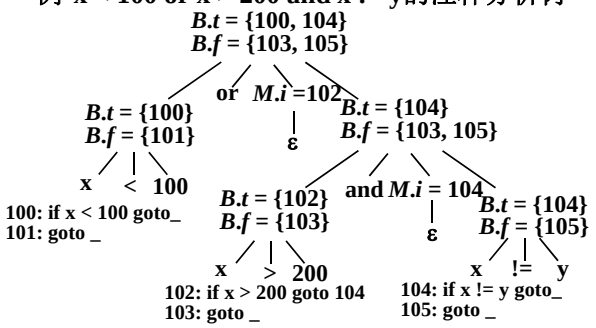
6.4 布尔表达式和控制流语句

• 例 x < 100 or x > 200 and x != y的注释分析树



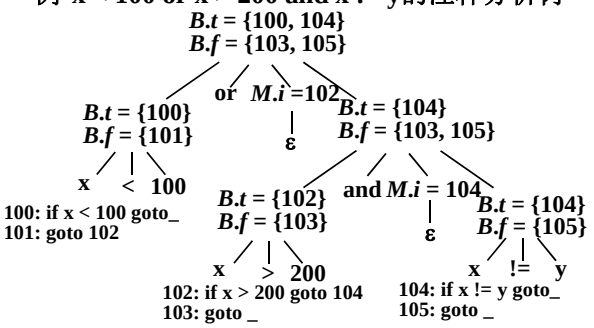
6.4 布尔表达式和控制流语句

• 例 x < 100 or x > 200 and x != y的注释分析树



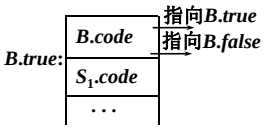
6.4 布尔表达式和控制流语句

• 例 x < 100 or x > 200 and x != y的注释分析树



6.4 布尔表达式和控制流语句

6.4.4 控制流语句的翻译



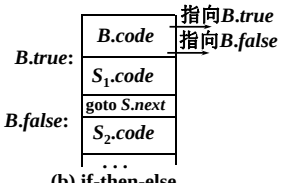
(a) if-then

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

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

6.4 布尔表达式和控制流语句

6.4.4 控制流语句的翻译



(b) if-then-else

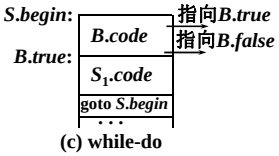
$S \rightarrow$ if B then $M_1\ S_1\ N$ else $M_2\ S_2$
{ *backPatch*(*B.truelist*, M_1.*instr*);
backPatch(*B.falselist*, M_2.*instr*);
S.nextlist = *merge*(*merge*(S_1.*nextlist*, *N.nextlist*), S_2.*nextlist*); }

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

$N \rightarrow \varepsilon$ { *N.nextlist*=*makeList*(*nextinstr*); *emit*('goto_');}

6.4 布尔表达式和控制流语句

6.4.4 控制流语句的翻译

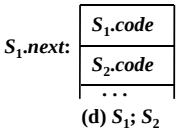


(c) while-do

$S \rightarrow$ while M_1 B do M_2 S_1
{ *backPatch*(S_1.*nextlist*, M_1.*instr*);
backPatch(*B.truelist*, M_2.*instr*);
S.nextlist = *B.falselist*; *emit*('goto', M_1.*instr*);}
$M \rightarrow \varepsilon$ { *M.instr* = *nextinstr*; }

6.4 布尔表达式和控制流语句

6.4.4 控制流语句的翻译



(d) S_1; S_2

$S \rightarrow S_1$; M S_2
{ *backPatch*(S_1.*nextlist*, *M.instr*);
S.nextlist = S_2.*nextlist*; }
$M \rightarrow \varepsilon$ { *M.instr* = *nextinstr*; }

6.4 布尔表达式和控制流语句

6.4.5 开关语句的翻译

```
switch E
  begin
      case V1: S1
      case V2: S2
      . . .
      case Vn-1: Sn-1
      default: Sn
  end
```

6.4 布尔表达式和控制流语句

- 分支数较少时

```
      t = E的代码
      if t != V1 goto L1
      S1的代码
      goto next
L1:   if t != V2 goto L2
      S2的代码
      goto next
L2:   . . .
      . . .
```

```
Ln-2: if t != Vn-1 goto Ln-1
      Sn-1的代码
      goto next
Ln-1: Sn的代码
next:
```

6.4 布尔表达式和控制流语句

- 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
      goto test
L1:   S1的代码
      goto next
L2:   S2的代码
      goto next
      . . .
Ln-1: Sn-1的代码
      goto next
```

```
Ln:   Sn的代码
      goto next
test: if t == V1 goto L1
      if t == V2 goto L2
      . . .
      if t == Vn-1 goto Ln-1
      goto Ln
next:
```

6.4 布尔表达式和控制流语句

- 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 V1    L1
      case V2    L2
      . . .
      case Vn-1  Ln-1
      case  t    Ln
next:
```

6.4 布尔表达式和控制流语句

6.4.6 过程调用的翻译

S → **call id (*Elist*)**
Elist → ***Elist, E***
Elist → ***E***

6.4 布尔表达式和控制流语句

• 过程调用**id(*E*$_1$, *E*$_2$, …, *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 id.*place, n*

6.4 布尔表达式和控制流语句

S → **call id (*Elist*)**
{ 为长度为***n***的队列中的每个***E.place***，执行
***emit*(‘param’, *E.place*);**
***emit*(‘call’, id.*plase, n*); }**
Elist → ***Elist, E***
{把***E.place***放入队列末尾}
Elist → ***E***
{将队列初始化，并让它仅含***E.place***}

本 章 要 点

- 中间代码的几种形式，它们之间的相互转换
- 符号表的组织和作用域信息的保存
- 数组元素定址的翻译方案，布尔表达式的两种不同实现方式
- 赋值语句和各种控制流语句的中间代码格式和生成中间代码的翻译方案
- 过程调用的中间代码格式和生成中间代码的翻译方案

例 题 1

Pascal语言的标准将for语句
for v := initial to final do stmt
定义成和下面的代码序列有同样的含义：

```
begin
    t1 := initial; t2 := final;
    if t1<= t2 then begin
        v := t1; stmt;
        while v <> t2 do begin
            v := succ(v); stmt
        end
    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
    t1 := initial; t2 := final;
    v := t1;
    while v <= t2 do begin
        stmt; v := succ(v)
    end
end
```

例　题　1

Pascal语言**for**语句

　　for v := initial to final do stmt

的中间代码结构如下:

```
        t1 = initial
        t2 = final
        if t1 > t2 goto L1
        v = t1
L2:     stmt
        if v == t2 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

```
    e1;
    while (e2) {
        stmt;
        e3;
    }
```

有同样的语义

例　题　2

C语言的**for**语句**for** $(e_1;e_2;e_3)$ **stmt**的中间代码结构如下

```
     e1的代码
L1:  e2的代码  ——→ 假转，由外层语句决定
L2:  e3的代码       真转（转到 stmt的代码）
     goto L1
     stmt的代码
     goto L2
```

例　题　3

- **Pascal**语言
  - **var a,b : array[1..100] of integer;**
  - **a:=b**　　——允许数组之间赋值
  - 若**a**和**b**是同一类型的记录，也允许赋值
- **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

例　题　3

- **Pascal**语言
  - **var a,b : array[1..100] of integer;**
  - **a:=b**　　——允许数组之间赋值
  - 若**a**和**b**是同一类型的记录，也允许赋值
- 仍然用中间代码复写语句 **x = y**，在生成目标代码时，必须根据它们类型对应的**size**，产生一连串的值传送指令

习　　题

- 第一次：**6.1**
- 第二次：**6.4, 6.9, 6.10**