第6章 中间代码生成

表达式的有向无环图

- 可以用和构造抽象语法树一样的SDD来构造

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

- 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

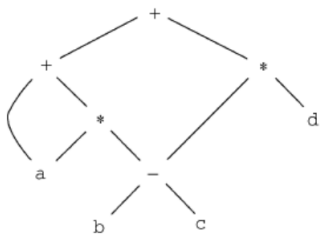


图 6-3 表达式 a+a*(b-c)+(b-c)*d 的 DAG

1) $p_1 = Leaf(\textbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\textbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\textbf{id}, entry\text{-}c)$
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) $p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\textbf{id}, entry\text{-}d)$
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

指令集合

- 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
- 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

四元式表示

- 赋值语句：a=b* -c + b* -c

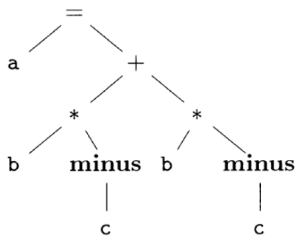
```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

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

三元式表示



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

静态单赋值

注意：等号左边右边的变量名都要改，定值处需要加上 FI 函数

- 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= Φ (x_1, x_2);
    y = x_3*a

类型和声明

这里老师一笔带过，类型表达式的SDT、计算类型和宽度的SDT、声明序列的SDT不会重点考。需要注意的：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

表达式代码的SDT

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：

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

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

$$\mid\ - E_1 \quad \{\ E.addr = \mathbf{new}\ Temp();\ gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr);\ \}$$

$$\mid\ (\ E_1\ ) \quad \{\ E.addr = E_1.addr;\ \}$$

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

数组引用翻译的SDT

不会考这么复杂，但是最好掌握

- 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

SDT与示例：

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

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

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

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

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

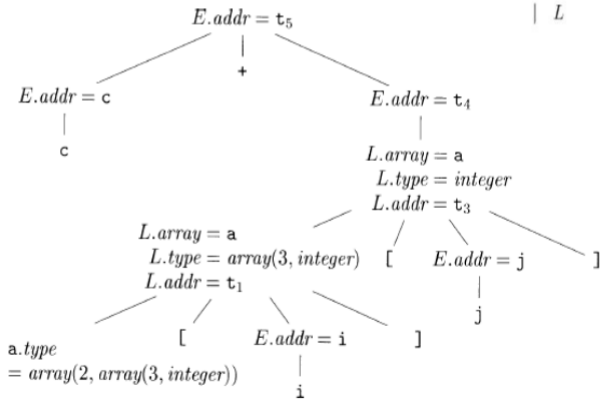


图 6-23 c+a[i][j]的注释语法分析树

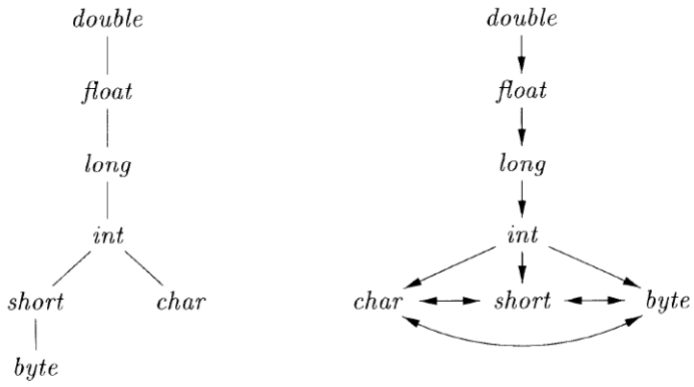
```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

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

考试时注意：调用widen函数（可能与课件不同），生成中间代码

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

控制流语句翻译的SDT

需要将语句的翻译和布尔表达式的翻译结合在一起

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

控制流语句翻译

此处B的true和false需要自己指定（因为产生式左边没有B）

next为继承属性，右边的可以用左边的推出

- 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

true和false为继承属性，右边的可以用左边的推出。

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

避免冗余的goto指令（但是在代码优化处需要避免）和利用"穿越"修改布尔表达式的语义规则不需要掌握

回填（重点！）

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

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

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

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

6) $B \rightarrow \mathbf{true}$ { $B.truelist = makelist(nextinstr)$;
$gen('\mathtt{goto} \; \_')$; }

7) $B \rightarrow \mathbf{false}$ { $B.falselist = makelist(nextinstr)$;
$gen('\mathtt{goto} \; \_')$; }

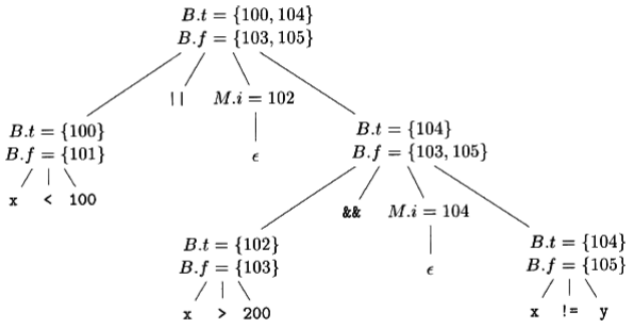
8) $M \rightarrow \epsilon$ { $M.instr = nextinstr$; }

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

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

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

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

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

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

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

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

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

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

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

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

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

Break、Continue的处理

一笔带过，不需要掌握