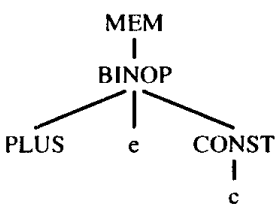


第9章 指令选择

指令 (in-struc-tion): 告诉计算机去执行一个特定操作的一条代码。

韦氏词典

中间表示 (Tree) 语言的每一个结点只表示一种操作, 如从存储器读取或存储、加或减, 以及条件转移等。真实的机器常常能用一条指令完成若干个基本操作。例如, 几乎所有的机器都可以用同一条指令完成与如下的树对应的取加操作:



找出实现一个给定的中间表示树的恰当机器指令序列是编译器在指令选择阶段要完成的工作。

树型

我们可以将一条机器指令表示成 IR 树的一段树枝 (fragment), 称之为树型 (tree pattern)。于是, 指令选择的任务就变成用树型的最小集合来覆盖 (tiling) 一棵树。

191 为了说明这个方法, 我们设计了一种指令集: Jouette 体系结构。图 9-1 给出了 Jouette 体系结构的算术指令和存取指令。在这种机器中, 寄存器 r_0 总是包含 0。

图 9-1 中双线上方的每一条指令都生成一个结果存放于寄存器中。最上面的第一项并不是一条真正的指令, 它只是表示 TEMP 结点是作为寄存器来实现的, 这种结点不需执行任何指令就能“生成一个存放在寄存器中的结果”。双线下方的指令不生成存放在寄存器中的结果, 它的执行只对存储器产生副作用。

192 图 9-1 还给出了每一条指令所实现的树型。有些指令对应有多种树型, 出现多种树型是由于可交换操作符 (如 + 和 *), 以及在有些情况下寄存器或者常数可以是 0 (如 LOAD 和 STORE 操作) 而导致的。在本章, 我们将稍微简化树的表示: BINOP(PLUS, x, y) 将写成 + (x, y), 并且不一定给出 CONST 结点和 TEMP 结点的实际值。

使用基于树的中间表示来实现指令选择的基本思想是, 用一些“瓦片”来“覆盖” (tiling) IR 树; 瓦片 (tile) 是与合法机器指令对应的树型, 指令选择的目的是用一组不重叠的瓦片来覆盖这棵 IR 树。

例如, Tiger 语言中的表达式 $a[i] := x$, 其中 i 是一个寄存器变量, a 和 x 是栈帧变量, 由该表达式生成的树可以有多种不同的覆盖方式。图 9-2 给出了它的两种覆盖和对应的指令序列 (记住, a 实际上是一个指向数组的指针的栈帧位移)。在这两种覆盖中, 瓦片 1、3 和 7 并不对应任何机器指令, 因为它们已经是含有正确值的寄存器 (TEMP)。

指令名	作用	树型
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	$\begin{array}{c} + \\ \swarrow \quad \searrow \end{array}$
MUL	$r_i \leftarrow r_j \times r_k$	$\begin{array}{c} * \\ \swarrow \quad \searrow \end{array}$
SUB	$r_i \leftarrow r_j - r_k$	$\begin{array}{c} - \\ \swarrow \quad \searrow \end{array}$
DIV	$r_i \leftarrow r_j / r_k$	$\begin{array}{c} / \\ \swarrow \quad \searrow \end{array}$
ADDI	$r_i \leftarrow r_j + c$	$\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
SUBI	$r_i \leftarrow r_j - c$	$\begin{array}{c} - \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
LOAD	$r_i \leftarrow M[r_j + c]$	$\begin{array}{c} \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\ \quad \quad \quad \\ + \quad + \quad \text{CONST} \quad \text{CONST} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$
STORE	$M[r_j + c] \leftarrow r_i$	$\begin{array}{c} \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\ \quad \quad \quad \\ + \quad + \quad \text{CONST} \quad \text{CONST} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$
MOVEM	$M[r_j] \leftarrow M[r_i]$	$\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\ \quad \end{array}$

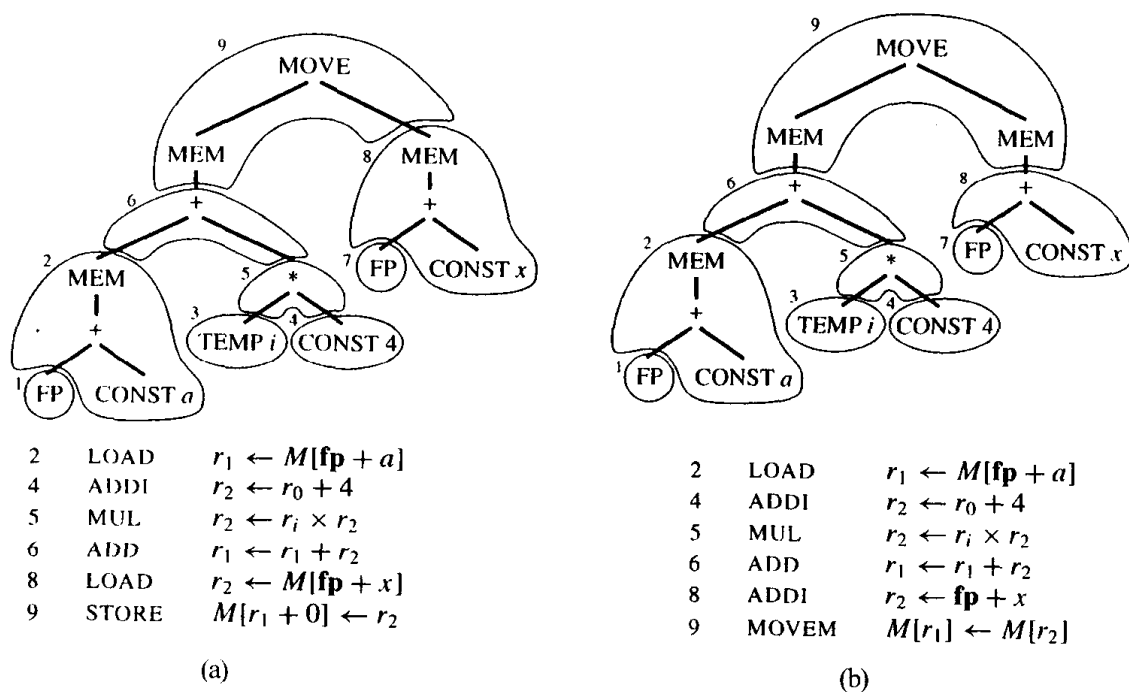
图 9-1 算术和存储器存取指令。 $M[x]$ 是地址为 x 的存储单元

图 9-2 用两种方式覆盖的一棵树

我们假定最终总是能得到一个“合理的”瓦片-树型 (tile-pattern) 集合——因为用一些每次只覆盖一个结点的小瓦片来覆盖一棵树总是可能的。在我们这个例子中，最终得到的覆盖将是：

```

ADDI     $r_1 \leftarrow r_0 + a$ 
ADD      $r_1 \leftarrow fp + r_1$ 
LOAD     $r_1 \leftarrow M[r_1 + 0]$ 
ADDI     $r_2 \leftarrow r_0 + 4$ 
MUL      $r_2 \leftarrow r_1 \times r_2$ 
ADD      $r_1 \leftarrow r_1 + r_2$ 
ADDI     $r_2 \leftarrow r_0 + x$ 
ADD      $r_2 \leftarrow fp + r_2$ 
LOAD     $r_2 \leftarrow M[r_2 + 0]$ 
STORE    $M[r_1 + 0] \leftarrow r_2$ 

```

对于一个合理的树型集合，让每个单独的 Tree 结点对应某个瓦片就足够了。一般情况下都可以做到这一点，例如，利用常数 0，便可以用 LOAD 指令覆盖单个 MEM 结点。

最佳覆盖与最优覆盖

树的最好覆盖对应于代价最小的指令序列：即最短的指令序列，或者当指令的执行时间各不相同，总执行时间最短的指令序列。

假设可以给每种指令一个代价，我们便可以将最优 (optimum) 覆盖定义为：其瓦片的代价之和可能是最小的覆盖。最佳 (optimal) 覆盖是指其中不存在两个相邻的瓦片能连接成一个代价更小的瓦片的覆盖。如果存在某个树型，它能进一步分割成几个具有较小组合代价的瓦片，则在开始之前就应当将该树型从瓦片清单中删除。

每一个最优覆盖同时也是最佳的¹，但反之不然。例如，假设除 MOVEM 指令以外，其他每一条指令的代价是一个单位，MOVEM 指令的代价为 m 个单位。则要么图 9-2a 是最优的（当 $m > 1$ 时），要么图 9-2b 是最优的（当 $m < 1$ 时），要么两个都是最优的（当 $m = 1$ 时）；但是两棵树都是最佳的。

最优覆盖基于理想代价模型。在实际中，单条指令的代价不仅仅与自己的属性有关，相邻的多条指令之间也会像第 20 章讨论的那样，以多种方式相互产生影响。

9.1 指令选择算法

现在已经有了一些较好的确定最优覆盖和最佳覆盖的算法，但正如预期的那样，最佳覆盖算法要更简单些。

复杂指令集计算机 (CISC) 具有一些一次能完成若干操作的指令。这些指令的瓦片相当大，尽管它们的最优覆盖和最佳覆盖之间的差别不是特别大，但至少有时是明显的。

现代大多数计算机都是精简指令集计算机 (RISC)。每一条 RISC 指令只完成少量操作（除 MOVEM 指令外，所有 Jouette 指令都是典型的 RISC 指令）。由于它们对应的瓦片很小且其代价一致，通常在最优与最佳覆盖之间完全不存在差别，因此，采用较简单的覆盖算法就足够了。

9.1.1 Maximal Munch 算法

本节描述的这个最佳覆盖算法叫做 Maximal Munch。它的实现相当简单：从树的根结点开始，

1. 原文用了两个含义相近的词 optimum 和 optimal，分别表示两种不同的“最好”情况。前者针对的是总的执行代价，后者针对的是瓦片的覆盖形状。我们也类似地处理，用“最优”表示总执行代价可能最小的覆盖，用“最佳”表示局部代价较小但覆盖的树型最大的瓦片组成的覆盖。——译者注

寻找适合它的最大瓦片，用这个瓦片覆盖根结点，同时也可能会覆盖根结点附近的其他几个结点。覆盖根结点后，遗留下了若干子树。然后，对每一棵子树再重复相同的算法。

当用瓦片进行覆盖的同时，也生成了与瓦片对应的指令。Maximal Munch 算法按逆序生成指令——虽然与根结点对应的指令是首先生成的，但毕竟只有当其他指令已经在寄存器中形成了操作数之后，才能执行与根结点对应的这条指令。

“最大瓦片”是覆盖结点数最多的瓦片。例如，ADD 操作对应的瓦片只有一个结点，SUBI 操作对应的瓦片有两个结点，STORE 和 MOVEM 操作对应的瓦片每一个都有三个结点。

当两个大小相等的瓦片都可以覆盖根结点时，可随意选择其中之一。如在图 9-2 的树中，STORE 和 MOVEM 两者都可以匹配，故可从中任选一个。

Maximal Munch 算法很容易用 C 来实现。我们只需要简单地编写两个递归函数，一个是用于语句的 munchStm，另一个是用于表达式的 munchExp。munchExp 中每一种情形的从句将匹配一个瓦片。这些从句按瓦片的优先级排列（最大瓦片优先级最高）。

程序 9-1 和程序 9-2 是基于 Maximal Munch 算法的 Jouette 代码生成器中的部分代码梗概。对图 9-2 中的树执行这段程序将匹配 munchStm 的第一种情形的从句，它将调用 munchExp 生成所有与 STORE 的操作数有关的指令，以及跟随在这些指令之后的 STORE 指令本身。程序 9-1 并没有说明如何选择寄存器，也没有为这些指令指明操作数的语法；我们这里关心的只是瓦片的树型匹配。

195

如果 Tree 语言的每一种结点类型都存在着一个单结点的瓦片树型，Maximal Munch 算法就不会因为没有可与某个子树匹配的瓦片树型而“陷入困境”。

196

程序 9-1 用 C 语言编写的 Maximal Munch 算法

```
static void munchStm(T_stm s) {
    switch(s->kind) {
        case T_MOVE: {
            T_exp dst = s->u.MOVE.dst, src = s->u.MOVE.src;
            if (dst->kind==T_MEM) {
                if (dst->u.MEM->kind==T_BINOP
                    && dst->u.MEM->u.BINOP.op==T_plus
                    && dst->u.MEM->u.BINOP.right->kind==T_CONST) {
                    T_exp e1 = dst->u.MEM->u.BINOP.left, e2=src;
                    /* MOVE(MEM(BINOP(PLUS,e1,CONST(i))),e2) */
                    munchExp(e1); munchExp(e2); emit("STORE");
                }
                else if (dst->u.MEM->kind==T_BINOP
                    && dst->u.MEM->u.BINOP.op==T_plus
                    && dst->u.MEM->u.BINOP.left->kind==T_CONST) {
                    T_exp e1 = dst->u.MEM->u.BINOP.right, e2=src;
                    /* MOVE(MEM(BINOP(PLUS,CONST(i),e1)),e2) */
                    munchExp(e1); munchExp(e2); emit("STORE");
                }
                else if (src->kind==T_MEM) {
                    T_exp e1 = dst->u.MEM, e2=src->u.MEM;
                    /* MOVE(MEM(e1),MEM(e2)) */
                    munchExp(e1); munchExp(e2); emit("MOVEM");
                }
                else {
                    T_exp e1 = dst->u.MEM, e2=src;
                    /* MOVE(MEM(e1),e2) */
                    munchExp(e1); munchExp(e2); emit("STORE");
                }
            }
            else if (dst->kind==T_TEMP) {
```

(续)

```

        T_exp e2=src;
        /* MOVE(TEMPi,e2) */
        munchExp(e2); emit("ADD");
    }
    else assert(0); /* destination of MOVE must be MEM or TEMP */
    break;

case T_JUMP: ...
case T_CUMP: ...
case T_NAME: ...
:

```

程序 9-2 C 函数 munchStm 的实现梗概

```

static void munchStm(T_stm s)
MEM(BINOP(PLUS,e1,CONST(i))) ⇒ munchExp(e1); emit("LOAD");
MEM(BINOP(PLUS,CONST(i),e1)) ⇒ munchExp(e1); emit("LOAD");
MEM(CONST(i)) ⇒ emit("LOAD");
MEM(e1) ⇒ munchExp(e1); emit("LOAD");
BINOP(PLUS,e1,CONST(i)) ⇒ munchExp(e1); emit("ADDI");
BINOP(PLUS,CONST(i),e1) ⇒ munchExp(e1); emit("ADDI");
CONST(i) ⇒ munchExp(e1); emit("ADDI");
BINOP(PLUS,e1,CONST(i)) ⇒ munchExp(e1); emit("ADD");
TEMP(i) ⇒ {}

```

9.1.2 动态规划

Maximal Munch 算法总在寻找一种最佳覆盖，但不一定是最优覆盖。而动态规划 (dynamic-programming) 算法却可以找到最优的覆盖。一般而言，动态规划是一种根据每个子问题的最优解找到整个问题的最优解的一种技术。在我们这里，子问题是每棵子树的覆盖。

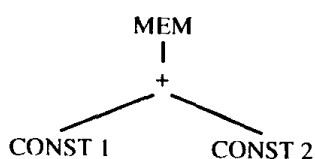
动态规划算法给树中每个结点指定一个代价，这个代价是可以覆盖以该结点为根的子树的最优指令序列的指令代价之和。

与自顶向下的 Maximal Munch 算法相反，动态规划算法是自底向上的。首先，它递归地求出结点 n 的所有儿子 (和孙子) 的代价，然后，将每一种树型 (瓦片种类) 与结点 n 进行匹配。

每个瓦片会有 0 个或多个叶子结点。在图 9-1 中，这些叶子结点是用其底端超出了瓦片的边来表示的。瓦片的这些叶结点便是可以连接子树的地方。

对每一个以代价 c 与结点 n 匹配的瓦片 t ，存在着 0 个或多个与该瓦片的叶结点对应的子树 s_i ，而且每一个子树的代价 c_i 都已经计算出来了 (因为该算法是自底向上的)。因此，匹配瓦片 t 的代价就是 $c + \sum c_i$ 。




在所有与结点 n 相匹配的瓦片 t_j 中，选择代价最小的那个瓦片，于是结点 n 的 (最小) 代价也计算出来了。例如，考虑这棵树：



197


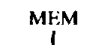

唯一与 CONST 1 匹配的瓦片是代价为 1 的 ADDI 指令。类似地，与 CONST 2 匹配的瓦片代价也

为1。有若干瓦片可与结点 + 匹配：

瓦片	指令	瓦片代价	叶结点代价	总代价
	ADD	1	1+1	3
	ADDI	1	1	2
	ADDI	1	1	2

其中，ADD 瓦片有两个叶结点，而 ADDI 瓦片只有一个叶结点。在匹配第一种 ADDI 树型时，我们说：“虽然已算出了 CONST 2 的覆盖代价，但我们并不打算使用这个信息”。因为如果选择使用第一种 ADDI 树型，CONST 2 便不会是什么瓦片的根，它的代价就会被忽略。在这种情况下，两个 ADDI 瓦片都能使结点 + 的代价最小，因此选择是任意的，结点 + 计算出来的代价是2。

现在，有若干瓦片可与 MEM 结点相匹配：

瓦片	指令	瓦片代价	叶结点代价	总代价
	LOAD	1	2	3
	LOAD	1	1	2
	LOAD	1	1	2

最后两种匹配都是最优的。

一旦求出了根结点的代价（也就是整棵树的代价），便开始指令流出（instruction emission）阶段。指令流出的算法如下：

Emission(node *n*)：对于在结点 *n* 选择的瓦片的每一个叶结点 *l_i*，执行 Emission(*l_i*)。然后流出在结点 *n* 匹配的指令。

Emission(*n*)并不是重复地作用于结点 *n* 的子结点，而是作用于与结点 *n* 相匹配的瓦片的叶结点。例如，在动态规划算法找到上面的简单树的最优代价以后，指令流出阶段将流出指令：

```
ADDI r1 ← r0 + 1
LOAD r1 ← M[r1 + 2]
```

但是对于任何以中间这个 + 结点为根的瓦片，并没有指令流出，因为这个 + 结点不是与根结点匹配的瓦片的叶结点。

9.1.3 树文法

对于具有复杂指令集以及若干类寄存器和寻址模式的机器，上述动态规划算法有一个很有用的推广算法。假设我们创建了一个“大脑分裂了的”Jouette 计算机，它有两类寄存器：*a* 寄存器用于地址，*d* 寄存器用于“数据”。这个“患有精神分裂症的”Jouette 计算机（类似于 Motorola 68000）的指令集如图 9-3 所示。

指令名	作用	树型
—	r_i	TEMP
ADD	$d_i \leftarrow d_j + d_k$	$d \begin{array}{c} + \\ \swarrow \quad \searrow \\ d \quad d \end{array}$
MUL	$d_i \leftarrow d_j \times d_k$	$d \begin{array}{c} * \\ \swarrow \quad \searrow \\ d \quad d \end{array}$
SUB	$d_i \leftarrow d_j - d_k$	$d \begin{array}{c} - \\ \swarrow \quad \searrow \\ d \quad d \end{array}$
DIV	$d_i \leftarrow d_j / d_k$	$d \begin{array}{c} / \\ \swarrow \quad \searrow \\ d \quad d \end{array}$
ADDI	$d_i \leftarrow d_j + c$	$d \begin{array}{c} + \\ \swarrow \quad \searrow \\ d \quad \text{CONST} \end{array}$ $d \begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \quad d \end{array}$ $d \text{ CONST}$
SUBI	$d_i \leftarrow d_j - c$	$d \begin{array}{c} - \\ \swarrow \quad \searrow \\ d \quad \text{CONST} \end{array}$
MOVEA	$d_j \leftarrow a_i$	$d \ a$
MOVED	$a_j \leftarrow d_i$	$a \ d$
LOAD	$d_i \leftarrow M[a_j + c]$	$d \ \text{MEM} \begin{array}{c} + \\ \swarrow \quad \searrow \\ a \quad \text{CONST} \end{array}$ $d \ \text{MEM} \begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \quad a \end{array}$ $d \ \text{MEM} \ \text{CONST}$ $d \ \text{MEM} \ a$
STORE	$M[a_j + c] \leftarrow d_i$	$\text{MOVE} \begin{array}{c} \text{MEM} \quad d \\ \swarrow \quad \searrow \\ \begin{array}{c} + \\ \swarrow \quad \searrow \\ a \quad \text{CONST} \end{array} \quad \text{CONST} \end{array}$ $\text{MOVE} \begin{array}{c} \text{MEM} \quad d \\ \swarrow \quad \searrow \\ \begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \quad a \end{array} \quad a \end{array}$ $\text{MOVE} \begin{array}{c} \text{MEM} \quad d \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{CONST} \end{array}$ $\text{MOVE} \begin{array}{c} \text{MEM} \quad d \\ \swarrow \quad \searrow \\ \text{MEM} \quad a \end{array}$
MOVEM	$M[a_j] \leftarrow M[a_i]$	$\text{MOVE} \begin{array}{c} \text{MEM} \quad \text{MEM} \\ \swarrow \quad \searrow \\ a \quad a \end{array}$

图 9-3 “患有精神分裂症的” Jouette 体系结构

每一个瓦片的根和叶子都必须带有标记 a 或 d ，以指明使用的是哪种类型的寄存器。现在，动态规划算法必须知道每一个结点使用 a 类寄存器时的最小代价，同时也必须知道使用 d 类寄存器的最小代价。

此时，一种有助的方法是用一个上下文无关文法来描述瓦片，该文法有非终结符 s （表示语句）、 a （表示其值存放到 a 寄存器的表达式）和 d （表示其值存放到 d 寄存器的表达式）。3.1 节描述了用于源语言语法的上下文无关文法，但这里使用它们的目的已截然不同。

LOAD、MOVEA 和 MOVED 指令的文法规则将像下面这样：

$$\begin{aligned}
 d &\rightarrow \text{MEM}(+(a, \text{CONST})) \\
 &\rightarrow \text{MEM}(+(\text{CONST}, a)) \\
 d &\rightarrow \text{MEM}(\text{CONST}) \\
 d &\rightarrow \text{MEM}(a) \\
 d &\rightarrow a \\
 a &\rightarrow d
 \end{aligned}$$

这种文法具有高度的歧义性：同一棵树有多种不同的分析结果（因为同一个表达式可以由许多不同的指令序列来实现）。由于这一原因，第3章描述的分析技术对于这种应用不是很有用。但是，一般的动态规划算法却能很好地适应这种情况：因为对文法中每个非终结符，要计算的只是每个结点的最小代价匹配。

199

动态规划算法虽然在概念上很简单，但用像C这样的通用程序设计语言来直接实现它却很繁琐。因此，人们开发了一些工具。这些工具是代码生成器的生成器，它们处理用于指明机器指令集的文法。对于文法中的每一条规则，都指明了代价和所要进行的动作。代价用于寻找最优覆盖，而与规则相匹配的动作则用于指令流出阶段。

同Yacc和Lex一样，代码生成器的生成器的输出通常是一个C程序，该程序用插入在适当点的动作代码（用C编写的）来操纵一个表驱动的匹配引擎。

这些工具十分方便。文法可以很好地指定类似于树的CISC指令的寻址模式。典型地，VAX机的文法有112条规则和20个非终结符；Motorola 68020的文法有141条规则和35个非终结符。但是，对于那种产生多个结果的指令，如VAX机中的地址自增指令，则很难用树型来表达。

代码生成器的生成器对于RISC机器而言则可能是大材小用。因为瓦片都非常小，且其数量也很少，故很少需要使用含有多个非终结符的文法。

9.1.4 快速匹配

对于每一个结点而言，Maximal Munch算法和动态规划算法都必须检查与该结点相匹配的所有瓦片。如果一个瓦片的每一个非叶子结点上标记的操作符都与树中对应结点的操作符（MEM、CONST等）相同，则该瓦片是匹配的。

比较简单的匹配算法是依次考察每一个瓦片，并对照树中相应的结点检查瓦片中的每一个结点。不过，还有更好的方法。为了在树结点 n 匹配一个瓦片，可用结点 n 的标号作为case语句的标号：

```
match(n) {
  switch (label(n)) {
    case MEM: ...
    case BINOP: ...
    case CONST: ...
  }
}
```

一旦选中了某个标号（如MEM）的从句，则可只考虑以该标号作为根的那些树型。另一个case语句则可以用结点 n 的子结点的标号来区分这些树型。

201

关于树型匹配判定树的组织和优化超出了本书讨论的范围。但是，为了有更好的性能，函数munchExp中那种自然排列的从句应当重写为按这种顺序排列的从句：它在进行比较时，对同一个树结点决不会考察两次。

9.1.5 覆盖算法的效率

Maximal Munch算法和动态规划算法的开销究竟有多大？

假设存在 T 个不同的瓦片，平均每个匹配的瓦片有 K 个非叶子（带标号的）结点。令 K' 表示在给定的子树中为确定应匹配哪个瓦片而需要检查的最大结点个数，该值近似于最大瓦片的大小。并且假定平均而言，每一个树结点都可以与 T' 个树型（瓦片）相匹配。对于典型的RISC

机器, 我们可以预期 $T=50, K=2, K'=4, T'=5$ 。

假设在输入树中存在 N 个结点。Maximal Munch 算法只需考虑在 N/K 个结点上的匹配, 因为, 一旦“吃掉”了根结点, 这个瓦片的非叶子结点就不再需要进行树型匹配了。

为了能找到与某个结点相匹配的所有瓦片, 必须检查的树结点数至多为 K' 个, 但 (当使用的是一种成熟的判定树时) 其中的每一个结点都只检查一次。然后, 算法需要比较每一个成功的匹配, 以查看它的代价是否为最小。因此, 每一个结点的匹配代价是 $K' + T'$, 总的代价则与 $(K' + T')N/K$ 成正比。

动态规划算法必须找出每一个结点的所有匹配, 因此它的代价与 $(K' + T')N$ 成正比。但是, 动态规划算法的比例常数比 Maximal Munch 算法的比例常数要大, 因为它需要对树进行两次遍历而不是一次。

K, K' 和 T' 都是常数, 因此所有这些算法的运行时间都是线性的。实际的测量表明, 与一个真实编译器所执行的其他处理相比, 这些指令选择算法都运行得非常快——即使是词法分析, 其执行时间也可能比指令选择要长。

9.2 CISC 机器

202

典型的现代 RISC 计算机具有如下一些特征:

- (1) 32 个寄存器。
- (2) 仅有一类整数/指针寄存器。
- (3) 算术运算仅对寄存器进行操作。
- (4) 采用形如 $r_1 \leftarrow r_2 \oplus r_3$ 的“三地址”指令。
- (5) 取指令和存指令只有 $M[\text{reg} + \text{const}]$ 寻址模式。
- (6) 每条指令的长度恰好为 32 位。
- (7) 每一条指令产生一个结果或一种作用。

20 世纪 70 年代至 80 年代中期之间设计的许多计算机都是复杂指令集计算机 (CISC)。这种计算机具有用较少位进行编码的复杂寻址方式。在计算机存储器容量较小并且很昂贵的情况下, 这种做法有其重要意义。CISC 计算机有下列典型特点:

- (1) 不多的几个寄存器 (一般 16、8 或 6 个)。
- (2) 寄存器分为不同的类型, 某些操作只在某类特定的寄存器上才能进行。
- (3) 算术运算可以通过不同的“寻址模式”访问寄存器或存储器。
- (4) 指令是形如 $r_1 \leftarrow r_1 \oplus r_2$ 的两地址指令。
- (5) 有若干不同的寻址模式。
- (6) 有由变长操作码加变长寻址模式形成的变长指令。
- (7) 指令具有副作用, 例如“自增”寻址方式。

20 世纪 90 年代以来设计的大多数计算机都是 RISC 结构的。但是, 20 世纪 90 年代以来安装的大多数通用计算机都是 CISC 计算机。例如, Intel 80386 及其后代产品 (486、Pentium)。

Pentium 计算机采用 32 位模式, 有 6 个通用寄存器、一个栈指针和一个帧指针。大多数指令可对这 6 个寄存器进行操作, 但是乘法和除法指令只能使用寄存器 eax 。与 RISC 机器中的“三地址”指令不同, Pentium 的算术指令一般都是“两地址”指令, 这就意味着目标寄存器必须与第一个源寄存器相同。大多数指令可以有两个寄存器操作数 ($r_1 \leftarrow r_1 \oplus r_2$), 或者一个寄存

器操作数和一个存储器操作数, 例如 $M[r_1 + c] \leftarrow M[r_1 + c] \oplus r_2$, 或者 $r_1 \leftarrow r_1 \oplus M[r_2 + c]$, 但不能是 $M[r_1 + c_1] \leftarrow M[r_1 + c_1] \oplus M[r_2 + c_2]$ 。

针对 CISC 机器的这些特点, 我们可以用如下快刀斩乱麻的方式来解决其难题:

(1) **寄存器较少**: 我们仍不受限制地生成 TEMP 结点, 并假设寄存器分配器能够很好地完成寄存器分配的工作。

203

(2) **寄存器分类**: Pentium 中的乘法指令要求将左操作数 (因此也是目标操作数) 放入寄存器 `eax` 中, 结果的高位 (对 Tiger 程序无用) 放至寄存器 `edx` 中。解决的方法是将操作数和结果显式地传送到相应的寄存器中。例如, 用下面的指令来实现 $t_1 \leftarrow t_2 \times t_3$:

```
mov eax, t2      eax ← t2
mul t3           eax ← eax × t3;  edx ← garbage
mov t1, eax      t1 ← eax
```

这看起来非常笨拙, 但是寄存器分配器的工作之一就是尽可能多地清除传送指令。如果寄存器分配器能够给 t_1 或 t_3 (或两者) 分配寄存器 `eax`, 则它既可以删除这两条传送指令中的一条, 也可以将两条全部删除。

(3) **两地址指令**: 我们用与前面相同的方法来解决这个问题: 即增加一条额外的传送指令。为了实现 $t_1 \leftarrow t_2 + t_3$, 我们生成

```
mov t1, t2      t1 ← t2
add t1, t3      t1 ← t1 + t3
```

然后寄希望于寄存器分配器能够将 t_1 和 t_2 分配到同一个寄存器中, 这样便可以删除这条传送指令。

(4) **算术运算可以访问存储器**: 指令选择阶段将每一个 TEMP 结点转换成一个“寄存器”引用。这些“寄存器”中的多数都将转变成存储器单元。寄存器分配器的溢出阶段必须能够有效地处理这种情况 (见第 11 章)。

对于使用存储器模式的操作数, 可以简单地在运算进行之前先将操作数取到寄存器, 运算完成之后再存入存储器。例如, 下面两个序列完成的计算是相同的:

```
mov eax, [ebp - 8]
add eax, ecx          add [ebp - 8], ecx
mov [ebp - 8], eax
```

右边的序列更加简洁 (并且占用了较少的机器代码空间), 但是这两个序列的执行速度是相同的。取数、寄存器-寄存器加和存储结果各需一个时钟周期的执行时间, 而存储器-寄存器加需要三个时钟周期的执行时间。在像 Pentium Pro 这样高度流水的机器中, 简单地数时钟周期数并不能反映事情的全貌, 但在这里结果是相同的: 无论使用的是什么指令, 处理器都必须执行取、加和存。

左边的序列有一个相当大的缺点: 它破坏了寄存器 `eax` 的值。因此, 在可能的情况下, 我们应尽量使用右边的序列。但这是寄存器分配的问题, 而不是指令执行速度的问题, 因此, 我们将它推迟到寄存器分配器再解决。

204

(5) **有若干种寻址模式**: 典型情况下, 能够完成 6 件事的寻址模式需要有 6 个执行步骤。因此, 这种指令执行起来并不比可替代它们的多条指令组成的序列快。它们只有两个优点: 一个是“破坏”的寄存器较少 (例如前例中的 `eax` 寄存器), 另一个是指令代码较短。多做一点工作, 可以使得树匹配时的指令选择能选择 CISC 寻址模式, 同时又能使程序的执行速度仍与使用简单的 RISC 指令时一样快。

(6) **变长指令**：这实际上不是编译器的问题。一旦选定了指令，流出具体编码就是汇编程序的事（尽管是单调乏味的）。

(7) **有副作用的指令**：有些机器具有“地址自增的”存储器取数指令，其效果如下：

$$r_2 \leftarrow M[r_1]; \quad r_1 \leftarrow r_1 + 4$$

这条指令产生两个结果，很难用一个树型来模拟。对这一问题，有以下三种解决方法：

- (a) 忽略地址自增指令，并希望它们会自动消失。这是一种会逐渐变得有效的解决方法，因为现代机器只有少数机器还存在多副作用指令。
- (b) 在采用树型匹配的代码生成器的上下文中，尽量用一种特别的方式来匹配特殊“方言”。
- (c) 使用完全不同的指令算法，该算法基于 DAG 样式，而不是树型。

这些解决方法中，有几种需要紧密地依靠寄存器分配器能删除传送指令，并能明智地进行溢出（见第 11 章）。

9.3 Tiger 编译器的指令选择

如程序 9-1 所示，用 C 实现“瓦片”的树型匹配是简单的（尽管冗长）。但是这段程序并没有给出对于每一种树型匹配应做何种处理。它做到的只是输出了指令名，但这些指令应当使用哪些寄存器呢？

在已用指令样式，即瓦片覆盖的树中，每一个瓦片的根对应于一个需要存放到寄存器的中间结果。寄存器分配的任务就是给每一个这样的结点指派一个寄存器号。

指令选择阶段也可以同时进行寄存器分配。但是，寄存器分配的很多方面都与特定目标机的指令集无关，并且为每一种目标机重复寄存器分配算法的做法也是很愚蠢的。因此，寄存器分配应当在指令选择之前或者之后进行。

在指令选择之前进行分配将无法知道哪些树结点需要寄存器来存储其结果，因为只有瓦片的根（而不是瓦片内其他带标记的结点）需要有明确的寄存器。因此，在指令选择之前无法非常准确地进行寄存器分配。但无论如何，有一些编译器为了避免在没有填充真实寄存器的情况下描述机器指令，确实是这样做了。

我们将在指令选择之后进行寄存器分配。指令选择阶段将在并不确切知道指令使用哪个寄存器的情况下生成指令。

9.3.1 抽象的汇编语言指令

我们设计了一种数据类型 `As_instr`，用于表示“没有指定寄存器的汇编语言指令”：

```
/* assem.h */
typedef struct {Temp_labelList labels;} *AS_targets;
AS_targets AS_Targets(Temp_labelList labels);

typedef struct {
    enum {I_OPER, I_LABEL, I_MOVE} kind;
    union {struct {string assem; Temp_tempList dst, src;
                  AS_targets jumps;} OPER;
          struct {string assem; Temp_label label;} LABEL;
          struct {string assem;
                  Temp_tempList dst, src;} MOVE;
```

```

        } u;
    } *AS_instr;

    AS_instr AS_Oper(string a, Temp_tempList d,
                    Temp_tempList s, AS_targets j);
    AS_instr AS_Label(string a, Temp_label label);
    AS_instr AS_Move(string a, Temp_tempList d, Temp_tempList s);

    void AS_print(FILE *out, AS_instr i, Temp_map m);

    :

```

OPER 中包含汇编语言指令 `assem`、操作数寄存器表 `src` 和结果寄存器表 `dst`, `src` 和 `dst` 都可以是空表。对于总是使得控制顺序执行下一条指令的操作, 有 `jump = NULL`; 其他的 `jump` 操作具有由它们可转移到的“目标”标号组成的一张表 (如果该表有可能下降到下一条指令执行, 则它必须明确地包括下一条指令)。

206

LABEL 是程序中转移可以到达的位置。它有一个 `assem` 成员和一个 `label` 成员, 前者用于指明汇编语言程序中标号的形式, 后者用于指出表示标号的那个符号。

MOVE 与 OPER 类似, 但只进行数据传送。如果某个 MOVE 指令的临时变量 `dst` 和 `src` 分配了同一个寄存器, 则该 MOVE 指令可以在稍后被删除。

调用 `AS_print(f, i, m)` 可将一条汇编指令表示为字符串的形式, 并输出到文件 `f`。 `m` 是一个临时变量映射 (temp mapping), 它给出每一个临时变量的寄存器指派 (或者只是寄存器的名字)。

`temp.h` 接口描述了对临时变量映射进行操作的函数:

```

/* temp.h */
:
typedef struct Temp_map_ *Temp_map;
Temp_map Temp_empty(void); /* create a new, empty map */
Temp_map Temp_layerMap(Temp_map over, Temp_map under);
void Temp_enter(Temp_map m, Temp_temp t, string s);
string Temp_look(Temp_map m, Temp_temp t);

Temp_map Temp_name(void);

```

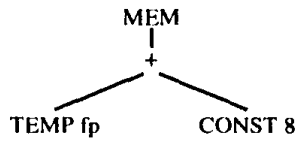
`Temp_map` 只是一张表, 表中每一项的键值是 `Temp_temp`, 绑定是字符串。但是, 一个映射可以压在另一个映射之上。例如, 如果 $\sigma_3 = \text{layer}(\sigma_1, \sigma_2)$, 这意味着 `look(σ_3, t)` 将首先尝试 `look(σ_1, t)`, 如果失败则继续尝试 `look(σ_2, t)`。另外, `enter(σ_3, t, a)` 的效果就是将 $t \mapsto a$ 送入 σ_2 。

这些 `Temp_map` 操作的主要使用者是寄存器分配器, 寄存器分配器决定每个临时变量使用的寄存器名字。但是 `Frame` 模块创建了 `Temp_map`, 用于描述所有预先分配的寄存器的名字 (如帧指针、栈指针, 等等)。因而为了有助于调试, 最好是用一个特殊的 `Temp_name` 映射将每一个临时变量 (如 t_{182}) 映射到它的“名字” (如字符串“t182”)。

机器无关性。 `As_instr` 类型与所选择的目标机汇编语言无关 (尽管它被调整成适合只有一类寄存器的机器)。如果目标机是 `Sparc`, 则 `assem` 字符串将是 `Sparc` 汇编语言。我将用 `Jouette` 汇编语言作为例子。

207

例如, 树



可以翻译为如下的 Jouette 汇编语言：

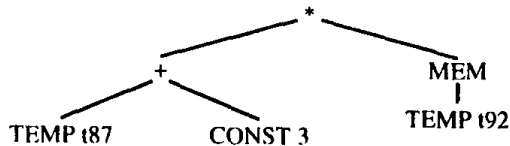
```
AS_Oper("LOAD 'd0 <- M['s0+8]",
        Temp_TempList(Temp_newtemp(), NULL),
        Temp_TempList(T_Temp(F_FP()), NULL),
        NULL)
```

这条指令需要解释一下。在寄存器分配之后，实际的 Jouette 汇编语言可能是：

```
LOAD r1 <- M[r27+8]
```

其中，假设寄存器 r_{27} 是帧指针 fp ，并且寄存器分配器决定将这个新的临时变量指派给寄存器 r_1 。但是，这条 Assem 指令并不知道有关寄存器的具体指派；它只涉及每条指令的源操作数和目的操作数。这条 LOAD 指令有一个源寄存器 $'s0$ 和一个目的寄存器 $'d0$ 。

另外还有一个有用的例子。树



可能被转换为

<i>assem</i>	<i>dst</i>	<i>src</i>
ADDI 'd0 <- 's0+3	t908	t87
LOAD 'd0 <- M['s0+0]	t909	t92
MUL 'd0 <- 's0*'s1	t910	t908,t909

208 其中，t908、t909 和 t910 都是由指令选择器新选择的临时变量。

在寄存器分配之后，汇编语言可能会是这样的：

```
ADDI r1 <- r12+3
LOAD r2 <- M[r13+0]
MUL r1 <- r1 * r2
```

instr 中的 *string* 可以引用源寄存器 $'s0, 's1, \dots, 's(k-1)$ ，以及目标寄存器 $'d0, 'd1$ ，等等。转移指令是引用标号 $'j0, 'j1$ 等的 OPER 指令。通常，条件转移指令（它可能分支，也可能下降执行）在 *jump* 表中有两个标号，但在 *assem* 字符串中只引用其中之一。

两地址指令。一些机器具有含两个操作数的算术指令，其中一个操作数既是源操作数，又是目标操作数。指令 `add t1, t2` 的作用同 $t_1 \leftarrow t_1 + t_2$ ，它可以描述为

<i>assem</i>	<i>dst</i>	<i>src</i>
add 'd0, 's1	t1	t1, t2

其中， $'s0$ 是隐含的，它不会在 *assem* 字符串中显式地出现。

9.3.2 生成汇编指令

现在，编写将 Tree 表达式转换为 Assem 指令的模式匹配从句的右部已是一件简单的事情。我将给出一些源自 *jouette* 代码生成器的例子，其中的思想也可用于实际计算机的代

码生成器中。

函数 `munchStm` 和 `munchExp` 自底向上产生 `Assem` 指令。函数 `munchExp` 返回一个存放结果的临时变量。

```
static Temp_temp munchExp(T_exp e);
static void munchStm(T_stm s);
```

程序 9-1 中 `munchExp` 中从句的“动作”可以按程序 9-3 和程序 9-4 所示编写。

程序 9-3 `munchExp` 的 `Assem` 指令

```
static Temp_temp munchExp(T_exp e) {
    switch( e )
    case MEM(BINOP(PLUS,e1,CONST(i))): {
        Temp_temp r = Temp_newtemp();
        emit(AS_Oper("LOAD 'd0 <- M['s0+" + i + "]\n",
                     L(r,NULL), L(munchExp(e1),NULL), NULL));
        return r; }
    case MEM(BINOP(PLUS,CONST(i),e1)): {
        Temp_temp r = Temp_newtemp();
        emit(AS_Oper("LOAD 'd0 <- M['s0+" + i + "]\n",
                     L(r,NULL), L(munchExp(e1),NULL), NULL));
        return r; }
    case MEM(CONST(i)): {
        Temp_temp r = Temp_newtemp();
        emit(AS_Oper("LOAD 'd0 <- M[r0+" + i + "]\n",
                     L(r,NULL), NULL, NULL));
        return r; }
    case MEM(e1): {
        Temp_temp r = Temp_newtemp();
        emit(AS_Oper("LOAD 'd0 <- M['s0+0]\n",
                     L(r,NULL), L(munchExp(e1),NULL), NULL));
        return r; }
    case BINOP(PLUS,e1,CONST(i)): {
        Temp_temp r = Temp_newtemp();
        emit(AS_Oper("ADDI 'd0 <- 's0+" + i + "\n",
                     L(r,NULL), L(munchExp(e1),NULL), NULL));
        return r; }
    case BINOP(PLUS,CONST(i),e1): {
        Temp_temp r = Temp_newtemp();
        emit(AS_Oper("ADDI 'd0 <- 's0+" + i + "\n",
                     L(r,NULL), L(munchExp(e1),NULL), NULL));
        return r; }
    case CONST(i): {
        Temp_temp r = Temp_newtemp();
        emit(AS_Oper("ADDI 'd0 <- r0+" + i + "\n",
                     NULL, L(munchExp(e1),NULL), NULL));
        return r; }
    case BINOP(PLUS,e1,e2): {
        Temp_temp r = Temp_newtemp();
        emit(AS_Oper("ADD 'd0 <- 's0+'s1\n",
                     L(r,NULL), L(munchExp(e1),L(munchExp(e2),NULL)), NULL));
        return r; }
    case TEMP(t):
        return t;
    :
}
```

程序 9-4 munchStm 的 Assem 指令

```

Temp_tempList L(Temp_temp h, Temp_tempList t) {return Temp_TempList(h,t);}
static void munchStm(T_stm s) {
    switch ( s )
    case MOVE(MEM(BINOP(PLUS,e1,CONST(i))),e2):
        emit(AS_Oper("STORE M['s0+" + i + "]" <- 's1\n",
                     NULL, L(munchExp(e1), L(munchExp(e2), NULL)), NULL));
    case MOVE(MEM(BINOP(PLUS,CONST(i),e1)),e2):
        emit(AS_Oper("STORE M['s0+" + i + "]" <- 's1\n",
                     NULL, L(munchExp(e1), L(munchExp(e2), NULL)), NULL));
    case MOVE(MEM(e1),MEM(e2)):
        emit(AS_Oper("MOVE M['s0] <- M['s1\n",
                     NULL, L(munchExp(e1), L(munchExp(e2), NULL)), NULL));
    case MOVE(MEM(CONST(i)),e2):
        emit(AS_Oper("STORE M[r0+" + i + "]" <- 's0\n",
                     NULL, L(munchExp(e2), NULL), NULL));
    case MOVE(MEM(e1),e2):
        emit(AS_Oper("STORE M['s0] <- 's1\n",
                     NULL, L(munchExp(e1), L(munchExp(e2), NULL)), NULL));
    case MOVE(TEMP(i), e2):
        emit(AS_Move("ADD    'd0 <- 's0 + r0\n",
                     i, munchExp(e2)));
    case LABEL(lab):
        emit(AS_Label(Temp_labelstring(lab) + ":\n", lab));
    :
}

```

函数 emit 只是将后面要返回的指令登记在指令表中，此表如程序 9-5 所示。assem.h 接口包含了指令表 AS_instrList 的数据结构和函数：

```

/* more of assem.h */
:
typedef struct AS_instrList_ *AS_instrList;
struct AS_instrList_ { AS_instr head; AS_instrList tail;};
AS_instrList AS_InstrList(AS_instr head, AS_instrList tail);

AS_instrList AS_splce(AS_instrList a, AS_instrList b);
void AS_printInstrList (FILE *out, AS_instrList iList,
                       Temp_map m);

typedef struct {
    string prolog; AS_instrList body; string epillog;
} *AS_proc;

```

程序 9-5 codegen 函数

```

/* codegen.c */
:
static AS_instrList iList=NULL, last=NULL;
static void emit(AS_instr inst) {
    if (last!=NULL)
        last = last->tail = AS_InstrList(inst,NULL);
    else last = iList = AS_InstrList(inst,NULL);
}

AS_instrList F_codegen(F_frame f, T_stmList stmList) {
    AS_instrList list; T_stmList sl;

```

(续)

```

: /* miscellaneous initializations as necessary */
for (sl=stmList; sl; sl=sl->tail) munchStm(sl->head);
list=iList; iList=last=NULL; return list;
}

```

9.3.3 过程调用

过程调用是用 $\text{EXP}(\text{CALL}(f, \text{args}))$ 来表示的, 函数调用则用 $\text{MOVE}(\text{TEMP } t, \text{CALL}(f, \text{args}))$ 来表示。这两个树型可以用如下瓦片来匹配:

```

case EXP(CALL(e, args)): {
    Temp_temp r = munchExp(e);
    Temp_tempList l = munchArgs(0, args);
    emit(AS_Oper("CALL 's0\n", calldefs, L(r, l), NULL));}

```

在这个例子中, 由 `munchArgs` 生成将所有实在参数传递到正确位置 (实参寄存器和/或存储器) 的代码。传递给 `munchArgs` 的整型参数为 i 则处理第 i 个参数; `munchArgs` 会用 $i+1$ 重复处理下一个参数, 依此类推。

`munchArgs` 返回的是一张表, 这张表中包含要传递给机器的 `CALL` 指令的所有临时变量。尽管这些临时变量不会显式地出现在汇编语言中, 但仍应当将它们作为 `CALL` 指令的源操作数列出, 以便活跃分析 (见第 10 章) 能够知道在此调用点需要保存它们的值。

`CALL` 指令可能会“破坏”某些寄存器中的值, 这些寄存器包括调用者保护的寄存器、返回地址寄存器和返回值寄存器。应将这些寄存器作为 `CALL` 指令的目标寄存器列出在表 `calldefs` 中, 以便编译器后面的各个阶段能知道这些寄存器在此曾被定值。

通常, 就任何一条指令而言, 只要有写另一个寄存器的副作用, 就需要进行这样的处理。例如: Pentium 的乘法指令使用寄存器 `edx` 来存放结果中无用的高位字节, 因此 `edx` 和 `eax` 都要作为乘法指令的目标寄存器列入到表中。(高位字节对于用汇编语言编写高精度的算术运算程序是非常有用的, 但是大多数高级程序设计语言都无法访问它们。) 212

9.3.4 无帧指针的情形

在如图 6-1 所示的栈帧布局中, 帧指针指向栈帧的一端, 栈指针则指向栈帧的另一端。在每次过程调用时, 栈指针寄存器的值将被复制到帧指针寄存器中, 然后栈指针本身再加上新栈帧的大小。

很多计算机的调用约定不使用帧指针, 而是使用一个“虚拟的帧指针”, 这个虚拟帧指针总是等于栈指针加栈帧的大小。这样做可以节省时间 (无复制指令) 和空间 (多了一个可用于其他目的的寄存器)。但是我们的 `Translate` 阶段已经生成了引用这个虚拟帧指针的树。因此, 函数 `codegen` 必须用 $\text{SP} + k + fs$ 来替代所有对 $\text{FP} + k$ 的引用, 其中 fs 是栈帧的大小。codegen 在用瓦片覆盖树的过程中可以识别出这种引用模式。

但是, 为了替代它们, `codegen` 必须知道 fs 的值, 而此时还不知道寄存器分配的情况, 因此无法确定 fs 的值。假设要在标号 `L14` 处流出函数 f 的代码, `codegen` 可以在它的汇编指令中只生成 `sp + L14_framesize`, 并期望函数 f 的入口代码 (由 `F_procEntryExit3` 生成的) 会包含汇编语言常数 `L14_framesize`。因此 `codegen` (程序 9-5) 要接收一个 `frame` 参数, 这

样它就可以知道名字 L14。

那种有“真实”帧指针的实现不需要对代码进行这种修改，并且可以忽略给 `codegen` 的 `frame` 参数。但是，既然使用真实的帧指针的实现浪费时间和空间，为什么还要这种有帧指针的实现呢？回答是这种实现使得在创建了栈帧之后，仍然可以支持栈的增长和收缩；有些语言允许动态分配数组位于栈帧内（例如，C 语言中使用 `alloca` 分配的数组）。不过，调用约定的设计者现在倾向于避免动态可调整的栈帧。

213

程序设计：指令选择

```
/* codegen.h */
AS_instrList F_codegen(F_frame f, T_stmList stmList);
```

使用 Maximal Munch 算法为你喜欢的指令集实现 IR 树到汇编指令的转换（令 μ 代表 Sparc、Mips、Alpha、Pentium 等）。如果你想要生成的是 RISC 机器的代码，却没有可对所生成代码进行测试的 RISC 机器，那么可以使用本书 Web 网页中介绍的 SPIM（由 James Larus 实现的一个 MIPS 模拟器）。

首先写出实现 `codegen.h` 中接口的模块 `μ codegen.c`，此模块用 Maximal Munch 转换算法将 IR 树转换为 Assem 数据结构。

在将你的 `codegen` 模块作用于 IR 树之前，应先用第 8 章描述的 Canon 模块简化它们。用函数 `AS_printInstrList` 将 `codegen` 模块得到的 Assem 树转换为 μ 汇编语言。因为你还没有进行寄存器分配，故只要将 `Temp_name` 传递给 `AS_print` 作为将临时变量转换到字符串的函数即可。

这样生成的是完全不需要使用寄存器名的“汇编”语言：指令将使用诸如 `t3`、`t283` 之类的名字。但是这些临时变量中有一些是“内建”的临时变量，它们是由 `Frame` 模块创建的用于表示特定机器寄存器（如 `Frame.FP`，见第 112 页）的临时变量。如果这些寄存器以它们本来的名字出现（例如，用 `fp` 而不用 `t1`），汇编语言程序就会比较容易阅读。

`Frame` 模块必须提供从这种特殊临时变量至其名字的映射，并将非特殊的临时变量映射为 `NULL`：

```
/* frame.h */
:
Temp_map F_tempMap;
```

于是，为了能够在寄存器分配之前就能显示出汇编语言，可用 `Temp_layerMap` 创建一个新函数，这个新函数首先尝试 `F_tempMap`，如果该函数返回 `NULL`，则转而使用函数 `Temp_name`。

寄存器表

生成下述寄存器表：对于每一个寄存器，都需要有一个给出其汇编语言表示的字符串和一个在 `Tree` 与 `Assem` 数据结构中引用它的 `Temp_temp`。

214

- `specialregs` μ 寄存器组成的表，它用于实现“特殊”寄存器，如 `RV` 和 `FP`，还有栈指针 `SP`、返回地址寄存器 `RA`，以及（某些机器上）0 号寄存器 `ZERO`。某些机器可能还有其他的特殊寄存器。
- `argregs` μ 寄存器组成的表，此表中的寄存器用于传递实在参数（包括静态链）。

- `calleesaves` μ 寄存器组成的表，此表中的寄存器是被调用过程（被调用者）必须保护并恢复以防止其改变的寄存器。
- `callersaves` μ 寄存器组成的表，此表中的寄存器是被调用者可能破坏的寄存器。

这4种寄存器表相互不能重叠，并且必须包括可能在 `Assem` 指令中出现的所有寄存器。这些表不能通过 `frame.h` 接口导出到外部，但在内部对 `Frame` 和 `codegen` 都很有用——例如，用于实现 `munchArgs` 和构造 `calldefs` 表。

实现 `frame.h` 接口中的 `F_procEntryExit2` 函数。

```
/* frame.h */
:
AS_instrList F_procEntryExit2(AS_instrList body);
```

这个函数在函数体的末尾添加了一条所谓的“下沉”（sink）指令，用以告诉寄存器分配器在过程的出口某些寄存器是活跃的。在 `Jouette` 机器的情况下，这个函数相当简单：

```
static Temp_tempList returnSink = NULL;

AS_instrList F_procEntryExit2(AS_instrList body) {
    if (!returnSink) returnSink =
        Temp_TempList(ZERO, Temp_TempList(RA,
                                           Temp_TempList(SP, calleesaves)));
    return AS_splICE(body, AS_InstrList(
        AS_Oper("", NULL, returnSink, NULL), NULL));
}
```

这意味着在函数结尾处，临时变量0、返回地址、栈指针，以及所有被调用者保护的寄存器都仍然是活跃的。使得临时变量 `zero` 在出口处是活跃的就意味着它始终都是活跃的，这可以防止寄存器分配器将它用于其他目的。同样的技巧也适用于机器可能具有的其他特殊寄存器。

\$TIGER/chap9 中包含的可用文件有：

- `canon.c`，规范化和轨迹生成。
- `assem.c`，`Assem` 模块。
- `main.c`，需要你修改的 `Main` 模块。

215

你的代码生成器将只处理每个过程的过程体或函数的函数体，而不处理过程的入口和出口指令序列。你可利用 `F_procEntryExit3` 函数的一个“掐头去尾”的版本：

```
AS_proc F_procEntryExit3(F_frame frame, AS_instrList body) {
    char buf[100];
    sprintf(buf, "PROCEDURE %s\n", S_name(frame->name));
    return AS_Proc(String(buf), body, "END\n");
}
```

推荐阅读

Cattell[1980]将机器指令表示成各种树型，发明了用于指令选择的 `Maximal Munch` 算法，建立了一个代码生成器的生成器，该生成器能够根据指令集的树型描述生成指令选择函数。Glanville 和 Graham[1978]将树型表示成 `LR(1)` 文法中的产生式，从而使得 `Maximal Munch` 算法可以使用多个非终结符来表示不同类型的寄存器和不同的寻址方式。但是描述指令集的文法的固有歧义性导致 `LR(1)` 方法存在问题；Aho 等人[1989]采用动态规划方法来分析树的文法，这种做法解决了歧义性

问题, 同时该文也介绍了自动代码生成器的生成器 Twig。动态规划可以在构造编译器的时候完成, 而不是在生成代码的时候完成 [Pelegri-Llopert and Graham 1988]; 利用这种技术, BURG 工具 [Fraser et al. 1992] 实现了一个与 Twig 相似但生成代码速度更快的接口。

习题

9.1 画出下面每一个表达式的树, 并使用 Maximal Munch 算法生成它们对应的 Jouette 机器指令。圈出其中的瓦片 (见图 9-2), 按照匹配的顺序对这些瓦片编号, 并给出所生成的 Jouette 指令序列。

- a. $\text{MOVE}(\text{MEM}(+(+(\text{CONST}_{1000}, \text{MEM}(\text{TEMP}_x))), \text{TEMP}_{fp})), \text{CONST}_0)$
- b. $\text{BINOP}(\text{MUL}, \text{CONST}_5, \text{MEM}(\text{CONST}_{100}))$

*9.2 考虑一个具有如下指令的计算机:

```
mult const1(src1), const2(src2), dst3
 $r_3 \leftarrow M[r_1 + \text{const}_1] * M[r_2 + \text{const}_2]$ 
```

这个机器中, r_0 总是 0, 并且 $M[1]$ 总是包含 1。

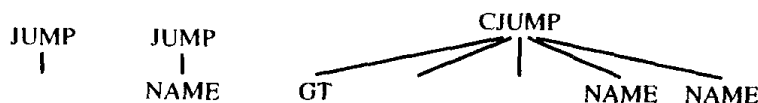
- a. 画出与这条指令 (和它的特殊情形) 对应的所有树型。
- b. 在 (a) 中选择一个较大的树型, 并说明如何写一个对此树型进行匹配的 C 语言的 if 语句, 使得它与 Tiger 编译器中使用的某个 Tree 表达式相匹配。

9.3 在 Jouette 计算机中有如下几种控制流指令

```
BRANCHGE  if  $r_i \geq 0$  goto L
BRANCHLT  if  $r_i < 0$  goto L
BRANCHEQ  if  $r_i = 0$  goto L
BRANCHNE  if  $r_i \neq 0$  goto L
JUMP      goto  $r_i$ 
```

其中, JUMP 指令的转移地址包含在寄存器中。

用这些指令实现下面的树型:



假设 CJUMP 之后总是跟随着它的 false 标号。给出实现每种树型的最好方法; 在某些情况下, 你可能会需要使用多条指令或创建一个新的临时变量。如何在不使用 BRANCHGT 指令的情况下实现 $\text{CJUMP}(\text{GT}, \dots)$?