

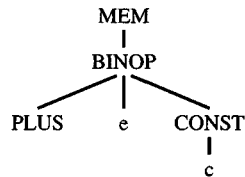
# 9

## Instruction Selection

**in-struc-tion:** a code that tells a computer to perform a particular operation

*Webster's Dictionary*

The intermediate representation (Tree) language expresses only one operation in each tree node: memory fetch or store, addition or subtraction, conditional jump, and so on. A real machine instruction can often perform several of these primitive operations. For example, almost any machine can perform an add and a fetch in the same instruction, corresponding to the tree



Finding the appropriate machine instructions to implement a given intermediate representation tree is the job of the *instruction selection* phase of a compiler.

### TREE PATTERNS

We can express a machine instruction as a fragment of an IR tree, called a *tree pattern*. Then instruction selection becomes the task of tiling the tree with a minimal set of tree patterns.

For purposes of illustration, we invent an instruction set: the *Jouette* architecture. The arithmetic and memory instructions of *Jouette* are shown in Figure 9.1. On this machine, register  $r_0$  always contains zero.

Name	Effect	Trees
—	$r_i$	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	$r_i \leftarrow r_j - c$	$\begin{array}{c} - \\ \swarrow \quad \searrow \\ \text{CONST} \end{array}$
LOAD	$r_i \leftarrow M[r_j + c]$	$\begin{array}{c} \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\   \quad   \quad   \quad   \\ + \quad + \quad + \quad + \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$
STORE	$M[r_j + c] \leftarrow r_i$	$\begin{array}{c} \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\   \quad   \quad   \quad   \\ + \quad + \quad + \quad + \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$
MOVEM	$M[r_j] \leftarrow M[r_i]$	$\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\   \quad   \end{array}$

**FIGURE 9.1.** Arithmetic and memory instructions. The notation  $M[x]$  denotes the memory word at address  $x$ .

Each instruction above the double line in Figure 9.1 produces a result in a register. The very first entry is not really an instruction, but expresses the idea that a TEMP node is implemented as a register, so it can “produce a result in a register” without executing any instructions at all. The instructions below the double line do not produce results in registers, but are executed only for side effects on memory.

For each instruction, the tree-patterns it implements are shown. Some instructions correspond to more than one tree pattern; the alternate patterns are obtained for commutative operators (+ and \*), and in some cases where a register or constant can be zero (LOAD and STORE). In this chapter we abbre-

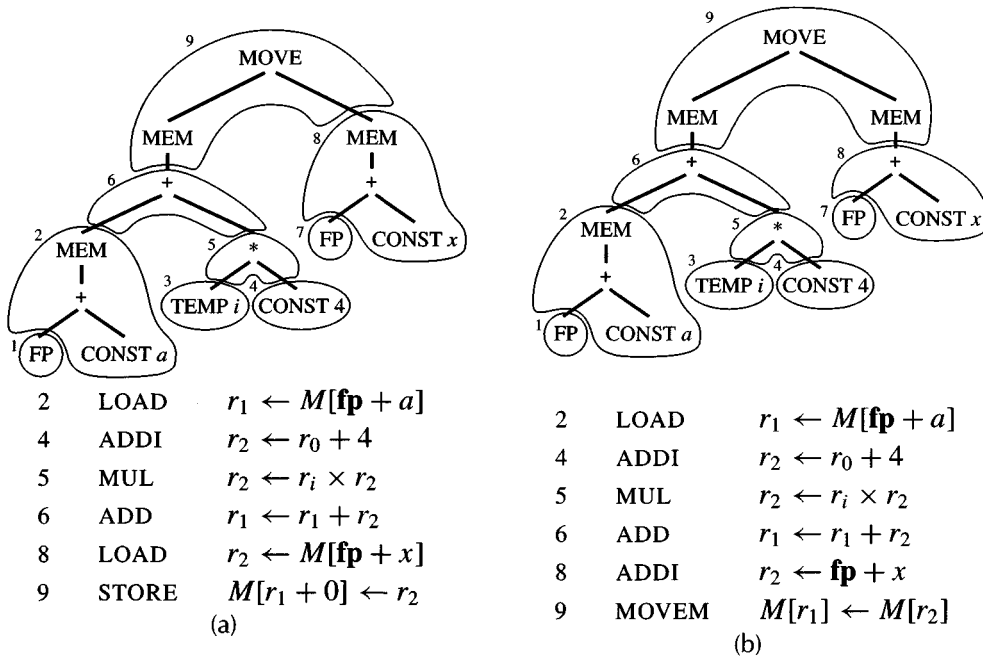


FIGURE 9.2. A tree tiled in two ways.

viate the tree diagrams slightly: BINOP(PLUS,  $x$ ,  $y$ ) nodes will be written as  $+(x, y)$ , and the actual values of CONST and TEMP nodes will not always be shown.

The fundamental idea of instruction selection using a tree-based intermediate representation is *tiling* the IR tree. The *tiles* are the set of tree patterns corresponding to legal machine instructions, and the goal is to cover the tree with nonoverlapping tiles.

For example, the Tiger-language expression such as  $a[i] := x$ , where  $i$  is a register variable and  $a$  and  $x$  are frame-resident, results in a tree that can be tiled in many different ways. Two tilings, and the corresponding instruction sequences, are shown in Figure 9.2 (remember that  $a$  is really the frame offset of the pointer to an array). In each case, tiles 1, 3, and 7 do not correspond to any machine instructions, because they are just registers (TEMPs) already containing the right values.

Finally – assuming a “reasonable” set of tile-patterns – it is always possible to tile the tree with tiny tiles, each covering only one node. In our example, such a tiling looks like this:

```

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

```

For a reasonable set of patterns, it is sufficient that each individual Tree node correspond to some tile. It is usually possible to arrange for this; for example, the LOAD instruction can be made to cover just a single MEM node by using a constant of 0, and so on.

### OPTIMAL AND OPTIMUM TILINGS

The best tiling of a tree corresponds to an instruction sequence of least cost: the shortest sequence of instructions. Or if the instructions take different amounts of time to execute, the least-cost sequence has the lowest total time.

Suppose we could give each kind of instruction a cost. Then we could define an *optimum* tiling as the one whose tiles sum to the lowest possible value. An *optimal* tiling is one where no two adjacent tiles can be combined into a single tile of lower cost. If there is some tree pattern that can be split into several tiles of lower combined cost, then we should remove that pattern from our catalog of tiles before we begin.

Every *optimum* tiling is also *optimal*, but not vice versa. For example, suppose every instruction costs one unit, except for MOVEM which costs  $m$  units. Then either Figure 9.2a is optimum (if  $m > 1$ ) or Figure 9.2b is optimum (if  $m < 1$ ) or both (if  $m = 1$ ); but both trees are optimal.

Optimum tiling is based on an idealized cost model. In reality, instructions are not self-contained with individually attributable costs; nearby instructions interact in many ways, as discussed in Chapter 20.

There are good algorithms for finding optimum and optimal tilings, but the algorithms for optimal tilings are simpler, as you might expect.

*Complex Instruction Set Computers (CISC)* have instructions that accomplish several operations each. The tiles for these instructions are quite large, and the difference between optimum and optimal tilings – while never very large – is at least sometimes noticeable.

Most architectures of modern design are *Reduced Instruction Set Computers (RISC)*. Each RISC instruction accomplishes just a small number of operations (all the *Jouette* instructions except MOVEM are typical RISC instructions). Since the tiles are small and of uniform cost, there is usually no difference at all between optimum and optimal tilings. Thus, the simpler tiling algorithms suffice.

### MAXIMAL MUNCH

The algorithm for optimal tiling is called *Maximal Munch*. It is quite simple. Starting at the root of the tree, find the largest tile that fits. Cover the root node – and perhaps several other nodes near the root – with this tile, leaving several subtrees. Now repeat the same algorithm for each subtree.

As each tile is placed, the corresponding instruction is generated. The Maximal Munch algorithm generates the instructions *in reverse order* – after all, the instruction at the root is the first to be generated, but it can only execute after the other instructions have produced operand values in registers.

The “largest tile” is the one with the most nodes. For example, the tile for ADD has one node, the tile for SUBI has two nodes, and the tiles for STORE and MOVEM have three nodes each.

If two tiles of equal size match at the root, then the choice between them is arbitrary. Thus, in the tree of Figure 9.2, STORE and MOVEM both match, and either can be chosen.

Maximal Munch is quite straightforward to implement in ML. Simply write two recursive functions, `munchStm` for statements and `munchExp` for expressions. Each clause of `munchExp` will match one tile. The clauses are ordered in order of tile preference (biggest tiles first); ML’s pattern-matching always chooses the first rule that matches.

Program 9.3 is a partial example of a *Jouette* code-generator based on the Maximal Munch algorithm. Executing this program on the tree of Figure 9.2 will match the first clause of `munchStm`; this will call `munchExp` to emit all the instructions for the operands of the STORE, followed by the STORE itself. Program 9.3 does not show how the registers are chosen and operand syntax is specified for the instructions; we are concerned here only with the pattern-matching of tiles.

---

## 9.1. ALGORITHMS FOR INSTRUCTION SELECTION

---

```
structure T = Tree

fun munchStm(T.MOVE(T.MEM(T.BINOP(T.PLUS,e1,T.CONST i)),e2)) =
    (munchExp(e1); munchExp(e2); emit "STORE")
| munchStm(T.MOVE(T.MEM(T.BINOP(T.PLUS,T.CONST i,e1)),e2)) =
    (munchExp(e1); munchExp(e2); emit "STORE")
| munchStm(T.MOVE(T.MEM(e1),T.MEM(e2))) =
    (munchExp(e1); munchExp(e2); emit "MOVEM")
| munchStm(T.MOVE(T.MEM(T.CONST i),e2)) =
    (munchExp(e2); emit "STORE")
| munchStm(T.MOVE(T.MEM(e1),e2)) =
    (munchExp(e1); munchExp(e2); emit "STORE")
| munchStm(T.MOVE(T.TEMP i, e2)) =
    (munchExp(e2); emit "ADD")

and munchExp(T.MEM(T.BINOP(T.PLUS,e1,T.CONST i))) =
    (munchExp(e1); emit "LOAD")
| munchExp(T.MEM(T.BINOP(T.PLUS,T.CONST i,e1))) =
    (munchExp(e1); emit "LOAD")
| munchExp(T.MEM(T.CONST i)) =
    (emit "LOAD")
| munchExp(T.MEM(e1)) =
    (munchExp(e1); emit "LOAD")
| munchExp(T.BINOP(T.PLUS,e1,T.CONST i)) =
    (munchExp e1; emit "ADDI")
| munchExp(T.BINOP(T.PLUS,T.CONST i,e1)) =
    (munchExp e1; emit "ADDI")
| munchExp(T.CONST i) =
    (emit "ADDI")
| munchExp(T.BINOP(T.PLUS,e1,e2)) =
    (munchExp e1; munchExp e2; emit "ADD")
| munchExp(T.TEMP t) =
    ()
```

---

### PROGRAM 9.3. Maximal Munch as ML pattern-matching.

---

If, for each node-type in the `Tree` language, there exists a single-node tile pattern, then Maximal Munch cannot get “stuck” with no tile to match some subtree.

### DYNAMIC PROGRAMMING

Maximal Munch always finds an optimal tiling, but not necessarily an optimum. A dynamic-programming algorithm can find the optimum. In general, dynamic programming is a technique for finding optimum solutions for a

whole problem based on the optimum solution of each subproblem; here the subproblems are the tilings of the subtrees.

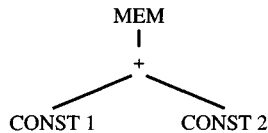
The dynamic-programming algorithm assigns a *cost* to every node in the tree. The cost is the sum of the instruction-costs of the best instruction sequence that can tile the subtree rooted at that node.

This algorithm works bottom-up, in contrast to Maximal Munch, which works top-down. First, the costs of all the children (and grandchildren, etc.) of node  $n$  are found recursively. Then, each tree-pattern (tile kind) is matched against node  $n$ .

Each tile has zero or more *leaves*. In Figure 9.1 the leaves are represented as edges whose bottom ends exit the tile. The leaves of a tile are places where subtrees can be attached.

For each tile  $t$  of cost  $c$  that matches at node  $n$ , there will be zero or more subtrees  $s_i$  corresponding to the leaves of the tile. The cost  $c_i$  of each subtree has already been computed (because the algorithm works bottom-up). So the cost of matching tile  $t$  is just  $c + \sum c_i$ .

Of all the tiles  $t_j$  that match at node  $n$ , the one with the minimum-cost match is chosen, and the (minimum) cost of node  $n$  is thus computed. For example, consider this tree:




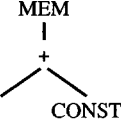
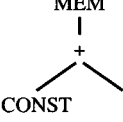
The only tile that matches CONST 1 is an ADDI instruction with cost 1. Similarly, CONST 2 has cost 1. Several tiles match the + node:

Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
	ADD	1	1+1	3
	ADDI	1	1	2
	ADDI	1	1	2

The ADD tile has two leaves, but the ADDI tile has only one leaf. In matching the first ADDI pattern, we are saying “though we computed the cost of tiling CONST 2, we are not going to use that information.” If we choose to use the first ADDI pattern, then CONST 2 will not be the root of any tile, and

its cost will be ignored. In this case, either of the two ADDI tiles leads to the minimum cost for the + node, and the choice is arbitrary. The + node gets a cost of 2.

Now, several tiles match the MEM node:

Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
	LOAD	1	2	3
	LOAD	1	1	2
	LOAD	1	1	2

Either of the last two matches will be optimum.

Once the cost of the root node (and thus the entire tree) is found, the *instruction emission* phase begins. The algorithm is as follows:

Emission(node  $n$ ): for each leaf  $l_i$  of the tile selected at node  $n$ , perform Emission( $l_i$ ). Then emit the instruction matched at node  $n$ .

Emission( $n$ ) does *not* recur on the children of node  $n$ , but on the *leaves of the tile* that matched at  $n$ . For example, after the dynamic programming algorithm finds the optimum cost of the simple tree above, the Emission phase emits

ADDI  $r_1 \leftarrow r_0 + 1$   
 LOAD  $r_1 \leftarrow M[r_1 + 2]$

but no instruction is emitted for any tile rooted at the + node, because this was not a leaf of the tile matched at the root.

## TREE GRAMMARS

For machines with complex instruction sets and several classes of registers and addressing modes, there is a useful generalization of the dynamic programming algorithm. Suppose we make a brain-damaged version of *Jouette* with two classes of registers: *a* registers for addressing, and *d* registers for “data.” The instruction set of the *Schizo-Jouette* machine (loosely based on the Motorola 68000) is shown in Figure 9.4.



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

**FIGURE 9.4.** The *Schizo-Jouette* architecture.

---

The root and leaves of each tile must be marked with  $a$  or  $d$  to indicate which kind of register is implied. Now, the dynamic programming algorithm must keep track, for each node, of the min-cost match as an  $a$  register, *and also* the min-cost match as a  $d$  register.

At this point it is useful to use a context-free grammar to describe the tiles; the grammar will have nonterminals  $s$  (for statements),  $a$  (for expressions calculated into an  $a$  register), and  $d$  (for expressions calculated into a  $d$  register). Section 3.1 describes the use of context-free grammars for source-language syntax; here we use them for quite a different purpose.

The grammar rules for the LOAD, MOVEA, and MOVED instructions might look like this:

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

Such a grammar is highly ambiguous: there are many different parses of the same tree (since there are many different instruction sequences implementing the same expression). For this reason, the parsing techniques described in Chapter 3 are not very useful in this application. However, a generalization of the dynamic programming algorithm works quite well: the minimum-cost match at each node *for each nonterminal of the grammar* is computed.

Though the dynamic programming algorithm is conceptually simple, it becomes messy to write down directly in a general-purpose programming language such as ML. Thus, several tools have been developed. These *code-generator generators* process grammars that specify machine instruction sets; for each rule of the grammar, a cost and an action are specified. The costs are used to find the optimum tiling, and then the actions of the matched rules are used in the *emission* phase.

Like Yacc and Lex, the output of a code-generator generator is usually a program in C or ML that operates a table-driven matching engine with the action fragments (written in C or ML) inserted at the appropriate points.

Such tools are quite convenient. Grammars can specify addressing modes of treelike CISC instructions quite well. A typical grammar for the VAX has 112 rules and 20 nonterminal symbols; and one for the Motorola 68020 has

141 rules and 35 nonterminal symbols. However, instructions that produce more than one result – such as autoincrement instructions on the VAX – are difficult to express using tree patterns.

Code-generator generators are probably overkill for RISC machines. The tiles are quite small, there aren't very many of them, and there is little need for a grammar with many nonterminal symbols.

### **FAST MATCHING**

Maximal Munch and the dynamic programming algorithm must examine, for each node, all the tiles that match at that node. A tile matches if each nonleaf node of the tile is labeled with the same operator (MEM, CONST, etc.) as the corresponding node of the tree.

The naive algorithm for matching would be to examine each tile in turn, checking each node of the tile against the corresponding part of the tree. However, there are better approaches. To match a tile at node  $n$  of the tree, the label at  $n$  can be used in a `case` statement:

```
match( $n$ ) =  
  case label( $n$ )  
  of MEM  $\Rightarrow$  ...  
     BINOP  $\Rightarrow$  ...  
     CONST  $\Rightarrow$  ...
```

Once the clause for one label (such as MEM) is selected, only those patterns rooted in that label remain in consideration. Another `case` statement can use the label of the child of  $n$  to begin distinguishing among those patterns.

The organization and optimization of decision trees for pattern matching is beyond the scope of this book. However, the ML compiler itself uses this algorithm for the translation of data-constructor pattern-matches such as the one shown in Figure 9.3. Thus, the seemingly naive sequence of clauses in function `munchExp` is actually compiled into a sequence of comparisons that never looks twice at the same tree node.

### **EFFICIENCY OF TILING ALGORITHMS**

How expensive are Maximal Munch and dynamic programming?

Let us suppose that there are  $T$  different tiles, and that the average matching tile contains  $K$  nonleaf (labeled) nodes. Let  $K'$  be the largest number of nodes that ever need to be examined to see which tiles match at a given subtree; this is approximately the same as the size of the largest tile. And suppose

that, on the average,  $T'$  different patterns (tiles) match at each tree node. For a typical RISC machine we might expect  $T = 50$ ,  $K = 2$ ,  $K' = 4$ ,  $T' = 5$ .

Suppose there are  $N$  nodes in the input tree. Then Maximal Munch will have to consider matches at only  $N/K$  nodes because, once a “munch” is made at the root, no pattern-matching needs to take place at the nonleaf nodes of the tile.

To find all the tiles that match at one node, at most  $K'$  tree nodes must be examined; but (with a sophisticated decision tree) each of these nodes will be examined only once. Then each of the successful matches must be compared to see if its cost is minimal. Thus, the matching at each node costs  $K' + T'$ , for a total cost proportional to  $(K' + T')N/K$ .

The dynamic programming algorithm must find all the matches at *every* node, so its cost is proportional to  $(K' + T')N$ . However, the constant of proportionality is higher than that of Maximal Munch, since dynamic programming requires two tree-walks instead of one.

Since  $K$ ,  $K'$ , and  $T'$  are constant, the running time of all of these algorithms is linear. In practice, measurements show that these instruction selection algorithms run very quickly compared to the other work performed by a real compiler – even lexical analysis is likely to take more time than instruction selection.

---

**9.2**

---

**CISC MACHINES**

A typical modern RISC machine has

1. 32 registers,
2. only one class of integer/pointer registers,
3. arithmetic operations only between registers,
4. “three-address” instructions of the form  $r_1 \leftarrow r_2 \oplus r_3$ ,
5. load and store instructions with only the M[reg+const] addressing mode,
6. every instruction exactly 32 bits long,
7. one result or effect per instruction.

Many machines designed between 1970 and 1985 are *Complex Instruction Set Computers* (CISC). Such computers have more complicated addressing modes that encode instructions in fewer bits, which was important when computer memories were smaller and more expensive. Typical features found on CISC machines include

1. few registers (16, or 8, or 6),

2. registers divided into different classes, with some operations available only on certain registers,
3. arithmetic operations can access registers or memory through “addressing modes,”
4. “two-address” instructions of the form  $r_1 \leftarrow r_1 \oplus r_2$ ,
5. several different addressing modes,
6. variable-length instructions, formed from variable-length opcode plus variable-length addressing modes,
7. instructions with side effects such as “auto-increment” addressing modes.

Most computer architectures designed since 1990 are RISC machines, but most general-purpose computers installed since 1990 are CISC machines: the Intel 80386 and its descendants (486, Pentium).

The Pentium, in 32-bit mode, has six general-purpose registers, a stack pointer, and a frame pointer. Most instructions can operate on all six registers, but the multiply and divide instructions operate only on the `eax` register. In contrast to the “3-address” instructions found on RISC machines, Pentium arithmetic instructions are generally “2-address,” meaning that the destination register must be the same as the first source register. Most instructions can have either two register operands ( $r_1 \leftarrow r_1 \oplus r_2$ ), or one register and one memory operand, for example  $M[r_1 + c] \leftarrow M[r_1 + c] \oplus r_2$  or  $r_1 \leftarrow r_1 \oplus M[r_2 + c]$ , but not  $M[r_1 + c_1] \leftarrow M[r_1 + c_1] \oplus M[r_2 + c_2]$

We will cut through these Gordian knots as follows:

1. **Few registers:** we continue to generate TEMP nodes freely, and assume that the register allocator will do a good job.
2. **Classes of registers:** The multiply instruction on the Pentium requires that its left operand (and therefore destination) must be the `eax` register. The high-order bits of the result (useless to a Tiger program) are put into register `edx`. The solution is to move the operands and result explicitly; to implement  $t_1 \leftarrow t_2 \times t_3$ :

<code>mov eax, t2</code>	$eax \leftarrow t_2$
<code>mul t3</code>	$eax \leftarrow eax \times t_3; \quad edx \leftarrow \textit{garbage}$
<code>mov t1, eax</code>	$t_1 \leftarrow eax$

This looks very clumsy; but one job that the register allocator performs is to eliminate as many move instructions as possible. If the allocator can assign  $t_1$  or  $t_3$  (or both) to register `eax`, then it can delete one or both of the move instructions.

3. **Two-address instructions:** We solve this problem in the same way as we solve the previous one: by adding extra move instructions. To implement  $t_1 \leftarrow$

$t_2 + t_3$  we produce

mov $t_1, t_2$	$t_1 \leftarrow t_2$
add $t_1, t_3$	$t_1 \leftarrow t_1 + t_3$

Then we hope that the register allocator will be able to allocate  $t_1$  and  $t_2$  to the same register, so that the move instruction will be deleted.

- 4. Arithmetic operations can address memory:** The instruction selection phase turns every TEMP node into a “register” reference. Many of these “registers” will actually turn out to be memory locations. The *spill* phase of the register allocator must be made to handle this case efficiently; see Chapter 11.

The alternative to using memory-mode operands is simply to fetch all the operands into registers before operating and store them back to memory afterwards. For example, these two sequences compute the same thing:

mov $\text{eax}, [\text{ebp} - 8]$	
add $\text{eax}, \text{ecx}$	add $[\text{ebp} - 8], \text{ecx}$
mov $[\text{ebp} - 8], \text{eax}$	

The sequence on the right is more concise (and takes less machine-code space), but *the two sequences are equally fast*. The load, register-register add, and store take 1 cycle each, and the memory-register add takes 3 cycles. On a highly pipelined machine such as the Pentium Pro, simple cycle counts are not the whole story, but the result will be the same: the processor has to perform the load, add, and store, no matter how the instructions specify them.

The sequence on the left has one significant disadvantage: it trashes the value in register  $\text{eax}$ . Therefore, we should try to use the sequence on the right when possible. But the issue here turns into one of register allocation, not of instruction speed; so we defer its solution to the register allocator.

- 5. Several addressing modes:** An addressing mode that accomplishes six things typically takes six steps to execute. Thus, these instructions are often no faster than the multi-instruction sequences they replace. They have only two advantages: they “trash” fewer registers (such as the register  $\text{eax}$  in the previous example), and they have a shorter instruction encoding. With some work, tree-matching instruction selection can be made to select CISC addressing modes, but programs can be just as fast using the simple RISC-like instructions.
- 6. Variable-length instructions:** This is not really a problem for the compiler; once the instructions are selected, it is a trivial (though tedious) matter for the assembler to emit the encodings.
- 7. Instructions with side effects:** Some machines have an “autoincrement” memory fetch instruction whose effect is

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

This instruction is difficult to model using tree patterns, since it produces two results. There are three solutions to this problem:

- (a) Ignore the autoincrement instructions, and hope they go away. This is an increasingly successful solution, as few modern machines have multiple-side-effect instructions.
- (b) Try to match special idioms in an ad hoc way, within the context of a tree pattern-matching code generator.
- (c) Use a different instruction algorithm entirely, one based on DAG-patterns instead of tree-patterns.

Several of these solutions depend critically on the register allocator to eliminate move instructions and to be smart about spilling; see Chapter 11.

---

**9.3**

---

**INSTRUCTION SELECTION FOR THE Tiger COMPILER**

Pattern-matching of “tiles” is quite simple in ML, as shown in Program 9.3. But this figure does not show what to do with each pattern match. It is all very well to print the name of the instruction, but which registers should these instructions use?

In a tree tiled by instruction patterns, the root of each tile will correspond to some intermediate result held in a register. Register allocation is the act of assigning register-numbers to each such node.

The instruction selection phase can simultaneously do register allocation. However, many aspects of register allocation are independent of the particular target-machine instruction set, and it is a shame to duplicate the register-allocation algorithm for each target machine. Thus, register allocation should come either before or after instruction selection.

Before instruction selection, it is not even known which tree nodes will need registers to hold their results, since only the roots of tiles (and not other labeled nodes within tiles) require explicit registers. Thus, register allocation before instruction selection cannot be very accurate. But some compilers do it anyway, to avoid the need to describe machine instructions without the real registers filled in.

We will do register allocation after instruction selection. The instruction selection phase will generate instructions without quite knowing which registers the instructions use.

#### ABSTRACT ASSEMBLY-LANGUAGE INSTRUCTIONS

We will invent a data type for “assembly language instruction without register assignments,” called `Assem.instr`:

```
structure Assem :
sig
  type reg = string
  type temp = Temp.temp
  type label = Temp.label

  datatype instr = OPER of {assem: string,
                           dst: temp list,
                           src: temp list,
                           jump: label list option}
    | LABEL of {assem: string,
               lab: Temp.label}
    | MOVE of {assem: string,
              dst: temp,
              src: temp}

  val format : (temp->string) -> instr -> string
end
```

An `OPER` holds an assembly-language instruction `assem`, a list of operand registers `src`, and a list of result registers `dst`. Either of these lists may be empty. Operations that always fall through to the next instruction have `jump=NONE`; other operations have a list of “target” labels to which they may jump (this list must explicitly include the next instruction if it is possible to fall through to it).

A `LABEL` is a point in a program to which jumps may go. It has an `assem` component showing how the label will look in the assembly-language program, and a `lab` component identifying which label-symbol was used.

A `MOVE` is like an `OPER`, but must perform only data transfer. Then, if the `dst` and `src` temporaries are assigned to the same register, the `MOVE` can later be deleted.

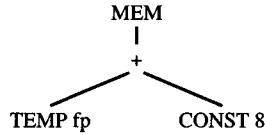
Calling `format(m)(i)` formats an assembly instruction as a string;  $m$  is a function that shows the register assignment (or perhaps just the name) of every `temp`.

**Machine-independence.** The `Assem.instr` data type is *independent* of the chosen target-machine assembly language (though it is tuned for machines with only one class of register). If the target machine is a Sparc, then the



assem strings will be Sparc assembly language. I will use *Jouette* assembly language for illustration.

For example, the tree



could be translated into *Jouette* assembly language as

```

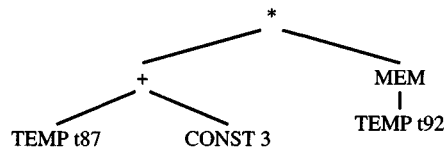
OPER{assem="LOAD 'd0 <- M['s0+8] ",
     dst=[Temp.newtemp()],
     src=[Frame.FP],
     jump=NONE}
  
```

This instruction needs some explanation. The actual assembly language of *Jouette*, after register allocation, might be

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

assuming that register  $r_{27}$  is the frame pointer  $fp$  and that the register allocator decided to assign the new temp to register  $r_1$ . But the Assem instruction does not know about register assignments; instead, it just talks of the sources and destination of each instruction. This LOAD instruction has one source register, which is referred to as 's0; and one destination register, referred to as 'd0.

Another example will be useful. The tree



could be translated as

<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

where t908, t909, and t910 are temporaries newly chosen by the instruction selector.

After register allocation the assembly language might look like:

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

The string of an *instr* may refer to *source registers* 's0, 's1, ... 's(*k* − 1), and *destination registers* 'd0, 'd1, etc. Jumps are *OPER* instructions that refer to labels 'j0, 'j1, etc. Conditional jumps, which may branch away or fall through, typically have two labels in the *jump* list but refer to only one of them in the *assem* string.

**Two-address instructions.** Some machines have arithmetic instructions with two operands, where one of the operands is both a source and a destination. The instruction *add t1, t2*, which has the effect of  $t_1 \leftarrow t_1 + t_2$ , can be described as

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

where 's0 is implicitly, but not explicitly, mentioned in the *assem* string.

## PRODUCING ASSEMBLY INSTRUCTIONS

Now it is a simple matter to write the right-hand sides of the pattern-matching clauses that “munch” *Tree* expressions into *Assem* instructions. I will show some examples from the *Jouette* code generator, but the same ideas apply to code generators for real machines.

The functions *munchStm* and *munchExp* will produce *Assem* instructions, bottom-up, as side effects. *MunchExp* returns the temporary in which the result is held.

```
munchExp: Tree.exp → Temp.temp
munchStm: Tree.stm → unit
```

The “actions” of the *munchExp* clauses of Program 9.3 can be written as shown in Programs 9.5 and 9.6.

The *emit* function just accumulates a list of instructions to be returned later, as shown in Program 9.7.

## PROCEDURE CALLS

Procedure calls are represented by *EXP*(*CALL*(*f*, *args*)), and function calls by *MOVE*(*TEMP t*, *CALL*(*f*, *args*)). These trees can be matched by tiles such as

```

structure A = Assem
fun munchStm(T.SEQ(a,b)) = (munchStm a; munchStm b)
  | munchStm(T.MOVE(T.MEM(T.BINOP(T.PLUS,e1,T.CONST i)),e2)) =
    emit(A.OPER{assem="STORE M['s0+" ^ int i ^ "]" <- 's1\n",
            src=[munchExp e1, munchExp e2],
            dst=[], jump=NONE})
  | munchStm(T.MOVE(T.MEM(T.BINOP(T.PLUS,T.CONST i,e1)),e2)) =
    emit(A.OPER{assem="STORE M['s0+" ^ int i ^ "]" <- 's1\n",
            src=[munchExp e1, munchExp e2],
            dst=[], jump=NONE})
  | munchStm(T.MOVE(T.MEM(e1),T.MEM(e2))) =
    emit(A.OPER{assem="MOVE M['s0] <- M['s1\n",
            src=[munchExp e1, munchExp e2],
            dst=[], jump=NONE})
  | munchStm(T.MOVE(T.MEM(T.CONST i),e2)) =
    emit(A.OPER{assem="STORE M[r0+" ^ int i ^ "]" <- 's0\n",
            src=[munchExp e2], dst=[], jump=NONE})
  | munchStm(T.MOVE(T.MEM(e1),e2)) =
    emit(A.OPER{assem="STORE M['s0] <- 's1\n",
            src=[munchExp e1, munchExp e2],
            dst=[], jump=NONE})
  | munchStm(T.MOVE(T.TEMP i, e2)) =
    emit(A.OPER{assem="ADD 'd0 <- 's0 + r0\n",
            src=[munchExp e2],
            dst=[i], jump=NONE})
  | munchStm(T.LABEL lab) =
    emit(A.LABEL{assem=lab ^ ":\n", lab=lab})

```

---

**PROGRAM 9.5.** Assem-instructions for munchStm.

---

```

munchStm(T.EXP(T.CALL(e,args))) =
  emit(A.OPER{assem="CALL 's0\n",
            src=munchExp(e)::munchArgs(0,args),
            dst=calldefs,
            jump=NONE})

```

In this example, `munchArgs` generates code to move all the arguments to their correct positions, in outgoing parameter registers and/or in memory. The integer parameter to `munchArgs` is *i* for the *i*th argument; `munchArgs` will recur with *i* + 1 for the next argument, and so on.

What `munchArgs` returns is a list of all the temporaries that are to be passed to the machine's `CALL` instruction. Even though these temps are never written explicitly in assembly language, they should be listed as “sources” of the instruction, so that liveness analysis (Chapter 10) can see that their values need to be kept up to the point of call.

---

### 9.3. INSTRUCTION SELECTION FOR THE TIGER COMPILER

---

```
and result(gen) = let val t = Temp.newtemp() in gen t; t end

and munchExp(T.MEM(T.BINOP(T.PLUS,e1,T.CONST i))) =
  result(fn r => emit(A.OPER
    {assem="LOAD 'd0 <- M['s0+" ^ int i ^ "]"\\n",
      src=[munchExp e1], dst=[r], jump=NONE}))
| munchExp(T.MEM(T.BINOP(T.PLUS,T.CONST i,e1))) =
  result(fn r => emit(A.OPER
    {assem="LOAD 'd0 <- M['s0+" ^ int i ^ "]"\\n",
      src=[munchExp e1], dst=[r], jump=NONE}))
| munchExp(T.MEM(T.CONST i)) =
  result(fn r => emit(A.OPER
    {assem="LOAD 'd0 <- M[r0+" ^ int i ^ "]"\\n",
      src=[], dst=[r], jump=NONE}))
| munchExp(T.MEM(e1)) =
  result(fn r => emit(A.OPER
    {assem="LOAD 'd0 <- M['s0+0]\\n",
      src=[munchExp e1], dst=[r], jump=NONE}))
| munchExp(T.BINOP(T.PLUS,e1,T.CONST i)) =
  result(fn r => emit(A.OPER
    {assem="ADDI 'd0 <- 's0+" ^ int i ^ "\\n",
      src=[munchExp e1], dst=[r], jump=NONE}))
| munchExp(T.BINOP(T.PLUS,T.CONST i,e1)) =
  result(fn r => emit(A.OPER
    {assem="ADDI 'd0 <- 's0+" ^ int i ^ "\\n",
      src=[munchExp e1], dst=[r], jump=NONE}))
| munchExp(T.CONST i) =
  result(fn r => emit(A.OPER
    {assem="ADDI 'd0 <- r0+" ^ int i ^ "\\n",
      src=[], dst=[r], jump=NONE}))
| munchExp(T.BINOP(T.PLUS,e1,e2)) =
  result(fn r => emit(A.OPER
    {assem="ADD 'd0 <- 's0+'s1\\n",
      src=[munchExp e1, munchExp e2], dst=[r],
      jump=NONE}))
| munchExp(T.TEMP t) = t
```

---

**PROGRAM 9.6.** Assem-instructions for munchExp.

---

A CALL is expected to “trash” certain registers – the caller-save registers, the return-address register, and the return-value register. This list of `calldefs` should be listed as “destinations” of the CALL, so that the later phases of the compiler know that something happens to them here.

In general, any instruction that has the side effect of writing to another register requires this treatment. For example, the Pentium’s multiply instruction writes to register `edx` with useless high-order result bits, so `edx` and `eax` are

```
fun codegen (frame) (stm: Tree.stm) : Assem.instr list =
let val ilist = ref (nil: A.instr list)
    fun emit x= ilist := x :: !ilist
    fun result(gen) = let val t = Temp.newtemp() in gen t; t end

    fun munchStm ...

    and munchExp ...

in munchStm stm;
    rev(!ilist)
end
```

---

**PROGRAM 9.7.** The codegen function.

---

both listed as destinations of this instruction. (The high-order bits can be very useful for programs written in assembly language to do multiprecision arithmetic, but most programming languages do not support any way to access them.)

### IF THERE'S NO FRAME POINTER

In a stack frame layout such as the one shown in Figure 6.2, the frame pointer points at one end of the frame and the stack pointer points at the other. At each procedure call, the stack pointer register is copied to the frame pointer register, and then the stack pointer is incremented by the size of the new frame.

Many machines' calling conventions do not use a frame pointer. Instead, the "virtual frame pointer" is always equal to stack pointer plus frame size. This saves time (no copy instruction) and space (one more register usable for other purposes). But our Translate phase has generated trees that refer to this fictitious frame pointer. The codegen function must replace any reference to  $FP+k$  with  $SP+k+fs$ , where  $fs$  is the frame size. It can recognize these patterns as it munches the trees.

However, to replace them it must know the value of  $fs$ , which cannot yet be known because register allocation is not known. Assuming the function  $f$  is to be emitted at label L14 (for example), codegen can just put `sp+L14_framesize` in its assembly instructions and hope that the prologue for  $f$  (generated by `Frame.procEntryExit3`) will include a definition of the assembly-language constant `L14_framesize`. Codegen is passed the `frame` argument (Program 9.7) so that it can learn the name L14.

Implementations that have a “real” frame pointer won’t need this hack and can ignore the `frame` argument to `codegen`. But why would an implementation use a real frame pointer when it wastes time and space to do so? The answer is that this permits the frame size to grow and shrink even after it is first created; some languages have permitted dynamic allocation of arrays within the stack frame (e.g., using `alloca` in C). Calling-convention designers now tend to avoid dynamically adjustable frame sizes, however.

---

**PROGRAM INSTRUCTION SELECTION**

---

```
signature CODEGEN =  
sig  
  structure Frame : FRAME  
  val codegen : Frame.frame -> Tree.stm -> Assem.instr list  
end
```

Implement the translation to Assem-instructions for your favorite instruction set (let  $\mu$  stand for *Sparc*, *Mips*, *Alpha*, *Pentium*, etc.) using Maximal Munch. If you would like to generate code for a RISC machine, but you have no RISC computer on which to test it, you may wish to use SPIM (a MIPS simulator implemented by James Larus), described on the Web page for this book.

First write the structure  `$\mu$ Gen`: `CODEGEN` implementing the “Maximal Munch” translation algorithm from IR trees to the `Assem` data structure.

Use the `Canon` module (described in Chapter 8) to simplify the trees before applying your `Codegen` module to them. Use the `format` function to translate the resulting `Assem` trees to  $\mu$  assembly language. Since you won’t have done register assignment, just pass `Temp.makestring` to `format` as the translation function from temporaries to strings.

This will produce “assembly” language that does not use register names at all: the instructions will use names such as `t3`, `t283`, and so on. But some of these temps are the “built-in” ones created by the `Frame` module to stand for particular machine registers (see page 155), such as `Frame.FP`. The assembly language will be easier to read if these registers appear with their natural names (e.g., `fp` instead of `t1`).

The `Frame` module must provide a mapping from the special temps to their names, and nonspecial temps to `NONE`:

```
signature FRAME =
sig
  :
  val tempMap: register Temp.Table.table
end
```

Then, for the purposes of displaying your assembly language prior to register allocation, make a new `temp→string` function that first tries `Frame.tempMap`, and if that returns `NONE`, resorts to `Temp.makestring`.

### REGISTER LISTS

Make the following lists of registers; for each register, you will need a `Frame.register` (that is, string) for its assembly-language representation and a `Temp.temp` for referring to it in `Tree` and `Assem` data structures.

`specialregs` a list of  $\mu$  registers used to implement “special” registers such as RV and FP and also the stack pointer SP, the return-address register RA, and (on some machines) the zero register ZERO. Some machines may have other special registers;

`argregs` a list of  $\mu$  registers in which to pass outgoing arguments (including the static link);

`calleesaves` a list of  $\mu$  registers that the called procedure (callee) must preserve unchanged (or save and restore);

`callersaves` a list of  $\mu$  registers that the callee may trash.

The four lists of registers must not overlap, and must include any register that might show up in `Assem` instructions. These lists are not exported through the `FRAME` signature, but they are useful internally for both `Frame` and `Codegen` – for example, to implement `munchArgs` and to construct the `calldefs` list.

Implement the `procEntryExit2` function of the `FRAME` signature.

```
structure Frame =
struct
  :
  val procEntryExit2 : frame * Assem.instr list ->
                                     Assem.instr list
end
```

This function appends a “sink” instruction to the function body to tell the register allocator that certain registers are live at procedure exit. In the case of the *Jouette* machine, this is simply:

---

## FURTHER READING

---

```
fun procEntryExit2(frame, body) =  
  body @  
  [A.OPER{assem=" ",  
    src=[ZERO, RA, SP]@calleesaves,  
    dst=[], jump=SOME[]}]}
```

meaning that the temporaries *zero*, *return-address*, *stack-pointer*, and all the callee-saves registers are still live at the end of the function. Having *zero* live at the end means that it is live throughout, which will prevent the register allocator from trying to use it for some other purpose. The same trick works for any other special registers the machine might have.

Files available in `$TIGER/chap9` include:

`canon.sml` Canonicalization and trace-generation.

`assem.sml` The Assem module.

`main.sml` A Main module that you may wish to adapt.

Your code generator will handle only the body of each procedure or function, but not the procedure entry/exit sequences. Use a “scaffold” version of `Frame.procEntryExit3` function:

```
fun procEntryExit3(FRAME{name,params,locals}, body) =  
{prolog = "PROCEDURE " ^ Symbol.name name ^ "\n",  
  body = body,  
  epilg = "END " ^ Symbol.name name ^ "\n"}
```

---

## FURTHER READING

---

Cattell [1980] expressed machine instructions as tree patterns, invented the Maximal Munch algorithm for instruction selection, and built a *code generator generator* to produce an instruction-selection function from a tree-pattern description of an instruction set. Glanville and Graham [1978] expressed the tree patterns as productions in LR(1) grammars, which allows the Maximal Munch algorithm to use multiple nonterminal symbols to represent different classes of registers and addressing modes. But grammars describing instruction sets are inherently ambiguous, leading to problems with the LR(1) approach; Aho et al. [1989] use dynamic programming to parse the tree grammars, which solves the ambiguity problem, and describe the *Twig* automatic code-generator generator. The dynamic programming can be done at compiler-construction time instead of code-generation time [Pelegri-Llopert



and Graham 1988]; using this technique, the *BURG* tool [Fraser et al. 1992] has an interface similar to Twig's but generates code much faster.

---

## EXERCISES

---

- 9.1** For each of the following expressions, draw the tree and generate *Jouette*-machine instructions using Maximal Munch. Circle the tiles (as in Figure 9.2), but number them *in the order that they are munched*, and show the sequence of *Jouette* instructions that results.

- MOVE(MEM(++(CONST<sub>1000</sub>, MEM(TEMP<sub>x</sub>)), TEMP<sub>fp</sub>)), CONST<sub>0</sub>)
- BINOP(MUL, CONST<sub>5</sub>, MEM(CONST<sub>100</sub>))

- \*9.2** Consider a machine with the following instruction:

mult const1(src1), const2(src2), dst3

$r_3 \leftarrow M[r_1 + \text{const}_1] * M[r_2 + \text{const}_2]$

On this machine,  $r_0$  is always 0, and  $M[1]$  always contains 1.

- Draw all the tree patterns corresponding to this instruction (and its special cases).
- Pick **one** of the bigger patterns and show how to write it as an ML pattern match, with the Tree representation used for the Tiger compiler.

- 9.3** The *Jouette* machine has control-flow instructions as follows:

BRANCHGE if  $r_i \geq 0$  goto  $L$

BRANCHLT if  $r_i < 0$  goto  $L$

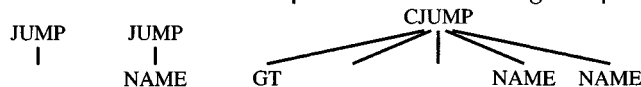
BRANCHEQ if  $r_i = 0$  goto  $L$

BRANCHNE if  $r_i \neq 0$  goto  $L$

JUMP goto  $r_i$

where the JUMP instruction goes to an address contained in a register.

Use these instructions to implement the following tree patterns:



Assume that a CJUMP is always followed by its false label. Show the best way to implement each pattern; in some cases you may need to use more than one instruction or make up a new temporary. How do you implement CJUMP(GT, ...) without a BRANCHGT instruction?