regarding the next use and liveness of x, y, and z.[4]

2. In the symbol table, set x to "not live" and "no next use."

3. In the symbol table, set y and z to "live" and the next uses of y and z to i. Note that the order of steps (2) and (3) may not be interchanged because x may be y or z.

If three-address statement i is of the form x := y or x := *op* y, the steps are the same as above, ignoring z.

Storage for Temporary Names

Although it may be useful in an optimizing compiler to create a distinct name each time a temporary is needed (see Chapter 10 for justification), space has to be allocated to hold the values of these temporaries. The size of the field for temporaries in the general activation record of Section 7.2 grows with the number of temporaries.

We can, in general, pack two temporaries into the same location if they are not live simultaneously. Since almost all temporaries are defined and used within basic blocks, next-use information can be applied to pack temporaries. For temporaries that are used across blocks, Chapter 10 discusses the data-flow analysis needed to compute liveness.

We can allocate storage locations for temporaries by examining each in turn and assigning a temporary to the first location in the field for temporaries that does not contain a live temporary. If a temporary cannot be assigned to any previously created location, add a new location to the data area for the current procedure. In many cases, temporaries can be packed into registers rather than memory locations, as in the next section.

For example, the six temporaries in the basic block (9.1) can be packed into two locations. These locations correspond to t_1 and t_2 in:

```
t1 := a * a
t2 := a * b
t2 := 2 * t2
t1 := t1 + t2
t2 := b * b
t1 := t1 + t2
```

9.6 A SIMPLE CODE GENERATOR

The code-generation strategy in this section generates target code for a sequence of three-address statement. It considers each statement in turn, remembering if any of the operands of the statement are currently in registers, and taking advantage of that fact if possible. For simplicity, we assume that

[4] If x is not live, then this statement can be deleted; such transformations are considered in Section 9.8.

for each operator in a statement there is a corresponding target-language operator. We also assume that computed results can be left in registers as long as possible, storing them only (a) if their register is needed for another computation or (b) just before a procedure call, jump, or labeled statement.[5]

Condition (b) implies that everything must be stored just before the end of a basic block.[6] The reason we must do so is that, after leaving a basic block, we may be able to go to several different blocks, or we may go to one particular block that can be reached from several others. In either case, we cannot, without extra effort, assume that a datum used by a block appears in the same register no matter how control reached that block. Thus, to avoid a possible error, our simple code-generation algorithm stores everything when moving across basic-block boundaries as well as when procedure calls are made. Later we consider ways to hold some data in registers across block boundaries.

We can produce reasonable code for a three-address statement `a := b + c` if we generate the single instruction `ADD Rj, Ri` with cost one, leaving the result `a` in register `Ri`. This sequence is possible only if register `Ri` contains `b`, `Rj` contains `c`, and `b` is not live after the statement; that is, `b` is not used after the statement.

If `Ri` contains `b` but `c` is in a memory location (called `c` for convenience), we can generate the sequence

```
ADD    c, Ri          cost = 2
```

or

```
MOV    c, Rj
ADD    Rj, Ri         cost = 3
```

provided `b` is not subsequently live. The second sequence becomes attractive if this value of `c` is subsequently used, as we can then take its value from register `Rj`. There are many more cases to consider, depending on where `b` and `c` are currently located and depending on whether the current value of `b` is subsequently used. We must also consider the cases where one or both of `b` and `c` is a constant. The number of cases that need to be considered further increases if we assume that the operator `+` is commutative. Thus, we see that code generation involves examining a large number of cases, and which case should prevail depends on the context in which a three-address statement is seen.

---

[5] However, to produce a *symbolic dump*, which makes available the values of memory locations and registers in terms of the source program's names for these values, it may be more convenient to have programmer-defined variables (but not necessarily compiler-generated temporaries) stored immediately upon calculation, should a program error suddenly cause a precipitous interrupt and exit.

[6] Note we are not assuming that the quadruples were actually partitioned into basic blocks by the compiler; the notion of a basic block is useful conceptually in any event.

Register and Address Descriptors

The code-generation algorithm uses descriptors to keep track of register contents and addresses for names.

1. A register descriptor keeps track of what is currently in each register. It is consulted whenever a new register is needed. We assume that initially the register descriptor shows that all registers are empty. (If registers are assigned across blocks, this would not be the case.) As the code generation for the block progresses, each register will hold the value of zero or more names at any given time.

2. An address descriptor keeps track of the location (or locations) where the current value of the name can be found at run time. The location might be a register, a stack location, a memory address, or some set of these, since when copied, a value also stays where it was. This information can be stored in the symbol table and is used to determine the accessing method for a name.

A Code-Generation Algorithm

The code-generation algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form `x := y` *op* `z` we perform the following actions:

1. Invoke a function *getreg* to determine the location `L` where the result of the computation `y` *op* `z` should be stored. `L` will usually be a register, but it could also be a memory location. We shall describe *getreg* shortly.

2. Consult the address descriptor for `y` to determine `y'`, (one of) the current location(s) of `y`. Prefer the register for `y'` if the value of `y` is currently both in memory and a register. If the value of `y` is not already in `L`, generate the instruction `MOV y',L` to place a copy of `y` in `L`.

3. Generate the instruction `OP z',L` where `z'` is a current location of `z`. Again, prefer a register to a memory location if `z` is in both. Update the address descriptor of `x` to indicate that `x` is in location `L`. If `L` is a register, update its descriptor to indicate that it contains the value of `x`, and remove `x` from all other register descriptors.

4. If the current values of `y` and/or `z` have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of `x := y` *op* `z`, those registers no longer will contain `y` and/or `z`, respectively.

If the current three-address statement has a unary operator, the steps are analogous to those above, and we omit the details. An important special case is a three-address statement `x := y`. If `y` is in a register, simply change the register and address descriptors to record that the value of `x` is now found only in the register holding the value of `y`. If `y` has no next use and is not

live on exit from the block, the register no longer holds the value of y.

If y is only in memory, we could in principle record that the value of x is in the location of y, but this option would complicate our algorithm, since we could not then change the value of y without preserving the value of x. Thus, if y is in memory we use *getreg* to find a register in which to load y and make that register the location of x.

Alternatively, we can generate a MOV y, x instruction, which would be preferable if the value of x has no next use in the block. It is worth noting that most, if not all, copy instructions will be eliminated if we use the block-improving and copy-propagation algorithm of Chapter 10.

Once we have processed all three-address statements in the basic block, we store, by MOV instructions, those names that are live on exit and not in their memory locations. To do this we use the register descriptor to determine what names are left in registers, the address descriptor to determine that the same name is not already in its memory location, and the live variable information to determine whether the name is to be stored. If no live-variable information has been computed by data-flow analysis among blocks, we must assume all user-defined names are live at the end of the block.

The Function *getreg*

The function *getreg* returns the location L to hold the value of x for the assignment x := y *op* z. A great deal of effort can be expended in implementing this function to produce a perspicacious choice for L. In this section, we discuss a simple, easy-to-implement scheme based on the next-use information collected in the last section.

1. If the name y is in a register that holds the value of no other names (recall that copy instructions such as x := y could cause a register to hold the value of two or more variables simultaneously), and y is not live and has no next use after execution of x := y *op* z, then return the register of y for L. Update the address descriptor of y to indicate that y is no longer in L.

2. Failing (1), return an empty register for L if there is one.

3. Failing (2), if x has a next use in the block, or *op* is an operator, such as indexing, that requires a register, find an occupied register R. Store the value of R into a memory location (by MOV R, M) if it is not already in the proper memory location M, update the address descriptor for M, and return R. If R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable occupied register might be one whose datum is referenced furthest in the future, or one whose value is also in memory. We leave the exact choice unspecified, since there is no one proven best way to make the selection.

4. If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L.

A more sophisticated *getreg* function would also consider the subsequent uses of x and the commutativity of the operator *op* in determining the register to hold the value of x. We leave such extensions of *getreg* as exercises.

Example 9.5. The assignment `d := (a-b) + (a-c) + (a-c)` might be translated into the following three-address code sequence

```
t := a - b
u := a - c
v := t + u
d := v + u
```

with d live at the end. The code-generation algorithm given above would produce the code sequence shown in Fig. 9.10 for this three-address statement sequence. Shown alongside are the values of the register and address descriptors as code generation progresses. Not shown in the address descriptor is the fact that a, b, and c are always in memory. We also assume that t, u and v, being temporaries, are not in memory unless we explicitly store their values with a MOV instruction.

STATEMENTS	CODE GENERATED	REGISTER DESCRIPTOR	ADDRESS DESCRIPTOR
		registers empty	
t := a - b	MOV a, R0	R0 contains t	t in R0
	SUB b, R0		
u := a - c	MOV a, R1	R0 contains t	t in R0
	SUB c, R1	R1 contains u	u in R1
v := t + u	ADD R1, R0	R0 contains v	u in R1
		R1 contains u	v in R0
d := v + u	ADD R1, R0	R0 contains d	d in R0
	MOV R0, d		d in R0 and memory

Fig. 9.10. Code sequence.

The first call of *getreg* returns R0 as the location in which to compute t. Since a is not in R0, we generate instructions MOV a, R0 and SUB b, R0. We now update the register descriptor to indicate that R0 contains t.

Code generation proceeds in this manner until the last three-address statement d := v + u has been processed. Note that R1 becomes empty because u has no next use. We then generate MOV R0, d to store the live variable d at the end of the block.

The cost of the code generated in Fig. 9.10 is 12. We could reduce this to

11 by generating `MOV R0,R1` immediately after the first instruction and removing the instruction `MOV a,R1`, but to do so requires a more sophisticated code-generation algorithm. The reason for the savings is that it is cheaper to load `R1` from `R0` than from memory. □

Generating Code for Other Types of Statements

Indexing and pointer operations in three-address statements are handled in the same manner as binary operations. The table in Fig. 9.11 shows the code sequences generated for the indexed assignment statements `a := b[i]` and `a[i] := b`, assuming `b` is statically allocated.

STATEMENT	i IN REGISTER Ri		i IN MEMORY Mi		i IN STACK	
	CODE	COST	CODE	COST	CODE	COST
a:=b[i]	MOV b(Ri),R	2	MOV Mi,R MOV b(R),R	4	MOV Si(A),R MOV b(R),R	4
a[i]:=b	MOV b,a(Ri)	3	MOV Mi,R MOV b,a(R)	5	MOV Si(A),R MOV b,a(R)	5

Fig. 9.11. Code sequences for indexed assignments.

The current location of `i` determines the code sequence. Three cases are covered depending on whether `i` is in register `Ri`, whether `i` is in memory location `Mi`, or whether `i` is on the stack at offset `Si` and the pointer to the activation record for `i` is in register `A`. The register `R` is the register returned when the function *getreg* is called. For the first assignment, we would prefer to leave `a` in register `R` if `a` has a next use in the block and register `R` is available. In the second assignment, we assume `a` is statically allocated.

The table in Fig. 9.12 shows the code sequences generated for the pointer assignments `a := *p` and `*p := a`. Here, the current location of `p` determines the code sequence.

STATEMENT	p IN REGISTER Rp		p IN MEMORY Mp		p IN STACK	
	CODE	COST	CODE	COST	CODE	COST
a:=*p	MOV *Rp,a	2	MOV Mp,R MOV *R,R	3	MOV Sp(A),R MOV *R,R	3
*p:=a	MOV a,*Rp	2	MOV Mp,R MOV a,*R	4	MOV a,R MOV R,*Sp(A)	4

Fig. 9.12. Code sequences for pointer assignments.

Three cases are covered depending on whether `p` is initially in register `Rp`, whether `p` is in memory location `Mp`, or whether `p` is on the stack at offset `Sp`

and the pointer to the activation record for `p` is in register `A`. The register `R` is the register returned when the function *getreg* is called. In the second assignment, we assume `a` is statically allocated.

Conditional Statements

Machines implement conditional jumps in one of two ways. One way is to branch if the value of a designated register meets one of the six conditions: negative, zero, positive, nonnegative, nonzero, and nonpositive. On such a machine a three-address statement such as `if x<y goto z` can be implemented by subtracting `y` from `x` in register `R`, and then jumping to `z` if the value in register `R` is negative.

A second approach, common to many machines, uses a set of *condition codes* to indicate whether the last quantity computed or loaded into a register is negative, zero, or positive. Often a compare instruction (`CMP` in our machine) has the desirable property that it sets the condition code without actually computing a value. That is, `CMP x, y` sets the condition code to positive if `x>y`, and so on. A conditional-jump machine instruction makes the jump if a designated condition $<$, $=$, $>$, $\leq$, $\neq$, or $\geq$ is met. We use the instruction `CJ<= z` to mean "jump to `z` if the condition code is negative or zero." For example, `if x<y goto z` could be implemented by

```
CMP   x, y
CJ<   z
```

If we are generating code for a machine with condition codes it is useful to maintain a condition-code descriptor as we generate code. This descriptor tells the name that last set the condition code, or the pair of names compared, if the condition code was last set that way. Thus we could implement

```
x := y + z
if x < 0 goto z
```

by

```
MOV   y, R0
ADD   z, R0
MOV   R0, x
CJ<   z
```

if we were aware that the condition code was determined by `x` after `ADD z, R0`.

9.7 REGISTER ALLOCATION AND ASSIGNMENT

Instructions involving only register operands are shorter and faster than those involving memory operands. Therefore, efficient utilization of registers is important in generating good code. This section presents various strategies for deciding what values in a program should reside in registers (register

allocation) and in which register each value should reside (register assignment).

One approach to register allocation and assignment is to assign specific values in an object program to certain registers. For example, a decision can be made to assign base addresses to one group of registers, arithmetic computation to another, the top of the run-time stack to a fixed register, and so on.

This approach has the advantage that it simplifies the design of a compiler. Its disadvantage is that, applied too strictly, it uses registers inefficiently; certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated. Nevertheless, it is reasonable in most computing environments to reserve a few registers for base registers, stack pointers and the like, and to allow the remaining registers to be used by the compiler as it sees fit.

Global Register Allocation

The code-generation algorithm in Section 9.6 used registers to hold values for the duration of a single basic block. However, all live variables were stored at the end of each block. To save some of these stores and corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (*globally*). Since programs spend most of their time in inner loops, a natural approach to global register assignment is to try to keep a frequently used value in a fixed register throughout a loop. For the time being, assume that we know the loop structure of a flow graph, and that we know what values computed in a basic block are used outside that block. The next chapter covers techniques for computing this information.

One strategy for global register allocation is to assign some fixed number of registers to hold the most active values in each inner loop. The selected values may be different in different loops. Registers not already allocated may be used to hold values local to one block as in Section 9.6. This approach has the drawback that the fixed number of registers is not always the right number to make available for global register allocation. Yet the method is simple to implement and it has been used in Fortran H, the optimizing Fortran compiler for the IBM-360 series machines (Lowry and Medlock [1969]).

In languages like C and Bliss, a programmer can do some register allocation directly by using register declarations to keep certain values in registers for the duration of a procedure. Judicious use of register declarations can speed up many programs, but a programmer should not do register allocation without first profiling the program.

Usage Counts

A simple method for determining the savings to be realized by keeping variable x in a register for the duration of loop L is to recognize that in our machine model we save one unit of cost for each reference to x if x is in a

register. However, if we use the approach of the previous section to generate code for a block, there is a good chance that after x has been computed in a block it will remain in a register if there are subsequent uses of x in that block. Thus we count a savings of one for each use of x in loop L that is not preceded by an assignment to x in the same block. We also save two units if we can avoid a store of x at the end of a block. Thus if x is allocated a register, count a savings of two for each block of L for which x is live on exit and in which x is assigned a value.

On the debit side, if x is live on entry to the loop header, we must load x into its register just before entering loop L. This load costs two units. Similarly, for each exit block B of loop L at which x is live on entry to some successor of B outside of L, we must store x at a cost of two. However, on the assumption that the loop is iterated many times, we may neglect these debits since they occur only once each time we enter the loop. Thus an approximate formula for the benefit to be realized from allocating a register to x within loop L is:

$$\sum_{\text{blocks } B \text{ in } L} (use(x, B) + 2 * live(x, B)) \tag{9.4}$$

where $use(x, B)$ is the number of times x is used in B prior to any definition of x; $live(x, B)$ is 1 if x is live on exit from B and is assigned a value in B, and $live(x, B)$ is 0 otherwise. Note that (9.4) is approximate, because not all blocks in a loop are executed with equal frequency and also because (9.4) was based on the assumption that a loop is iterated "many" times. On other machines a formula analogous to (9.4), but possibly quite different from it, would have to be developed.

Example 9.6. Consider the basic blocks in the inner loop depicted in Fig. 9.13, where jump and conditional jump statements have been omitted. Assume registers R0, R1, and R2 are allocated to hold values throughout the loop. Variables live on entry into and on exit from each block are shown in Fig. 9.13 for convenience, immediately above and below each block, respectively. There are some subtle points about live variables that we address in Chapter 10. For example, notice that both e and f are live at the end of B_1, but of these, only e is live on entry to B_2 and only f on entry to B_3. In general, the variables live at the end of a block are the union of those live at the beginning of each of its successor blocks.

To evaluate (9.4) for x = a we observe that a is live on exit from B_1 and is assigned a value there, but is not live on exit from B_2, B_3, or B_4. Thus, $\sum_{B \text{ in } L} 2*live(\mathtt{a}, B) = 2$. Also, $use(\mathtt{a}, B_1) = 0$, since a is defined in B_1 before any use. Also, $use(\mathtt{a}, B_2) = use(\mathtt{a}, B_3) = 1$ and $use(\mathtt{a}, B_4) = 0$. Thus, $\sum_{B \text{ in } L} use(\mathtt{a}, B) = 2$. Hence the value of (9.4) for x = a is 4. That is, four units of cost can be saved by selecting a for one of the global registers. The values of (9.4) for b, c, d, e, and f are 6, 3, 6, 4, and 4, respectively.

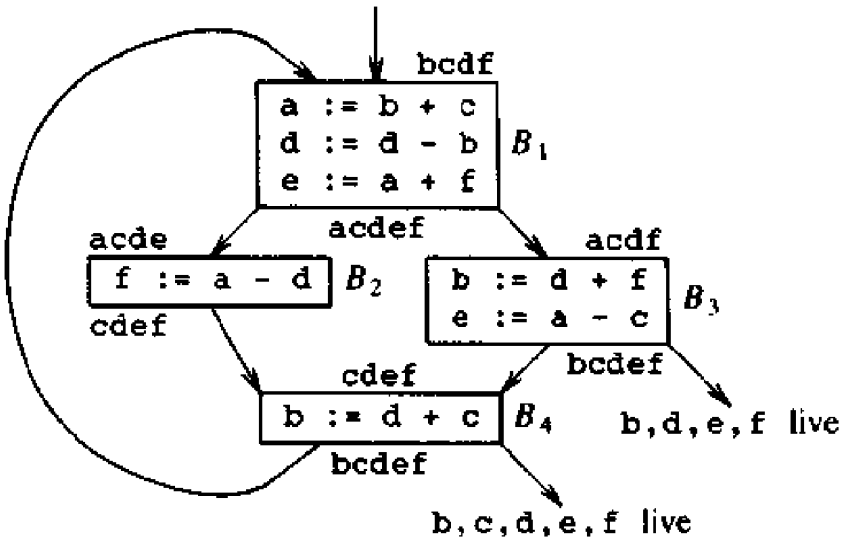

Fig. 9.13. Flow graph of an inner loop.

Thus we may select `a`, `b`, and `d`, for registers `R0`, `R1`, and `R2`, respectively. Using `R0` for `e` or `f` instead of `a` would be another choice with the same apparent benefit. Fig. 9.14 shows the assembly code generated from Fig. 9.13, assuming that the strategy of Section 9.6 is used to generate code for each block. We do not show the generated code for the omitted conditional or unconditional jumps that end each block in Fig. 9.13, and we therefore do not show the generated code as a single stream, as it would appear in practice. It is worth noting that, if we did not adhere strictly to our strategy of reserving `R0`, `R1`, and `R2`, we could use

```
SUB  R2, R0
MOV  R0, f
```

for B_2, saving a unit since `a` is not live on exit from B_2. A similar saving could be realized at B_3. □

Register Assignment for Outer Loops

Having assigned registers and generated code for inner loops, we may apply the same idea to progressively larger loops. If an outer loop L_1 contains an inner loop L_2, the names allocated registers in L_2 need not be allocated registers in $L_1 - L_2$. However, if name x is allocated a register in loop L_1 but not L_2, we must store x on entrance to L_2 and load x if we leave L_2 and enter a block of $L_1 - L_2$. Similarly, if we choose to allocate x a register in L_2 but not L_1 we must load x on entrance to L_2 and store x on exit from L_2. We leave as an exercise the derivation of a criterion for selecting names to be allocated registers in an outer loop L, given that choices have already been made for all loops nested within L.

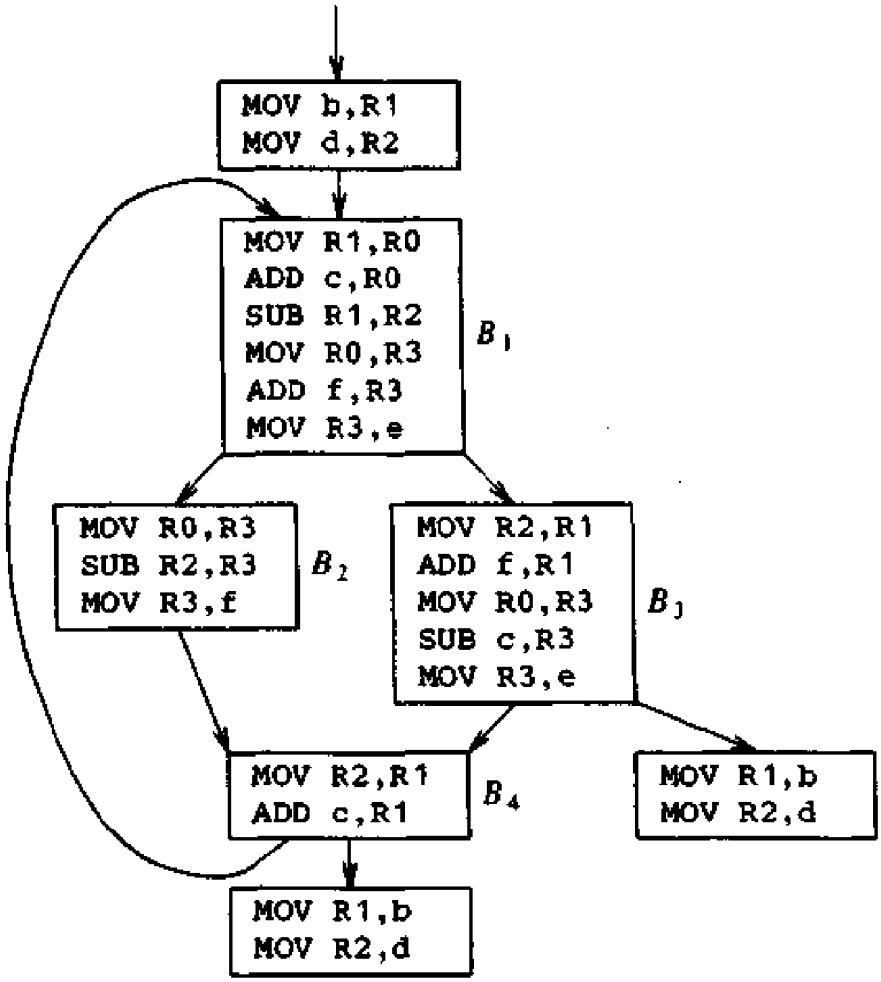

Fig. 9.14. Code sequence using global register assignment.

Register Allocation by Graph Coloring

When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored (*spilled*) into a memory location in order to free up a register. Graph coloring is a simple, systematic technique for allocating registers and managing register spills.

In this method, two passes are used. In the first, target-machine instructions are selected as though there were an infinite number of symbolic registers; in effect, names used in the intermediate code become names of registers and the three-address statements become machine-language statements. If access to variables requires instructions that use stack pointers, display pointers, base registers, or other quantities that assist access, then we assume that these quantities are held in registers reserved for each purpose. Normally, their use is directly translatable into an access mode for an address mentioned in a machine instruction. If access is more complex, the access must be broken into several machine instructions, and a temporary symbolic register (or several) may need to be created.

Once the instructions have been selected, a second pass assigns physical registers to symbolic ones. The goal is to find an assignment that minimizes the cost of the spills.

In the second pass, for each procedure a *register-interference graph* is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined. For example, a register-interference graph for Figure 9.13 would have nodes for names a and d. In block B_1, a is live at the second statement, which defines d; therefore, in the graph there would be an edge between the nodes for a and d.

An attempt is made to color the register-interference graph using k colors, where k is the number of assignable registers. (A graph is said to be *colored* if each node has been assigned a color in such a way that no two adjacent nodes have the same color.) A color represents a register, and the coloring makes sure that no two symbolic registers that can interfere with each other are assigned the same physical register.

Although the problem of determining whether a graph is k-colorable is NP-complete in general, the following heuristic technique can usually be used to do the coloring quickly in practice. Suppose a node n in a graph G has fewer than k neighbors (nodes connected to n by an edge). Remove n and its edges from G to obtain a graph G'. A k-coloring of G' can be extended to a k-coloring of G by assigning n a color not assigned to any of its neighbors.

By repeatedly eliminating nodes having fewer than k edges from the register-interference graph, either we obtain the empty graph, in which case we can produce a k-coloring for the original graph by coloring the nodes in the reverse order in which they were removed, or we obtain a graph in which each node has k or more adjacent nodes. In the latter case a k-coloring is no longer possible. At this point a node is spilled by introducing code to store and reload the register. Then the interference graph is appropriately modified and the coloring process resumed. Chaitin [1982] and Chaitin et al. [1981] describe several heuristics for choosing the node to spill. A general rule is to avoid introducing spill code into inner loops.

9.8 THE DAG REPRESENTATION OF BASIC BLOCKS

Directed acyclic graphs (dags) are useful data structures for implementing transformations on basic blocks. A dag gives a picture of how the value computed by each statement in a basic block is used in subsequent statements of the block. Constructing a dag from three-address statements is a good way of determining common subexpressions (expressions computed more than once) within a block, determining which names are used inside the block but evaluated outside the block, and determining which statements of the block could have their computed value used outside the block.

A *dag for a basic block* (or just *dag*) is a directed acyclic graph with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or

constants. From the operator applied to a name we determine whether the *l*-value or *r*-value of a name is needed; most leaves represent *r*-values. The leaves represent initial values of names, and we subscript them with 0 to avoid confusion with labels denoting "current" values of names as in (3) below.

2. Interior nodes are labeled by an operator symbol.

3. Nodes are also optionally given a sequence of identifiers for labels. The intention is that interior nodes represent computed values, and the identifiers labeling a node are deemed to have that value.

It is important not to confuse dags with flow graphs. Each node of a flow graph can be represented by a dag, since each node of the flow graph stands for a basic block.

```
(1)   t1 := 4 * i
(2)   t2 := a [ t1 ]
(3)   t3 := 4 * i
(4)   t4 := b [ t3 ]
(5)   t5 := t2 * t4
(6)   t6 := prod + t5
(7)   prod := t6
(8)   t7 := i + 1
(9)   i := t7
(10)  if i <= 20 goto (1)
```

Fig. 9.15. Three-address code for block B_2.

Example 9.7. Figure 9.15 shows the three-address code corresponding to block B_2 of Fig. 9.9. Statement numbers starting from (1) have been used for convenience. The corresponding dag is shown in Fig. 9.16. We discuss the significance of the dag after giving an algorithm to construct it. For the time being let us observe that each node of the dag represents a formula in terms of the leaves, that is, the values possessed by variables and constants upon entering the block. For example, the node labeled t_4 in Fig. 9.16 represents the formula

```
b [4 * i]
```

that is, the value of the word whose address is `4*i` bytes offset from address `b`, which is the intended value of t_4. □

Dag Construction

To construct a dag for a basic block, we process each statement of the block in turn. When we see a statement of the form `x := y+z`, we look for the

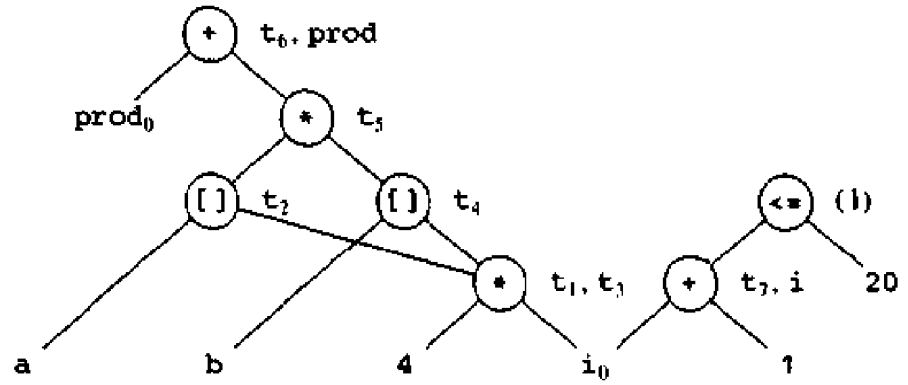

Fig. 9.16. Dag for block of Fig. 9.15.

nodes that represent the "current" values of y and z. These could be leaves, or they could be interior nodes of the dag if y and/or z had been evaluated by previous statements of the block. We then create a node labeled + and give it two children; the left child is the node for y, the right the node for z. Then we label this node x. However, if there is already a node denoting the same value as y + z, we do not add the new node to the dag, but rather give the existing node the additional label x.

Two details should be mentioned. First, if x (not x_0) had previously labeled some other node, we remove that label, since the "current" value of x is the node just created. Second, for an assignment such as x := y we do not create a new node. Rather, we append label x to the list of names on the node for the "current" value of y.

We now give the algorithm to compute a dag from a block. The algorithm is almost the same as Algorithm 5.1, except for the additional list of identifiers we attach to each node here. The reader should be warned that this algorithm may not operate correctly if there are assignments to arrays, if there are indirect assignments through pointers, or if one memory location can be referred to by two or more names, due to **EQUIVALENCE** statements or the correspondences between actual and formal parameters of a procedure call. We discuss the modifications necessary to handle these situations at the end of this section.

Algorithm 9.2. Constructing a dag.

Input. A basic block.

Output. A dag for the basic block containing the following information:

1. A *label* for each node. For leaves the label is an identifier (constants permitted), and for interior nodes, an operator symbol.
2. For each node a (possibly empty) list of attached identifiers (constants not permitted here).

Method. We assume the appropriate data structures are available to create nodes with one or two children, with a distinction between "left" and "right" children in the latter case. Also available in the structure is a place for a label for each node and the facility to create a linked list of attached identifiers for each node.

In addition to these components, we need to maintain the set of all identifiers (including constants) for which there is a node associated. The node could be either a leaf labeled by that identifier or an interior node with that identifier on its attached identifier list. We assume the existence of a function *node*(*identifier*), which, as we build the dag, returns the most recently created node associated with *identifier*. Intuitively, *node*(*identifier*) is the node of the dag that represents the value that *identifier* has at the current point in the dag construction process. In practice, an entry in the symbol-table record for *identifier* would indicate the value of *node*(*identifier*).

The dag construction process is to do the following steps (1) through (3) for each statement of the block, in turn. Initially, we assume there are no nodes, and *node* is undefined for all arguments. Suppose the "current" three-address statement is either (i) `x := y` *op* `z`, (ii) `x :=` *op* `y`, or (iii) `x := y`.[7] We refer to these as cases (i), (ii), and (iii). We treat a relational operator like `if i <= 20 goto` as case (i), with `x` undefined.

1. If *node*(`y`) is undefined, create a leaf labeled `y`, and let *node*(`y`) be this node. In case (i), if *node*(`z`) is undefined, create a leaf labeled `z` and let that leaf be *node*(`z`).

2. In case (i), determine if there is a node labeled *op*, whose left child is *node*(`y`) and whose right child is *node*(`z`). (This check is to catch common subexpressions.) If not, create such a node. In either event, let *n* be the node found or created. In case (ii), determine whether there is a node labeled *op*, whose lone child is *node*(`y`). If not, create such a node, and let *n* be the node found or created. In case (iii), let *n* be *node*(`y`).

3. Delete `x` from the list of attached identifiers for *node*(`x`). Append `x` to the list of attached identifiers for the node *n* found in (2) and set *node*(`x`) to *n*. □

Example 9.8. Let us return to the block of Fig. 9.15 and see how the dag of Fig. 9.16 is constructed for it. The first statement is t_1 `:= 4 * i`. In step (1), we must create leaves labeled `4` and i_0. (We use the subscript 0, as before, to help distinguish labels from attached identifiers in pictures, but the subscript is not really part of the label.) In step (2), we create a node labeled `*`, and in step (3) we attach identifier t_1 to it. Figure 9.17(a) shows the dag at this stage.

[7] Operators are assumed to have at most two arguments. The generalization to three or more arguments is straightforward.

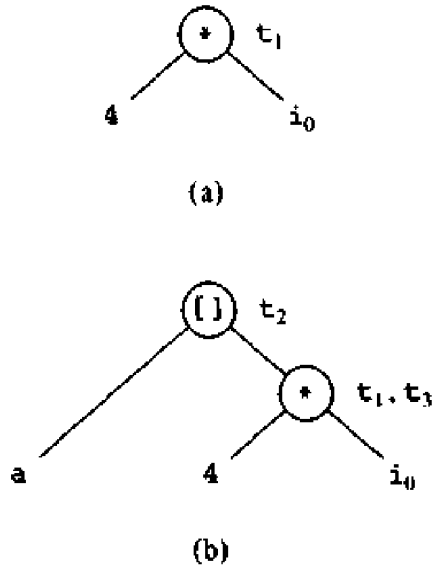

Fig. 9.17. Steps in the dag construction process.

For the second statement, t_2 := a[t_1], we create a new leaf labeled a and find the previously created *node*(t_1). We also create a new node labeled [ ] to which we attach the nodes for a and t_1 as children.

For statement (3), t_3 := 4 * i, we determine that *node*(4) and *node*(i) already exist. Since the operator is *, we do not create a new node for statement (3), but rather append t_3 on the identifier list for node t_1. The resulting dag is shown in Fig. 9.17(b). The value-number method of Section 5.2 can be used to discover quickly that the node for 4 * i already exists.

We invite the reader to complete the construction of the dag. We mention only the steps taken for statement (9), i := t_7. Before statement (9), *node*(i) is the leaf labeled i_0. Statement (9) is an instance of case (iii); therefore, we find *node*(t_7), append i to its identifier list, and set *node*(i) to *node*(t_7). This is one of only two statements — the other is statement (7) — where the value of *node* changes for an identifier. It is this change that ensures that the new node for i is the left child of the <= operator node constructed for statement (10). □

Applications of Dags

There are several pieces of useful information that we can obtain as we are executing Algorithm 9.2. First, note that we automatically detect common subexpressions. Second, we can determine which identifiers have their values used in the block; they are exactly those for which a leaf is created in step (1) at some time. Third, we can determine which statements compute values that could be used outside the block. They are exactly those statements S whose node n constructed or found in step (2) still has *node*(x) = n at the end of the dag construction, where x is the identifier assigned by statement S. (Equivalently, x is still an attached identifier for n.)

Example 9.9. In Example 9.8, all statements meet the above constraint because the only times *node* is redefined — for `prod` and `i` — the previous value of *node* was a leaf. Thus all interior nodes can have their value used outside the block. Now, suppose we inserted before statement (9) a new statement s that assigned a value to `i`. At statement s we would create a node m and set *node*(`i`) = m. However, at statement (9) we would redefine *node*(`i`). Thus the value computed at statement s could not be used outside the block. □

Another important use to which the dag may be put is to reconstruct a simplified list of quadruples taking advantage of common subexpressions and not performing assignments of the form `x := y` unless absolutely necessary. That is, whenever a node has more than one identifier on its attached list, we check which, if any, of those identifiers are needed outside the block. As we mentioned, finding live variables for the end of a block requires a data-flow analysis called "live-variable analysis" discussed in Chapter 10. However, in many cases we may assume no temporary name such as $t_1, t_2, \ldots, t_7$ in Fig. 9.15 is needed outside the block. (But beware of how logical expressions are translated; one expression may wind up spread over several basic blocks.)

We may in general evaluate the interior nodes of the dag in any order that is a topological sort of the dag. In a topological sort, a node is not evaluated until all of its children that are interior nodes have been evaluated. As we evaluate a node, we assign its value to one of its attached identifiers `x`, preferring one whose value is needed outside the block. We may not, however, choose `x` if there is another node m whose value was also held by `x` such that m has been evaluated and is still "live." Here, we define m to be live if its value is needed outside the block or if m has a parent which has not yet been evaluated.

If there are additional attached identifiers $y_1, y_2, \ldots, y_k$ for a node n whose values are also needed outside the block, we assign to them with statements $y_1 := x, y_2 := x, \ldots, y_k := x$. If n has no attached identifiers at all (this could happen if, say, n was created by an assignment to `x`, but `x` was subsequently reassigned), we create a new temporary name to hold the value of n. The reader should beware that in the presence of pointer or array assignments, not every topological sort of a dag is permissible; we shall attend to this matter shortly.

Example 9.10. Let us reconstruct a basic block from the dag of Fig. 9.16, ordering nodes in the same order as they were created: t_1, t_2, t_4, t_5, t_6, t_7, (1). Note that statements (3) and (7) of the original block did not create new nodes, but added labels t_3 and `prod` to the identifier lists of nodes t_1 and t_6, respectively. We assume none of the temporaries t_i are needed outside the block.

We begin with the node representing `4*i`. This node has two identifiers attached, t_1 and t_3. Let us pick t_1 to hold the value `4*i`, so the first reconstructed statement is

```
t1 := 4 * i
```

just as in the original basic block. The second node considered is labeled t_2. The statement constructed from this node is

```
t2 := a [ t1 ]
```

also as before. The node considered next is labeled t_4 from which we generate the statement

```
t4 := b [ t1 ]
```

The latter statement uses t_1 as an argument rather than t_3 as in the original basic block, because t_1 is the name chosen to carry the value `4*i`.

Next we consider the node labeled t_5 and generate the statement

```
t5 := t2 * t4
```

For the node labeled t_6, `prod`, we select `prod` to carry the value, since that identifier and not t_6 will (presumably) be needed outside the block. Like t_3, temporary t_6 disappears. The next statement generated is

```
prod := prod + t5
```

Similarly, we choose `i` rather than t_7 to carry the value `i+1`. The last two statements generated are

```
i := i + 1
if i <= 20 goto (1)
```

Note that the ten statements of Fig. 9.15 have been reduced to seven by taking advantage of the common subexpressions exposed during the dag construction process, and by eliminating unnecessary assignments. □

Arrays, Pointers, and Procedure Calls

Consider the basic block:

```
x := a[i]
a[j] := y                    (9.5)
z := a[i]
```

If we use Algorithm 9.2 to construct the dag for (9.5), `a[i]` would become a common subexpression, and the "optimized" block would turn out to be

```
x := a[i]
z := x                       (9.6)
a[j] := y
```

However, (9.5) and (9.6) compute different values for `z` in the case `i = j` and `y ≠ a[i]`. The problem is that when we assign to an array `a`, we may be changing the *r*-value of expression `a[i]`, even though `a` and `i` do not change. It is therefore necessary that when processing an assignment to array

a, we *kill* all nodes labeled [], whose left argument is a plus or minus a constant (possibly zero).[8] That is, we make these nodes ineligible to be given an additional identifier label, preventing them from being falsely recognized as common subexpressions. It is thus required that we keep a bit for each node telling whether or not it has been killed. Further, for each array a mentioned in the block, it is convenient to have a list of all nodes currently not killed but that must be killed should we assign to an element of a.

A similar problem occurs if we have an assignment such as *p:=w, where p is a pointer. If we do not know what p might point to, every node currently in the dag being built must be killed in the sense above. If node n labeled a is killed and there is a subsequent assignment to a, we must create a new leaf for a and use that leaf rather than n. We later consider the constraints on the order of evaluation caused by the killing of nodes.

In Chapter 10, we discuss methods whereby we could discover that p could only point to some subset of the identifiers. If p could only point to r or s, then only *node*(r) and *node*(s) must be killed. It is also conceivable that we could discover that i = j is impossible in block (9.5), in which case the node for a[i] need not be killed by a[j] := y. However, the latter type of discovery is not usually worth the trouble.

A procedure call in a basic block kills all nodes, since in the absence of knowledge about the called procedure, we must assume that any variable may be changed as a side effect. Chapter 10 discusses how we may establish that certain identifiers are not changed by a procedure call, and then nodes for these identifiers need not be killed.

If we intend to reassemble the dag into a basic block and may not want to use the order in which the nodes of the dag were created, then we must indicate in the dag that certain apparently independent nodes must be evaluated in a certain order. For example, in (9.5), the statement z := a[i] must follow a[j] := y, which must follow x := a[i]. Let us introduce certain edges $n \rightarrow m$ in the dag that do not indicate that m is an argument of n, but rather that evaluation of n must follow evaluation of m in any computation of the dag. The rules to be enforced are the following.

1. Any evaluation of or assignment to an element of array a must follow the previous assignment to an element of that array if there is one.
2. Any assignment to an element of array a must follow any previous evaluation of a.
3. Any use of any identifier must follow the previous procedure call or indirect assignment through a pointer if there is one.
4. Any procedure call or indirect assignment through a pointer must follow all previous evaluations of any identifier.

[8] Note that the argument of [] indicating the array name could be a itself, or an expression like a−4. In the latter case, the node a would be a grandchild, rather than a child of the node [].

That is, when reordering code, uses of an array a may not cross each other, and no statement may cross a procedure call or an assignment through a pointer.

9.9 PEEPHOLE OPTIMIZATION

A statement-by-statement code-generation strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying "optimizing" transformations to the target program. The term "optimizing" is somewhat misleading because there is no guarantee that the resulting code is optimal under any mathematical measure. Nevertheless, many simple transformations can significantly improve the running time or space requirement of the target program, so it is important to know what kinds of transformations are useful in practice.

A simple but effective technique for locally improving the target code is *peephole optimization*, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the *peephole*) and replacing these instructions by a shorter or faster sequence, whenever possible. Although we discuss peephole optimization as a technique for improving the quality of the target code, the technique can also be applied directly after intermediate code generation to improve the intermediate representation.

The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements. In general, repeated passes over the target code are necessary to get the maximum benefit. In this section, we shall give the following examples of program transformations that are characteristic of peephole optimizations:

- redundant-instruction elimination
- flow-of-control optimizations
- algebraic simplifications
- use of machine idioms

Redundant Loads and Stores

If we see the instruction sequence

```
(1) MOV   R0, a
(2) MOV   a, R0
```
(9.7)

we can delete instruction (2) because whenever (2) is executed, (1) will ensure that the value of a is already in register R0. Note that if (2) had a label[9] we

[9] One advantage of generating assembly code is that labels will be present, facilitating peephole optimizations such as this. If machine code is generated, and peephole optimization is desired, we can use a bit to mark the instructions that would have labels.

could not be sure that (1) was always executed immediately before (2) and so we could not remove (2). Put another way, (1) and (2) have to be in the same basic block for this transformation to be safe.

While target code such as (9.7) would not be generated if the algorithm suggested in Section 9.6 were used, it might be if a more naive algorithm like the one mentioned at the beginning of Section 9.1 were used.

Unreachable Code

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable `debug` is 1. In C, the source code might look like:

```
#define debug 0
...
if ( debug ) {
    print debugging information
}
```

In the intermediate representation the if-statement may be translated as:

```
    if debug = 1 goto L1
    goto L2                                    (9.8)
L1: print debugging information
L2:
```

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of `debug`, (9.8) can be replaced by:

```
    if debug ≠ 1 goto L2
    print debugging information                (9.9)
L2:
```

Now, since `debug` is set to 0 at the beginning of the program,[10] constant propagation should replace (9.9) by

```
    if 0 ≠ 1 goto L2
    print debugging information                (9.10)
L2:
```

As the argument of the first statement of (9.10) evaluates to a constant **true**, it can be replaced by `goto L2`. Then all the statements that print debugging aids are manifestly unreachable and can be eliminated one at a time.

[10] To tell that `debug` has the value 0 we need to do a global "reaching definitions" data-flow analysis, as discussed in Chapter 10.

Flow-of-Control Optimizations

The intermediate code generation algorithms in Chapter 8 frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

```
        goto L1
        ...
L1:     goto L2
```

by the sequence

```
        goto L2
        ...
L1:     goto L2
```

If there are now no jumps to `L1`,[11] then it may be possible to eliminate the statement `L1: goto L2` provided it is preceded by an unconditional jump. Similarly, the sequence

```
        if a < b goto L1
        ...
L1:     goto L2
```

can be replaced by

```
        if a < b goto L2
        ...
L1:     goto L2
```

Finally, suppose there is only one jump to `L1` and `L1` is preceded by an unconditional goto. Then the sequence

```
        goto L1
        ...
L1:     if a < b goto L2          (9.11)
L3:
```

may be replaced by

```
        if a < b goto L2
        goto L3
        ...
L3:                               (9.12)
```

While the number of instructions in (9.11) and (9.12) is the same, we

[11] If this peephole optimization is attempted, we can count the number of jumps to each label in the symbol-table entry for that label; a search of the code is not necessary.

sometimes skip the unconditional jump in (9.12), but never in (9.11). Thus, (9.12) is superior to (9.11) in execution time.

Algebraic Simplification

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. However, only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

```
x := x + 0
```

or

```
x := x * 1
```

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 is invariably cheaper to implement as `x*x` than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented (approximated) as multiplication by a constant, which may be cheaper.

Use of Machine Idioms

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like `i:=i+1`.

9.10 GENERATING CODE FROM DAGS

In this section, we show how to generate code for a basic block from its dag representation. The advantage of doing so is that from a dag we can more easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples. Central to our discussion is the case where the dag is a tree. For this case we can generate code that we can prove is optimal under such criteria as