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

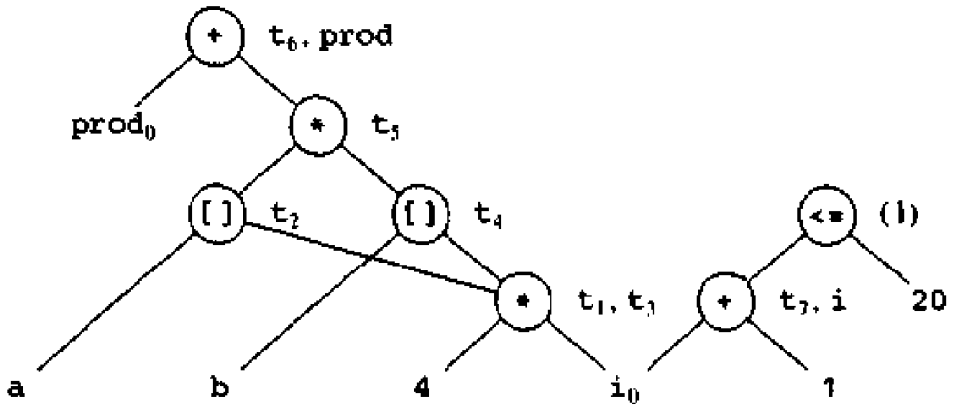

Fig. 9.16. Dag for block of Fig. 9.15.

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.

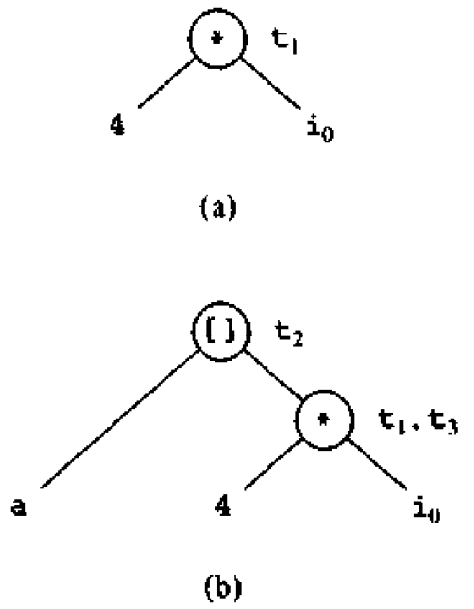

Fig. 9.17. Steps in the dag construction process.

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.)