Exercise 8.3.3: Generate code for the following three-address statements again assuming stack allocation and assuming `a` and `b` are arrays whose elements are 4-byte values.

a) The four-statement sequence

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

b) The three-statement sequence

```
x = a[i]
y = b[i]
z = x * y
```

c) The three-statement sequence

```
x = a[i]
y = b[x]
a[i] = y
```

A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. Compilers: Principles, Techniques, & Tools, 2nd ed. Addison-Wesley, 2007. ("The Dragon Book")

8.4 Basic Blocks and Flow Graphs

This section introduces a graph representation of intermediate code that is helpful for discussing code generation even if the graph is not constructed explicitly by a code-generation algorithm. Code generation benefits from context. We can do a better job of register allocation if we know how values are defined and used, as we shall see in Section 8.8. We can do a better job of instruction selection by looking at sequences of three-address statements, as we shall see in Section 8.9.

The representation is constructed as follows:

1. Partition the intermediate code into *basic blocks*, which are maximal sequences of consecutive three-address instructions with the properties that

   (a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.

   (b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.

2. The basic blocks become the nodes of a *flow graph*, whose edges indicate which blocks can follow which other blocks.

The Effect of Interrupts

The notion that control, once it reaches the beginning of a basic block is certain to continue through to the end requires a bit of thought. There are many reasons why an interrupt, not reflected explicitly in the code, could cause control to leave the block, perhaps never to return. For example, an instruction like `x = y/z` appears not to affect control flow, but if z is 0 it could actually cause the program to abort.

We shall not worry about such possibilities. The reason is as follows. The purpose of constructing basic blocks is to optimize the code. Generally, when an interrupt occurs, either it will be handled and control will come back to the instruction that caused the interrupt, as if control had never deviated, or the program will halt with an error. In the latter case, it doesn't matter how we optimized the code, even if we depended on control reaching the end of the basic block, because the program didn't produce its intended result anyway.

Starting in Chapter 9, we discuss transformations on flow graphs that turn the original intermediate code into "optimized" intermediate code from which better target code can be generated. The "optimized" intermediate code is turned into machine code using the code-generation techniques in this chapter.

8.4.1 Basic Blocks

Our first job is to partition a sequence of three-address instructions into basic blocks. We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction. In the absence of jumps and labels, control proceeds sequentially from one instruction to the next. This idea is formalized in the following algorithm.

Algorithm 8.5: Partitioning three-address instructions into basic blocks.

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are *leaders*, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.

2. Any instruction that is the target of a conditional or unconditional jump is a leader.

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program. □

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Figure 8.7: Intermediate code to set a 10×10 matrix to an identity matrix

Example 8.6: The intermediate code in Fig. 8.7 turns a 10×10 matrix a into an identity matrix. Although it is not important where this code comes from, it might be the translation of the pseudocode in Fig. 8.8. In generating the intermediate code, we have assumed that the real-valued array elements take 8 bytes each, and that the matrix a is stored in row-major form.

for i from 1 to 10 **do**
    for j from 1 to 10 **do**
        $a[i, j] = 0.0$;
for i from 1 to 10 **do**
    $a[i, i] = 1.0$;

Figure 8.8: Source code for Fig. 8.7

First, instruction 1 is a leader by rule (1) of Algorithm 8.5. To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 9, 11, and 17. By rule (2), the targets of these jumps are leaders; they are instructions 3, 2, and 13, respectively. Then, by rule (3), each instruction following a jump is a leader; those are instructions 10 and 12. Note that no instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

We conclude that the leaders are instructions 1, 2, 3, 10, 12, and 13. The basic block of each leader contains all the instructions from itself until just before the next leader. Thus, the basic block of 1 is just 1, for leader 2 the block is just 2. Leader 3, however, has a basic block consisting of instructions 3 through 9, inclusive. Instruction 10's block is 10 and 11; instruction 12's block is just 12, and instruction 13's block is 13 through 17. □

8.4.2 Next-Use Information

Knowing when the value of a variable will be used next is essential for generating good code. If the value of a variable that is currently in a register will never be referenced subsequently, then that register can be assigned to another variable.

The *use* of a name in a three-address statement is defined as follows. Suppose three-address statement i assigns a value to x. If statement j has x as an operand, and control can flow from statement i to j along a path that has no intervening assignments to x, then we say statement j *uses* the value of x computed at statement i. We further say that x is *live* at statement i.

We wish to determine for each three-address statement $x = y + z$ what the next uses of x, y, and z are. For the present, we do not concern ourselves with uses outside the basic block containing this three-address statement.

Our algorithm to determine liveness and next-use information makes a backward pass over each basic block. We store the information in the symbol table. We can easily scan a stream of three-address statements to find the ends of basic blocks as in Algorithm 8.5. Since procedures can have arbitrary side effects, we assume for convenience that each procedure call starts a new basic block.

Algorithm 8.7: Determining the liveness and next-use information for each statement in a basic block.

INPUT: A basic block B of three-address statements. We assume that the symbol table initially shows all nontemporary variables in B as being live on exit.

OUTPUT: At each statement i: $x = y + z$ in B, we attach to i the liveness and next-use information of x, y, and z.

METHOD: We start at the last statement in B and scan backwards to the beginning of B. At each statement i: $x = y + z$ in B, we do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and y.

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.

Here we have used $+$ as a symbol representing any operator. If the three-address statement i is of the form $x = +\, y$ or $x = y$, the steps are the same as above, ignoring z. Note that the order of steps (2) and (3) may not be interchanged because x may be y or z. □

8.4.3 Flow Graphs

Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B. There are two ways that such an edge could be justified:

- There is a conditional or unconditional jump from the end of B to the beginning of C.
- C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

We say that B is a *predecessor* of C, and C is a *successor* of B.

Often we add two nodes, called the *entry* and *exit*, that do not correspond to executable intermediate instructions. There is an edge from the entry to the first executable node of the flow graph, that is, to the basic block that comes from the first instruction of the intermediate code. There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program. If the final instruction of the program is not an unconditional jump, then the block containing the final instruction of the program is one predecessor of the exit, but so is any basic block that has a jump to code that is not part of the program.

Example 8.8: The set of basic blocks constructed in Example 8.6 yields the flow graph of Fig. 8.9. The entry points to basic block B_1, since B_1 contains the first instruction of the program. The only successor of B_1 is B_2, because B_1 does not end in an unconditional jump, and the leader of B_2 immediately follows the end of B_1.

Block B_3 has two successors. One is itself, because the leader of B_3, instruction 3, is the target of the conditional jump at the end of B_3, instruction 9. The other successor is B_4, because control can fall through the conditional jump at the end of B_3 and next enter the leader of B_4.

Only B_6 points to the exit of the flow graph, since the only way to get to code that follows the program from which we constructed the flow graph is to fall through the conditional jump that ends B_6. □

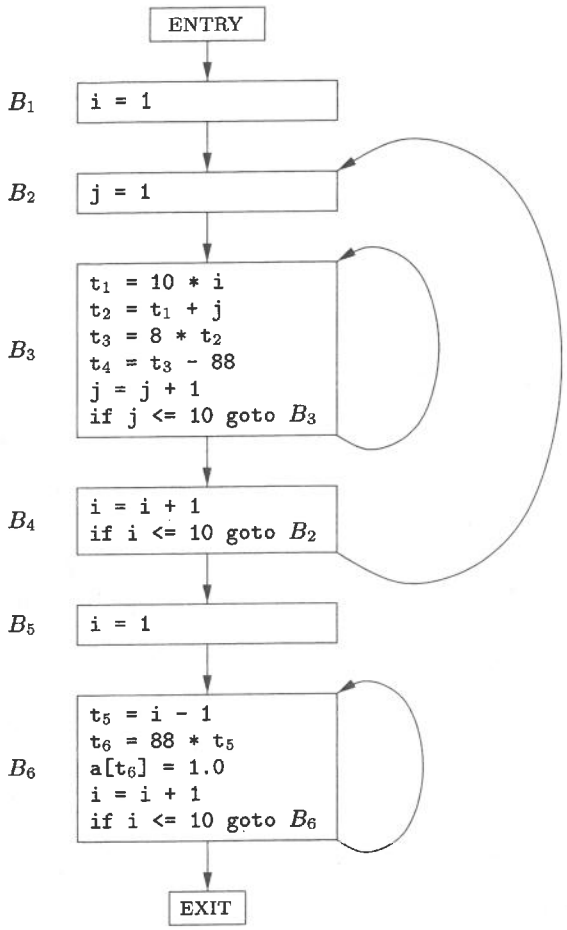

Figure 8.9: Flow graph from Fig. 8.7

8.4.4 Representation of Flow Graphs

First, note from Fig. 8.9 that in the flow graph, it is normal to replace the jumps to instruction numbers or labels by jumps to basic blocks. Recall that every conditional or unconditional jump is to the leader of some basic block, and it is to this block that the jump will now refer. The reason for this change is that after constructing the flow graph, it is common to make substantial changes to the instructions in the various basic blocks. If jumps were to instructions, we would have to fix the targets of the jumps every time one of the target instructions was changed.

Flow graphs, being quite ordinary graphs, can be represented by any of the data structures appropriate for graphs. The content of nodes (basic blocks) need their own representation. We might represent the content of a node by a

pointer to the leader in the array of three-address instructions, together with a count of the number of instructions or a second pointer to the last instruction. However, since we may be changing the number of instructions in a basic block frequently, it is likely to be more efficient to create a linked list of instructions for each basic block.

8.4.5 Loops

Programming-language constructs like while-statements, do-while-statements, and for-statements naturally give rise to loops in programs. Since virtually every program spends most of its time in executing its loops, it is especially important for a compiler to generate good code for loops. Many code transformations depend upon the identification of "loops" in a flow graph. We say that a set of nodes L in a flow graph is a *loop* if

1. There is a node in L called the *loop entry* with the property that no other node in L has a predecessor outside L. That is, every path from the entry of the entire flow graph to any node in L goes through the loop entry.
2. Every node in L has a nonempty path, completely within L, to the entry of L.

Example 8.9: The flow graph of Fig. 8.9 has three loops:

1. B_3 by itself.
2. B_6 by itself.
3. $\{B_2, B_3, B_4\}$.

The first two are single nodes with an edge to the node itself. For instance, B_3 forms a loop with B_3 as its entry. Note that the second requirement for a loop is that there be a nonempty path from B_3 to itself. Thus, a single node like B_2, which does not have an edge $B_2 \rightarrow B_2$, is not a loop, since there is no nonempty path from B_2 to itself within $\{B_2\}$.

The third loop, $L = \{B_2, B_3, B_4\}$, has B_2 as its loop entry. Note that among these three nodes, only B_2 has a predecessor, B_1, that is not in L. Further, each of the three nodes has a nonempty path to B_2 staying within L. For instance, B_2 has the path $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2$. □

8.4.6 Exercises for Section 8.4

Exercise 8.4.1: Figure 8.10 is a simple matrix-multiplication program.

a) Translate the program into three-address statements of the type we have been using in this section. Assume the matrix entries are numbers that require 8 bytes, and that matrices are stored in row-major order.

b) Construct the flow graph for your code from (a).

c) Identify the loops in your flow graph from (b).

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0.0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Figure 8.10: A matrix-multiplication algorithm

Exercise 8.4.2: Figure 8.11 is code to count the number of primes from 2 to n, using the sieve method on a suitably large array a. That is, $a[i]$ is `TRUE` at the end only if there is no prime $\sqrt{i}$ or less that evenly divides i. We initialize all $a[i]$ to `TRUE` and then set $a[j]$ to `FALSE` if we find a divisor of j.

a) Translate the program into three-address statements of the type we have been using in this section. Assume integers require 4 bytes.

b) Construct the flow graph for your code from (a).

c) Identify the loops in your flow graph from (b).

```
for (i=2; i<=n; i++)
    a[i] = TRUE;
count = 0;
s = sqrt(n);
for (i=2; i<=s; i++)
    if (a[i]) /* i has been found to be a prime */ {
        count++;
        for (j=2*i; j<=n; j = j+i)
            a[j] = FALSE; /* no multiple of i is a prime */
    }
```

Figure 8.11: Code to sieve for primes

8.5 Optimization of Basic Blocks

We can often obtain a substantial improvement in the running time of code merely by performing *local* optimization within each basic block by itself. More thorough *global* optimization, which looks at how information flows among the basic blocks of a program, is covered in later chapters, starting with Chapter 9. It is a complex subject, with many different techniques to consider.

8.5.1 The DAG Representation of Basic Blocks

Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph). In Section 6.1.1, we introduced the DAG as a representation for single expressions. The idea extends naturally to the collection of expressions that are created within one basic block. We construct a DAG for a basic blockas follows:

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.

2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.

3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.

4. Certain nodes are designated *output nodes*. These are the nodes whose variables are *live on exit* from the block; that is, their values may be used later, in another block of the flow graph. Calculation of these "live variables" is a matter for global flow analysis, discussed in Section 9.2.5.

The DAG representation of a basic block lets us perform several code-improving transformations on the code represented by the block.

a) We can eliminate *local common subexpressions*, that is, instructions that compute a value that has already been computed.

b) We can eliminate *dead code*, that is, instructions that compute a value that is never used.

c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.

d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.

8.5.2 Finding Local Common Subexpressions

Common subexpressions can be detected by noticing, as a new node M is about to be added, whether there is an existing node N with the same children, in the same order, and with the same operator. If so, N computes the same value as M and may be used in its place. This technique was introduced as the "value-number" method of detecting common subexpressions in Section 6.1.1.

Example 8.10: A DAG for the block

```
a = b + c
b = a - d
c = b + c
d = a - d
```

is shown in Fig. 8.12. When we construct the node for the third statement `c = b + c`, we know that the use of `b` in `b + c` refers to the node of Fig. 8.12 labeled `-`, because that is the most recent definition of `b`. Thus, we do not confuse the values computed at statements one and three.

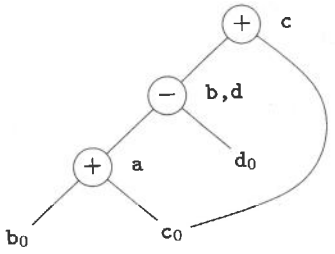

Figure 8.12: DAG for basic block in Example 8.10

However, the node corresponding to the fourth statement `d = a - d` has the operator `-` and the nodes with attached variables `a` and d_0 as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add `d` to the list of definitions for the node labeled $-$. □

It might appear that, since there are only three nonleaf nodes in the DAG of Fig. 8.12, the basic block in Example 8.10 can be replaced by a block with only three statements. In fact, if `b` is not live on exit from the block, then we do not need to compute that variable, and can use `d` to receive the value represented by the node labeled $-$. in Fig. 8.12. The block then becomes

```
a = b + c
d = a - d
c = d + c
```

However, if both b and d are live on exit, then a fourth statement must be used to copy the value from one to the other.[1]

Example 8.11: When we look for common subexpressions, we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed. Thus, the DAG method will miss the fact that the expression computed by the first and fourth statements in the sequence

```
a = b + c;
b = b - d
c = c + d
e = b + c
```

is the same; namely $b_0 + c_0$. That is, even though b and c both change between the first and last statements, their sum remains the same, because $b + c = (b - d) + (c + d)$. The DAG for this sequence is shown in Fig. 8.13, but does not exhibit any common subexpressions. However, algebraic identities applied to the DAG, as discussed in Section 8.5.4, may expose the equivalence. □

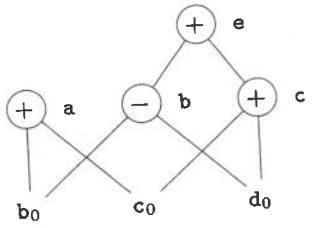

Figure 8.13: DAG for basic block in Example 8.11

8.5.3 Dead Code Elimination

The operation on DAG's that corresponds to dead-code elimination can be implemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Example 8.12: If, in Fig. 8.13, a and b are live but c and e are not, we can immediately remove the root labeled e. Then, the node labeled c becomes a root and can be removed. The roots labeled a and b remain, since they each have live variables attached. □

[1] In general, we must be careful, when reconstructing code from DAG's, how we choose the names of variables. If a variable x is defined twice, or if it is assigned once and the initial value x_0 is also used, then we must make sure that we do not change the value of x until we have made all uses of the node whose value x previously held.

8.5.4 The Use of Algebraic Identities

Algebraic identities represent another important class of optimizations on basic blocks. For example, we may apply arithmetic identities, such as

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

to eliminate computations from a basic block.

Another class of algebraic optimizations includes local *reduction in strength*, that is, replacing a more expensive operator by a cheaper one as in:

EXPENSIVE		CHEAPER
x^2	=	$x \times x$
$2 \times x$	=	$x + x$
$x/2$	=	$x \times 0.5$

A third class of related optimizations is *constant folding*. Here we evaluate constant expressions at compile time and replace the constant expressions by their values.[2] Thus the expression $2 * 3.14$ would be replaced by 6.28. Many constant expressions arise in practice because of the frequent use of symbolic constants in programs.

The DAG-construction process can help us apply these and other more general algebraic transformations such as commutativity and associativity. For example, suppose the language reference manual specifies that $*$ is commutative; that is, $x*y = y*x$. Before we create a new node labeled $*$ with left child M and right child N, we always check whether such a node already exists. However, because $*$ is commutative, we should then check for a node having operator $*$, left child N, and right child M.

The relational operators such as $<$ and $=$ sometimes generate unexpected common subexpressions. For example, the condition $x > y$ can also be tested by subtracting the arguments and performing a test on the condition code set by the subtraction.[3] Thus, only one node of the DAG may need to be generated for $x - y$ and $x > y$.

Associative laws might also be applicable to expose common subexpressions. For example, if the source code has the assignments

```
a = b + c;
e = c + d + b;
```

the following intermediate code might be generated:

[2] Arithmetic expressions should be evaluated the same way at compile time as they are at run time. K. Thompson has suggested an elegant solution to constant folding: compile the constant expression, execute the target code on the spot, and replace the expression with the result. Thus, the compiler does not need to contain an interpreter.

[3] The subtraction can, however, introduce overflows and underflows while a compare instruction would not.

```
a = b + c
t = c + d
e = t + b
```

If t is not needed outside this block, we can change this sequence to

```
a = b + c
e = a + d
```

using both the associativity and commutativity of +.

The compiler writer should examine the language reference manual carefully to determine what rearrangements of computations are permitted, since (because of possible overflows or underflows) computer arithmetic does not always obey the algebraic identities of mathematics. For example, the Fortran standard states that a compiler may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated. Thus, a compiler may evaluate $x * y - x * z$ as $x * (y - z)$, but it may not evaluate $a+(b-c)$ as $(a+b)-c$. A Fortran compiler must therefore keep track of where parentheses were present in the source language expressions if it is to optimize programs in accordance with the language definition.

8.5.5 Representation of Array References

At first glance, it might appear that the array-indexing instructions can be treated like any other operator. Consider for instance the sequence of three-address statements:

```
x = a[i]
a[j] = y
z = a[i]
```

If we think of `a[i]` as an operation involving `a` and `i`, similar to $a + i$, then it might appear as if the two uses of `a[i]` were a common subexpression. In that case, we might be tempted to "optimize" by replacing the third instruction `z = a[i]` by the simpler `z = x`. However, since `j` could equal `i`, the middle statement may in fact change the value of `a[i]`; thus, it is not legal to make this change.

The proper way to represent array accesses in a DAG is as follows.

1. An assignment from an array, like `x = a[i]`, is represented by creating a node with operator =[] and two children representing the initial value of the array, a_0 in this case, and the index `i`. Variable `x` becomes a label of this new node.

2. An assignment to an array, like `a[j] = y`, is represented by a new node with operator []= and three children representing a_0, `j` and `y`. There is no variable labeling this node. What is different is that the creation of

this node *kills* all currently constructed nodes whose value depends on a_0. A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

Example 8.13: The DAG for the basic block

```
x = a[i]
a[j] = y
z = a[i]
```

is shown in Fig. 8.14. The node N for x is created first, but when the node labeled `[]=` is created, N is killed. Thus, when the node for z is created, it cannot be identified with N, and a new node with the same operands a_0 and i_0 must be created instead. □

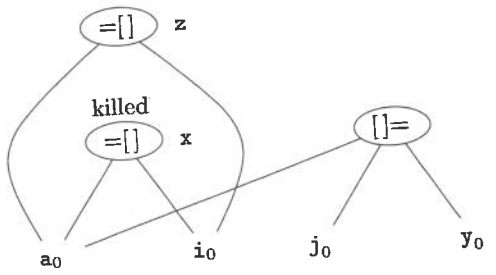

Figure 8.14: The DAG for a sequence of array assignments

Example 8.14: Sometimes, a node must be killed even though none of its children have an array like a_0 in Example 8.13 as attached variable. Likewise, a node can kill if it has a descendant that is an array, even though none of its children are array nodes. For instance, consider the three-address code

```
b = 12 + a
x = b[i]
b[j] = y
```

What is happening here is that, for efficiency reasons, b has been defined to be a position in an array a. For example, if the elements of a are four bytes long, then b represents the fourth element of a. If j and i represent the same value, then b[i] and b[j] represent the same location. Therefore it is important to have the third instruction, b[j] = y, kill the node with x as its attached variable. However, as we see in Fig. 8.15, both the killed node and the node that does the killing have a_0 as a grandchild, not as a child. □

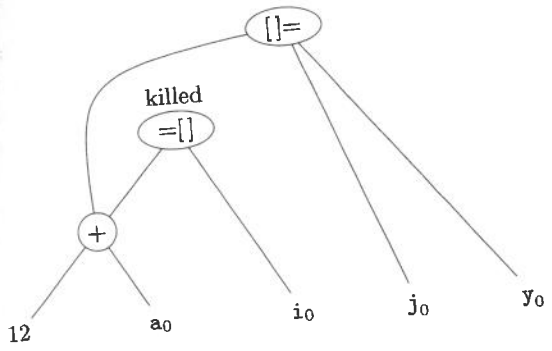

Figure 8.15: A node that kills a use of an array need not have that array as a child

8.5.6 Pointer Assignments and Procedure Calls

When we assign indirectly through a pointer, as in the assignments

```
x = *p
*q = y
```

we do not know what `p` or `q` point to. In effect, `x = *p` is a use of every variable whatsoever, and `*q = y` is a possible assignment to every variable. As a consequence, the operator $=*$ must take all nodes that are currently associated with identifiers as arguments, which is relevant for dead-code elimination. More importantly, the $*=$ operator kills all other nodes so far constructed in the DAG.

There are global pointer analyses one could perform that might limit the set of variables a pointer could reference at a given place in the code. Even local analysis could restrict the scope of a pointer. For instance, in the sequence

```
p = &x
*p = y
```

we know that `x`, and no other variable, is given the value of `y`, so we don't need to kill any node but the node to which `x` was attached.

Procedure calls behave much like assignments through pointers. In the absence of global data-flow information, we must assume that a procedure uses and changes any data to which it has access. Thus, if variable `x` is in the scope of a procedure P, a call to P both uses the node with attached variable `x` and kills that node.

8.5.7 Reassembling Basic Blocks From DAG's

After we perform whatever optimizations are possible while constructing the DAG or by manipulating the DAG once constructed, we may reconstitute the three-address code for the basic block from which we built the DAG. For each

node that has one or more attached variables, we construct a three-address statement that computes the value of one of those variables. We prefer to compute the result into a variable that is live on exit from the block. However, if we do not have global live-variable information to work from, we need to assume that every variable of the program (but not temporaries that are generated by the compiler to process expressions) is live on exit from the block.

If the node has more than one live variable attached, then we have to introduce copy statements to give the correct value to each of those variables. Sometimes, global optimization can eliminate those copies, if we can arrange to use one of two variables in place of the other.

Example 8.15: Recall the DAG of Fig. 8.12. In the discussion following Example 8.10, we decided that if `b` is not live on exit from the block, then the three statements

```
a = b + c
d = a - d
c = d + c
```

suffice to reconstruct the basic block. The third instruction, `c = d + c`, must use `d` as an operand rather than `b`, because the optimized block never computes `b`.

If both `b` and `d` are live on exit, or if we are not sure whether or not they are live on exit, then we need to compute `b` as well as `d`. We can do so with the sequence

```
a = b + c
d = a - d
b = d
c = d + c
```

This basic block is still more efficient than the original. Although the number of instructions is the same, we have replaced a subtraction by a copy, which tends to be less expensive on most machines. Further, it may be that by doing a global analysis, we can eliminate the use of this computation of `b` outside the block by replacing it by uses of `d`. In that case, we can come back to this basic block and eliminate `b = d` later. Intuitively, we can eliminate this copy if wherever this value of `b` is used, `d` is still holding the same value. That situation may or may not be true, depending on how the program recomputes `d`. □

When reconstructing the basic block from a DAG, we not only need to worry about what variables are used to hold the values of the DAG's nodes, but we also need to worry about the order in which we list the instructions computing the values of the various nodes. The rules to remember are

1. The order of instructions must respect the order of nodes in the DAG. That is, we cannot compute a node's value until we have computed a value for each of its children.

2. Assignments to an array must follow all previous assignments to, or evaluations from, the same array, according to the order of these instructions in the original basic block.

3. Evaluations of array elements must follow any previous (according to the original block) assignments to the same array. The only permutation allowed is that two evaluations from the same array may be done in either order, as long as neither crosses over an assignment to that array.

4. Any use of a variable must follow all previous (according to the original block) procedure calls or indirect assignments through a pointer.

5. Any procedure call or indirect assignment through a pointer must follow all previous (according to the original block) evaluations of any variable.

That is, when reordering code, no statement may cross a procedure call or assignment through a pointer, and uses of the same array may cross each other only if both are array accesses, but not assignments to elements of the array.

8.5.8 Exercises for Section 8.5

Exercise 8.5.1: Construct the DAG for the basic block

```
d = b * c
e = a + b
b = b * c
a = e - d
```

Exercise 8.5.2: Simplify the three-address code of Exercise 8.5.1, assuming

a) Only a is live on exit from the block.

b) a, b, and c are live on exit from the block.

Exercise 8.5.3: Construct the basic block for the code in block B_6 of Fig. 8.9. Do not forget to include the comparison $i \leq 10$.

Exercise 8.5.4: Construct the basic block for the code in block B_3 of Fig. 8.9.

Exercise 8.5.5: Extend Algorithm 8.7 to process three-statements of the form

a) `a[i] = b`

b) `a = b[i]`

c) `a = *b`

c) `*a = b`

Exercise 8.5.6: Construct the DAG for the basic block

```
a[i] = b
*p = c
d = a[j]
e = *p
*p = a[i]
```

on the assumption that

a) p can point anywhere.

b) p can point only to b or d.

! **Exercise 8.5.7:** If a pointer or array expression, such as `a[i]` or `*p` is assigned and then used, without the possibility of being changed in the interim, we can take advantage of the situation to simplify the DAG. For example, in the code of Exercise 8.5.6, since p is not assigned between the second and fourth statements, the statement `e = *p` can be replaced by `e = c`, regardless of what p points to. Revise the DAG-construction algorithm to take advantage of such situations, and apply your algorithm to the code of Example 8.5.6.

Exercise 8.5.8: Suppose a basic block is formed from the C assignment statements

```
x = a + b + c + d + e + f;
y = a + c + e;
```

a) Give the three-address statements (only one addition per statement) for this block.

b) Use the associative and commutative laws to modify the block to use the fewest possible number of instructions, assuming both x and y are live on exit from the block.

8.6 A Simple Code Generator

In this section, we shall consider an algorithm that generates code for a single basic block. It considers each three-address instruction in turn, and keeps track of what values are in what registers so it can avoid generating unnecessary loads and stores.

One of the primary issues during code generation is deciding how to use registers to best advantage. There are four principal uses of registers:

- In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation.
- Registers make good temporaries — places to hold the result of a subexpression while a larger expression is being evaluated, or more generally, a place to hold a variable that is used only within a single basic block.