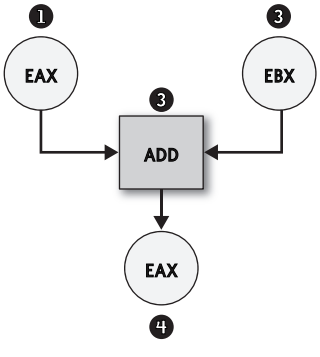
Converting this graph representation of the interpreted basic blocks into Java bytecode is a simple depth-first traversal of the graph from each sink in turn. Each node in the graph takes the topmost elements of the stack as input and then leaves its result at the top, ready for processing by any child nodes. See Figure 9-9.

NOTE

Classically optimal traversal order for the graph requires that at each node, the ascendants should be evaluated in depth order. The node with the longest path to the farthest source is evaluated first, and that with the shortest path last. On a register-based target, this will produce the shortest code because it eliminates the majority of the register juggling. On a stack-based machine such as the JVM, the same decision-making process will give rise to code with a minimal stack depth. In JPC we neglect such niceties and rely on the JVM to iron out the difference, the extra complication of tracking all the node depths is simply not worth the effort.



	code	stack
1	load eax	eax
2	load eax load ebx	ebx eax
3	load eax load ebx add	ebx+eax
4	load eax load ebx add store eax	<empty>

FIGURE 9-9. Representing x86 operations as a directed acyclic graph

This sink-by-sink depth-first parsing causes natural optimization of the graph, as shown in Figure 9-10. Orphaned sections of the graph that correspond to unused code cannot be accessed via the sinks and so will be automatically removed, as the parse will never reach them. Reused code sections will find themselves evaluated multiple times in one parse of the graph. We can cache their result in a local variable and simply load the result on subsequent visits to the node, thus saving any code duplication.

The code associated with the node in a tree is then represented by a single static function compiled from source as part of the JPC codebase. The code we generate is then just a sequence of pushes of processor variables and immediates onto the stack, followed by a sequence of `invokestatic` calls for each node, and finally a sequence of pops back into the processor object.