FIGURE e2.15.1 The structure of a modern optimizing compiler consists of a number of passes or phases. Logically, each pass can be thought of as running to completion before the next occurs. In practice, some passes may handle one procedure at a time, essentially interleaving with another pass.

FIGURE e2.15.2 An abstract syntax tree for the while example. The roots of the tree consist of the informational tokens such as numbers and names. Long chains of straight-line descendents are often omitted in constructing the tree.

FIGURE e2.15.3 The while loop example is shown using a typical intermediate representation. In practice, the names save, i, and k would be replaced by some sort of address, such as a reference to either the local stack pointer or a global pointer, and an offset, similar to the way save[i] is accessed. Note that the format of the MIPS instructions is different, because they are intermediate representations here: the operations are capitalized and the registers use RXX notation.

FIGURE e2.15.4 A control flow graph for the while loop example. Each node represents a basic block, which terminates with a branch or by sequential fall-through into another basic block that is also the target of a branch. The IR statements have been numbered for ease in referring to them. The important transformation performed was to move the while test and conditional branch to the end. This eliminates the unconditional branch that was formerly inside the loop and places it before the loop. This transformation is so important that many compilers do it during the generation of the IR. The mult was also replaced with (“strength-reduced to”) an sll.

FIGURE e2.15.5 The control flow graph showing the representation of the while loop example after code motion and induction variable elimination. The number of instructions in the inner loop has been reduced from 10 to 6.

FIGURE e2.15.6  The control flow graph showing the representation of the while loop example after code motion and induction variable elimination and register allocation, using the MIPS register names. The number of IR statements in the inner loop has now dropped to only four from six before register allocation and ten before any global optimizations. The value of i resides in $t2 at the end of the loop and may need to be stored eventually to maintain the program semantics. If i were unused after the loop, not only could the store be avoided, but also the increment inside the loop could be eliminated completely!

FIGURE e2.15.7  Major types of optimizations and explanation of each class. The third column shows when these occur at different levels of optimization in gcc. The GNU organization calls the three optimization levels medium (O1), full (O2), and full with integration of small procedures (O3).

FIGURE e2.15.8  Java bytecode architecture versus MIPS. Although many bytecodes are simple, those in the last half-dozen rows above are complex and specific to Java. Bytecodes are one to five bytes in length, hence their name. The Java mnemonics use the prefix i for 32-bit integer, a for reference (address), s for 16-bit integers (short), and b for 8-bit bytes. We use I8 for an 8-bit constant and I16 for a 16-bit constant. MIPS uses registers for operands, but the JVM uses a stack. The compiler knows the maximum size of the operand stack for each method and simply allocates space for it in the current frame. Here is the notation in the Meaning column: TOS: top of stack; NOS: next position below TOS; NNOS: next position below NOS; pop: remove TOS; pop2: remove TOS and NOS; and push: add a position to the stack. \*NOS and \*NNOS mean access the memory location pointed to by the address in the stack at those positions. Const[] refers to the runtime constant pool of a class created by the JVM, and Frame[] refers to the variables of the local method frame. The only missing MIPS instructions from Figure 2.1 are nor, andi, ori, slti, and lui. The missing bytecodes are a few arithmetic and logical operators, some tricky stack management, compares to 0 and branch, support for branch tables, type conversions, more variations of the complex, Java-specific instructions plus operations on floating-point data, 64-bit integers (longs), and 16-bit characters.

FIGURE e2.15.9 An initial Java procedure that performs a sort on the array v. Changes from Figures 2.24 and 2.26 are highlighted.

FIGURE e2.15.10 A revised Java procedure that sorts on the array v that can take on more types. Changes from Figure e2.15.9 are highlighted.

FIGURE e2.15.11 MIPS assembly code of the procedure swap in Figure 2.24.

FIGURE e2.15.12 MIPS assembly version of the method body of the Java version of sort. The new code is highlighted in this figure. We must still add the code to save and restore registers and the return from the MIPS code found in Figure 2.27. To keep the code similar to that figure, we load v.length into $s3 instead of into a temporary register. To reduce the number of lines of code, we make the simplifying assumption that compareTo is a leaf procedure and we do not need to push registers to be saved on the stack.