(implemented as separate Java threads) to summarize and record the raw profile data. Periodically, a *Controller* thread analyzes the current profile data and uses an analytic model to determine which, if any, methods should be scheduled for optimizing compilation. These decisions are made using a standard 2-competitive solution to the "ski rental" problem* from online algorithms. A method is not selected for optimization until the expected benefit (speedup in future invocations) from optimizing it exceeds the expected cost (compile time). These cost-benefit calculations are made by combining the online profile data (how often has the candidate method been sampled in the current execution?) with (offline) empirically derived constants that describe the expected relative speedup and compilation costs of each optimization level of the optimizing compiler (known as the compiler's DNA).

## Optimizing Compilation

As a metacircular runtime, Jikes RVM compiles itself instead of relying on another compiler to ensure good performance. Metacircularity creates a virtuous cycle: our strong desire to write clean, elegant, and efficient Java code in the virtual machine implementation has driven us to develop innovative compiler optimizations and runtime implementation techniques. In this section, we present the optimizing compiler, which is composed of many phases that are organized into three main stages:

1. High-level Intermediate Representation (HIR)
2. Low-level Intermediate Representation (LIR)
3. Machine-level Intermediate Representation (MIR)

All of the stages operate on a control-flow graph composed of basic blocks, which are composed of a list of instructions, as shown in Figure 10-5. An instruction is composed of operands and an operator. As the operator gradually becomes more machine-specific, the operator can be changed (mutated) during the lifetime of the instruction. The operands are organized into those that are defined and those that are used. The main kinds of operands are constant operands and operands that encode a register. A basic block is a list of instructions where branch instructions may only occur at the end of the list. Special instructions mark the beginning and end of the basic block. The control-flow graph connects the different regions of code by edges. Exceptions are treated specially and are described later in "Factored control flow graph." Because each of the three main stages of compilation use the same basic intermediate representation, a number of optimizations are applied in more than one of the stages. We describe the main tasks of the three compiler stages next.

* See *http://en.wikipedia.org/wiki/Ski_rental_problem* for more information.