operations can, in addition to delivering a result (as a mathematical function does), modify the *state* of the computation: either a global state or, in a more modular approach, some part of that state (for example, the contents of a specific object).

Although prominent in all presentations of functional programming, this property is not visible in the examples discussed here, perhaps because they follow from an initial problem analysis that already whisked the state away in favor of functional constructs. It is possible, for example, that a nonfunctional model of the notion of valuation of a financial contract would have used, instead of a function that yields a sequence (the value process), an operation that transforms the state to update the value.

It is nevertheless possible to make general comments on this fundamental decision of functional approaches. The notion of state is hard to avoid in any model of a system, computerized or not. One might even argue that it is the central notion of computation. (It has been argued [Peyton Jones 2007] that stateless programming helps address issues of concurrent programming, but there is not enough evidence yet to draw a general conclusion.) The world does not clone itself as a result of each significant event. Neither does the memory of our computers: it just overwrites its cells. It is always possible to model such state changes by positing a sequence of values instead, but this can be rather artificial (as suggested by the alternative answer to the earlier riddle: functional programmers never change a bulb, they buy a new lamp with a new cable, a new socket, and a new bulb).

Recognizing the impossibility of ignoring the state for such operations as input and output, and the clumsiness of earlier attempts (Peyton Jones and Wadler 1993), modern functional languages, in particular Haskell, have introduced the notion of monad (Wadler 1995). Monads embed the original functions in higher-order functions with more complex signatures; the added signature components can serve to record state information, as well as any extra elements such as an error status (to model exception handling) or input-output results.

Using monads to integrate the state proceeds from the same general idea—used in the reverse direction—as the technique described in the last section for obtaining lazy behavior by modeling infinite sequences as an abstract data type: to emulate in a framework A a technique T that is *implicit* in a framework B, program in A an *explicit* version of T or of the key mechanism making T possible. T is infinite lists in the first case (the "key mechanism" is infinite lists evaluated finitely), and the state in the second case.

The concept of monad is elegant and obviously useful for semantic descriptions of programming languages (especially for the denotational semantics style). One may wonder, however, whether it is the appropriate solution as a mechanism to be used directly by programmers. Here we must be careful to consider the right arguments. The obvious objection to monads—that they are difficult to teach to ordinary programmers—is irrelevant; innovative ideas considered hard at the time of their introduction can fuse into the mainstream as educators develop ways to explain them. (Both recursion and object-oriented programming were once considered beyond the reach of "Joe the Programmer.") The important question is