SIDE BAR: Programming Languages and Design Decisions

There is a correspondence between the categories of design decisions listed in Section 2 and the variety of approaches to programming language design found in modern programming languages. This is not accidental but reflects the fact that programming languages are designed to make program development easier. They do this by providing a variety of abstraction mechanisms.

The languages Algol 60, Algol 68, and Pascal introduced and systematized control and data structures. They provided mechanisms to support decomposition of procedures into statements and data into its components. Of course procedural abstraction has been with us since the early days of programming languages in the form of procedures and functions.

Variables, too, have been part of programming since its beginning, but the explicit trade offs between data and procedures have become more prominent with the advent of functional programming languages. Programming in a functional language is difficult for a traditional programmer accustomed to using variables.

Similarly, logic programming languages, such as Prolog, highlight the *function/relation* dichotomy. The same kind of conceptual barriers confront a new Prolog programmer used to more traditional styles of programming or even used to a functional style.

The issue of *encapsulation* is explicitly stressed in Ada, Modula, Clu, etc. The programmer expresses the functional interface to a module in a separate construct from the implementation. The idea is to insulate the remainder of the code from subsequent maintenance activities that alter the implementation of the module while leaving the functional interface unchanged.

Generalization/specialization is a primary consideration in Smalltalk. The class hierarchy expresses how subclasses specialize their parents. Dynamic binding is used to invisibly delegate responsibility for computation to the least general class able to handle it. Ada supports compile-time specialization of generic packages and procedures by data type and functional parameters.

Representation is supported in most programming languages, but Clu emphasizes the distinction between *representation* and *specialization* by providing separate language constructs for expressing them. Gypsy has features for describing both ideal behavior and implementation details. It supports the proof of their equivalence via semi-automated means.