A panel session—Whither automatic programming

SESSION CHAIRMAN—ROBERT BALZER
Information Sciences Institute

Panel Members

Thomas S. Standish—University of California
Michael Hammer—MIT Laboratory for Computer Science

PANEL OVERVIEW—Robert Balzer

The preceding sessions have displayed some current research into the area of Automatic Programming. These efforts are small laboratory prototypes which attempt to automate particular portions of the programming process. It is quite clear that to progress significantly beyond our current state of software development technology, increased automation, in some form, must occur.

The questions facing this panel, and addressed below, are when, in what form, and to what extent this automation will occur?

THE FUTURE OF AUTOMATIC PROGRAMMING—
Thomas A. Standish

In automatic programming, we strive to build systems that will perform problem acquisition, algorithm and data representation synthesis, optimization, and concrete program generation—systems which automatically acquire specifications of the program behavior required and which will build programs to satisfy them.

I speculate that the attainment of technological capability to build such automatic programming systems flexibly across a spectrum of dissimilar, important problem domains is a long way off—perhaps centuries and most likely decades. Before we succeed, we shall likely have had to develop deep understanding of the interrelation between knowledge systems of many sorts—knowledge systems in which problems are posed and in whose terms solutions must be delivered (the so-called problem domains), knowledge systems describing the media in which problem solving computations must execute at the lowest levels (the so-called concrete solution domains), and knowledge systems that cooperate together to provide the scaffolding in which program synthesis can take place (the intermediate problem-solving knowledge systems, which offer systems of reasoning and representation).

The exploration required to develop an understanding of relationships between these several sorts of knowledge systems is likely to lead to a deep reorganization of the foundations of epistemology itself. For example, the doctrine of reductionism—which holds that phenomena at higher levels of science can be explained by reduction to laws and facts at lower levels—is a doctrine which is far too simplistic and inadequate to provide understanding of how to build program synthesizing systems. Reductionism may describe the end product of program synthesis—i.e., the already synthesized program—but it seems inadequate to describe the process of synthesis itself.

The doctrine of reductionism is important in programming. Every large software system is living testimony to the idea of reductionism. In such systems, we organize the data and operations at given levels to represent the information and problem solving processes required at higher levels. When we engage in "top-down programming" or "programming by stepwise refinement," we consciously organize the act of large system construction into a number of levels of reductionistic choice, each carefully selected to be intellectually manageable and to permit flexibility in future maintenance.

While every synthesized software system is an example of successful reductionism in which the external high-level behaviors of the system have been reduced to a composition of the lowest level behaviors of the underlying machine, there may be a paradox. The knowledge systems in the heads of programmers who build the overall system may not be similarly reducible. Rather, such knowledge systems may exhibit phenomena akin to the logical independence of the parallel postulate from the remaining axioms of Euclidean geometry.

The history of science teaches us that large increments in technological power are seldom purchased through cheap tricks or shallow understanding. Rather, deep basic understanding of a degree of indirectness scarcely imagined by original explorers seems more characteristic (recall here the history of the conquest of flight, or the history of the quest to transmute the elements).

It seems to me that basic understanding of how to represent knowledge systems, and how to use one knowledge system to represent and solve problems posed in another, is a basic requirement of the task of building program synthesis systems. Perhaps it is better to make an indirect assault.