WINOGRAD79
Winograd, Terry; Beyond Programming Languages; CACM 22, 7 (July 1979) pp. 391-401.

Many experts today are of the opinion that available programming languages are inadequate for building computer systems. Indeed, the quandary which we face may be much more serious than that of the early 1950's which led to the development of high level languages, since the problems for which solutions are currently being sought are much harder to solve, and the cost of not solving them is correspondingly higher. Perhaps at least part of the problem is due to the fact that the view of programming and programming languages has not changed to reflect current trends.

A widely accepted view is that programmers are expected to design algorithms for carrying out tasks, and to specify these algorithms as a precise series of instructions which computers are expected to follow. High level programming languages exist to simplify the writing of these instructions by providing "basic building blocks" which are of higher level than those of the computer. The problem with this approach, however, is that the nature of computer programming has changed: computers are now more commonly just components in larger, more complex systems (as opposed to being complete systems on their own), and the "building blocks" required for systems development are not at the level of programming language constructs, but rather are "subsystems" or "packages" of data structures, programs and protocols. Additionally, much more time (and money) is spent integrating, modifying and explaining existing programs, rather than writing new programs.

The emphasis should be on a declarative, rather than imperative, approach. Rather than specifying sequences of instructions to be followed to complete a task, instead describe, in as formal a manner as possible, what is to be done, and to what. This approach is consistent with current work in specification languages, structured programming formalisms and denotational theories of programming semantics, all of which emphasize the description of the results of the computations, rather than the instructions followed to obtain those results.

As an example of why this shift is required, Winograd presents a "motivating example" of an extension to a room scheduling system. While the proposed system is quite complex (Winograd describes it as "just at the edge of our programming powers today"), each individual component, or some variation thereupon, definitely is in existence now. The difficulty comes in trying to integrate the various components into a single entity. An added complexity is also presented if it is assumed that the person having to make the changes to the existing components will not be completely familiar with them, and may even be a new programmer.

Clearly, there are multiple "domains" to such a complex system. Winograd describes three such domains (subject, interaction, and implementation), and demonstrates how there are multiple viewpoints possible within each of these domains. Rather than simplifying the task, these independent viewpoints and domains would seem to further complicate matters. How, then, is one to specify computer applications?

Examining how people deal with understanding complex problems, it has been noted that allowing imprecision (when precision is not required) can actually serve to reduce complexity. This phenomenon, and others like it, were first observed in the study of natural languages, so it would seem logical to attempt to apply various principles from the area of Artificial Intelligence to the problem at hand.

The problem presented is clearly a very difficult one, and one which cannot be solved within a single paper. While Winograd proposes some plausible approaches, much more work is required before a feasible alternative to today's high level programming languages and computer systems can be developed. The dilemma being faced, though, is a very real one, and the directions indicated within this paper appear to be very promising.