THE FIFTH GENERATION

第五代

Kyoko Okamoto

SECOND OF THREE ARTICLES

SWIFT MECHANICAL LOGIC

By JANET RALOFF

Logic is the art of making truth prevail.
—La Bruyère, *Characters* 1688

The human mind learns or calculates largely through inference—the reasoning that underlies logical deduction. Does that make reasoning through inference the best way to program a machine to learn or calculate? The Japanese think so, which is why the soul of their "fifth-generation" computer system (SN: 5/26/84, p. 330), is expected to be based upon the nonconventional and still largely experimental class of computer dialects known as logic-programming languages. The Japanese are attempting to engineer a new genre of computers whose basic unit of operation is the inference—hence their name, "inference engines."

It is because these machines are to possess such a high level of intelligence—technically, artificial intelligence—that a conscious decision has been made to program their electronic brains with logic languages. No mere number crunchers, the goal is to have these devices deduce split-second conclusions from incomplete data, sort of a speeded-up version of the educated guess.

Though researchers developing artificial-intelligence systems have made substantial progress toward demonstrating deductive reasoning in a machine, none has approached the Japanese goal of one billion logic-inferences per second (lips)—the famous gigalips. "Your better implementations [representations of a program language on a specific computer] are approaching 100,000 logic inferences a second," notes Ross Overbeek of Argonne National Laboratory outside Chicago, whereas "the Japanese are talking about execution rates on the order of 10 million to a billion." How to get from here to there "is a problem that fascinates me personally," Overbeek says. And that's evidenced in the building of software (computer programs) for constructing inference engines he and E.L. Lusk have undertaken and written about (*Automated Reasoning*, Prentice-Hall Inc., 1984).

Unlike most of his U.S. colleagues similarly involved with artificial-intelligence engineering, Overbeek has abandoned use of LISP, a functional-programming language, for PROLOG, the predominant logic-programming language. "See, I tend to agree with the Japanese," he says, "that expert systems will be the main application—certainly a central application—of computers in the 1990s, and that logic programming is probably the right vehicle for implementing expert systems."

These expert systems are computer programs that contain both the knowledge and ability to make deductions in a narrow, specialized field, such as heart disease diagnosis. "And the appeal of logic programming for this application," explains Gary Lindstrom at the Massachusetts Institute of Technology in Cambridge, "is that the program representation is in a rule format."

A typical rule might be: *If* a person smokes, *then* that person has an elevated risk of developing heart disease. "This format is nice because you can collect bodies of knowledge in a fragmentary and accumulating kind of way," Lindstrom says. "You don't have to write the whole program from top to bottom before you can try it out."

What's more, he says, "LISP is written in an algorithmic fashion in that one writes pieces of the program—called functions—which do specific tasks when explicitly invoked in a planned sequence." By contrast, the rules of logic programming "are invoked in unforeseen sequences by the deduction that's done in a particular problem," Lindstrom says.

"Logic programming, fundamentally, is a way of doing an orderly search in attempting to solve a problem," he points out. For example, an expert system might be asked to determine whether John has measles. First it would phrase the problem as a premise to be proved, such as: John has measles. Then the program would search for useful rules in establishing the conditions for measles, such as: If John has red spots on his face, then he may have measles or acne; if he has acne, then it's likely he's under 18; if John has measles, then he probably has a fever; and if he has acne, then it's unlikely he has a fever.

In its attempt to reason, the program would apply various combinations of these and other rules to establish whether there are sufficient data to indicate John has measles. "In fact, what's really going on in the execution of a logic program is theorem proving," Lindstrom explains. A program systematically searches for data, in the form of rules, that will help it either prove or refute the correctness of the original premise—in this case, that John has measles. Along the way, this search will undoubtedly follow some false leads. When the futility of such a path becomes obvious, the program must backtrack and begin again along an alternate route.

One of logic programming's strengths is that it has been specifically designed to employ searching and backtracking as fundamental operations. By contrast, Lindstrom points out, in LISP, "Though you can program searches, they are not an inherent part of its execution."

"We [who program with a logic language] believe that practically everything to be solved must be logical," explains Harvard University mathematician Jan Komorowski, another convert from LISP to PROLOG. "I don't want to say that humans think in logic, but at least we try to think in a logical way." So, he asks, why program in languages like LISP that take a roundabout mathematical path to emulate logic when there is a language that allows one to use logic directly?

"We find logic is perhaps the best way to store information in a computer," Komorowski says. "So we use a logic [language] to express information, a logic query to ask questions and logical deduction to answer those questions. We don't use anything but logic."

Computer linguist Alan Robinson at Syracuse University in New York also prefers logic programming. "But then I'm in

Fifth-generation computers are expected to be engines of inference

the very special situation," he explains, "that what people call logic programming actually grew out of some work I started 20 years ago in computational deductive processing"—known as resolution.

"The whole point of logic programming," he says, "is that you're explicitly doing axiomatic descriptions of problems and deducing answers from those axioms. The deduction system is resolution." A machine-oriented inferencing system, "resolution works on two sentences—both with an 'if..., then...' structure—to deduce a third sentence of the same structure," Robinson explains. In the first step, the internal syntactic structure of the initial two sentences is analyzed with the goal of making a substitution. "The key idea," Robinson says, "is to try to create by manipulation a case which looks like:

If A, then B [sentence 1]
and
If B, then C [sentence 2]

where B is now common to both sentences." From that, through inference, one can deduce:

If A, then C. [sentence 3]

In most problems, when the computations begin, the Bs will not appear to be identical. Through a process called unification, however, two things that initially look different are changed so that they eventually look alike. This process usually involves having a program search for and substitute appropriate, actual values for variables. According to Robinson, "unification is something that goes on at least half the time in a logic-programming machine."

Functional programming—more popular than logic programming, but still rather uncommon itself—is characterized by the application of mathematical functions to a symbol. "What you do," Robinson says, "is successively replace expressions like (+3,2) with 5. [That's the LISP notation: the function comes first; in this case it's addition, as signified by the +. What follows is the 'argument'—3,2—to which the function will be applied.] Functional programming computation is just replacement of applicative combinations by their result. It's mathematical, but not necessarily numerical. LISP involves functions applied to symbolic entities of any kind, not just numbers."

PROLOG, developed in France around 1970, has until lately provoked little interest outside Europe. LISP, which was developed in the United States beginning around 1960, continues to be the predominant logic- oriented language used for artificial-intelligence research in this country. However, the Japanese decision to go with PROLOG for their fifth-generation project has both heightened interest in the logic language and stimulated intense debate over its merits relative to LISP.

What most people don't stop to realize, Robinson contends, is that like PROLOG, LISP too is deductive. "When you compute with functions, you're also working with axioms," he says. "What you write down as a definition is an axiom about that function," he notes. Computing in LISP, he explains, amounts to "substituting equals for equals to get equals—classical, euclidean logical inference sets." Therefore, "functional programming is deduction with equations," he says, while logic programming is "computing with conditional sentences."

In an interview with SCIENCE NEWS, Kazuhiro Fuchi, director of research on the Japanese fifth-generation project, acknowledged, "We are designing at present a PROLOG machine." But he added that he had no commitment to PROLOG and in fact suspects the prototype system his researchers ultimately develop will use either a modified and customized son-of-PROLOG, or some altogether new logic language. However, he said, for the present, "if we want a logic-programming language, there is little to choose from but PROLOG."

Robinson would also like to see some choice, which is why he developed LOGLISP. "As its name is intended to suggest," he explains, "it's LISP and it's also logic." By hybridizing the two types of programming in this language, he says one gets "the advantages of both and something neither of them has—the possibility to do both types of computing at the same time." The version that exists today is essentially the standard LISP language to which logic-language features have been added. "What we're going to do now is start over, not with LISP, but from scratch," he says. "We'll design a single language in which people will find a means to do both functional and logic programming. But the language will have a rationale which encourages people not to think of it that way"—as being a merger—but rather as a comfortable and flexible approach to deductive programming.

Does it make a difference in which deductive "tongue" a computer has been programmed? Several years ago, computers created specifically to handle LISP instructions had a definite speed advantage, Overbeek says. "But current research indicates you can produce a PROLOG machine that would be capable of pretty impressive [inference] execution rates — rates similar to those available with [commercial] LISP machines," he says. Moreover, he adds, "I think PROLOG is an easier notation to work with, unifying data-base theory with some of the issues of knowledge representation. And it's probably easier to teach."

Komorowski points to another advantage: "It's usually very difficult to connect a data base to a programming language" so that you can retrieve information from it. "And LISP [users] have more problems connecting than we [PROLOG users] do," he says, "because logic languages accommodate the inclusion of knowledge"—a representation of those data bases—within a program. What's more, writing certain types of programs can be much easier with logic programming, he maintains.

By way of example he points to the writing of codes for a compiler—a computer program that translates a high level language, such as PROLOG, into the lower machine-oriented language from which computers take their actual instructions. Writing a compiler program "is about 10 times shorter in PROLOG," Komorowski says. "That doesn't mean that we develop a faster compiler," he says, "but we can write a compiler much faster." That's important, he explains, because "human resources are usually more expensive than computer time."

All who use logic-oriented programming note it's unlikely their languages will supplant the more conventional ones like FORTRAN, BASIC and PASCAL. Explains Komorowski, "There are some domains we don't do at all," and number crunching is one of them. Why? "We don't yet know how," he says. "The logic of this is not really well understood." □

Next: Lightning-quick machines