Southern Illinois University Carbondale
OpenSIUC

Publications | Department of Medical Education

7-1988

Bridging the Gap between Object-oriented and Logic Programming

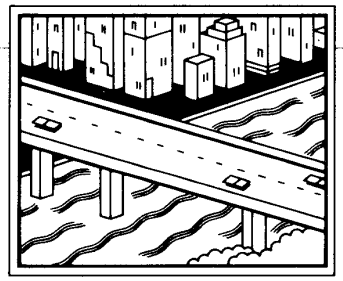
Timothy Koschmann
Southern Illinois University Carbondale

Martha Walton Evans
Illinois Institute of Technology

Follow this and additional works at: http://opensiuc.lib.siu.edu/meded_pubs
©1988 IEEE.
Published in *IEEE Software*, Vol. 5, Issue 4 (July 1988) at 10.1109/52.17800
Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

Recommended Citation
Koschmann, Timothy and Evans, Martha Walton, "Bridging the Gap between Object-oriented and Logic Programming" (1988). *Publications.* Paper 8.
http://opensiuc.lib.siu.edu/meded_pubs/8

This Article is brought to you for free and open access by the Department of Medical Education at OpenSIUC. It has been accepted for inclusion in Publications by an authorized administrator of OpenSIUC. For more information, please contact opensiuc@lib.siu.edu.



Bridging the Gap between Object-Oriented and Logic Programming

Timothy Koschmann, *Xerox AI Systems*
Martha Walton Evens, *Illinois Institute of Technology*

Object-oriented and logic programming each have advantages. This interface bridges the two styles, letting you take equal advantage of both.

In recent years, many programmers have begun to experiment with alternative programming styles. Object-oriented and logic programming have attracted growing interest among software designers, particularly among those working on knowledge-based applications. Both styles of programming offer real advantages over traditional programming methods.

Object-oriented programming is particularly useful for problems in which data objects can be categorized hierarchically. The notions of inheritance and data encapsulation encourage a structured implementation style and enhance the maintainability of programs. But object-oriented environments lack generalized facilities for deductive retrieval and pattern matching, functions that are basic to most knowledge-based applications.

On the other hand, logic programming languages like Prolog have built-in facilities for deductive retrieval through chronological backtracking and pattern matching via unification. Prolog's declarative style provides a natural way to represent rule-based knowledge. Unfortunately, Prolog also has deficiencies: Its declarative style is clumsy for tasks that are implicitly procedural in nature, such as prompting a user for a series of pieces of information. Also, because Prolog is a relatively new language in the US, it does not yet have a library of routines for graphics, manipulating windows, creating menus, or monitoring mouse events.

The best solution would seem to be one where you could use object-oriented and logic programming in a common application, using each for what it does best. This is the philosophy behind the notion of multiparadigm programming.[1] The question that must be addressed is how you combine object-oriented and logic programming so the best features of each paradigm are preserved. (The box on p. 39 describes each style.)

0740-7459/88/0700/0036/$01.00 ©1988 IEEE

One strategy might be to implement one paradigm in the other, either by implementing objects in an existing logic environment or by implementing a Prolog-like search facility in an object-oriented environment. Both approaches have been tried:

- There are several ways to implement objects in a logic programming framework. For example, Malpas has described a technique for implementing classes, instances, methods, and inheritance in Prolog.[2]
- The Xerox Palo Alto Research Center has tried the opposite approach of providing logical variables and backtracking search in an object-based system: The Common Log system was designed to provide the features of a Prolog-like language implemented on top of the Common Lisp Object Specification.

There is a third approach: an interface between an existing object-based system and a logic-based system. Abbott calls such an interface a Boolean bridge.[3] There are two advantages to creating such a bridge: First, you can access all the support tools of both environments. Second, there is no degradation of performance due to one language being implemented top of another.

This article describes an interface that was developed between Loops[4] and Xerox Quintus Prolog. Loops is an extension to the Xerox AI Environment to support object-oriented programming; Xerox Quintus Prolog is a version of Prolog that runs on Xerox Lisp Machines.

The interface has three layers. At the lowest level, a set of Prolog predicates gives the Prolog programmer access to Loops objects. This lowest level is the bridge from Prolog to Loops. At the next level, programming tools in the Loops environment let object methods be defined in Prolog. At the highest level, the Prolog programmer may treat Prolog clauses as Loops objects that can be manipulated outside the Prolog database.

You may use each layer independently. For example, in some applications, there may be no need for clause objects or Prolog methods — but the ability to manipulate Loops objects from the Prolog environment may still be valuable.

My colleagues and I at Xerox AI Systems used the interface to implement Rotamer, a fairly complex knowledge-based application.[5] Rotamer uses a truth-maintenance system (implemented with Prolog methods) to cache information acquired in the course of exploring a large search space. Our early concerns were that the overhead caused by the calling between Lisp and Prolog would lead to major performance problems. These fears proved to be groundless. On the Xerox Lisp Machine, the cost in machine cycles for a call to a Lisp function from Prolog is only slightly more expensive than a simple function call in Lisp.

Our early concerns were that the overhead between Lisp and Prolog would cause major performance problems. These fears were groundless.

Accessing Loops from Prolog

Figure 1 shows how Prolog is implemented on the Xerox Lisp Machine. In this implementation, Prolog and Lisp each have their own set of microcode. The microcode sets are stored in two different banks of control-store chips. In a sense, the Lisp Machine becomes a Prolog Machine when it is executing Prolog microcode. Because both sets of microcode are resident at all times, there is no special overhead associated with decoding Prolog instructions versus Lisp instructions. To the programmer, Lisp and Prolog appear to be two separate worlds, each with its own name spaces and syntactical rules. Access to the Lisp world from Prolog is achieved through built-in predicates that let you invoke Lisp functions while in Prolog and capture the returned results.

The Xerox Quintus Prolog/Lisp interface was readily extensible to provide a Prolog/Loops interface. The principle of data encapsulation that underlies object-oriented programming dramatically simplified the task of writing an interface to the Loops system. In Loops, everything is represented as objects. All communication to and from Loops objects, with the exception of some low-level data-access routines, occurs through the message-sending mechanism.

Therefore, only a little coding is required to provide the Prolog programmer the full range of capabilities to manipulate the Loops objects available to the Lisp programmer. A set of predicates is required so the Prolog programmer can retrieve the contents of an object's slots, insert new values into the slots of a specified object, and send messages to an object. This functionality is provided through three Prolog procedures: get_value/3, put_value/3, and send_message/4.

Using the Loops access predicates, the Prolog programmer can easily create objects, manipulate the contents of object slots, and execute object methods. For example, you could use the following Prolog procedure to perform a slot-value substitution in a Loops object:

```
substitute_slot_value(Object,Slot,
    Old_value,New_value) :-
  get_value(Object,Slot,Old_value),
  put_value(Object,Slot,New_value).
```

The first goal checks to see if the value

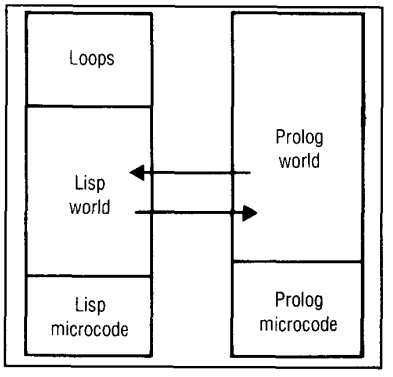


Figure 1. Lisp and Prolog methods.

stored in a slot matches the target value. If this goal succeeds, the second goal will always succeed and will install the new value in the slot as a side effect.

The Prolog/Loops interface is completely adequate for the Prolog programmer who merely wants to write code that will reason using the information represented (explicitly or implicitly) in the Loops object world. However, some programmers may want to define methods that refer to Prolog procedures. Indeed, they may want to define entire methods in Prolog. To satisfy these programmers, the interface must allow calls to Prolog procedures from the Lisp world.

Prolog procedures as methods

Loops was initially designed to offer the features of a Smalltalk-like language to the Lisp programmer. It provides facilities to define classes and methods and to create instances. Loops comes with a generous collection of system-defined classes and a set of elegant programming tools. These tools include ones that display and manipulate class hierarchies. You can invoke most common programming operations with a mouse.

In the Loops environment, object methods are invoked by sending a message to the object. The syntax for sending a message is

```
(← $ObjectName Selector arg1 arg2 ...
  argN)
```

In this case, $ObjectName is the handle for the object to which the message is being sent, and Selector is the name of the method to be invoked.

Loops methods are themselves objects. Loops methods have two components: a method object and a functional definition. To work, the Loops/Prolog interface needed a way to define methods in Prolog that would let the methods be invoked via the Loops message-sending mechanism. Ideally, it should be transparent to the sender whether the method is defined in Lisp or Prolog.

Message-sending mechanism. Xerox Quintus Prolog provided a way through a Lisp function named PROLOG to facilitate the creation of Prolog queries from Lisp. If a Loops programmer wanted to invoke the Prolog procedure foo/3, using the function call

```
(PROLOG 'foobar (LIST 'arg1 *VALUE*
  'arg3))
```

would have the same effect as the Prolog query

```
foobar(arg1,X,arg3).
```

The interface treats the Prolog procedure as though it were determinate — it will always return only the first answer. Variables are passed by substituting the global symbol *VALUE* for each variable in the argument list. The above call to PROLOG would return a list containing the first value to which the variable *X* had been unified.

The PROLOG function provides a bridge from the Lisp world to the Pr[illegible] world. You could simply embed calls to PROLOG into Loops, but these would not be true Prolog methods because they would not be visible to the Loops environment. In this simple approach to integrating Prolog into Loops methods, object-oriented Loops features such as method inheritance and method combination may not work properly.

The solution to this problem is to integrate Prolog-defined methods more closely into the Loops system. This can be done by specializing the class Method in Loops to define a new class called PrologMethod. PrologMethods have four components: a method object, a functional definition, the text representing the Prolog source code, and a set of clauses that have been interpreted or compiled into the Prolog database.

Sending messages to invoke a PrologMethod object could be done precisely the same way you would send a message to invoke a method defined in Lisp: When a message is directed to an object, the SEND

```
% Prolog Method
% Class: Class
% Selector: PatternMatch

'$Class.PatternMatch'(Self,Arg1,Arg2,...,Result) :-
  goal_1,
  goal_2,
  .
  .
  .
  goal_K.

'$Class.PatternMatch'(Self,Arg1,Arg2,...,Result) :-
  goal_1,
  goal_2,
  .
  .
  .
  goal_K.
```

Figure 2. Template for a Prolog method.

function determines which definition of that method to use by checking the list of superclasses for that object. If the appropriate method definition happens to be written in Prolog, the PROLOG function will be called instead of applying a Lisp function to the list of arguments. The SEND function would return the first element of the list returned by PROLOG.

It would also be convenient if the way you define a Prolog method were compatible with the way you define a Lisp method. Most Loops programmers define new methods through a special browser window known as the Class Inheritance Lattice (see Figure 5 for an example).

The Class Inheritance Lattice is your interface to an object-oriented application. With a mouse, you can select menu options like Create Method, Edit Method, and Edit Class Description. In the extended interface, a menu option for creating a Prolog method would be added.

If you selected this item, you would be prompted for a method name. A Prolog method template similar to the one in Figure 2 would then appear on the screen. The system would fill in the Class and Selector fields. It would compose the procedure name by concatenating the class name with the selector name and inserting a $ at the beginning of the name.

Following the convention used in Loops methods, the first argument to the Prolog procedure would be a pointer to the object to which the message was sent. You would

Object-oriented versus logic programming

Object-oriented and logic programming both represent radical departures from how programmers have traditionally designed programs and solved problems. Because both styles have the full expressive power of a Turing machine, they are equivalent in power. But the two styles are not necessarily equivalent in the ease of implementation; that usually depends on the application.

Object-oriented programming. Object-oriented programming was first developed as a convenient approach to implement simulation problems and distributed operating systems. The notions of objects, classes, and message sending were first introduced in the Simula language. Organizing code and data around the objects they represent proved to be a very general and natural way to conceptualize applications.

Objects can possess local storage in the form of slots, and they have a repertoire of behaviors called methods. An object's method is invoked by sending a message to the object. Most object-oriented languages use three types of objects: classes, metaclasses, and instances.

A class represents a template for all members of a set of objects. For example, the class Programmer defines a set of objects that represent people who write programs. The class Programmer may have a superclass, such as Person or Employee, and it can in turn have subclasses such as LispHacker.

A class object contains a description of the class's members. For example, the class Programmer defines a set of objects that represent people who write programs. It may specify that all Programmer objects should have a slot to store the list of programming languages they use. The class Programmer may have a superclass, such a Person or Employee. It can also have subclasses such as LispHacker. Members of a class are called instances of the class. For example, Tim might be an instance of the class LispHacker.

Metaclasses are a special category of class objects whose instances are always classes. A major convenience of the object-oriented paradigm is that objects can inherit default characteristics (methods and slot values) from their superclasses. The class Programmer, for example, could inherit variables such as Employer and Employee Number from the superclass Employee.

Logic programming. In logic programming, you program by creating a database of axioms. The program is executed by entering a theorem and asking the system to find a proof given the set of axioms. Prolog, the most popular example of this programming style, uses first-order predicate logic to derive theorems from the set of axioms in a database. Prolog databases are constructed of facts and rules. An example of a Prolog fact is

```
temperature(jones,101,oral).
```

declarative reading of this statement might be "The patient Jones has an oral temperature of 101 degrees." Prolog rules are if-then statements of the form

```
fever(Patient) :-
  temperature(Patient,Temp,oral),
  Temp > 100.
```

This rule would read, "If a patient has an oral temperature in excess of 100 degrees, that patient has a fever." With this pair of Prolog clauses, you could ask the system to prove a theorem that Jones has a fever. Even though there are no facts in the database about Jones or fevers, Prolog could derive the theorem using the fever rule and the single fact about Jones's temperature.

Styles' strengths. Both styles have their strengths.

Object-oriented programming is particularly useful for problems where data objects can be categorized hierarchically. The notions of inheritance and data encapsulation encourage a structured implementation style and enhance program maintainability. Indeed, it has been said that object-oriented programming is to the 1980s what structured programming was to the 1970s. Object-oriented programming has been used extensively in model building, computer-based simulations, and as a tool for knowledge representation in expert systems.

Prolog, with its built-in facilities for backtracking and unification, is a natural choice for any application requiring deductive retrieval. Logic programming has been used in many areas, including computational linguistics, database research, and knowledge-based systems.

```
(PrologMethod ((Class PatternMatch) self Slot Pattern)
                                                    (* tdk: "20-Apr-87 16:36")
(* Metamethod for finding all instance of Class with Slot values equal to Pattern)
  (CAR (PROLOG (QUOTE $Class.PatternMatch)
               (LIST self Slot Pattern (QUOTE *VALUE*)))))
```

Figure 3. Functional definition for a PrologMethod.

edit the template to specify other arguments and to add the appropriate clauses. The text editor has options to interpret or compile the Prolog code in the window.

Example. As a simple example of a PrologMethod's use, say that you wanted to create a method to search through the list of instances belonging to a certain class to find all instances that hold a given value in a specified slot. You can take advantage of Prolog's built-in backtracking and unification mechanisms to conduct this search. You will define PatternMatch as a PrologMethod of the metaclass $Class. Using the template provided for PrologMethods, you can define the functional definition as shown in Figure 3.

As with all Loops methods, the first argument to the method is always bound to the object to which the message was directed. In this case, you are adding two arguments to hold the name of the instance variable and the value you want to match. The global variable *VALUE* represents a variable in your Prolog query. You will use this variable to hold the list of objects that match the pattern. When using *VALUE*, the result returned by the function PROLOG is always a list — in this case, yielding a list in a list. What you really want then is the first element (the CAR) of the list returned by PROLOG.

When you exit from the editor window after creating the functional definition for your PrologMethod, you will be asked to create a TEdit window to edit the Prolog source code for the method. Like in the functional definition, the system provides a special template to help you create the Prolog source definition for the PrologMethod. The naming convention for PrologMethods is similar to the Loops convention for the Prolog predicate that will be associated with the PrologMethod. The name given to the Prolog procedure (the functor) is constructed from the name of the class for which the PrologMethod is defined and from the name of the selector for the method. Figure 4 shows the example's PrologMethod after editing.

This simple pattern matcher could be modified to use more complex patterns such as ranges and match values for multiple slots. The advantages of using Prolog become obvious when you design methods to perform more complicated pattern matching.

```
% Prolog Method
% Class: Class
% Selector: PatternMatch

'$Class.PatternMatch'(Self,Slot,Pattern,MatchList) :-
  send_message(Self,'AllInstances!',[],ObjectList),
  match_objects(ObjectList,Slot,Pattern,MatchList)._

match_objects([],_,_,[]).
match_objects([Object|Tail],Slot,Pattern,[Object|Rest]) :-
  get_value(Object,Slot,Pattern),
  match_objects(Tail,Slot,Pattern,Rest).
match_objects([_|Tail],Slot,Pattern,MatchList) :-
  match_objects(Tail,Slot,Pattern,MatchList).
```

Figure 4. Example PrologMethod.

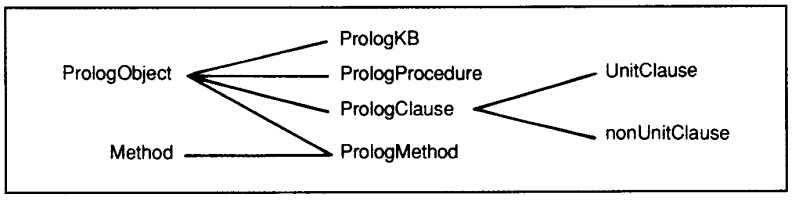


Figure 5. Class-inheritance browser for PrologObject.

Rule-based approaches. Organizing Prolog code as methods of objects is very similar to how Loops handles rule-based programming. Loops supports a rule language that lets you write decision code in a production-like, non-Lisp syntax. Loops rules are embedded in special Method objects known as RuleSetMethod objects. When RuleSetMethod objects are compiled, the rules are simultaneously translated into Lisp code and converted into individual Rule objects.

But Loops discourages you from manipulating the Rule objects. The objects representing the rules in RuleSetMethods are not documented. In fact, most Loops programmers working with RuleSetMethods may not even be aware of their existence. They exist for the convenience of the Loops system implementers, not for application programmers.

This sharply contrasts to how rule-based programming is done in Intellicorp's KEE expert-system shell. In KEE, you create

and manipulate rule objects explicitly.[6] In KEE, rules are first-class objects. KEE rule frames have predefined slots such as PREMISE, ACTION, and ASSERTION — and you can add new slots if you want.

Prolog clauses as objects

There are many situations where you may want to treat Prolog facts and rules as objects. Different sets of clauses may represent different (perhaps contradictory) views of a problem. When clauses are represented as objects, it is easy to switch views by asserting and retracting sets of clause objects. Another situation is during the design of knowledge-engineering tools. Representing clauses as objects allows greater flexibility in the design of sophisticated tools for creating and maintaining large knowledge bases.

Representing Prolog clauses as objects is straightforward. Figure 5 shows one scheme that can define Prolog clauses and other objects composed of Prolog clauses. Figure 5 contains a Loops class-inheritance browser for the class PrologObject. PrologClause is shown as a subclass of PrologObject. The classes UnitClause, representing Prolog facts, and nonUnitClause, representing Prolog rules, are specializations of PrologClause. PrologClause objects would have slots to store things like the functor name, number of arguments, and list of arguments.

PrologProcedure objects would be composite objects containing an ordered list of PrologClause objects with the same functor name and number of arguments. PrologKB objects would be composite objects containing a list of PrologProcedure objects. Loops clauses may have more than one superclass; auxiliary superclasses that provide extra slots or methods to their subclasses are called mixins.[7] PrologObject serves as a mixin for the class PrologMethod.

The class PrologObject provides certain slots for storing information, such as the creation date and the author, that are inherited by all its subclasses. This kind of documentation is important when creating and maintaining a large knowledge base.

Bobrow et al. presented the idea of organizing source code into object-based structures known as definition groups.[8] Representing source code as objects can significantly simplify management for large projects undertaken by programmer teams. Using objects to represent Prolog program elements would let the software librarian for a project team apply these ideas to logic programming as well.

Objects can also store information describing the deductive relationships between clause objects. This would include information about what other clauses may have been used to derive this clause, as well as what assumptions were made for this clause. You can use dependency information of this type to construct truth-maintenance systems,[9] which are frequently used in expert systems to keep track of the assumptions underlying a conclusion.

When object-oriented and logic programming styles are combined, they unite synergistically. Loops and Prolog complement each other very constructively.

When object-oriented and logic programming styles are combined, they unite synergistically: Loops and Prolog complement each other very constructively. Loops's frame-like inheritance and data-encapsulation facilities, combined with Prolog's built-in facilities for pattern matching and deductive retrieval, produce an ideal environment for prototyping knowledge-based applications.

Loops also supports a style of programming known as access-oriented programming.[1] You can attach procedures to specified slots; these procedures get invoked whenever the slots are accessed. A slot with an attached procedure is called an active value. You can use active values to implement demons, drive gauges, and implement data probes, among other things. The Loops/Prolog interface is completely compatible with the use of active values, so you can easily combine object-oriented, access-oriented, and logic programming.

The notion of representing inferential data both as objects and as Prolog clauses offers special advantages to the system designer: In Prolog, a rule ceases to exist as an entity in the database when it is retracted. However, if the rule is also defined as an object, it can be asserted and retracted at will without ever being in danger of being lost from the system. This feature is useful for implementing systems that are based on propagation of constraints. If constraints are represented as Prolog clauses, the set of constraints defining a problem could be expanded or contracted to reflect the addition or relaxation of constraints.

The ability to combine two or more programming paradigms produces an extremely flexible and powerful environment for developing applications of all kinds. The advantages of such an approach are clear for large knowledge-based applications:

For experienced developers, the use of high-level languages in a multiparadigm environment may be a cost-effective alternative to the use of commercial expert-system shells, which are often constrained in their abilities.

Seasoned programmers should be able to prototype systems — using a combination of object-oriented, access-oriented, and logic programming — in little more time than it would take to prototype the application with a shell.

The major benefit to the knowledge engineer is greater flexibility in the choice of a knowledge-representation scheme. The knowledge engineer would be free to design a custom system to satisfy the special requirements of each application using the appropriate paradigm for each. ❖

References

1. M. Stefik, D.G. Bobrow, and K.M. Kahn, "Integrating Access-Oriented Programming into a Multiparadigm Environment," *IEEE Software*, Jan. 1986, pp. 10-18.
2. J. Malpas, *Prolog: A Relational Language and*

Its Applications, Prentice-Hall, Englewood Cliffs, N.J., 1987.

3. R. Abbott, "Knowledge Abstraction," *Comm. ACM*, Aug. 1987, pp. 664-671.
4. M. Stefik et al., "Knowledge Programming in Loops: Reprint on an Experimental Course," *AI Magazine*, Fall 1983, pp. 3-13.
5. T.D. Koschmann, *Conformational Analysis Using a Simplified Truth-Maintenance System Implemented with Object-Oriented and Logic Programming*, PhD dissertation, Computer Science Dept., Illinois Inst. of Technology, Chicago, 1987.
6. R. Fikes and T. Kehler, "The Role of Frame-based Representation in Reasoning," *Comm. ACM*, Sept. 1985, pp. 904-920.
7. M. Stefik and D. Bobrow, "Object-Oriented Programming: Themes and Variations," *AI Magazine*, Winter 1986, pp. 40-62.
8. D.G. Bobrow, D.S. Fogelsong, and M.S. Miller, "Definition Groups: Making Sources into First-Class Objects," *Trans. ACM Conf. Object-Oriented Programming*, ACM, New York, 1986.
9. J. de Kleer, "An Assumption-Based TMS," *Artificial Intelligence*, Vol. 28, No. 2, pp. 127-162.

Timothy Koschmann is a senior member of the engineering staff at Xerox AI Systems. Before joining Xerox, he was the chief computer scientist for the Chicago Medical School. His research interests include reasoning systems and the design of object-oriented environments.

Koschmann received a BA in philosophy from the University of Missouri at Kansas City, an MS in psychology from the University of Wisconsin at Milwaukee, and a PhD in computer science from the Illinois Institute of Technology.

Martha Walton Evens is a professor of computer science at the Illinois Institute of Technology. Her research interests include computational linguistics and knowledge-based systems.

Evens received a BA in mathematics from Bryn Mawr College, a Fulbright scholarship in Paris, an MA in mathematics from Radcliffe College, and a PhD in computer science from Northwestern University. She is a past president of the ACM and a director of AFIPS.

Address questions about this article to Koschmann at Xerox AI Systems, 3000 Des Plaines River Dr., Des Plaines, IL 60018; ARPAnet koschmann.dpmw4000@xerox.com.

Reader Service Number 6