Part I

BY TIM FININ

UNDERSTANDING FRAME LANGUAGES

Representing knowledge about some aspect of the world is fundamental to most AI systems. This is true of all kinds of AI systems: expert systems, natural language interfaces, text understanding systems, perceptual systems, planning systems, etc. It is also true for all kinds of domains over which these systems operate.

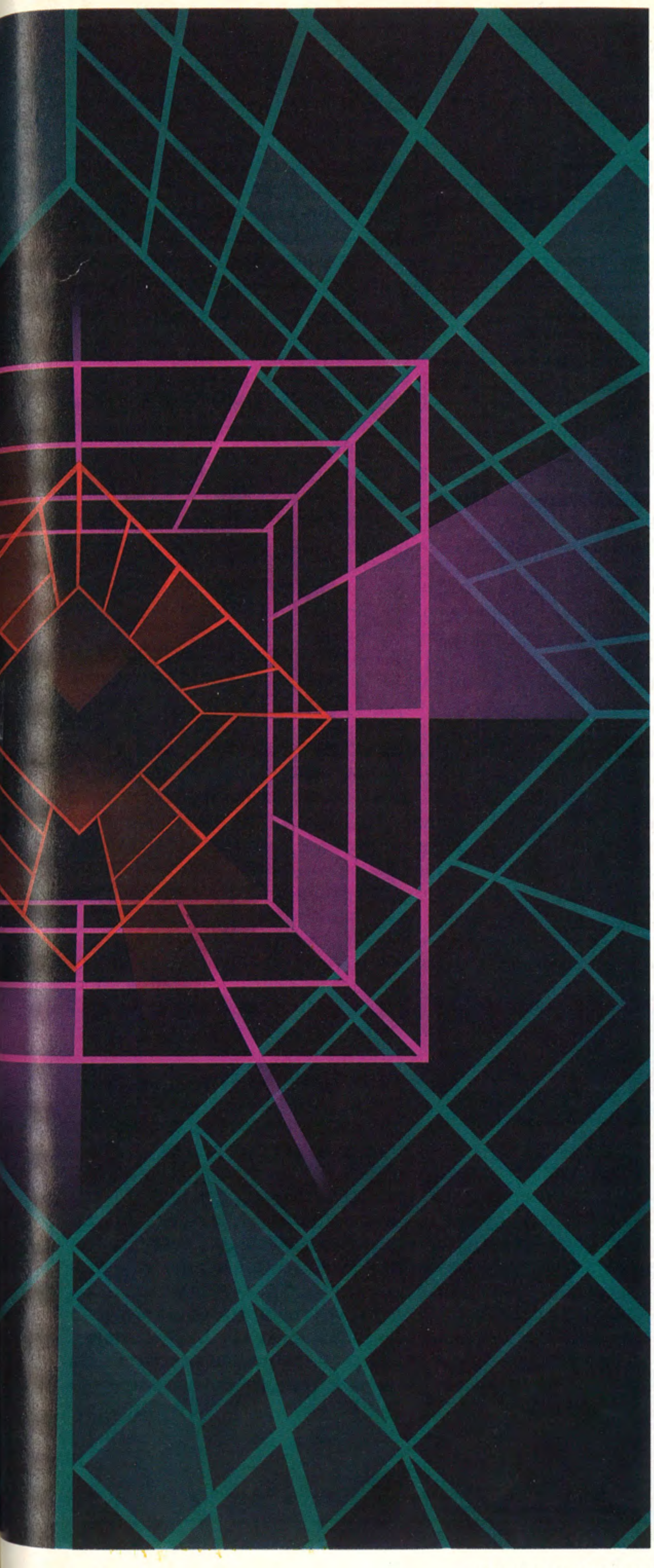
Over the last 25 years the AI community has devised and experimented with a variety of special purpose languages for representing knowledge. These languages attempt to provide AI programmers with a tool to ease the task of encoding domain knowledge and to allow the system to effectively and efficiently use this knowledge.

Although no single representation language is likely to be optimal or even satisfactory for all types of systems or all domains, a small number of generic types of representation languages have been found to have very attractive properties for a wide class of applications. Frame-based representation languages (FBRLs) form one of these classes. Other popular types of representation languages include those based on first-order logic, on production rules, or on some combination of frames, logic, and rules.

In very general terms, the need to represent some aspect of the world could be satisfied by any system in which we can name and refer to objects in the domain (whether they be primitive or complex, concrete or abstract) and to describe relations between them. The semantic network was an early representation technique that provided this capability in a straightforward manner.

A semantic network is simply a graph containing a set of named nodes and a set of associative links between them. Early work on semantic networks as AI representational

ILLUSTRATION: ADRIAN DAY



A simple language called PFL can meet many AI knowledge representation needs

systems[1-3] showed the formalism to be flexible and powerful as well as intuitively attractive from a psychological point of view.

The current notion of a frame-based representation language evolved out of attempts to remedy the deficiencies (both theoretical and practical) found in simple semantic network systems as well as a desire to provide a uniform set of representational services. Although these problems will not be discussed in this article, the interested reader will find excellent discussions in several sources.[4,5] In this article we will describe the concepts and mechanisms that underlie FBRLs by discussing a particular implementation, a simple pedagogical frame language (PFL). Part II, to appear in the December issue of *AI EXPERT,* will demonstrate some of the features of Common LISP in the context of the PFL program.

PFL is was written for pedagogical purposes—it does not attempt to be very powerful, expressive, or efficient. It is deliberately kept simple, both in its features and implementation. PFL is written in Common LISP and has been run on a Symbolics LISP Machine, a VAX (in VAXLISP), and a Hewlett-Packard Bobcat.

Although PFL is quite simple (amounting to less than 250 lines of commented Common LISP code), it is sufficiently powerful to support the representational needs of many AI applications. We will use the university world (the world of students, instructors, courses, departments, and so on) as an example in which to discuss the concepts in this article.

SEMANTICS AND TAXONOMY

Given the importance of knowledge representation to AI, it is natural to be concerned about the meaning one can ascribe to descriptions built in a given knowledge representation language. Pat Hayes[6] provided an early attempt to specify formally the semantics of FBRLs in terms of first-order logic. David Etherington and Ray Reiter[7] have looked in more detail at the problem of formalizing the exception mechanism commonly found in such languages. Many languages, such as those in the KL-ONE family (for example, KL-ONE, NIKL, and KRYPTON) have been designed to include just those features for which a formal account of meaning can be given.

On the other hand, most of the frame-based languages used today (such as KEE, KnowledgeCraft, ART, and HP-RL) do not attempt to provide a careful account of their semantics. These languages include a large and rich set of features that have been found to be of practical utility but would be very difficult to formalize.

Our PFL language, although rather simple, is in this second category. Instead of attempting a formal account for the semantics of PFL, we will instead give an operational description. The meaning of a network of PFL frames will be given in the form of descriptions of the functions used to create, modify, and access them. In particular, we will describe a small set of special slots and slot facets, which have special meaning in PFL.

FBRLs are essentially object-oriented languages in which a representation consists of a set of frames. In PFL, as in most FBRLs, a frame can represent either an individual object in the domain (for example, George Washington, the integer three, the king of France in 1985, the largest prime number) or a generic class of objects (for example, U.S. presidents, positive integers, heads of European states, prime numbers greater than 100). Note that representing an individual does not imply its existence in any possible world.

FBRLs typically have a number of features that distinguish them from other representational systems. These are:

Generalization hierarchy. The frames are organized into a generalization hierarchy in which frames inherit information from their ancestors.

Slots. A frame has a number of subunits, called slots, which can take on values or describe, in general terms, constraints on what their values can be.

Limited reasoning services. The functions for creating, modifying, and accessing the representation provide a limited number of reasoning functions, such as attribute inheritance, default reasoning, constraint checking, and classification of new frames.

The core of a representation is a collection of frames organized into a generalization/specialization hierarchy (also commonly referred to as an abstraction hierarchy, *isa* hierarchy, or concept lattice) defined by primitive directed links. In PFL, we will refer to these links as *AKO* for "a kind of" links. An *AKO* link exists between frames *F1* and *F2* just in case we wish *F1* to be seen as a specialization of *F2*. For example, Figure 1 shows a fragment of the abstraction taxonomy for the university world. The most general concept defined, *university-person,* is specialized into two subconcepts: *student* and *staff.* As the figure shows, these are further refined into still more specialized concepts. The heavily shaded frame, *john,* represents an individual person in the domain. Note that a frame may have more than one immediate ancestor, such as *teaching fellow,* which is both a kind of *graduate student* and a kind of *instructor.* Using PFL,

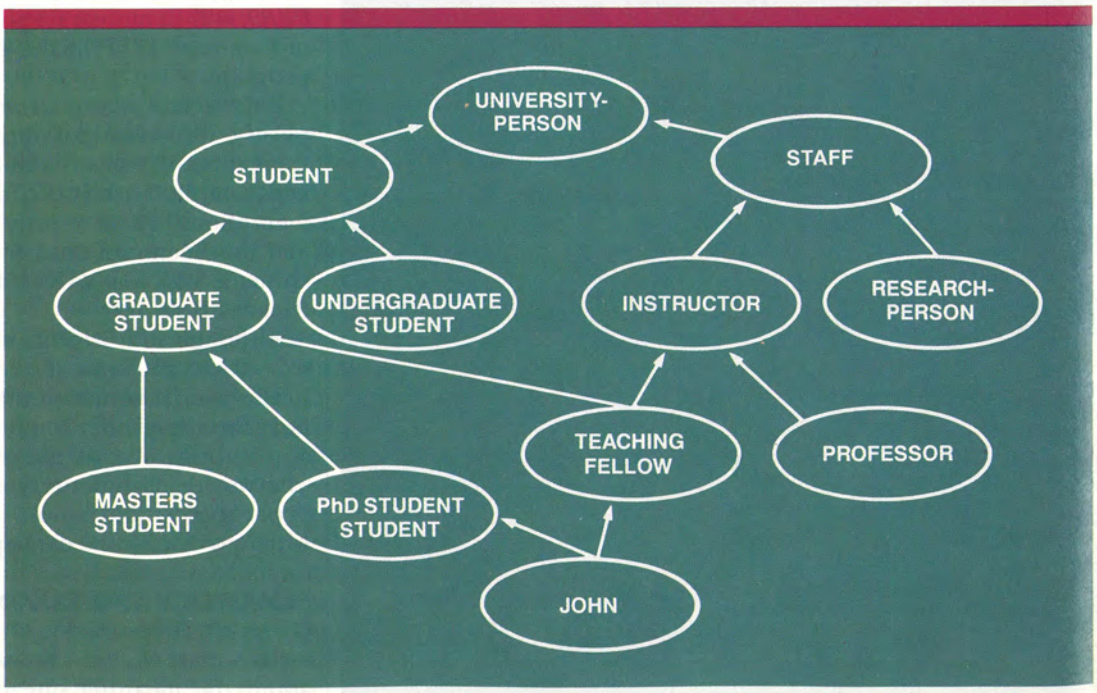


FIGURE 1. A fragment of the abstraction taxonomy for the university world.

we could define the subtaxonomy below *instructor* in Figure 1 by evaluating the following expressions:

```
(fdefineq professor instructor)
(fdefineq teaching-fellow instructor)
(fdefineq john (teaching-fellow PhD-student))
```

The *fdefineq* function is used to define a frame and to optionally specify its immediate ancestors in the taxonomy and, perhaps, some additional attributes. Note that the frame *john* is defined as being both a kind of *teaching fellow* and also a kind of *Ph.D. student.* The ability to define one concept as the intersection of several more general concepts is extremely useful and greatly adds to the expressive power of this kind of language.

The *AKO* link is significant in that it allows one frame to inherit information from another under certain restrictions. The specifics of inheritance is one of the principle ways in which frame-based languages differ. Some systems (for example, KL-ONE,[8] NIKL,[9] and KRYPTON[10]) specify that a frame necessarily inherits all information from all of its ancestors.

The approach we have taken in PFL is typical of many simple object-oriented languages and FBRLs (for example, LOOPS[11] and FRL[12]). A frame inherits information from its ancestors unless overridden by local information. Before discussing inheritance in more detail, we will describe the structure and content of a frame's slots.

One important concept in FBRLs is the notion of subsumption. One concept subsumes another if it includes it; for example, we might say that the concept of "student" subsumes that of "graduate student." In a FBRL, a frame *F1* subsumes another *F2* whenever *F2* and all of its descendants are necessarily descendants of *F1.*In some FBRLs (such as those in the KL-ONE family) the system itself has the responsibility of taking new frames to be added to the knowledge base and determining all of the subsumption relationships between it and the exisiting frames. This process is known as classification (see discussion of automatic classification[13,14]).

In PFL, however, the subsumption relationships are determined by the *AKO* links. A frame *F1* subsumes another *F2* if there is a chain of *AKO* links between *F2* and *F1.* PFL also provides a mechanism by which the knowledge-base designer can attach special procedures to frames that will be used to check candidate subsumption relationships. This facility is discussed later in the article.

Given we have a way to define objects and place them in a taxonomy, we need to have a way to describe various attributes an object has. In PFL, as in most frame languages, we express such attributes through a frame's slots. We can think of a slot as a subobject of a frame which can take on one or more values and have several additional properties specified. Collectively, we will refer to the value(s) and the other properties as facets.

For example, we could evaluate the following expression to describe the fact that *john* is both a *teaching fellow* and a *Ph.D. student* whose major is computer science and course is number 121:

```
(fdefineq john (phd-student teaching-fellow)
   (major (:value computerScience))
   (course (:value cse121)))
```

In defining a frame with the *fdefineq* function, any arguments after the second define the frame's slots and give them initial values for some of their facets. Each slot specification has the form of list beginning with the name of a slot (for example, *major*) followed by any number of facet descriptions. These have the form of a list beginning with the facet's name and continuing with the data in the facet (for example, *(.value cse121)*).

Once we have defined a frame, we can easily add and remove slots, facets of slots, or information in the facets. For example, to reflect the fact that *john* has changed his major to electrical engineering and his advisor is Mr. Chips, we could evaluate the following expressions:

```
(fremove 'john 'major :value 'computer-science)
(fput 'john 'major :value 'ee)
(fput 'john 'advisor :value mister-chips)
```

Each of these functions takes four arguments: the names of a frame, slot, and facet, and the datum to be removed or added.

FACETS STRUCTURE SLOTS

A frame's slot is much more than a place to put values. The other slot facets provide a structure that allows us to specify useful information about the slot. The structure used by PFL and the functions for manipulating these slot structures are general in that one can create facets of any name on a slot. This is a very useful feature that makes it easy to extend the representation system or to customize it for a particular application.

Eight facets have special, predefined meaning in PFL.

:value holds the actual values for the slot.

:default holds any default values for the slot. These may be used (under certain circumstances) when a value is needed for the slot, but the *:value* facet is empty.

:type constrains the possible values for the slot. The data must be frame names. For an expression to be a legal value for a slot, it must be subsumed by all of the frames in the *:type* facet. PFL will refuse to put a value which does not satisfy this type constraint.

:min contains a single positive integer that

A frame's slot is much more than a place to put a value specifies the minimum number of values needed by the slot for the frame to be well formed. The system will refuse to remove from a slot a value that would cause it to have fewer than this minimum number of values.

:max contains a single positive integer that specifies the maximum number of values allowed in the slot for the frame to be well formed. PFL will refuse to add a value to a slot if it would result in the slot holding more than this maximum number of values.

:if-needed holds procedures (known as demons) to be run whenever a value is needed for the slot but none is present. These must be two argument functions, the arguments being the names of the frame and slot for which values are sought. The procedures are invoked in order until one of them returns a non-nil result, which is interpreted as a list of virtual values for this slot.

:if-added holds demon procedures to be run whenever a value is added to the slot. These must be three argument functions (frame, slot, and value). All of the procedures are run and their return values ignored.

:if-removed holds demons to be run whenever a value is removed to the slot. These must be three argument functions (frame, slot, and value). Again, all of the procedures are run and their return values ignored.

To see how these facets can be used to build useful descriptions, consider a description of the generic concept for a student:

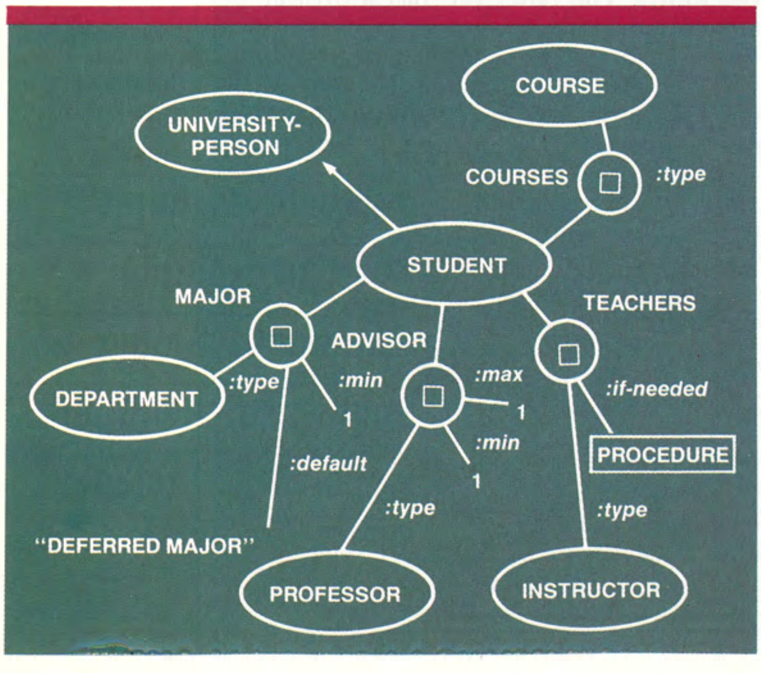
```
(fdefineq student university-person
   (major (:type department)
          (:default "deferred major")
          (:min 1))
   (advisor (:type professor)
            (:if-added add-inverse)
            (:min 1)
            (:max 1))
   (teachers (:type instructor)
             (:if-needed
               (lambda (f s)
                 (collect (lambda (x)
                            (fvalues course 'instructors))
                          (fvalues f courses)))))
   (courses (:type course)))
```

Graphically, we will depict this concept as shown in Figure 2 (this follows the graphical conventions used in the KL-ONE family of FBRLs). Each slot is shown as a small circle with an inscribed square. The individual facets of the slot are shown as arcs between the slot and their data.

This definition illustrates how the various facets in PFL can be used to describe general concepts that provide partial descriptions of the various slots. It encodes the following information:

A student is a kind of *university-person* and, in general, inherits all of its attributes. In addition, a student has four local slots: major, advisor, teachers and courses. A student must have at least one major, which must be a kind of department. If a student's major is not known, assume it is "deferred major." A student must have exactly one advisor who must be of type *professor* and can have any number of *teachers*. If the teachers are not known, they can be computed by a particular procedure. A student has any number of courses, all of which must be of type *course*.

FIGURE 2.
A graphic representation of how facets can be used to build useful descriptions.



INHERITANCE

Knowledge representation systems offer their users two things. They provide a language for building an explicit knowledge base of descriptions to represent objects and relations in some domain, and they provide a set of inferential services to allow access to implicit information that can be automatically inferred from the explicit knowledge base. For FBRLs, the typical services include attribute inheritance, default reasoning, attached procedures, consistency maintenance, and classification.

In PFL, these services are built into the functions for accessing and updating a frame's slots and facets. Of these services, the inheritance of slots, facets, and data within a facet is the most important. In general, a frame inherits all of the information in its ancestors except where overridden by locally defined data. This occurs at the level of slots within a frame as well as at the level of facets within a slot.

In PFL, the procedure for retrieving a particular facet of a slot in a frame is:

- If there is a facet local to the frame, then use its data.
- If there is no local facet, then recursively search the immediate ancestors of the frame until one provides the data.

The extraction of a value from a slot is more complex and involves the intertwining of inheritance, procedural attachment, and default assumptions. If we seek a value for slot *S* of frame *F*, then the following steps are tried in order:

1. If a local slot *S* contains a *:value* facet, its data are taken as the values.
2. If a local slot *S* contains a *:default* facet, its data are the values.
3. If a local slot *S* contains an *:if-needed* facet, its data are procedures that are invoked sequentially until one returns a non-nil value. This value is then taken as a list of the values for the slot.
4. The immediate ancestors of the frame are recursively checked in a depth first search until one provides a value.

PROCEDURES

In addition to *if-needed* procedures, which can compute values for a slot when none is explicitly recorded, PFL also provides procedures that will be automatically invoked whenever a value is added or removed from a slot. These are quite useful for maintaining a consistent knowledge base.

For example, we may wish to have a spouse slot on the generic frame for a person which represents a symmetric relationship. That is, if *A* is the spouse of *B*, then it is the case that *B* is the spouse of *A*. This can be easily accomplished via *if-added* and *if-removed* procedures as the following example shows:

```
(fdefineq person animal
   . . .
   (spouse (:type person)
           (:max 1)
           (:if-added 'add-symmetric)
           (:if-removed 'remove-symmetric))
   . . .)
(defun add-symmetric (frame slot value)
      (fput value slot :value frame))
(defun remove-symmetric (frame slot value)
      (fremove value slot :value frame))
```

Whenever a value is added to a slot of a frame, a search is made for local or inherited *if-added* procedures. Any procedures found are then called with the frame, slot, and value involved. Thus if we add the value *mary* to the spouse slot of *john*, this will trigger the invocation of the general procedure *add-symmetric* with arguments *john spouse* and *mary*, which will cause *john* to be added as the *spouse* of *mary*.

Another common technique is to use *if-added* and *if-removed* procedures to maintain inverse relations. For example, we may wish to have an advisor slot on the *student* frame and an advisee slot on the *professor* frame. If we assert that the advisor of student *S* is professor *P*, then the following will cause the fact that an advisee of professor *P* is the student *S*. A similar transaction will transpire upon the removal of a value from either the advisor or advisee slot.

```
(fdefineq student
   . . .
   (advisor (:min 1)
            (:type professor)
            (:if-added 'add-inverse)
            (:if-removed 'remove-inverse))
      . . .)
(fdefine professor
      . . .
      (advisee (:type student)
               (:if-added 'add-inverse)
               (:if-removed 'remove-inverse))
      . . .)
   (fdefine advisor slot (inverse (:value advisee)))
   (fdefine advisee slot (inverse (:value advisor)))
   (defun add-inverse (frame slot value)
      (fput value slot :value frame))
   (defun remove-inverse (frame slot value)
      (fremove value slot :value frame))
```

For this to work we introduce frames to represent the slots *advisor* and *advisee*. This allows us to express that these two slots define inverse relationships.

SLOTS WITH SPECIAL MEANING

A frame's slot represents a potential relationship between that frame and some other object (that is, the values of the slot). In general, the meaning of a slot is totally up to the knowledge base designer. Three slots, however, have special meaning in PFL. These slots are used to represent the taxonomic links between frames and to hold procedures that are used to tell when the frame subsumes or is subsumed by some other frame or expression. These special slots are:

AKO. The values in this slot are the names of frames that are the immediate subsumers (parents) in the abstraction hierarchy.

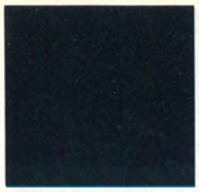
subsumes-if. The values should be one-argument predicates. If any is true of an arbitrary LISP expression *S*, then *S* is subsumed by the frame.

subsumed-if. The values should be one-argument predicates. If any is true of an arbitrary expression *S*, the frame is subsumed by *S*.

These last two slots provide a way for the knowledge base designer to bridge the gap between frames and arbitrary LISP expressions. For example, using the *:type* facet we can constrain a slot to be filled with values subsumed by a given set of frames.

Often, however, we would like to represent values using not frames but some other kind of LISP expression. To represent a person's age, for example, it is natural to use integers. We can accomplish this by defining a frame *valid-age* to represent the class of legal age values and providing it with a *subsumes-if* slot filled with a predicate to recognize the appropriate LISP expressions.

The meaning of a slot is totally up to the knowledge base engineer



```
(fdefineq person
   . . .
   (age (:min 1)
        (:max 1)
        (:type valid-age))
   . . . )
(fdefineq valid-age thing
   (subsumes-if (:value valid-agep)))
(defun valid-agep (n)
   ; returns T iff n is reasonable as a person's age.
   (and (integer n)
        (> = n 0)
        (< n 150)))
```

Before placing a value in a slot, PFL will check to see if the value is subsumed by all of the frames in the slot's *:type* facet. In PFL we have extended the subsumption relation to be one that can exist between two frames or a frame and an arbitrary LISP expression. Basically, one expression, *S1,* subsumes another, *S2,* if any of the following conditions holds:

- Both *S1* and *S2* are frames and there is a chain of *AKO* links between *S2* and *S1.*
- *S1* is a frame that has a *subsumes-if* slot, one of whose values is a predicate, which is true of *S2.*
- *S2* is a frame that has a *subsumed-if* slot, one of whose values is a predicate, true of *S1.*

This makes it easy to have PFL recognize arbitrary expressions (such as LISP integers between 0 and 150) as being subsumed by a frame (for example, *valid-age*) as well as enabling one to experiment with various definitions of subsumption.

This ends our initial discussion of frame-based representation languages and of PFL in particular. Part II of this article, to appear in the December issue of *AI EXPERT,* will discuss the implementation of PFL both from the point of view of implementation techniques particular to FBRLs and more general issues of good Common LISP programming.

The reader wishing to further his or her understanding the role of knowledge representation languages in AI might start with an introductory AI text such as *Artificial Intelligence,*[15] *LISP,*[16] and *Introduction to Artificial Intelligence.*[17] Additional details can be found in "The Role of Frame-Based Representation in Reasoning,"[18] which describes frame-based representation systems from the perspective of their use in expert systems applications,[19] and in *Readings in Knowledge Representation,* which is a collection of 31 papers on knowledge representation. Also available are detailed descriptions of particular frame-based languages, such as FRL,[14] SRL,[20] KRL,[21] KEE,[22] and HP-RL.[23] AI

REFERENCES

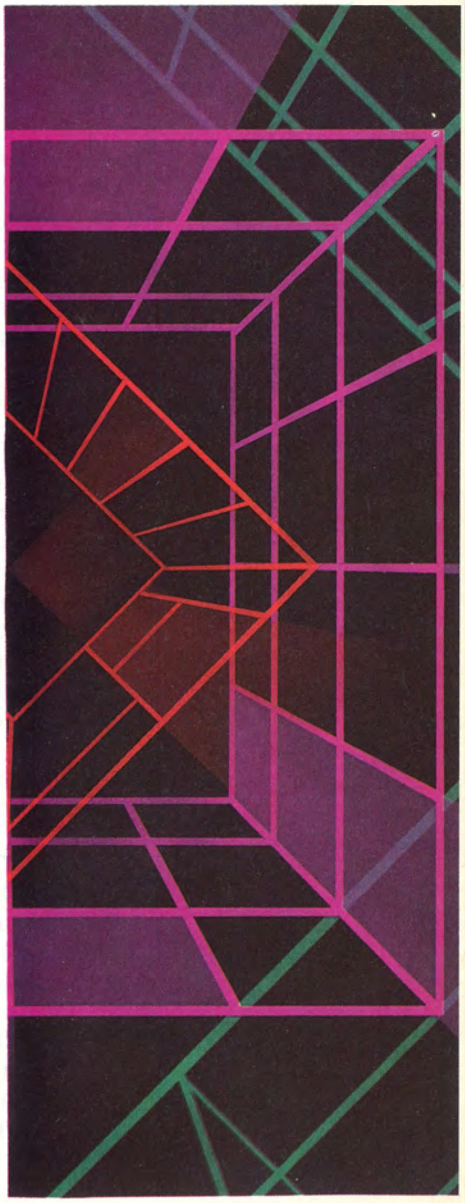
1. M. Ross Quillian. "Word Concepts: A Theory and Simulation of some Basic Semantic Capabilities." *Behavioral Science* 12 (1967): 410-430.
2. M. Ross Quillian. "Semantic Memory." In *Semantic Information Processing,* edited by Marvin Minsky. Cambridge: MIT Press, 1968.
3. S.C. Shapiro. "A Net Structure for Semantic Information Storage, Deduction, and Retrieval." In *Proceedings of the Second International Joint Conference on Artificial Intelligence.* Los Altos, Calif.: IJCAI (Aug. 1971): 512-523.
4. R. Brachman. "On the Epistemological Status of Semantic Networks." In *Associative Networks: Representation and Use of Knowledge by Computers,* edited by N.V. Findler. New York: Academic Press, 1979: 3-50. Reprinted in *Readings in Knowledge Representation,* edited by R.J. Brachman and H.J. Levesque. Los Altos, Calif.: Morgan Kaufmann, 1985.
5. W. Woods. "What's in a Link: Foundations for Semantic Networks." In *Representation and Understanding: Studies in Cognitive Science,* edited by D.G. Bobrow and A.M. Collins. New York: Academic Press, 1975: 35-82. Reprinted in *Readings in Knowledge Representation,* edited by R.J. Brachman and H.J. Levesque. Los Altos, Calif.: Morgan Kaufmann, 1985.
6. Pat Hayes. "The Logic of Frames." In *Readings in Knowledge Representation.* Edited by R.J. Brachman and H.J. Levesque. Los Altos, Calif: Morgan Kaufmann, 1985: 287-295.
7. Etherington, David, and Ray Reiter. "On Inheritance Hierarchies with Exceptions." In *Proceedings of the National Conference on Artificial Intelligence.* AAAI, 1983.
8. Brachman, R.J., and Schmolze, J.G. "An Overview of the KL-ONE Knowledge Representation System." *Cognitive Science* 9 (1985): 171-216.
9. Kaczmarek, T., R. Bates, and G. Robins. "Recent Developments in NIKL." In *Proceedings of the 1986 Annual National Conference in Artificial Intelligence.* AAAI, Aug. 1986.
10. Brachman, R.J., Pigman, V.G., and Levesque, H.J. *An Essential Hybrid Reasoning System: Knowledge and Symbol Level Accounts of KRYPTON.* Los Altos, Calif.: IJCAI, 1985: 532-539.
11. Bobrow, D.G., and Stefik, M. "The Loops Manual." Technical Report KB-VLSI-81-13. Xerox PARC, 1981.
12. Roberts, B.R., and I.P. Goldstein. *The FRL Manual.* MIT-AI Memo 409. Massachusetts Institute of Technology, 1977.
13. Finin, Tim. "Interactive Classification as a Knowledge Aquisition Tool." In *Expert Database Systems,* edited by Larry Kerschberg. Menlo Park, Calif.: Benjamin/Cummings Publishing Co., 1985.
14. Schmolze, J.G., and Lipkis, T.A. "Classification in the KL-ONE Knowledge Representation System." In *Proceedings of the International Joint Conference on Artificial Intelligence.* Karlsruhe, Fed. Rep. of Germany, 1983.
15. Winston, P. *Artificial Intelligence.* Addison-Wesley, 1983.
16. Winston, P., and B.K.P. Horn. *LISP.* Addison-Wesley, 1984.
17. McDermott, D.E., and Charniak. *Introduction to Artificial Intelligence.* Addison-Wesley, 1985.
18. Fikes, R., and T. Kehler. "The Role of Frame-Based Representation in Reasoning." *Communication of the ACM* 28, 9 (1985): 904-920.
19. Brachman, R.J., and H.J. Levesque, eds. *Readings in Knowledge Representation.* Los Altos, Calif.: Morgan Kaufmann, 1985.
20. Fox, M., J. Wright and D. Adam. "Experiences with SRL: An Analysis of a Frame-based Knowledge Representation." In *Expert Database Systems* edited by Larry Kerschberg. Menlo Park, Calif.: Benjamin/Cummings Publishing Co., 1985.
21. Kehler, T.P., and Clemenson, G.D. "An Application Development System for Expert Systems." *Systems & Software* (Jan. 1984).
22. Lanam, D., R. Letsinger, S. Rosenberg, P. Huyun, and M. Lemon. "Guide to the Heuristic Programming and Representation Language Part 1: Frames." AT-MEMO-83-3. Application and Technology Lab., Computer Research Center, Hewlett-Packard, Jan. 1984.
23. Bobrow, D., and T. Winograd. "An Overview of KRL, a Knowledge Representation Language." In *Readings in Knowledge Representation,* edited by R.J. Brachman and H.J. Levesque. Los Altos, Calif.: Morgan Kaufmann, 1985: 263-285. Originally appeared in *Cognitive Science* 1, 1 (1977): 3-46.

Tim Finin, Ph.D., is an assistant professor in the Dept. of Computer and Information Science at the University of Pennsylvania, Philadelphia, Pa. His current research interests include knowledge representation, expert systems, and computational linguistics.

Part II

UNDERSTANDING FRAME LANGUAGES

Implementing PFL



Knowledge representation is fundamental to AI. Over the past 25 years, many special-purpose languages have been developed for representing knowledge in AI systems. Frame-based representation languages (FBRLs) form a class that has achieved wide popularity.

In Part I of this article, we discussed the concepts that underlie frame-based representation languages and introduced a pedagogical frame language (PFL). In Part II we give a functional description of PFL, discuss implementation issues, and exhibit portions of the Common LISP code that implements PFL.

The purposes of PFL and this article are twofold: to describe in the most concrete terms possible (by using code) some of the concepts and mechanisms that underlie frame-based representation languages, and to demonstrate some of the features of Common LISP in the context of a complete, useful program.

PFL—THE BASICS

The primary PFL functions can be classified into those used to create, access, modify, or display frames. The complete list is:

(fdefine F . . .). Defines the frame *F*

(fget Fr S F D). Returns data facet *F* of slot *S* of frame *Fr*

(fvalue(s) F S). Returns data in the *:value* facet of slot *S* of frame *F*

(fslots F). Returns names of slots in frame *F*

(ffacets F S). Returns names of facets in slot *S* of frame *F*

(fput Fr S F D). Adds datum *D* to facet *F* from slot *S* of frame *Fr*

(fremove Fr S F D). Removes *D* from facet *F* of slot *S* of frame *Fr*

(ferase F). Removes all local information

ILLUSTRATION: ADRIAN DAY

from frame *F* then deletes it

(framep F). True if *F* is the name of a frame

(fsubsumes F1 F2). True if *F1* subsumes *F2*

(fshow F). Shows definition of frame *F*.

The easiest way to create a frame is just to refer to it. For example, saying *(fget 'john 'age ':value)* has the side effect of establishing *john* as a frame if it is not one already. Two functions are provided for explicitly creating a frame and giving it some information:

```
(fdefine frame-name parents &rest slots)

(fdefineq frame-name parents &rest slots)
```

The function *fdefineq* is just like *fdefine* except that it does not evaluate its arguments. A typical call to *fdefineq* looks like this:

```
(fdefineq cs-student student
   (major (:value computer-science)))
```

This expression defines *cs-student* to be a kind of student whose major is computer science. The first argument provides the frame's name, the second its immediate subsumers in the taxonomy, and the remaining arguments (if any) its slots. The slots specifying arguments are lists with the structure:

```
(slot-name facet1 facet2 . . . facetn)
```

where a facet looks like:

```
(facet-name datum1 datum2 . . . datumn)
```

A more elaborate example of a frame definition is:

```
 (fdefineq course university-thing
       (name (:type string)(:min 1)(:max 1))
       (number (:min 1) (:max 1))
       (department (:type dept) (:min 1))
       (prerequisites (:type course))
       (lecturer (:type instructor)(:min 1))
       (teaching-assistant (:type ta))
       (students
              (:type student)
              (:min 1)
              (:if-added check-prerequisites)))
```

Note that redefining a frame will cause the old definition to be overwritten.

ACCESSING PARTS OF FRAMES

Four basic functions are provided to extract information from a frame. The functions *fget* and *fvalues* extract data stored in the facets of the frame slots. *Fget* retrieves data from arbitrary facets of a frames slot; *fvalues* retrieves data from the *:value* facet.

The functions *fslots* and *ffacets* operate on a more schematic level, retrieving the slots of a frame and the facets of a slot, respectively. Whether or not inheritance is done (and defaults and demons are used) is controlled through the use of optional keyword parameters.

The main frame-accessing function is *fget*:

```
(fget frame slot facet &key inherit)
```

It takes three required arguments, which specify a frame, slot, and facet, and the optional keyword parameter *inherit*. It returns the data in the specified frame, slot, and facet. Some examples are:

```
(fget 'john 'advisor :if-needed)
(fget 'john 'advisor :type
   :inherit t)
(fget 'john 'advisor :type
   :inherit nil)
```

The optional keyword parameter *inherit* controls whether or not inheritance is used. If it is not specified, then its value defaults to that of the global variable **finherit**.

The function *fvalues* is used to get the data in a slot's *:value* facet:

```
(fvalues frame slot &key inherit default demons)
```

It is distinct from *fget* because the data in this facet can be represented explicitly or computed from the *:default* or *:if-needed* facets. This function takes two required parameters (which specify the frame and slot) and up to three optional keyword parameters (which control inheritance, the use of default values, and the use of *if-needed* demons). Some examples are:

```
(fvalues stud 'courses)
(fvalues stud 'courses :inherit nil)
(fvalues stud 'courses
   :inherit t
   :default nil
   :if-needed nil)
```

The optional keyword parameters are:

inherit. If nonnil, then data can be inherited from any of the frame's parents, provided there are no local values, default values, or *if-needed* demons.

default. If nonnil, then a default value will be sought, provided there is no local explicit value.

demons. If nonnil, then *if-needed* demons can be invoked to compute values, provided there are no local or default values.

If any of these optional keyword variables are not specified, then their values are provided by the global variables **finherit**, **fdefault**, and **fdemons**. These variables are initially all set to *T*.

The function *fslots* returns a list of the slot names in the frame *frame*:

```
(fslots frame &key inherit)
```

If the keyword parameter *inherit* is nil, then these names will include only the local slots. Otherwise, the list will include all local and inherited slots.

The function *ffacets* returns a list of all the facet names in frame *frame* and slot *slot*:

```
(ffacets frame slot &key inherit)
```

As with the function *fslots,* the keyword parameter *inherit* determines whether just the local or the local and inherited slots are returned.

These last two functions are useful in writing functions that operate on arbitrary frame structures. Writing a function to display the information in a frame, for example, requires iterating over all the slots in the frame and then iterating over all the facets in each slot:

```
(defun fshow (frame)
   "displays a frame"
   (format t "~%frame ~S" frame)
   (foreach slot in (fslots frame)
      (format t "~%      slot ~S:" slot)
      (foreach facet in (ffacets frame slot)
            (format t "~%        ~S = " facet)
            (foreach datum in
                  (fget frame slot facet)
                  (format t "~S "      datum))))
frame)
```

ADDING, REMOVING INFORMATION

This next function adds a new datum to a given frame, slot, and facet after checking any appropriate type and cardinality constraints:

```
(fput frame slot facet datum &key type demons
   inherit number)
```

If the datum is already a local value in the facet, then nothing is done. If the facet is *:value* and the keyword parameter *:type* is nonnil, then the datum is checked for proper type. The datum is then added to the facet and, if the facet is *:value* and the keyword parameter *:demons* is nonnil, any demons are run.

The *:inherit* keyword parameter controls whether or not inheritance is used in gathering the demons and type information. The *:number* keyword parameter controls the application of any cardinality constraints (such as those specified by the *:min* and *:max* facets.

Two primitive functions are provided for removing information from the frame system: *fremove,* which removes a given datum from the facet of a frame's slot, and *ferase,* which erases an entire frame. These two functions look like this:

```
(fremove frame slot facet datum &key demons
   inherit number)
(ferase frame &key demons inherit)
```

The *fremove* function removes the datum from the frame, slot, and facet after checking any appropriate cardinality constraint. If we are removing a value (for example, if the facet is *:value*) then any appropriate *:min* constraints are checked before the value is removed and any *if-removed* demons are run (provided the *:demons* parameter is nonnil). The keyword parameter *:inherit* controls whether or not these demons and *:min* constraints will be inherited.

The *ferase* function removes all local data in all local facets of all local slots of the frame *frame* via calls to *fremove.* This may, of course, trigger *if-removed* demons. After the data have been removed, the frame itself is deleted. The optional keyword parameters *demons* and *inherit* are simply passed on to *fremove.*

FUNCTIONS, GLOBAL VARIABLES

The following predicate returns *T* if its argument is the name of a frame:

```
(framep expr)
```

The following function returns *T* if *expr1* can be shown to subsume *expr2*:

```
(fsubsumes expr1 expr2)
```

One of the two arguments must be a frame. Expression *E1* subsumes *E2* if one of the following is true:

- Both are frames and are equal or there is a chain of *AKO* links from *E2* to *E1*
- *E1* is a frame and there is a predicate in its *subsumes-if* slot, which is true of *E2*
- *E2* is a frame and there is a predicate in its *subsumed-if* slot, which is true of *E1.*

The last two methods are provided to compare frames with other, nonframe objects. For example, we can define a frame "number" that subsumes all LISP numbers and a frame *ageValue* which subsumes numbers between 0 and 120, as follows:

```
(fdefineq number thing
   (subsumes-if (:value numberp)))
(fdefineq ageValue number
   (subsumes-if (:value validAgeP)))
(defun validAgeP (X)
   (and (numberp X)
      (< = 0 X 120)))
```

Two simple functions are provided for displaying the definition of a frame. The first of these is *fshow*:

```
(fshow frame &key inherit)
```

The second is *fshow-values*:

```
(fshow-values frame &key inherit demons default)
```

The first function displays the specified

The easiest way to create a frame is just to refer to it frame, including all of its slots, facets, and data. If the keyword parameter *inherit* is nil, only the local information will be displayed.

The second function displays only the values for the slots in the specified frame. The keyword parameters are similar to those for *fvalues*. Again, the use of inheritance, default values, and demons is controlled by optional keyword parameters.

PFL has a small number of global variables that control its operation. These variables include:

frames. This variable is bound to a list of the names of all of the frames in existence. Its initial value is nil.

finherit. This is the default value for the keyword parameter *inherit,* and initially is *T.* It determines whether or not inheritance is used in seeking data from facets in a slot.

fdemons. The default value for the keyword parameter *demons.* Initially, it is *T.* It determines whether or not *if-needed, if-added,* and *if-removed* demons are invoked when seeking, adding, or removing, respectively, values from a slot.

ftype. The default value for the keyword parameter *type.* Initially, it is *T.* It determines whether or not type checking is done when values are added to a slot.

fdefault. The default value for the keyword parameter *default.* Initially, it is *T.* It determines whether or not the *:default* facet is used when looking for a value for a slot but none is found.

fnumber. The default value for the keyword parameter *number.* Initially, it is *T.* It determines whether or not the *:min* and *:max* constraints are checked when adding and removing values from slots.

IMPLEMENTATION

This section describes some of the details of PFL's Common LISP implementation. Good programming practice dictates that any large system should be broken up into smaller modules.

I have followed this practice and decomposed this PFL implementation into the files: PFL.LISP, PFLVARIABLES.LISP, PFLMACROS.LISP, PFLBASE.LISP, PFLDISPLAY.LISP, and PFLTHING.LISP. These files can be found on the *AI EXPERT* Bulletin Board Service and CompuServe forum under the single file name PFL.LSP.

The file PFL.LISP defines the PFL system and can be used to load PFL. Note the use of the read-time conditionals (such as *#+ symbolics*) to customize PFL to run on different systems.

The file PFLVARIABLES.LISP defines and intializes all of the global variables used in PFL. We follow the popular LISP convention that the names of global variables begin and end with an asterisk.

The file PFLMACROS.LISP contains the definitions of macros and general utilities that are used throughout the system. It is important to have these organized into a separate file since the macro definitions are needed whenever any other module is compiled.

The main body of the system is contained in PFLBASE.LISP. This is the largest and most important file. Several functions for displaying frames are found in PFLDISPLAY.LISP. Finally, some standard frames that are to be included in every taxonomy are defined in the file PFLTHING.LISP

REPRESENTING A FRAME

In PFL a frame is represented as a list containing the frame's name and a sublist for each of its local slots. Each slot list has a name and any number of facets. Each facet has a name and any number of data. Here is the structure schematically:

```
(frame-name
   (slot1-name . . . )
   (slot2-name . . . )
   (slot3-name
```

LISTING 1.
The *fget* function.

```
(defun fget
       (frame slot facet
         &key (inherit *finherit*)
              (demons *fdemons*)
              (default *fdefault*))
   (if (equal facet :value)
       (fvalues frame slot facet
         :inherit inherit
         :demons demons
         :default default)
       (fget1 frame slot facet inherit)))

(defun fget1 (frame slot facet inherit?)
   "returns data in frame, slot and facet"
   (or (fget-local frame slot facet)
       (if inherit?
          (forsome parent in (fparents frame)
              (fget1 parent slot facet t)))))

(defun fvalues (f s
                &key (inherit *finherit*)
                     (demons *fdemons*)
                     (default *fdefault*)
                     (finitial f))
   "returns values from frame F slot S"
   (or (fget-local f s :value)
       (and default
            (fget-local f s :default))
       (and demons
            (forsome demon in
              (fget-local f s :if-needed)
              (listify (funcall demon finitial s))))
       (and inherit
            (forsome parent in (fparents f)
              (fvalues parent s
                :inherit t
                :demons demons
                :default default
                :finitial finitial)))))
```

```
    (facet1-name . . . )
    (facet2-name datum1
                 datum2
                 . . . )
  . . . )
. . . )
```

And here is an example of a list structure for a frame used to represent a person:

```
(person
  (ako (:value animal))
  (gender (:type gender-term)
          (:min 1)
          (:max 1))
  (spouse (:type person)
          (:if-added add-inverse)))
```

The current implementation stores a structure that represents a frame under the "frame" property of the frame's name. In addition, the global variable **frames** is bound to a list of the names of all current frames. Thus the function that creates a frame is relatively straightforward:

```
(defun fcreate (f)
  "creates a frame with name F"
  (setq *frames* (adjoin f *frames*))
  (setf (get f 'frame) (list f)))
```

Indexing the frames in this way has the advantages of being very easy to implement and allowing for extremely fast access. The chief disadvantage is that it only allows a single, global frame system to exist since Common LISP's property list system is global. This limitation makes it impossible, for example, to maintain two separate knowledge bases and to quickly switch between them.

A typical alternative to this scheme is to maintain a hash-table into which all current frames are placed, indexed by their names. Doing this enables you to have multiple frame systems by creating several hash tables.

ACCESSING DATA IN FRAMES

One of the most basic operations on a frame structure involves associating the data with a particular frame, slot, and facet. This operation can be accomplished very easily by the internal PFL function *fget-local*:

```
(defun fget-local (frame slot facet)
  ;; returns the data in a facet w/o
  ;; inheritance or demons.
  (cdr (assoc
       facet
       (cdr (assoc
            slot
            (cdr (frame frame)))))))
```

where the function *frame* returns the structure that represents the specified frame, creating the frame if necessary:

```
(defun frame (f)
  (or (get f 'frame)
     (fcreate f)))
```

Since the CDR of nil is defined to be nil in Common LISP, this process will work even if the frame does not have the slot or facet in question.

Of course, attempting to get data may in general require that it be inherited from the frame's ancestors. If the facet in question is the *:value* facet, then we may have to look for corresponding *:default* or *:if-needed* facets.

The *fget* function (Listing 1) takes three required arguments, specifying a frame, slot, and facet, and returns a list of the data found.

```
(defun fput-add (frame slot facet datum)
   ;; adds datum to given (frame,slot,facet)
   (rplacd (last (ffacet frame slot facet))
           (list datum)))

(defun ffacet (frame slot facet)
   ;; returns the expression representing
   ;; given (frame, slot,facet), creating
   ;; it if neccessary.
   (extend facet (extend slot (frame frame))))

(defun extend (key alist)
   ;; like assoc, but adds key KEY if
   ;; its not in the alist AlIST.
   (or (assoc key (cdr alist))
       (cadr (rplacd (last alist)
                     (list (list key))))))
```

LISTING 2.
The *ffacet* function.

LISTING 3.
Putting a value into a facet of a slot.

```
(defun fput (frame slot facet datum
             &key (demons *fdemons*)
                  (type *ftype*)
                  (inherit *finherit*)
                  (number *fnumber*))
  ;; adds a datum to a slot.
  (cond ((member datum (fget-local frame slot facet))
         datum)
        ((equal facet :value)
         (fput-value frame slot datum
            demons type inherit number))
        (t
         (fput-add frame slot facet datum)
         datum)))

(defun fput-value (frame slot datum demons?
                   type? inherit? number?)
   ;; adds value to slot if the types are ok
   ;; & slot isn't full, then runs demons
   (unless
     (and type? (not (fcheck-types frame slot datum)))
     (unless (and number?
                  (not (fcheck-max frame slot)))
       (fput-add frame slot :value datum)
       (if demons?
           (foreach demon in
             (fget frame slot :if-added
                   :inherit inherit?)
             (funcall demon frame slot datum)))
       datum)))
```

Whether or not inheritance is done is controlled by the optional keyword parameter *:inherit*. If the facet in question is the *:value* facet, then we can control whether or not demons can be invoked to compute the values and whether or not default values will be accepted through the use of the optional keyword parameters *:demons* and *default*. Note that *fget* simply passes the job onto *fget-value* or *fgetl*, depending on whether or not the facet sought is *:value*.

New data is added to a facet of a slot using *rplacd* to modify the list representing the facet and appending the new datum to the end of the list. The *ffacet* function (Listing 2) is used to get this facet structure, creating it if necessary.

In general, putting a value into a facet of a slot is somewhat more complex. If the value is already present, then nothing need be done. If the facet in question is the *:value* facet, we must check to see if the candidate value satisfies all the types associated with the slot (as specified in the *:type* facet). Then we must ensure that the slot is still open to receiving additional values (checking the *:min* facet). Finally, we must trigger any *if-added* demons associated with the slot. The functions in Listing 3 accomplish this process.

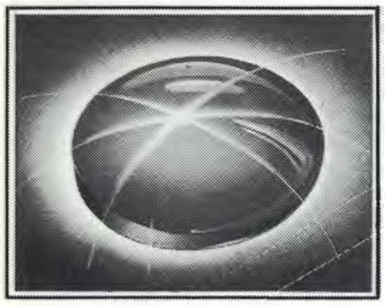
Removing data from a facet of a frame's slot is relatively easy (Listing 4). We first must ensure that the datum is indeed a locally stored one. Then, if the facet in question is the *:value* facet, we must verify that the datum's removal will not leave too few data in the facet. The actual removal can then be done with a simple call to delete. Finally, if we are removing a value, we must run any *if-removed* demons associated with the slot. AI

LISTING 4.
Removing data from a facet of a slot.

```
(defun fremove (frame slot facet datum
                &key (demons *fdemons*)
                     (inherit *finherit*)
                     (number *fnumber*))
  ;; removes datum from frame's slot's facet
  (when (and (member datum
                     (fget-local frame slot facet))
             (or (not (eq facet :value))
                 (fcheck-min frame slot)))
        (delete datum (ffacet frame slot facet))
        (if (and (eq facet :value) demons)
            (foreach demon in
              (fget frame slot :if-removed
                    :inherit inherit)
              (funcall demon frame slot datum)))))
```

If you'd like to acquire the extended, full-length version of PFL, the PFL.LSP file can be downloaded off any of the AI EXPERT *BBS nodes or from* AI EXPERT*'s account on CompuServe.*

Tim Finin, Ph.D., is an assistant professor in the Dept. of Computer and Information Science at the University of Pennsylvania, Philadelphia, Pa.



SAPIENS STAR SAPPHIRE
A Common LISP Compiler for the IBM PC™

- ★ 710 functions
- ★ C source code for Libraries
- ★ Lexical and dynamic scoping
- ★ Compiles to C
- ★ Unlimited multi-dimensional arrays
- ★ Object-oriented programming (flavors)
- ★ Numerical functions (bignums up to 128 digits)

Star Sapphire is a Common LISP compiler for the IBM PC. LISP is high-level general purpose programming language especially suited for writing artificial intelligence applications. All 710 functions of Common LISP are supported in Star Sapphire. The memory-intensive requirements of LISP are met with an 8 megabyte configurable workspace, Sapiens V8. Speed test results show a virtual memory overhead of less than 10%. It is primarily written in C.

Star Sapphire compiles programs to C. This increases speed and efficiency, and allows for porting code to other C environments. Compiling to C also permits the combination of LISP and C source code.

Star Sapphire includes a LISP to C translator, the runtime libraries, a runtime optimizer, and code samples, including graphics and an expert system engine. The IBM ROM BIOS is accessible through a library of keyboard, screen and graphics routines. The graphics library draws points, lines, and circles.

Star Sapphire is suitable for AI programming directly on the IBM PC, as well as for use as an AI delivery system. Developers can port programs written on larger machines to the desktop computer. Our C interface allows for delivery of commercial applications for the PC market.

SYSTEM REQUIREMENTS: The system requires a DOS-based C compiler which supports huge model. It needs 640K of RAM and a hard disk. Programs developed with *Star Sapphire* will run on a system with 256K of RAM. A hard disk is recommended for large applications. The virtual memory manager uses 16-128 kilobytes of RAM at the programmer's discretion.

PRICE: $495.00

Sapiens Software, 236 Mora St., Santa Cruz, CA 95060 • 408/458-1990