KR: an Efficient Knowledge Representation System

Dario Giuse

CMU-RI-TR-87-23

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

October 1987

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under contract F33615-84-K-1520. The views and conclusions are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

List of Figures

Abstract

KR is a very efficient semantic network knowledge representation language implemented in Common Lisp. It provides basic mechanisms for knowledge representation which include user-defined inheritance, relations, and the usual repertoire of knowledge manipulation functions. The system is simple and compact and does not include some of the more complex functionality often found in other knowledge representation systems. Because of its simplicity, however, KR is highly optimized and offers good performance. These qualities make it suitable for many applications which require a mixture of good performance and flexible knowledge representation.

1. Introduction

The Dante Project [Giuse 86] is a broad investigation into the issue of building man-machine interfaces. Having developed and refined user-interface techniques, we now intend to combine what we have learned into one integrated environment: the Uniform Workstation Interface. The Uniform Workstation Interface will provide an integrated interface system for a very heterogeneous environment of workstations and multiprocessors connected through a local area network.

Work within the Dante Project proceeds along several dimensions. One such dimension is the development of so-called *interactions styles*, i.e., the basic components that together form the Uniform Workstation Interface. Each interaction style corresponds to a traditional user interface paradigm such as form filling or menu selection. Another dimension is the integration of the different interaction styles into a coherent unit. Our strategy is to achieve this goal through a *common representation system* that underlies the different components of the Uniform Workstation Interface. This representation system serves two purposes: It allows the different components to communicate with one another, and it provides an explicit representation of the user and the computing environment.

The knowledge representation system we are developing is named KR and is described in this document. It is implemented in Common Lisp [Steele 84] and provides compact, very efficient knowledge representation. KR was specifically developed with interactive applications in mind, and thus good performance was the primary design concern.

In spite of its simplicity, KR has many potential applications outside the immediate Dante environment. It is fairly well integrated with LISP and provides a natural extension of the LISP philosophy. Because of its efficiency, KR is ideally suited for a number of applications which require flexible representation of knowledge but cannot afford the performance overhead often associated with full-fledged knowledge representation systems. Moreover, KR is entirely written in portable Common Lisp and thus can be used for a wide variety of situations.

The first section of the document presents the general principles behind the design of KR and its relationship with similar knowledge representation systems. The following section describes the main concepts of the system, including the notions of schema, slot, value, inheritance, and relation. The central portion of the document describes in detail the functional interface to the system and gives a complete specification of all the functions that comprise the interface. The following section describes how to load the system and presents several points about the best way to use KR to represent knowledge. Finally, the last section presents an evaluation of the actual performance of the system.

2. The KR Philosophy

This section briefly describes the most significant design choices in KR. Such choices have a profound effect on both the internal design of the system and on the appearance of user-level code that uses it. In this section we will assume that the reader is familiar with at least the general concepts of knowledge representation, and especially semantic network systems.

2.1 Structure of the System

KR is a knowledge representation system implemented in Common Lisp. It can be described as a semantic network system, since it stores knowledge as a network of chunks of information. Such systems are often referred to as "frame systems".

The main feature of semantic network systems is the flexibility they provide in representing knowledge. Unlike more traditional data-storage systems, such as for instance relational data bases, semantic networks are built out of completely unstructured chunks. Each chunk (known as a *frame* or *schema*) can store any arbitrary piece of information and is not in any way restricted to a particular format or data structure. The general way to represent information is as *attribute-value pairs*.

A program or user is free to use a schema in any given way and to store as much information as needed in it. Moreover, schemata[1] can be modified as needed, even after they have been created. Relational data bases, by comparison, force each chunk to be in one of a small group of possible formats, and the format of a chunk cannot be modified after creation.

The other important property that KR shares with most semantic network systems is that certain values in a schema can be interpreted as links to other schemata. This enables the system to support very complex network structures, which can be freely extended and modified by application programs. KR provides simple mechanisms that enable an application program to specify the structure of a network and how the system should handle the existing knowledge.

2.2 Simple versus Complex

KR is a very simple knowledge representation system. Simplicity results in two desirable properties: The system is easy to maintain and extend, and it performs fairly well. While the first property is intuitive, the second property deserves a little explanation.

It is certainly true that fine-tuning a simple system for performance is easier than fine-tuning a complex system. This is indeed what happened with KR, which we first implemented in a straightforward way and then fine-tuned very extensively to achieve good performance.

A common objection to this approach, however, maintains that where a simple system fails to implement a particular functional capability the application program must implement that capability itself. This might conceivably introduce an overall loss of efficiency. Advocates of this objection conclude that a knowledge representation system must implement all possible functions that will ever be required.

We believe that this argument is flawed. Our personal observation has been that it is quite difficult to provide simultaneously the right type of extended functionality and the right performance. The crucial problem is that system implementors often cannot anticipate exactly how the extended functionality will be used; as a result, they have to implement it in a completely general fashion. In most situations, unfortunately, complete generality means poor performance. Given any particular problem, system-defined general purpose solutions are typically inferior to solutions that use problem-specific information. In some sense, system-defined general purpose solutions are equivalent to brute force algorithms, since they have no information whatsoever about the particular problem.

The ironical consequence is that the "extended" functionality often gets bypassed completely for performance reasons, and users end up implementing it differently. What was supposed to alleviate the problem ends up making things worse: Application programmers go through the frustration of first basing their code on system-defined functionality, then finding out that it is too slow, and finally having to re-implement it in an ad-hoc fashion.

KR takes an entirely different approach. It recognizes that extended functionality cannot be implemented efficiently without detailed knowledge of how it will be used. KR, therefore, makes it easy for an

---

[1] The plural of *schema* is *schemata*

application program to implement particular solutions, but does not try to provide a "complete" set of solutions for all possible problems. Rather than providing a monolithic system, complete unto itself, KR simply extends the LISP language. Functions expressible in LISP are never duplicated, and the system only implements the lowest level of knowledge representation.

A consequence of this design choice is that the application developer must be more involved with the details of the implementation. This seems entirely logical, however, in view of the previous considerations: The application developer has much better knowledge of the particular problem, and can ultimately provide a more efficient solution.

2.3 Error Handling

Most Algol- and Pascal-like programming languages perform type checking at compile time, the idea being to catch errors as soon as possible. One could say that such systems assume that the programmer is in error, unless proven otherwise. This idea is reasonable for novice programmers but is overly restrictive for experienced programmers. Significant portions of most large Pascal programs, for instance, are purely devoted to type conversions among similar objects (such as arrays of the same basic type that simply happen to have a different number of elements).

LISP, on the other hand, takes an entirely different approach: type checking is performed at run-time. Rather than trying to prevent errors at any cost, LISP gives the programmer more freedom and simply informs him or her when an error does occur. To put it differently, LISP assumes that the programmer is right unless proven otherwise.

A similar dichotomy exists in knowledge representation systems. Some systems (such as SRL [Wright and Fox 83] and CRL [Carnegie Group 86], for instance) take the position that the programmer is wrong unless proven otherwise. In CRL, for example, one cannot assign a value to a slot in a schema[2] without first creating the slot. Failure to do so causes an error, unless the programmer explicitly overrides the default.

Unlike those systems, KR follows the LISP philosophy when handling errors: it assumes that what the programmer is doing is correct, and tries to do the reasonable thing if possible. One could view this as a simple form of "Do What I Mean" (DWIM) behavior. As an example, consider again the case where a programmer tries to assign a value to a non-existing slot in a schema. Rather than generating an error, KR first creates the slot and then gives it the new value. This is almost always the behavior the programmer intended.

A consequence of this approach is that traditional patterns of usage become simplified. To continue our example, the typical pattern for assigning a value to a slot in CRL is as follows:

- Create an empty slot;
- Assign a new value.

or (even worse in terms of performance and code legibility):

- Check to see if the slot exists;
- If not, create an empty slot;
- Assign a new value.

The complexity arises purely from the desire to prevent error messages, rather than from the problem itself. The corresponding code in KR, on the other hand, is simply a value assignment. The programmer can assume that if the slot is not there the system will do the right thing and create the slot before using it.

---

[2]See below for an explanation of terms like *value*, *slot*, and *schema*.

We believe that this approach is more intuitive and leads to a more natural programming style.

3. Main Concepts in KR

3.1 Schema, Slot, Value

Knowledge in KR is represented as a network of schemata. The *schema* data structure is the basic unit of representation and consists of a *name*, a set of *slots*, and a set of *values* for each slot.

The user can assemble a *network* of schemata by using a schema name as the value in the slot of another schema, which causes the two schemata to become linked. Networks that correspond to arbitrarily complex graphs can be constructed this way.

The name of a schema is always a symbol. In particular, we recommend that users employ only *keywords* as schema names. This choice makes any schema directly accessible from any Lisp package. It also has another advantage: it reduces the possibility of conflicts. In the current implementation of KR, slots and values are stored on the property-list of the schema name. Using keywords, which normally have empty property-lists, makes conflicts with existing symbols much less likely.

A schema may have any number of *slots*, which are simply attribute-value pairs. The slot name indicates the attribute name; the slot values (if any) indicate its values. Slot names are also symbols, and again we recommend that keywords be used. All slots in a schema must have distinct names, but different schemata may very well have slots with the same name.

Each slot can contain zero or more *values*. Values are the actual data items stored in the schema, and may be of any Lisp type. KR provides functions to add, delete, and retrieve values from a given slot in a schema.

The printed representation of a schema shows the schema name followed by slot/value pairs, each one on a separate line. The whole schema is surrounded by double curly braces. Consider a schema for John's pet, Fido:

```
{{fido
  :is-a :dog :pet
  :owner :John
  :color :brown
  :age 5
  }}
```

The schema is named :FIDO and contains four slots, named *is-a*, *owner*, *color*, and *age*. The slot *age* contains one value, the integer 5. The slot *color* also contains one value, the keyword :BROWN. The slot *is-a* contains two values, :DOG and :PET.

3.2 Inheritance

The main function of values is to provide information about the object represented by a schema. In the previous example, for instance, a query for Fido's age would return the value "5".

Values can also perform another function: They can establish *connections between schemata*. Consider the *owner* slot in the example above: if we interpret :JOHN as a schema name, than the slot tells us that

the :FIDO schema is somehow related to the :JOHN schema. Given the name of the slot, we might reasonably assume this to mean that John owns Fido.

KR makes it possible to use such connections to perform *inheritance*, i.e., to control the way information is inherited by a particular schema from some other schema to which it is connected. Inheritance allows information to be arranged in a hierarchical fashion, with lower-level schemata inheriting most of their general features from higher-level nodes and possibly providing local refinements or modifications. A connection that enables inheritance of values is called a *relation* (see section 3.3).

3.2.1 An Example of Inheritance

The most common example of inheritance is provided by the *is-a* relation. If schema A is connected to schema B by the *is-a* relation,[3] then values that are not present in A may be inherited from B.

Consider, for instance, the :FIDO schema in our previous example. If we were to ask "How many legs does Fido have?" we would not be able to find the correct answer by just looking at the :FIDO schema. Let us suppose, however, that we had also defined another schema:

```
{{dog
  :is-a: :mammal
  :owner:
  :legs: 4}}
```

Since we said that Fido *is-a* dog, the value can be inherited from the :DOG schema through the *is-a* slot. The answer would thus be "Fido has 4 legs." Inheritance is possible in this case because *is-a* is defined by the system to be a relation.

3.2.2 The Role of Inheritance

Inheritance achieves three purposes: It reduces network size, it helps maintain consistency, and it allows local knowledge to override global knowledge. That inheritance reduces network size is obvious, since whenever a piece of information for a schema is the same as in a more general one, we need not repeat it in the more specialized schema. In the example above, we do not say that Fido has four legs, nor that it has a tail or that it barks. :FIDO can inherit all of these properties from the parent concept :DOG.

Inheritance helps maintain consistency because it allows any piece of information to be stored only once. When a change is needed, the information is simply modified in one place. Multiple updates are unnecessary since the change will be immediately apparent in the rest of the network.

Finally, inheritance allows local redefinition of global knowledge. A particular schema can assert a different, local value for some piece of inherited knowledge by simply providing a local slot with the same name and a new value. Its children would then inherit the new value, since the inheritance process stops as soon as a value is found.

3.3 Relations

Slots like *is-a* that enable knowledge to be inherited from other parts of a network are called *relations*. Inheritance along a relation is typically defined to proceed depth-first and may include any number of steps (in other words, the search terminates if a value is found or if no other schema can be reached via the relation).

---

[3] In other words, if the name of the schema B appears as a value in the *is-a* slot of schema A.

KR allows the user to define new relations as desired. This is achieved through the function **create-relation** (see section 4.4), which performs all the necessary bookkeeping operations.

Any relation, including user-defined ones, may be declared to have an inverse link. If this is the case, KR will automatically generate an inverse link any time the relation is used to link one schema to another. Imagine, for instance, that we defined *pet-of* to be a relation having *has-pet* as its inverse. Writing :JOHN in the *pet-of* slot of :FIDO would automatically add :FIDO to the *has-pet* slot of :JOHN, thereby creating a reverse link.

3.4 Link Maintenance

KR automatically maintains all the links and inverse links described above, and the application programmer does not have to worry about them. This is probably one of the most convenient features of the system.

Imagine, for instance, that the two schemata A and B are linked by a certain relation and inverse relation. This means that schema A will have the name of schema B as the value in one of its slots. If the program decides to delete schema B, then, it is essential that the link from A to B also disappear. Failure to do so would cause the reference in A to be dangling: it would be an error to try to follow the reference, since the schema being pointed to (i.e., B) would no longer exist.

KR carefully keeps track of similar situations whenever they occur and corrects them instantly. The KR function that deletes schema B will automatically follow all the reverse pointers and make sure that any reference to B disappears as well.

In a similar manner, whenever the name of a schema is assigned as a value to a slot which happens to be a relation, KR automatically creates an inverse link. This ensures that the state of the knowledge representation system is completely consistent at any point in time, independent of the particular sequence of operations.

4. Program Interface

The KR program interface allows a program or a user to create and modify schemata, slots, and values. The interface is available as a set of functions defined and exported by the "KR" package. See section 5.1 for instructions on how to load KR onto your system.

4.1 Notation

In order to simplify the notation we will use the following conventions:

- The notation *<object>* indicates any Lisp object, which may or may not be a schema.
- The notation *<schema>* indicates that a function expects a valid schema as an argument. An error will typically be signalled if this is not the case.
- The notation *<slot>* indicates a valid slot name for a schema, i.e., a symbol (and more specifically a keyword).
- The notation *<schema-name>* indicates that a valid name for a schema (i.e., a keyword) must be supplied. It is not necessary for a schema by that name to already exist.

The notation "==>" will indicate the result of evaluating a LISP form. The notation "<==>" will indicate that

two forms are equivalent, i.e., one produces exactly the same effect as the other. Finally, the LISP comment line "; prints:" will be used to indicate the printed output produced by evaluating a LISP expression.

4.2 Example Schemata

The following sections use certain schemata as examples. We present here the definitions of those schemata once and for all:

```
;;; Define the :OWNER relation and its inverse, :HAS-PET.
(create-relation :owner T '(:has-pet))     ; T means inheritance is on.

(create-schema :dog
  (:is-a :mammal)
  (:legs 4))

(create-schema :fido
  (:is-a :dog)
  (:owner :john)
  (:age 4.5))

(create-schema :john
  (:is-a :person :lawyer)
  (:address "13 Elm Street"))
```

Figure 4-1 shows all the user-defined schemata after those definitions have been executed. Relations are indicated as an arrow going from a schema to the one it is related to.

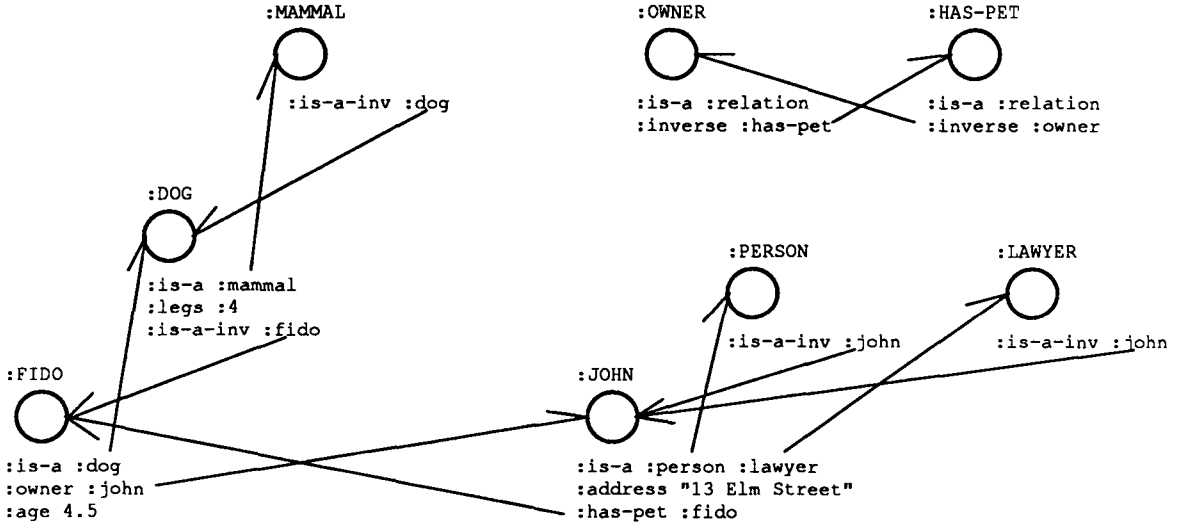

Figure 4-1: The resulting network of schemata

4.3 Predicates and Query Functions

The functions in this group give information about a schema or a slot. All functions whose name ends in "-P" are predicates, i.e., they return a value which is simply used as a Boolean.

(SCHEMA-P *object*) [*Function*]
A predicate that returns NIL if <object> is not a valid schema, non-nil otherwise.
Examples:

```
(schema-p :fido) ==> T      ; or a system-dependent non-nil value
(schema-p :waffle) ==> NIL
```

(RELATION-P *object*) [*Macro*]
A predicate that returns NIL if <object> is not a relation, or a non-nil value if it is the name of a schema and the schema is declared to be a relation.
Examples:

```
(relation-p :has-pet) ==> T
(relation-p :color) ==> NIL
```

(IS-A-P *schema1 schema2*) [*Function*]
A predicate that returns T if <schema1> is related to <schema2> by the *is-a* relation, either directly or through an inheritance chain.
Examples:

```
(is-a-p :fido :dog) ==> T
(is-a-p :fido :mammal) ==> T
(is-a-p :fido :canine) ==> NIL
```

(HAS-SLOT-P *schema slot*) [*Function*]
A predicate that returns T if the <schema> contains a slot named <slot>, NIL otherwise. Note that <slot> must be local to <schema>, and inherited slots are not considered.
Examples:

```
(has-slot-p :fido :is-a)  ==> T
(has-slot-p :fido :legs) ==> NIL   ; Slot is not local
```

(GET-SLOTS *schema*) [*Function*]
Returns a list of all the slot names in <schema>. The list only includes local slots and does not report slots that might be inherited.
Example:

```
(get-slots :fido) ==> (:AGE :OWNER :IS-A)
```

(GET-ALL-SLOTS *schema*) [*Macro*]
Returns a list of all the slot names in <schema>, including slots that are not local but may be inherited through an inheritance chain.

Example:

```
(get-all-slots :fido) ==>
  (:LEGS :HAS-PET :IS-A-INV :IS-A :ADDRESS :OWNER :AGE)
```

Note that the example above returns a somewhat surprising list of slot names for :fido (including :has-pet). This is because we declared :owner to be an inheritance relation.

(PS *object*) [*Function*]
Prints out the current schema corresponding to <object>, if one exists, or nothing if <object> is not a valid schema.
Example:

```
(ps :fido)    ; prints out:

{{FIDO
  AGE: 4.5
  OWNER: :JOHN
  IS-A: :DOG
  }}
```

4.4 Schema Manipulation Functions

This group includes functions that create, modify, and delete whole schemata. The function **create-relation** does so implicitly, since relations are also represented by KR as schemata.

(CREATE-SCHEMA *schema-name* &rest *slot-definitions*) [*Macro*]
This macro creates and returns a new schema named <schema-name>. <slot-definitions>, if present, are used to create new slots and values for the schema. Each slot definition should be a list whose CAR is the name of a slot and whose CDR is a (possibly empty) list of values for that slot.
Note: if <schema-name> already exists, the schema is modified in place and will contain the union of its previous slots and the slots specified by create-schema. Previous slots which are mentioned in the call will retain whatever values they had before the operation.
Example:

```
(create-schema :timmy (:is-a :cat) (:age 1.5) (:color :brown :white))
```

(CREATE-FRESH-SCHEMA *schema-name* &rest *slot-definitions*) [*Macro*]
This function is similar to CREATE-SCHEMA, except that it always deletes the schema <schema-name> (if it exists) before creating a new schema. The schema is guaranteed to include only the slots and values specified in the call.

(COPY-SCHEMA *schema*) [*Function*]
Creates and returns a new schema which is an identical copy of <schema>. The newly created schema is automatically given a unique name. All the slots in the new schema contain a copy of the values in the corresponding slot of <schema>. Corresponding lists of values, in other words, will be **equal** in the LISP sense, but not **eq**.
Examples:

```
(copy-schema :fido) ==> T1806
(ps 't1806)      ; prints:

{{T1806
  AGE: 4.5
  OWNER: :JOHN
  IS-A: :DOG
  }}
```

(DELETE-SCHEMA *schema-name*) [*Function*]
Destroys the schema named by <schema-name>. Returns T if the schema was destroyed, NIL if it did not exist.

(CREATE-RELATION *schema-name inherits-p inverses*) [*Function*]
Creates a new schema named <schema-name> and declares it to be a relation. The new relation will have <inverses> (a list of relations) as its inverse relations. If <inherits-p> is non-nil, <schema-name> will become a relation with inheritance, and values may be inherited through it. As a side effect, a schema called <schema-name> is created and linked to the :relation schema through an *is-a* link; all of the <inverses> schemata are also linked to <schema-name>.
Example:

```
(create-relation :has-subsystems nil '(:part-of :subsystem-of))
```

The previous function call defines the non-inheritance relation :HAS-SUBSYSTEMS and its two inverses, :PART-OF and :SUBSYSTEM-OF.

4.5 Slot Manipulation Functions

This group includes functions which create, modify, and delete slots in a schema. It also includes a convenient way to iterate a user-defined function over all the slots in a schema.

(CREATE-SLOT *schema slot-name*) [*Function*]
Creates slot <slot-name> in <schema>. The slot will initially be empty.
Examples:

```
(create-slot :fido :color) ==> NIL
(ps :fido)       ; prints:

{{FIDO
  COLOR:
  AGE: 4.5
  OWNER: :JOHN
  IS-A: :DOG
  }}
```

(DELETE-SLOT *schema slot-name*) [*Function*]
Destroys the slot named <slot-name> from <schema>. Values previously stored in the slot, if any, are lost.

(DO-SLOTS *schema function*) [*Function*]
Iterates <function> over all the slots of the <schema>. The <function>, which should be a LISP function of two arguments, is applied in turn to each of the local slots of the <schema>; the first argument is the schema itself, and the second argument is the name of the slot. The <function> is called purely for side effects, and DO-SLOTS simply returns NIL.
Note that the same result can be achieved with an explicit iteration over the list returned by GET-SLOTS, but in general DO-SLOTS avoids the allocation of storage implicit in the latter.
Example:

```
(do-slots :fido #'(lambda (schema slot)
                    (format t "Slot ~S has values ~S~%"
                            slot (get-values schema slot))))
Slot :COLOR has values NIL
Slot :AGE has values (4.5)
Slot :OWNER has values (:JOHN)
Slot :IS-A has values (:DOG)
```

4.6 Value Manipulation Functions

This group includes the most commonly used KR functions, i.e., the one which retrieve or modify the value(s) in a slot.

(GET-VALUE *schema slot-name*) [*Macro*]
Returns the first value in the slot <slot-name> from the <schema>. If the slot is empty or not present, it returns NIL. Inheritance may be used when looking for a value.
Examples:

```
(get-value :fido :is-a) ==> :DOG
(get-value :fido :legs) ==> 4        ; inherit the value from :DOG
(get-value :john :is-a) ==> :PERSON  ; first value only
```

A **setf** form is defined for GET-VALUE, so that one can write, for instance,

```
(setf (get-value :fido :owner) :Bill)
```

(GET-VALUES *schema slot-name*) [*Macro*]
Returns all the values in <slot-name> from the <schema>, as a list. If the slot is empty or not present, it returns NIL. Inheritance may be used when looking for values.
Examples:

```
(get-values :fido :is-a) ==> (:DOG)
(get-values :john :is-a) ==> (:PERSON :LAWYER)  ; all values
```

A **setf** form is also defined for GET-VALUES. For instance,

```
(setf (get-values :fido :owner) '(:Bill :Jill))
```

(GET-LOCAL-VALUES *schema slot-name*) [*Macro*]
Similar to GET-VALUES, but only local slots are examined and inheritance is never used.
Examples:

```
(get-local-values :fido :is-a) ==> (:DOG)
(get-local-values :fido :legs) ==> NIL        ; no inheritance
```

(DOVALUES (*variable schema slot-name*) &rest *body*) [*Macro*]
This macro lets you iterate over all the values in slot <slot-name> for the <schema>. The <body> is repeatedly executed with <variable> bound to each value in turn. It is an error for <body> to modify the structure of the slot.
Example:

```
(dovalues (owner-name :fido :owner)
  (format t "Fido is a pet of ~A, who lives at ~A.~%"
          owner-name (get-value owner-name :ADDRESS)))      ; prints:

Fido is a pet of JOHN, who lives at 13 Elm Street.
```

(DO-ALL-VALUES (*variable schema slot-name*) &rest *body*) [*Macro*]
This is similar to DOVALUES, except that in this case when inheritance is used to find the slot. In this case, all the parents of the <schema> are explored, whereas DOVALUES would stop whenever a parent with the slot is reached. The difference is only important when <schema> has multiple parents.

(SET-VALUE *schema slot-name object*) [*Function*]
Causes slot <slot-name> in <schema> to contain <object> as its single value. Note that

```
(set-value s slot value) <==> (setf (get-value s slot) value)
```

because of the **setf** form described above.
Example:

```
(set-value :fido :color :brown) ==> (:brown)
(ps :fido)    ; prints:

{{FIDO
  COLOR: :BROWN
  AGE: 4.5
  OWNER: :JOHN
  IS-A: :DOG
  }}
```

(SET-VALUES *schema slot-name object-list*) [*Function*]
Causes slot <slot-name> in <schema> to contain the values specified by <object-list>.
Note that

```
(set-values s slot values) <==> (setf (get-values s slot) values)
```

Examples:

```
(set-values :fido :owner '(:peter :paul :mary))
```

(APPEND-VALUE *schema slot-name object*) [*Function*]
Adds one more value, <object>, to the end of the list of values in <slot-name>. The new value will appear

last in the values returned by GET-VALUES.
Examples:

```
(append-value :fido :color :white)      ; and now
{{FIDO
  COLOR: :BROWN :WHITE
  AGE: 4.5
  OWNER: :JOHN
  IS-A: :DOG
  }}
```

(DELETE-VALUE-N *schema slot-name position*) [*Function*]
Deletes the n-th value from a slot. <position>, a 0-based integer, indicates which value should be deleted from slot <slot-name> in the <schema>. <position> must be between 0 and the position of the last value in the slot.
Example:

```
(delete-value-n :fido :color 1)         ; and now
{{FIDO
  COLOR: :BROWN
  AGE: 4.5
  OWNER: :JOHN
  IS-A: :DOG
  }}
```

5. Using the System

5.1 How to Load KR

Until the Dante software is made a part of the official distribution system, you will have to load KR by hand. The easiest method to accomplish this is to execute the following expression from within LISP:

```
(unless (get :dante-modules :kr)
  (setf (lisp::search-list "kr:")
        '("/../herbie/usr/dante/kr/release/code"))
  (load "kr:dante-loader"))
```

Like all Dante subsystems, KR follows a special convention that lets an application program determine whether the subsystem is already loaded. After loading KR, the special keyword **:dante-modules** will have its **:kr** property set to a non-nil value. The typical way to check for this is shown in the expression above.

The system defines a package of its own, namely the "KR" package. All the function names described in the Program Interface section of this document are exported from the KR package, and all you need to do is to add the following line to your program:

```
(use-package "KR")
```

5.2 Internal Representation

This section briefly describes the internal representation for schemata, slots, and values. Only information that may be useful at the application-program level is presented here. Such information can be considered a part of the external contract of the KR system, and application programs can safely rely on the details presented here.

Schemata are simply represented as symbols. No special information is attached to a symbol to indicate that it is a KR schema.

Slots are represented as part of the P-list of a symbol. In particular, each slot corresponds directly to an entry in the P-list. Application programs should never depend on this particular implementation, and should not modify the P-list of a symbol to modify slot information.

Values are represented as a list which is the value of an entry in the P-list. A slot with one value is represented as a list of one element. Values are always internally stored in the same order as shown by the PS function. A list of values is always a simple list, and it contains no additional information whatsoever. Consequently, all the ordinary LISP list-manipulation functions can be used on lists of values. Moreover, the list returned by GET-VALUES is always guaranteed to be EQ to the list of values internally stored in the schema.

5.3 Style Notes

5.3.1 List Representation

The fact that **setf** forms are defined for the two access functions **get-value** and **get-values** makes it possible to obtain quite a few interesting combinations while keeping the functional interface to KR very simple. This is a typical example of how following the LISP philosophy can greatly simplify the external interface of a knowledge representation system.

The operation of adding a new value to the front of a slot, for instance, does not require a special KR function. One simply writes:

(push *value* **(get-values** *schema slot***))**

Similarly, in order to add a value to a slot only if it is not already there, one simply writes one of the following (depending on whether a special test function is required):

(pushnew *value* **(get-values** *schema slot***))**
or
(pushnew *value* **(get-values** *schema slot***) :test #'** *some-test-function***)**

As another example, no special function is needed to eliminate the first value from a slot. One simply writes:

(pop (get-values *schema slot***))**

Other commonly-used KR idioms also arise from the fact that values are stored as lists. To find out how many values are in a slot, for instance, one uses the function LENGTH:

(length (get-values *schema slot***))**

To search for a given value in a slot, one can use the functions FIND, POSITION, MEMBER, or any of the variations provided by Common Lisp.

5.3.2 Adding and Deleting Values

One should not use destructive LISP functions to add or delete values from a slot, even though those functions might "work" in some cases. We recommend that the KR access functions (such as SET-VALUE and SET-VALUES, or the SETF methods for GET-VALUE and GET-VALUES) be used in all cases to achieve the same effect.

The reason to avoid direct destructive operations is that such operations may leave the system in an inconsistent state when the slot being operated upon is a relation. Remember that slots that happen to be relations must be handled specially because of the reverse links. Strictly speaking, this only applies to symbols, but we prefer to simply state the following rule of thumb: *Do not use destructive operations to alter the contents of a slot.*

5.4 Usage Hints

5.4.1 Inheritance Relations

KR allows you to freely define new relations that perform inheritance. As a general rule, however, we recommend that you consider carefully whether such relations are really required for your application. Two problems can arise from excessive usage of inheritance relations:

- Poor performance.
- Confusion.

Inheritance relations may affect the system's performance since they turn what is normally a simple hierarchical network into a tangled graph. Every time a slot is accessed and a value is not present locally, KR may have to proceed up the hierarchy following several relations, instead of just the *is-a* relation. There are clearly cases when this is justified by the additional functionality, however, and one should evaluate advantages and disadvantages of the choice on a case-by-case basis.

The second factor, i.e., confusion, is somewhat less intuitive. We will refer back to the example in section 4.3 to illustrate this point. That example is repeated here for convenience:

```
(get-all-slots :fido) ==>
  (:LEGS :HAS-PET :IS-A-INV :IS-A :ADDRESS :OWNER :AGE)
```

Remember that we had linked :FIDO to :JOHN via the *owner* slot, and we had declared *owner* to be an inheritance relation. Getting the list of all slots, then, returned surprising things like *has-pet*, even though :FIDO is a dog and thus is not supposed to have any pets. What happened is that the *owner* relation opened up all the slots in :JOHN for inheritance, and thus :FIDO was suddenly endowed with all the properties that would normally only belong to a person. If :JOHN had had a *salary* slot and a *languages-spoken* slot, those would also have been inherited by :FIDO!

Again, there are cases when user-defined inheritance relations are quite useful. An example occurs when a network is used to represent a situation with multiple hierarchies. In such cases it is natural to define inheritance relations to support the multiple hierarchies, rather than writing special-purpose code to do the same thing.

5.4.2 The IS-A Hierarchy

The IS-A hierarchy constitutes the most natural way to structure a network hierarchically. There are cases, however, where we feel that using the IS-A hierarchy is not appropriate because of stylistic or performance reasons. The most typical example is when the IS-A hierarchy is used to express minor or insignificant differences among certain schemata. In such situations it might be more appropriate to use a

separate slot to express the difference.

As an illustration of this point we will use an example from the Chinese Tutor [Giuse 87], an intelligent language tutor for beginner-level Chinese which uses KR to represent all of its internal data structures. A particular entity of the Chinese language is a *glyph*, i.e., the printed or written representation of a character. Glyphs are represented in the program as KR schemata.

As it turns out, different types of glyphs exist in Chinese (in particular, the complex form and the simplified form of a Chinese character correspond to different glyphs). The very first version of the Chinese Tutor used the *is-a* hierarchy to differentiate among the different types of glyphs, so that the original structure of the network looked like the one shown in figure 5-1.

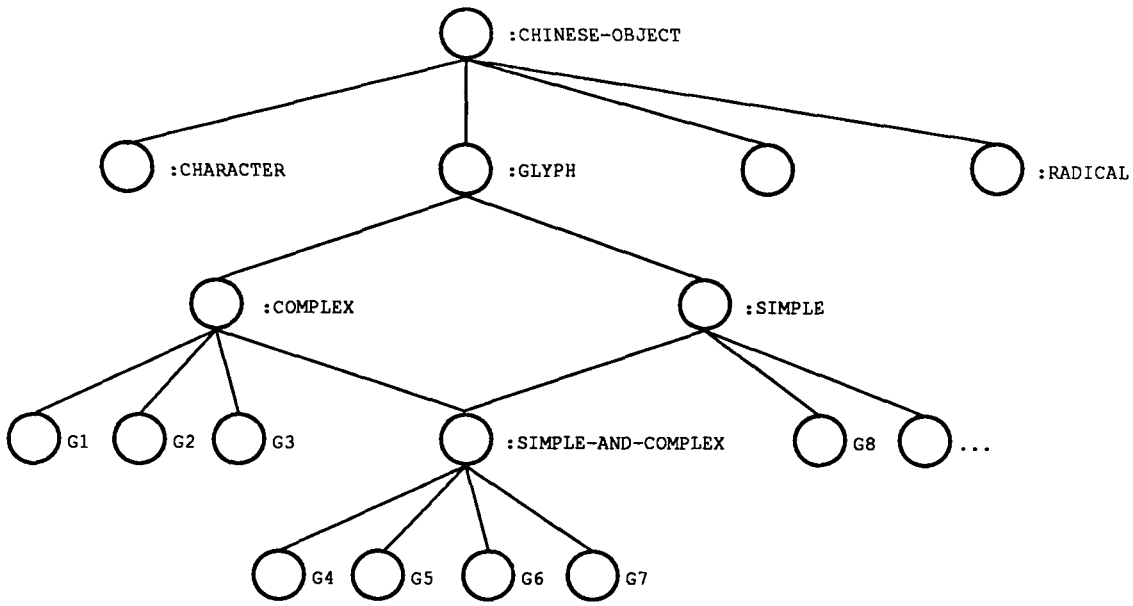

Figure 5-1: The original IS-A hierarchy for glyphs

Note, in particular, how some glyphs appeared two levels below the :GLYPH schema, whereas others appeared three levels below because of the peculiar position of the :SIMPLE-AND-COMPLEX schema. This solution was not ideal. The *is-a* hierarchy was used to represent essentially minor semantical differences, rather than a true hierarchical structure. As a consequence, a common set of operations became unnecessarily complicated and expensive. These operations all followed the same pattern, i.e., they needed to access all the glyphs in the system *independent* of those minor semantical differences. Keeping track of all the glyphs was difficult because they could appear in several subtrees and possibly at different levels in the hierarchy.

The second version of the system eliminated the problem by making all the glyphs immediate children of the :GLYPH schema. This is illustrated in figure 5-2.

Keeping track of all the glyphs, then, simply became a matter of looking into the *is-a-inv* slot of the :GLYPH schema. The minor differences among glyphs are now encoded in a different slot, which does not serve any hierarchical function. The slight increase in space for the extra slot is more than justified in view of better performance and much cleaner code.

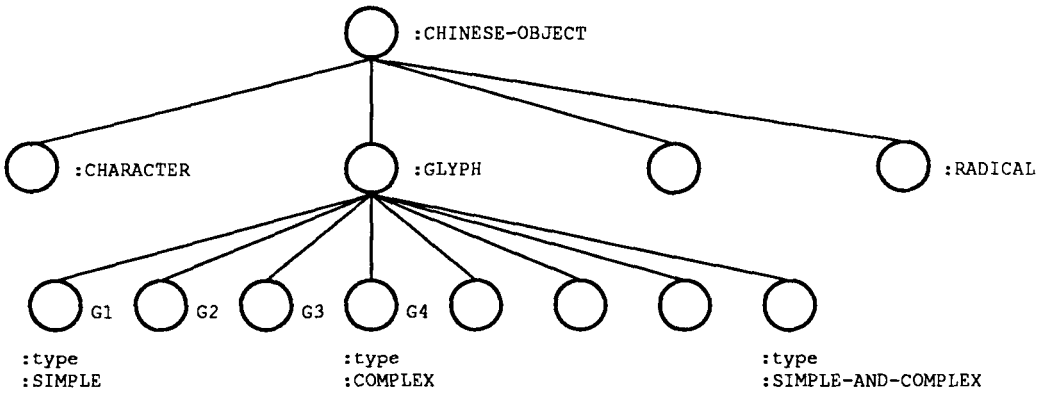

Figure 5-2: The new IS-A hierarchy for glyphs

6. Performance of the KR System

This section presents a simple evaluation the performance of the KR system. These figures were collected on an IBM RT running CMU Common Lisp under the Mach operating system. The RT used to obtain these figures had 10 Mbytes of central memory.

All figures refer to compiled code and are expressed in microseconds per function call. Statistics were collected from within Lisp by executing each function call within a tight loop for a given number of times and timing the loop. A few microseconds should be subtracted from each number to eliminate the loop overhead; 3 or 4 microseconds is probably a reasonable factor.

6.1 Value Access and Modification

GET-VALUE:	40.5
GET-VALUE with 1 level of inheritance:	142.3
GET-VALUE with 2 levels of inheritance:	208.2
GET-VALUE with 3 levels of inheritance:	328.7
GET-VALUES:	40.0
GET-VALUES with 1 level of inheritance:	137.1
GET-VALUES with 2 levels of inheritance:	239.0
GET-VALUES with 3 levels of inheritance:	327.1
GET-LOCAL-VALUES:	10.6
SET-VALUE:	94.4
SET-VALUES:	127.8

6.2 Predicates

RELATION-P:	11.1
HAS-SLOT-P:	100.8
SCHEMA-P:	29.1
IS-A-P:	75.7

6.3 Discussion

The figures above indicate that KR performs quite well. To put those figures in perspective, consider that an empty function call and return in the same environment takes about 14 microseconds. The time to execute the simplest and most commonly used access functions, GET-VALUE and GET-VALUES, is of the order of 3 function calls.

It is also worth mentioning that none of the functions in the tables above allocate any memory at all. This eliminates a common cause of inefficiency, namely, the allocation of temporary storage ("garbage") which has to be eliminated later on.

It is somewhat difficult to provide fair comparisons with other existing knowledge representation systems. Such comparisons are always prone to criticism unless all conditions are absolutely identical, which is difficult to obtain. Just as one point in the spectrum, however, we will mention that the corresponding execution times for CRL running on a Symbolics 3640 Lisp Machine are significantly longer than the ones reported above. Considering that Common Lisp benchmarks typically perform 1.2 to 2.2 times better on a Symbolics than on RT/PC, we might conclude that the above functions in KR are anywhere between 7 and 13 times faster than in CRL. Significantly, it appears that the most expensive functions (such as functions involving inheritance) are even more efficient, relatively speaking, than the simpler ones.

As a final point of comparison we will mention that the time to access a slot in a Common Lisp structure in the same environment is 10.9 microseconds. Compared to this, the corresponding function in KR (i.e., GET-VALUE) is 3.7 times slower. Given that access to Common Lisp structures is very highly optimized in Lisp, it seems that the performance penalty for using KR is amply justified by the much greater flexibility offered by the system.

7. Summary

KR is a simple, very efficient knowledge representation system for Common Lisp. It implements the basic paradigm of semantic network systems and offers such features as inheritance, user-defined relations, and user-defined inheritance.

The main emphasis of the system is on efficiency. Unlike many semantic network knowledge representation systems, KR does not try to provide a monolithic system, but rather aims at extending the Common Lisp philosophy in a natural way. The system is highly optimized and provides a solid substrate upon which application programs can build more elaborate knowledge manipulation algorithms. The program interface to the system consists of a small number of carefully tuned functions; these functions are easy to understand and to use.

Because it adopts the fundamental LISP philosophy, KR fits in very naturally with Common Lisp based application programs. Because of its simplicity, the system is quite small and entirely portable. We feel

that these characteristic make it a useful knowledge representation language, and one whose range of applicability extends well beyond the original environment it was developed for.

References

[Carnegie Group 86]
Knowledge Craft Reference Manual
Carnegie Group, Inc., Pittsburgh, PA, 1986.

[Giuse 86] Dario Giuse.
Research in Uniform Workstation Interfaces - Research Proposal to DARPA
1986.

[Giuse 87] Dario Giuse.
LISP as a Rapid Prototyping Environment: the Chinese Tutor.
submitted to the International Journal on Lisp and Symbolic Computation , 1987.

[Steele 84] Guy L. Steele.
Common LISP - The Language.
Digital Press, Burlington, MA, 1984.

[Wright and Fox 83]
M. Wright, M. Fox.
SRL: Schema Representation Language.
Technical Report, Carnegie-Mellon University, December, 1983.