January 1990

KB: A Knowledge Representation Package for Common Lisp

Jeffrey Esakov
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-90-03.

KB: A Knowledge Representation Package for Common Lisp

Abstract

KB is a frame-based knowledge representation package. It is written as a Common Lisp package, and is comprised of a set of functions for representing semantic knowledge and relationships among data represented.

KB encourages the use of the object-oriented programming metaphor by requiring that a set of operators be defined for each concept (object). Inheritance is supported for both data types and for operators.

KB has a well-defined programming interface through which a user interface language can be easily developed. The semantics of *KB* are straightforward and allow a programmer considerable flexibility in developing an application.

KB borrows heavily from the Flavors system in syntax and semantics (and in documentation!).

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-90-03.

KB: A KNOWLEDGE REPRESENTATION PACKAGE FOR COMMON LISP

Jeffrey Esakov

MS-CIS-90-03
GRAPHICS LAB 30

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

January 1990

KB
A Knowledge Representation Package for Common Lisp

Jeffrey Esakov

Abstract

KB is a frame-based knowledge representation package. It is written as a Common Lisp package, and is comprised of a set of functions for representing semantic knowledge and relationships among data represented. **KB** encourages the use of the object-oriented programming metaphor by requiring that a set of operators be defined for each concept (object). Inheritance is supported for both data types and for operators.

KB has a well-defined programming interface through which a user interface language can be easily developed. The semantics of **KB** are straightforward and allow a programmer considerable flexibility in developing an application.

KB borrows heavily from the Flavors system in syntax and semantics (and in documentation!).

KB
A Knowledge Representation Package for Common Lisp

Jeffrey Esakov

October 25, 1989

Introduction to KB

KB is a frame-based knowledge representation package. It is written as a Common Lisp package, and is comprised of a set of functions for representing semantic knowledge and relationships among data represented. **KB** encourages the use of the object-oriented programming metaphor by requiring that a set of operators be defined for each concept (object). Inheritance is supported for both data types and for operators.

KB has a well-defined programming interface through which a user interface language can be easily developed. The semantics of **KB** are straightforward and allow a programmer considerable flexibility in developing an application.

KB borrows heavily from the Flavors system in syntax and semantics (and in documentation!).

Concepts and Instances

There are two types of objects in **KB**, *concepts* and *instances*. A concept represents a class or collection. A concept may contain a set of *roles* which describe properties of the concept (and are similar to members of a structure). A concept may also inherit roles from a previously defined concept. In the object-oriented programming metaphor, a concept is called an *abstract data type* and roles are called *instance variables*.

An instance is a specific incarnation of a concept. To request an action or access the value of a role of an instance, a *message* must be sent to the instance. The action which is performed by an instance upon receiving a particular message is defined by the instance's *message handler* for the message. In the object-oriented programming metaphor, a message handler is called a *method*.

The following example illustrates the definition of a concept (the system prompt is "kb>").

```
kb> (defconcept circle () (center radius))
CIRCLE
kb> (defhandler (circle :diameter) ()
      (* 2 radius))
:DIAMETER
kb> (defhandler (circle :translate) (dx dy)
      "Move the center of the circle."
      (let ((point (send-message self :center)))
        (send-message self
             :center (list (+ (car point) dx) (+ (cadr point) dy)))))
:TRANSLATE
kb>
```

The **defconcept** macro creates a concept called `circle`. The circle concept has two roles: the coordinates of the center of the circle, and its radius. Every instance of a circle will have its own set of values for these roles.

The syntax of **defhandler** is nearly identical to that of **defun**. The function name is replaced by *(concept message)*. The lambda list is of the same form as, and has the full generality of, a **defun** lambda list.

In this example, the messages were keywords, but that is not a requirement of the **KB** package. In **KB**, a message can be any atom. The print name of the symbol is used in determining the specific message name. This allows one to avoid difficulties which may arise due to using the package facility of Common Lisp.

The handlers that were defined in this example compute the diameter of a circle and translate the center point. That is, if an instance receives the message `:diameter`, it should return two times the radius and if it receives the message `:translate`, it should translate the value of its `center` role (which is a list of two numbers) by the amount of its two arguments.

In this example, there were two additional message handlers implicitly defined by **defconcept**. The macro **defconcept** automatically defines one message handler for each of its roles. The corresponding handler is invoked whenever a message which is the name of a role is sent. Handlers of these types optionally have a single argument. If the argument is not specified, the value of the role is returned. If the argument is specified, the value of the role is set to the argument and the value of the role is returned. Although these handlers are defined automatically, they are of a similar nature to those defined explicitly using **defhandler**. Through the use of options described below, these implicitly-defined "role" message handlers provide a powerful facility for operating on the data.

Sending Messages

The function **send-message** sends a message to an instance. The syntax of **send** is:

send-message *instance message* **&rest** *args* [*Function*]

The **KB** package automatically finds the appropriate handler corresponding to the message. The remaining arguments are passed to the handler and are bound to the variables in the lambda list of the **defhandler**.

Modifying the "Role" Message Handlers

It is possible to modify a "role" message handler so that special actions will be taken when the role value is set or accessed.

```
kb> (defconcept inc () ((number :if-needed (setf number (1+ number))
                                :if-set (setf number (1- new-value)))))
INC
kb>
```

Whenever the value of the role `number` is accessed, the current value will be incremented and that modified value will be stored and returned. Whenever the value of the role is set, the actual value assigned to the role will be one less than that which was passed as an argument. This uses the fact that after a role is set, a message is sent to the instance to get the value (and that is returned).

The values that a role of a particular concept can take on can be limited to a specific range. Alternatively, it is possible to give a predicate form which when evaluated will indicate whether the new value for the role is acceptable.

```
kb> (defconcept person () (name
                           (gender :range '(male female))
                           (age :range-predicate (numberp new-value))))
PERSON
kb>
```

The `gender` role of the concept `person` is defined such that only the values `male` or `female` are valid. Any attempt to use the "role" message handler to assign a different value will fail.

The **age** role will only allow numeric values. The variable **new-value** is defined by **KB** to be the value that is to be assigned to the role. The **:range-predicate** can be arbitrarily complex.

Creating Instances

To create an instance of a concept, use the **create-instance** macro. The first argument to **create-instance** is the name of a concept. There can be additional optional arguments which will cause messages to be sent to the newly created instance. An instance identifier (actually, an uninterned symbol) is returned.

For example, entering `(setf a-circle (create-instance 'circle))` creates an instance of a circle. However, the instance variables are unbound, so the "role" message handlers can be invoked to assign values:

```
kb> (send-message a-circle :center '(0 0))
(0 0)
kb> (send-message a-circle :radius 5)
5
kb>
```

There is an alternate way to specify initial values. The messages can be included as arguments to **create-instance**:

```
kb> (create-instance 'circle :send-message :center '(0 0)
                             :send-message :radius 5)
```

Alternatively, the concept can be defined so that whenever an instance is created, the roles are automatically given initial values:

```
kb> (defconcept circle ()
                ((center :default-value (list 0 0))
                 (radius :default-value 5)))
CIRCLE
kb>
```

Whenever an instance is created, the default expressions will be evaluated and assigned as the initial values of the roles.

Role and Message Handler Inheritance

A concept may inherit roles and message handlers from other concepts. Inheritance allows you to take a general concept and build a more specialized concept on top of it. The concept **man** can inherit roles and methods from the concept **person**, with one distinction being the value of the role **gender** is **male**. Similarly, the concept **woman** would set the value of the **gender** role to **female**. For both men and women, it should not be possible to change the value of that role.

```
kb> (defconcept person () (name age gender) "A person")
PERSON
kb> (defconcept man (person) ((gender :fixed-value 'male)))
MAN
kb> (defconcept woman (person) ((gender :fixed-value 'female)))
WOMAN
kb>
```

The concepts **man** and **woman** inherit the roles **name** and **age** from the concept **person**, as well as the message handlers for those roles. The role **gender** is not inherited, but is set to a fixed value appropriately. In this case, it is not possible to change the value of gender.

As can be seen, the second argument to **defconcept** specifies the parent concepts. Parent concepts must be defined before they are referenced. This eliminates the possibility of inadvertently creating a circular reference.

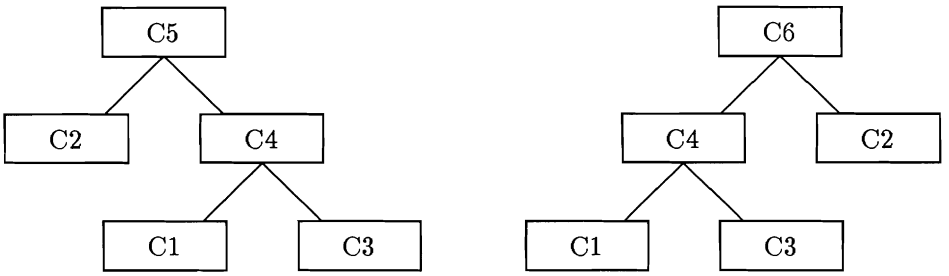
The order of role and message handler inheritance is important since the same role or handler may be defined in different parents. The search order is based upon the order of parent concepts in the **defconcept** macro call. The concept being defined can be thought of as the root of an N-ary tree with a link pointing to each of its parents (the terminology is somewhat inverted from the normal tree terminology). Similarly, each parent is the root of its own N-ary tree. The order of the nodes in the N-ary tree is the same as was listed in the third argument to **defconcept**. This tree is searched in pre-order (root node, left node, right node) to find roles and handlers. The searching always starts from the root of the tree.

Suppose the following concepts were defined:

```
kb> (defconcept c1 () (a b c))
C1
kb> (defconcept c2 () (d e f))
C2
kb> (defconcept c3 () (a d g))
C3
```

```
kb> (defconcept c4 (c1 c3) ())
C4
kb> (defconcept c5 (c2 c4) (b))
C5
kb> (defconcept c6 (c4 c2) (b))
C6
kb>
```

For concepts `c5` and `c6`, the trees would be drawn as follows:



Thus, `c5` will inherit roles `d, e,` and `f` from concept `c2`, roles `a` and `c` from concept `c1`, and role `g` from concept `c3`. The role `b` is not inherited.

Concept `c6` will inherit roles `a and c` from concept `c1`, roles `d` and `g` from concept `c3` and roles `e` and `f` from concept `c2`. The role `b` is not inherited.

The Root Concept

The concept `kb::root-concept` is a system-provided concept that supplies a set of useful message handlers. This concept is automatically inherited when a new concept is defined (it is placed as the right-most parent), but will not show up on any list of ancestors. This concept defines handlers for the following messages:

- `:unclaimed`
- `:describe`
- `:which-messages`
- `:message-handledp`
- `:send-if-handled`

These message handlers should not be redefined for the concept `kb::root-concept`. With the exception of the message handler for `:describe`, it should not be necessary to define message handlers for new concepts to handle these messages. Furthermore, should a message handler for `:describe` be defined, it must have the same calling sequence as the system defined message handler.

Relationship to Flavors

There are several distinctions between the **KB** package and Flavors:

1. **KB** tracks inheritance relationships and makes that information available to an application.
2. **KB** maintains a lists of instances making it easy to operate on them as a group.
3. The syntax and semantics of **KB** is such that it is easier to specify ranges of values, and those ranges are more syntactically visible.
4. Flavors allows a finer grain of control over mixing flavors and method inheritance.

compile-handlers

Purpose: The variable ***compile-handlers*** controls whether newly-defined message handler functions are compiled.

Syntax: ***compile-handlers*** [*Variable*]

Remarks: If the variable ***compile-handlers*** is **nil**, message handler functions are not compiled. If the variable ***compile-handlers*** is not **nil**, message handler functions are compiled.

If a message handler is to be debugged, then it must not be compiled. To do this, the value of ***compile-handlers*** at the time the handler is defined must be **nil**.

concept-instancep

Purpose: The function **concept-instancep** tests whether its argument *object* is an instantiation of a concept.

Syntax: **concept-instancep** *object* [*Function*]

Examples:

```
kb> (defconcept example () (A B))
EXAMPLE
kb> (setf instance (create-instance 'example))
#:EXAMPLE524
kb> (concept-instancep instance)
T
kb> (concept-instancep 'foo)
NIL
kb>
```

concept-list

Purpose: The function **concept-list** returns a list of all defined concepts.

Syntax: **concept-list** [*Function*]

Examples:

```
kb> (defconcept example () (A B))
EXAMPLE
kb> (concept-list)
(EXAMPLE KB::ROOT-CONCEPT)
```

conceptp

Purpose: The function **conceptp** tests whether its argument *object* is a named concept.

Syntax: **conceptp** *object* [*Function*]

Examples:

```
kb> (defconcept example () (A B))
EXAMPLE
kb> (conceptp 'example)
T
kb> (conceptp 'foo)
NIL
kb>
```

create-instance

Purpose: The macro **create-instance** creates an instance of a concept.

Syntax: **create-instance** *concept* {`:send-message` *message* `&rest` *args*}* [*Macro*]

Remarks: The *concept* argument is a quoted concept name. It is evaluated immediately upon executing the macro.

Each message and corresponding set of message parameter arguments are evaluated after the previous message has been sent. These arguments are evaluated with the symbol **self** bound to the instance being created. This allows the newly-created instance to send messages to itself.

After all the messages have been processed, the message **:required-roles-setp** is sent to the newly-created instance to verify that all required roles have been bound.

This macro returns two values: an instance identifier (an uninterned symbol) and the result of sending the message **:required-roles-setp** to the new instance.

Any number of messages may be sent to the newly created instance, but the return value of the message handlers are not available. That is,

```
(create-instance 'foo :send-message msg1 'arg :send-message msg2)
```

is equivalent to

```
(let ((instance (create-instance 'foo)))
  (send-message instance msg1 'arg)
  (send-message msg2)
  (values instance (send-message instance :required-roles-setp)))
```

Examples:

```
kb> (defconcept foo () (a b c))
FOO
kb> (create-instance 'foo
                     :send-message :a 10
                     :send-message :b 20
                     :send-message 'c 30))
#:FOO429
T
kb> (send-message '#:FOO429 :describe)
Instance 429 of FOO
A: 10
B: 20
C: 30
kb>
```

defconcept

Purpose: The macro **defconcept** creates a new concept.

Syntax: **defconcept** *concept-name* [*Macro*]
({ *concept* }*)
({ *role* | (*role* [**:default-value** *form* | **:fixed-value** *form*]
[**:if-needed** *form*] [**:if-set** *form*]
[**:range** *value* | **:range-predicate** *form*]
[**:required** *value*])}*)
documentation

Remarks: The *concept-name* argument is the name of the concept being created. This argument should not be quoted.

The second argument is a list of parents from which roles and message handlers should be inherited. Roles and handlers are inherited from the first ancestor (in a preorder traversal – left-to-right) defining that entity.

The third argument is a list of roles of the concept.

The last argument is ignored. It can be used as a documentation string.

For the second argument, if the list form for specifying the role is used, any of the keywords-value pairs may be left unspecified. **:default-value** indicates an initial value for the role. **:fixed-value** indicates a value for the role, but also indicates that the value of the role can not be changed. The *form*s associated with these keywords are evaluated whenever an instance is created. If neither of these keywords are specified, the role is initially unbound. Only one of **:default-value** and **:fixed-value** may be specified.

The **:if-needed** *form* is evaluated when a message is sent to an instance requesting the value of a role. If the **:if-needed** keyword is not specified, the current value of the role is returned.

The **:if-set** *form* is evaluated when a message is sent to an instance setting the value of a role. If the **:if-set** keyword is not supplied, the role is set to the new value and the current value of the role is returned.

:range *value* is a quoted list of valid values for the role. **:range-predicate** is a form which is evaluated when the value of a role is being set. If the *form* evaluates to **t**, the new value is allowed. If the *form* evaluates to **nil**, the new value is not allowed and the role value will not be changed. Only one of **:range** and **:range-predicate** may be specified. If neither **:range** nor **:range-predicate** is specified, the role may take on any value.

The **:required** *value* indicates whether the role must be bound to a value when an instance of this concept is created. *value* is not evaluated and should be either **t** or **nil**.

The variable **self** may be used in the **:if-needed**, **:if-set**, and **:range-predicate** forms. The variable **new-value** may be used in the **:if-set** and **:range-predicate** forms.

To get the current value of any role from within the **:if-needed**, **:if-set**, or **:range-predicate** forms, the role name should be used. It is suggested that, in general, only the current role name should be used explicitly and the values of the other roles should be obtained by sending messages.

Name shadowing is not handled correctly for role names within a **defconcept**. Do not create any local variables within the *forms* with the same name as any role.

When a concept with roles is defined, a message handler is automatically created for each role. The message name is the name of the role and it automatically has one **&optional** argument. If the "role" message is sent to an instance with no arguments, the value of the role is returned (as specified by the **:if-needed** keyword). If the "role" message is sent to an instance with the optional argument specified, then the argument is taken to be the new value for the role and it is set as specified by the **:if-set** keyword. The return value when setting a value is determined by recursively invoking the "role" message handler with no arguments.

Examples:

```
kb> (defconcept ex1 () (a))
EX1
kb> (setf in1 (create-instance 'ex1))
#:EX1549
kb> (send-message in1 :a)
KB::UNBOUND
kb> (defconcept ex2 () ((a :default-value 'foo)))
EX2
kb> (setf in2 (create-instance 'ex2))
#:EX2550
kb> (send-message in2 :a)
FOO
kb> (defconcept ex3 () ((a :fixed-value 'bar)))
EX3
kb> (setf in3 (create-instance 'ex3))
#:EX3551
kb> (send-message in3 :a)
BAR
kb> (send-message in3 :a 'test)
Can't change fixed-value role A.
BAR
kb> (defconcept ex4 () ((a :if-needed (progn (princ "Getting a")
                                             (if (numberp a) (+ a 10) a)))))
EX4
```

```
kb> (setf in4 (create-instance 'ex4 :send-message :a 10))
Getting a
#:EX4552
kb> (send-message in4 :a)
Getting a
20
kb> (send-message in4 :a 'foo)
Getting a
FOO
kb> (defconcept ex5 () ((a :if-set (progn (princ "Setting a")
                                          (if (numberp new-value)
                                              (setf a (+ new-value 10))
                                            (setf a new-value)))
                           :required t)))
EX5
kb> (setf in5 (create-instance 'ex5))
Required role A is not set.
#:EX5553
kb> (send-message in5 :a 23)
Setting a
33
kb> (send-message in5 :a 'foo)
Setting a
FOO
kb> (defconcept ex6 () ((a :range '(a b c))))
EX6
kb> (setf in6 (create-instance 'ex6))
#:EX6554
kb> (send-message in6 :a 'foo)
Can't set role A to FOO.
KB::UNBOUND
kb> (send-message in6 :a 'c)
C
kb> (defconcept ex7 () ((a :range-predicate (numberp new-value))))
EX7
kb> (setf in7 (create-instance 'ex7 :send-message :a 10))
#:EX6555
kb> (send-message in7 :a 'foo)
Can't set role A to FOO.
10
kb>
```

defhandler

Purpose: The macro **defhandler** creates or modifies a message handler used by an instance.

Syntax: **defhandler** (*concept message*) *lambda-list* [*Macro*]
{*declaration* | *documentation*}* {*form*}*

Remarks: The *forms* of the handler are evaluated whenever *message* is sent to an instance of *concept*. Message handlers can be defined at any time and are immediately available to instances.

The variable **self** is bound to the instance to which the message has been sent.

Role values may be accessed directly by a message handler by using the role name. It is suggested, however, that role values be set and obtained by sending the appropriate messages.

Name shadowing is not handled correctly for role names within a **defhandler**. Do not create any local variables within the *forms* with the same name as any role.

When a handler is defined, the **KB** package either defines or compiles (depending upon the value of the variable ***compile-handlers***) a function whose name is a composite of *concept* and *message* of the form *concept-message*.

Examples:

```
kb> (defconcept circle () ((center :default-value '(0 0))
                           radius))
CIRCLE
kb> (defhandler (circle :diameter) (&optional (value nil))
      "Set or get the diameter of the circle."
      (if value
          (send-message self :radius (/ value 2)))
      (* 2 (send-message self :radius)))
:DIAMETER
kb> (setf instance (create-instance 'circle))
#:CIRCLE421
kb> (send-message instance :diameter 10)
10
kb> (send-message instance :radius)
5
kb> (defhandler (circle :translate) (dx dy)
      "Move the center of the circle."
      (let ((point (send-message self :center)))
        (send-message self
              :center (list (+ (car point) dx) (+ (cadr point) dy)))))
:TRANSLATE
```

```
kb> (send-message instance :center)
(0 0)
kb> (send-message instance :translate 5 -1)
5 -1
kb> (send-message instance :translate 1 2)
6 1
kb>
```

delete-instance

Purpose: The function **delete-instance** deletes an instance of a concept.

Syntax: **delete-instance** *instance* [*Function*]

Remarks: When an instance is deleted, knowledge of that instance is removed from **KB**.However, references to that instance may remain.

Only use this function when you are experimenting with the **KB** package.

Examples:

```
kb> (defconcept foo () (a))
FOO
kb> (create-instance 'foo)
#:FOO439
kb> (send-message '#:foo439 :a 10)
10 kb> (delete-instance '#:foo439) T kb> (send-message '#:foo439 :a 10)
Can't send a message to #:FOO439.  Not an instance.
kb>
```

:describe

Purpose: The message handler **:describe** displays a description of an instance.

Syntax: (send-message *instance* **:describe** **&optional** *indent*) [*Message Handler*]

Remarks: This message handler is defined by the concept **kb::root-concept**. Since **kb::root-concept** is automatically inherited by all other concepts, this message handler will be inherited as well.

The description of an instance is displayed on ***standard-output*** consists of the roles and their values. The description is indented *indent* spaces.

This message handler does not return any values.

Examples:

```
kb> (defconcept example () (A B))
EXAMPLE
kb> (setf instance (create-instance 'example :send-message :a 10))
#:EXAMPLE435
kb> (send-message instance :describe)
Instance 435 of EXAMPLE
A: 10
B: KB::UNBOUND
kb> (send-message instance
                  :b (create-instance 'example :send-message :a 'foo))
#:EXAMPLE439
kb> (send-message instance :describe)
Instance 439 of EXAMPLE
A: 10
B: Instance 437 of EXAMPLE
   A: FOO
   B: KB::UNBOUND
kb>
```

describe-concept

Purpose: The function **describe-concept** returns a form which when executed will create the concept passed as an argument.

Syntax: **describe-concept** *concept-name* [*Function*]

Remarks: Currently, the form that is returned may be a somewhat more verbose version of what was actually specified to define the concept.

The form that is returned can be printed to a file and saved for posterity.

Examples:

```
kb> (defconcept example () (a))
A
kb> (describe-concept 'example)
(DEFCONCEPT EXAMPLE ((A :DEFAULT-VALUE (QUOTE KB::UNBOUND)
:IF-SET (SETF A NEW-VALUE) :IF-NEEDED A :RANGE-PREDICATE T)) ()
"Children:  NIL")
kb>
```

get-ancestors

Purpose: The macro **get-ancestors** returns a list of the names of the ancestor concepts of a concept.

Syntax: **get-ancestors** *object* [*Macro*]

Remarks: All concepts have at least one ancestor, the concept **kb::root-concept**. If the argument to this function *object* is not a concept name, **nil** is returned.

The ancestors are returned in the same order in which they are searched for role and method inheritance.

Examples:

```
kb> (defconcept grandparent () (g1))
GRANDPARENT
kb> (defconcept parent (grandparent) (p1))
PARENT
kb> (defconcept child (parent) (c1))
CHILD
kb> (get-parents 'child)
(PARENT)
kb> (get-ancestors 'child)
(PARENT GRANDPARENT KB::ROOT-CONCEPT)
kb>
```

get-children

Purpose: The function **get-children** returns a list of the names of the child concepts of a concept.

Syntax: **get-children** *object* [*Function*]

Remarks: If the argument to this function *object* is not a concept name, **nil** is returned. Thus, it is not possible to distinguish between a concept with no children and a symbol which is not a concept name.

The children are not returned in any particular order.

Examples:

```
kb> (defconcept parent () (p1))
PARENT
kb> (defconcept child (parent) (c1))
CHILD
kb> (get-children 'parent)
(CHILD)
kb> (get-children 'child)
NIL
kb>
```

get-descendents

Purpose: The macro **get-descendents** returns a list of the names of the descendent concepts of a concept.

Syntax: **get-descendents** *object* [*Macro*]

Remarks: If the argument to this function *object* is not a concept name, **nil** is returned. Thus, it is not possible to distinguish between a concept with no descendents and a symbol which is not a concept name.

The descendents are not returned in any particular order.

Examples:

```
kb> (defconcept grandparent () (g1))
GRANDPARENT
kb> (defconcept parent (grandparent) (p1))
PARENT
kb> (defconcept child (parent) (c1))
CHILD
kb> (get-children 'grandparent)
(PARENT)
kb> (get-descendents 'grandparent)
(PARENT CHILD)
kb>
```

get-instances

Purpose: The function **get-instances** returns a list of the instances of a concept.

Syntax: **get-instances** *object* [*Function*]

Remarks: If the argument to this function *object* is not a concept name, **nil** is returned. Thus, it is not possible to distinguish between a concept with no instances and a symbol which is not a concept name.

The instances are not returned in any particular order.

Examples:

```
kb> (defconcept example () (a))
EXAMPLE
kb> (create-instance 'example)
#:EXAMPLE524
kb> (create-instance 'example)
#:EXAMPLE525
kb> (get-instances 'example)
(#:EXAMPLE529 #:EXAMPLE525)
kb>
```

get-parents

Purpose: The function **get-parents** returns a list of the names of the parent concepts of a concept.

Syntax: **get-parents** *object* [*Function*]

Remarks: If the argument to this function *object* is not a concept name, **nil** is returned. Thus, it is not possible to distinguish between a concept with no parents and a symbol which is not a concept name.

The parents are returned in the same order in which they are searched for role and method inheritance.

Examples:

```
kb> (defconcept parent () (p1))
PARENT
kb> (defconcept child (parent) (c1))
CHILD
kb> (get-parents 'parent)
NIL
kb> (get-parents 'child)
(PARENT)
kb>
```

get-roles

Purpose: The function **get-roles** returns a list of the names of the roles of a concept.

Syntax: **get-roles** *object* [*Function*]

Remarks: If the argument to this function *object* is not a concept name, **nil** is returned. Thus, it is not possible to distinguish between a concept with no roles and a symbol which is not a concept name.

The roles are not returned in any particular order.

Examples:

```
kb> (defconcept parent () (p1))
PARENT
kb> (defconcept child (parent) (c1))
CHILD
kb> (get-roles 'parent)
(P1)
kb> (get-roles 'child)
(C1 P1)
kb>
```

kb::root-concept

Purpose: The concept **kb::root-concept** is the top level concept which is automatically inherited by all other concepts.

Remarks: There are no roles associated with the concept **kb::root-concept**. The concept is used strictly for the message handlers it defines. Since this concept is inherited by all other concepts, the message handlers are automatically available to every concept (but can be redefined as desired).

The following concepts are defined:

:unclaimed Invoked when an message is sent to an instance for which there is no message handler defined.

:which-messages Returns a list of messages supported by an instance.

:message-handledp A message of one argument which returns **t** if the argument is a message for which a handler is defined.

:send-if-handled Sends a message to itself only if there is a handler defined for the message.

:describe Displays to ***standard-output*** a description of the instance. The description is a list of the roles and their values.

:message-handledp

Purpose: The message handler **:message-handled** tells you if a message is handled by the instance.

Syntax: (send-message *instance* **:message-handledp** *message*) [*Message Handler*]

Remarks: This message handler is defined by the concept **kb::root-concept**. Since **kb::root-concept** is automatically inherited by all other concepts, this message handler will be inherited as well.

If *message* is handled by *instance*, **t** is returned. Otherwise, **nil** is returned.

Examples:

```
kb> (defconcept example () (A))
EXAMPLE
kb> (defhandler (example :exercise) ()
      ;;; ...
)
:EXERCISE
kb> (setf instance (create-instance 'example))
#:EXAMPLE435
kb> (send-message instance :which-messages)
(:EXERCISE A :SEND-IF-HANDLED :DESCRIBE :MESSAGE-HANDLEDP :WHICH-MESSAGES
:UNCLAIMED)
kb> (send-message instance :message-handledp :A)
T
kb> (send-message instance :message-handledp 'A)
T
kb> (send-message instance :message-handledp :FOO)
NIL
kb>
```

new-value

Purpose: The variable **new-value** is used by the role message handlers. It is bound to the value being assigned to a role.

Syntax: **new-value** [*Variable*]

Remarks: The variable **new-value** can be used in the **:if-set** and **:range-predicate** fields of **defconcept**.

The variable **new-value** is not a special variable.

Examples:

```
kb> (defconcept circle ()
                  ((radius
                      :default-value 15
                      ;; radius may not change by more than 5 units.
                      :if-set (let ((diff (- new-value radius)))
                                (setf radius
                                      (if (> (abs diff) 5)
                                          (+ radius (* 5 (signum diff)))
                                        new-value))))
                   center))
CIRCLE
kb>
```

:required-roles-setp

Purpose: The message handler **:required-roles-setp** tells you if the required roles are bound.

Syntax: (send-message *instance* **:required-roles-setp**) [*Message Handler*]

Remarks: This message handler is defined by the concept **kb::root-concept**. Since **kb::root-concept** is automatically inherited by all other concepts, this message handler will be inherited as well.

If all roles which have been marked as required when the concept was defined are bound in *instance*, **t** is returned. Otherwise, **nil** is returned and an error message is displayed.

This message handler is automatically called by **create-instance**.

select-instances

Purpose: The function **select-instances** is used to obtain a list of instances which meet arbitrary selection criteria.

Syntax: **select-instances** *selectf* `&optional` *name* [*Function*]

Remarks: The function *selectf* argument is a lambda-form or named function which is called to select the instances. It is called with a single instance as an argument. If the instance is to be selected, the selection function should return **T**. If the instance is not to be selected, the selection function should return **nil**.

The optional argument *name* is a concept name. If specified, only instances of that concept and its descendents are considered for selection. If the optional argument *name* is not specified, then all instances are considered for selection.

Examples:

```
kb> (defconcept example () (a))
EXAMPLE
kb> (create-instance 'example :send-message :a 'foo)
#:EXAMPLE524
kb> (create-instance 'example :send-message :a '(foo bar))
#:EXAMPLE525
kb> (defconcept dunsil () (a))
DUNSIL
kb> (create-instance 'dunsil :send-message :a 'dunsil-foo)
#:DUNSIL528
kb> (defun atoma (instance)
      (atom (send-message instance :a)))
ATOMA
kb> (select-instances #'atoma)
(#:EXAMPLE524 #:DUNSIL528)
kb> (select-instances #'atoma 'example)
(#:EXAMPLE524)
```

self

Purpose: The variable **self** is used by message handlers to send themselves messages. It is bound to the instance receiving the message that caused the execution of the **defhandler**.

Syntax: **self** [*Variable*]

Remarks: The variable **self** is used when a message handler wishes to make use of another message handler of the same instance.

This variable can also be used within the **:if-needed**, **:if-set**, and **:range-predicate** keywords of **defconcept**.

The variable **self** is not a special variable.

Examples:

```
kb> (defconcept circle ()
                ((radius
                    :range-predicate (ckradius self radius new-value))
                center))
CIRCLE
kb> (defhandler (circle :circumference) ()
      (* 2 pi (send-message self :radius)))
:CIRCUMFERENCE
kb>
```

:send-if-handled

Purpose: The message handler **:send-if-handled** sends *message* to *instance* only if there is a handler for that message.

Syntax: (send-message *instance* **:send-if-handled** *message* **&rest** *args*) [*Message Handler*]

Remarks: This messages handler is defined for the concept **kb::root-concept**. Since **kb::root-concept** is automatically inherited by all other concepts, this message handler will be inherited as well.

The message handler **:send-if-handled** returns whatever values that the called message handler returns. If there is no message handler defined, this message handler returns **nil**.

send-message

Purpose: The function **send-message** sends a message to an instance.

Syntax: **send-message** *instance message* **&rest** *args* [*Function*]

Remarks: The **KB** package determines which handler to invoke to handle the message. The name of the message is determined by evaluating *message* and considering the print name of the result. This allows one to avoid difficulties which may arise due to using the package facility of Common Lisp.

If no handler has been defined (or inherited) for the message, *instance* is sent the message **:unclaimed**. A default handler for this message is defined for **kb::root-concept**.

The value returned by the message handler is returned by **send-message**.

Examples:

```
kb> (defconcept a () ())
A
kb> (defhandler (a :message) () (princ "In message") nil)
:MESSAGE
kb> (defhandler (a :echo) (&key arg)
      (princ "In echo, arg is ")
      (princ arg)
      (terpri)
      arg)
:ECHO
kb> (setf instance (create-instance 'a))
#:A326
kb> (send-message instance :message)
In message
NIL
kb> (send-message instance :echo :arg 'foo)
In echo, arg is FOO
FOO
kb> (send-message instance :echo :arg t)
In echo, arg is T
T
kb>
```

unboundp

Purpose: The function **unboundp** tests whether its argument *value* is an unbound value.

Syntax: **unboundp** *value* [*Function*]

Examples:

```
kb> (defconcept example () (A B))
EXAMPLE
kb> (setf instance (create-instance 'example :send-message :a 10))
#:EXAMPLE435
kb> (send-message instance :describe)
Instance 435 of EXAMPLE
A: 10
B: KB::UNBOUND
kb> (unboundp (send-message instance :a))
NIL
kb> (unboundp (send-message instance :b))
T
kb>
```

:unclaimed

Purpose: The message handler **:unclaimed** is invoked when a message is sent to an instance for which there is no message handler defined.

Syntax: (send-message *instance* **:unclaimed** *message* **&rest** *args*) [*Message Handler*]

Remarks: This message handler is defined for the concept **kb::root-concept**. Since **kb::root-concept** is automatically inherited by all other concepts, this message handler will be inherited as well.

It prints the message, **Unclaimed message:** *messagename*. It returns the value **kb::unclaimed**.

There is no reason why this message should be sent directly to an instance.

Examples:

```
kb> (defconcept example () (A))
EXAMPLE
kb> (setf instance (create-instance 'example))
#:EXAMPLE435
kb> (send-message instance :message-with-no-handler)
Unclaimed message:  MESSAGE-WITH-NO-HANDLER
KB::UNCLAIMED
kb>
```

undefconcept

Purpose: The function **undefconcept** deletes a concept.

Syntax: **undefconcept** concept-name [*Function*]

Remarks: The function **undefconcept** will "undefine" the concept *concept-name*. The children of *concept-name* will lose that concept as a parent. Instances of the children will still keep the roles inherited from the deleted parent, however, the message handlers that are inherited from the parent will be deleted.

Only use this function when you are experimenting with the **KB** package.

It is not possible to delete the concept **kb::root-concept**.

Examples:

```
kb> (defconcept a () ())
A
kb> (conceptp 'a)
T
kb> (undefconcept 'a)
T
kb> (conceptp 'a)
NIL
kb>
```

verbose

Purpose: The variable ***verbose*** controls the printing of error messages.

Syntax: ***verbose*** [*Variable*]

Remarks: If ***verbose*** is **nil**, no error messages are printed.
The default value of ***verbose*** is **t**.

what-concept

Purpose: The function **what-concept** returns the name of the concept of which its argument *object* is an instance.

Syntax: **what-concept** *object* [*Function*]

Remarks: If *object* is not a concept instance, **what-concept** returns **nil**.

Examples:

```
kb> (defconcept example () (A B))
EXAMPLE
kb> (setf instance (create-instance 'example))
#:EXAMPLE524
kb> (what-concept instance)
EXAMPLE
kb>
```

:which-messages

Purpose: The message handler **:which-messages** returns a list of messages for which handlers are defined for the instance.

Syntax: (send-message *instance* **:which-messages**) [*Message Handler*]

Remarks: This messages handler is defined for the concept **kb::root-concept**. Since **kb::root-concept** is automatically inherited by all other concepts, this message handler will be inherited as well.

Examples:

```
kb> (defconcept example () (A))
EXAMPLE
kb> (defhandler (example :exercise) ()
      ;;; ...
)
kb> (setf instance (create-instance 'example))
#:EXAMPLE435
kb> (send-message instance :which-messages)
(:EXERCISE A :SEND-IF-HANDLED :DESCRIBE :MESSAGE-HANDLEDP :WHICH-MESSAGES
:UNCLAIMED)
kb>
```